Bull. Nov. Comp. Center, Comp. Science, 34 (2012), 93–103

Graph algorithm interactive visualization

D. S. Gordeev

Abstract.

This paper describes a new method of graph algorithm visualization based on a dynamic approach. Graph algorithms are algorithms processing graphs. The main advantages of this approach are the possibility to set an algorithm as an input parameter, to set the graph as an input parameter, and also to adjust visualization flexibly. Visualization of algorithms is carried out by means of a set of configurable visual effects. The class of hierarchical graphs is used as an input parameter. This allows using any type of input graphs and presenting additional data appearing during the algorithm work as part of a single visualized graph model. This approach can be used both in research and for education.

Keywords: graph, hierarchical graphs, algorithm, visualization, visual effect

1. Introduction

The graph theory methods have been used successfully to model various problems that arise in computer science and in practical applications. For example, constructing the syntax trees in programming, graph coloring problems in the design of electrical chains, finding the shortest way in the tasks of creating computer games, etc. More information can be found in [1]. Graph drawing is a useful way of representation of these 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. 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. There are methods presenting transformations both in a static and dynamic form. For example, Figure 1 shows visualization of a sorting algorithm with a static image. This example is described in more detail in [2].

The following figure shows an example of visualization of a dynamic system algorithm. More details about this system can be found in [3, 4]

These methods 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 visualization. The main distinguishing feature of working visualizers is their narrow specialization in the sense that a new visualizer should be created for each new algorithm.

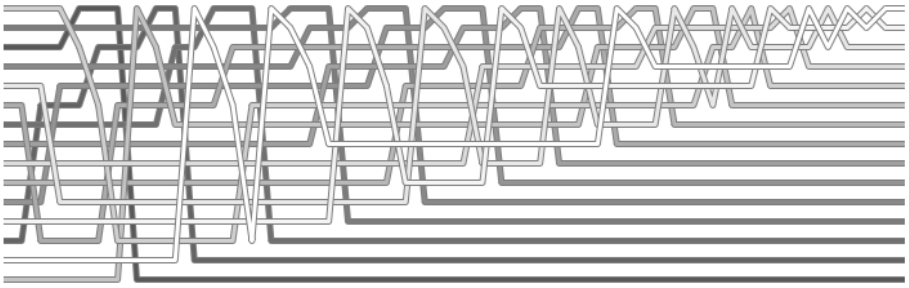

Figure 1. Visualization of a heapsort algorithm using static image

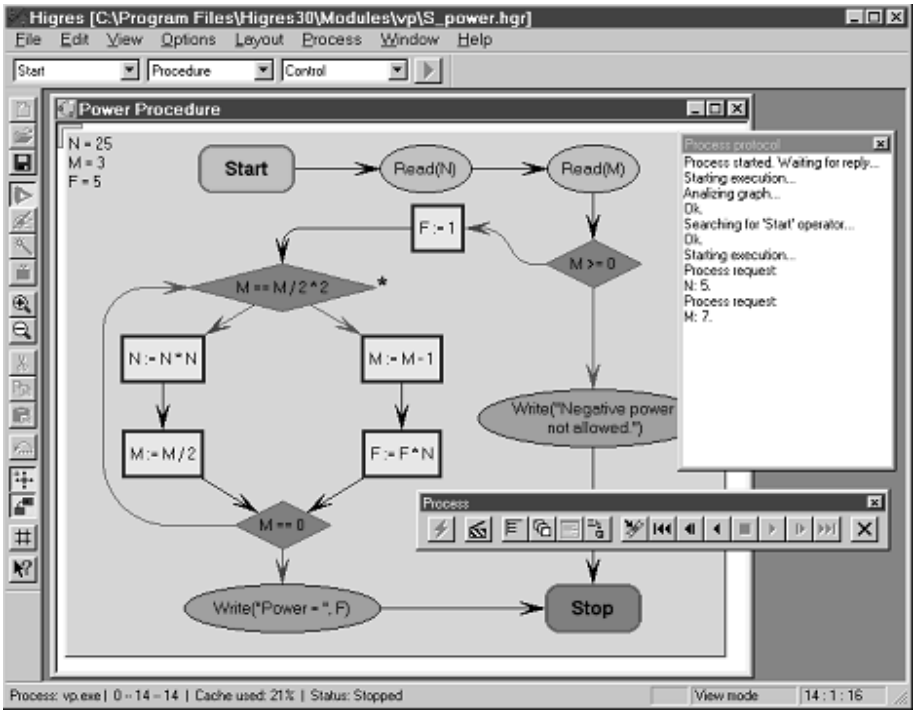

Figure 2. The algorithm is implemented in the form of an external module to the Higres system

