Reasoning about graph algorithm visualization

Dmitry S. Gordeev
A.P. Ershov Institute of Informatics Systems,
Siberian Branch of the Russian Academy of Sciences
Novosibirsk, Russia
gds@iis.nsk.su

ABSTRACT

A method of graph algorithm visualization based on an implicit visual effect generation approach is described. The approach develops an idea to establish an algorithm as an input as well as input graph. Visualization of algorithms is carried out by means of a set of configurable visual effects. We consider a class of hierarchical graphs as a class of input graphs. This allows using wide set of input graphs and presenting additional data appearing during the algorithm work as part of a single visualized graph model. Described approach is be used both in research and education.

Keywords:
Graph, hierarchical graph, algorithm, visualization, visual effect.

INTRODUCTION

Algorithm visualization is a recurring subject in the field of data visualization. Researchers mostly use them in order to build some educational courses in computer science field [1, 2]. According to review papers a lot of algorithm visualization works are focused on construction new visualization for specific algorithm or for very limited number of ones, and also have different characteristics of effectiveness of visualization. It's remarked in reviews that it's quite difficult to create effective algorithm visualization. Thus most of works have low quality and often it's hard to find working samples for demonstration in education class.

Most of these visualizations cover simple cases of algorithms and data structures. The review paper [2] remarks that there is need for visualization working with more complex data structures like B-trees or NP-hardness. It does make sense in order to familiarize students with more complex concepts of computer science. Often authors of review makes conclusion that there is need for estimation of algorithm visualization effectiveness. Usually they estimate effectiveness in