2. Algorithm visualization methods

This paper describes a method for constructing visualizations based on visual effects generated by an input algorithm. Visualization of a graph is a graphic representation of graph elements. Usually graph elements 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 [5]. 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 a visual effect 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. Visualizers often 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. The disadvantages of some visualizers can also include excessive declarative instructions in the text of the algorithm. Figure 3 shows the frame constructed by the algorithm visualization system named Leonardo [6, 7]. As it can be seen, the Leonardo system uses directives in a specific format, /** Not VisualUpdate **/, and similar. This declarative structure is used to group visual effects or to run them directly.

3. Interactive visualization model

A new algorithm visualization model based on the dynamic approach 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 in terms of 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 [8]. 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.

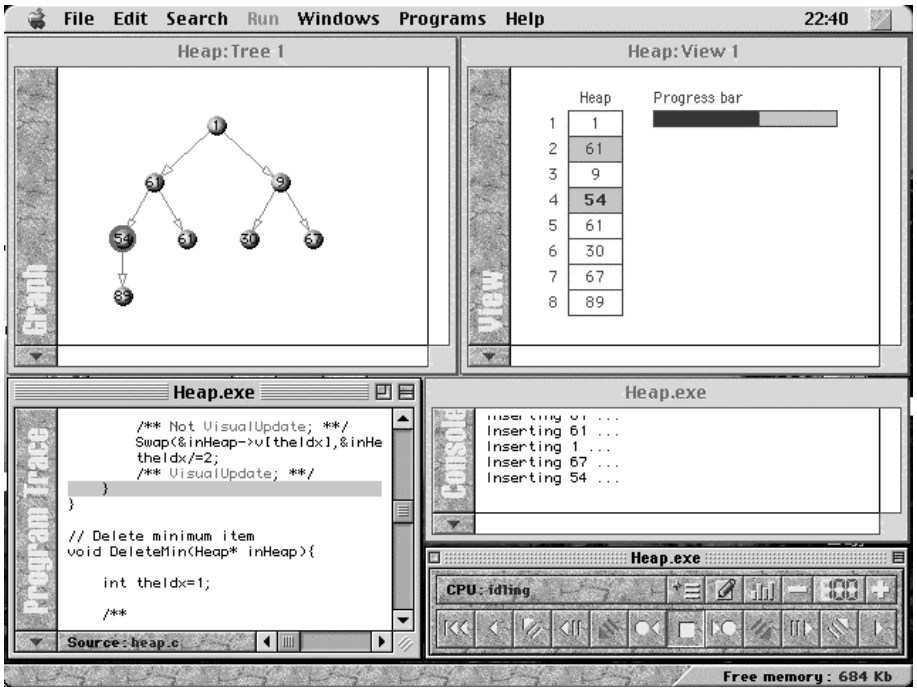

Figure 3. An example of visualization of operations on binary heaps in the Leonardo system, AC Programming Environment for Reversible Execution and Software Visualization

```
    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 a hierarchical marked graph. The hierarchical graph H is a tuple of two elements: the first is a graph G and the second is a tree of fragments. Each fragment is a subgraph of the graph G. For any two fragments U and V, only one of the following properties holds: U is a subgraph of V, V is a subgraph of U, or U equals V. More details about hierarchical graphs can be found in [1]. 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(IfCondition(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(IfCondition(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 Pruefer encoding algorithm constructs a sequence of numbers by the given tree graph. During the coding process, the vertices of the graph are removed one by one. To perform this operation, the RemoveVertex(...) instruction should be used, which leads to generation of a visual effect of the corresponding vertex disappearing. Here is an example of the Pruefer 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.

4. Algorithm visualization system

A system of graph algorithm visualization has been constructed using the suggested model. This system consists of several components: an algorithm execution module, a graph editor and a graph algorithm visualizer. It can be assumed without loss of generality that data are passed between components in a text form. This is useful if the components are 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 execution 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 restriction on the algorithm implementation language since many programming languages allow graph structures to be used in the program 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 inefficient 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 graphical 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 deep-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.

The figure below shows an example of visualization of the deep-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 deep-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 state 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.

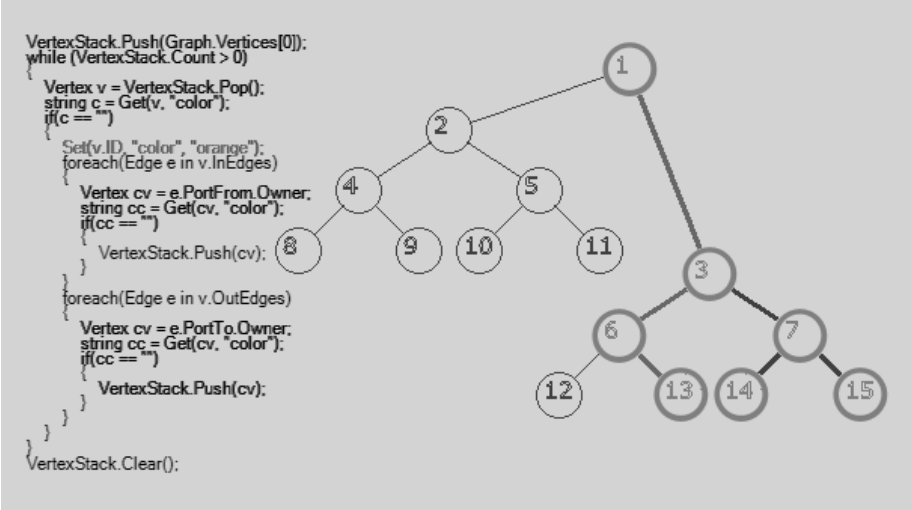

Figure 4. Visualization of the deep-first search algorithm. This is one of intermediate images

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 deep-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. Figure 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 ren-

dering 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 according 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 deep-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 represented 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.

5. Conclusion

This paper describes the model of interactive visualization of graph algorithms, providing the capability to build the algorithm visualization with the help of a flexible system of visual effects and using the algorithm as an input parameter. Also, the paper describes a method to improve understanding of the graph algorithm visualization using additional information generated during execution of the input algorithm. A system for graph algorithm visualization has been created. It implements visualization in two steps: first, the algorithm text is transformed into a program 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 visualization system allows testing of the proposed method of graph algorithm visualization. A substantial set of graphic effects significantly improves control of the algorithm visualization. Interactivity of visualization is supported by the capability to configure visual effects, to change the text of the algorithm and to build visualization once again. The implemented system allows us to observe the performed changes immediately. Any hierarchical graph can be a parameter for the visualization system. The class of algorithms admissible for our system is a subject for further research. At the moment, no restrictions are found and the set of implemented visual effects is predefined. An opportunity to extend the set of effects dynamically also is a matter for further research.

References

[1] Kasyanov V.N., Yevstigneyev V.A. Graphs in programming: processing, visualization and application.– SPb. BHV-Petersburg, 2003. – 1104 with. silt. ISBN 5-94157-184-4

[2] http://corte.si/posts/code/visualisingsorting/

[3] Lisitsyn I.A., Kasyanov V.N. Higres — visualization system for clustered graphs and graph algorithms // Proc. of Graph Drawing 99. – Berlin a.o.: Springer Verlag, 1999. – P. 82–89. — (Lect. Notes in Comput. Sci.; Vol. 1731).

[4] Higres system. – Available at http://pcosrv.iis.nsk.su/higres/

[5] Demetrescu C., Finocchi I., Stasko J.T. Specifying Algorithm Visualizations: Interesting Events or State Mapping? // Proc. of Dagstuhl Seminar on Software Visualization – Lect. Notes Comput. Sci. – 2001. – P. 16–30.

[6] Demetrescu C., Finocchi I. 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, Czech Republic, February 1999.. – P. 55–62.

[7] Leonardo system. – Available at
http://www.dis.uniroma1.it/∼demetres/Leonardo/

[8] Gordeev D.S. Model of interactive visualization of graph algorithms. // Works of KIS 2011 / Working seminar "The knowledge-intensive software". – Novosibirsk: A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, 2011. – P. 58–62.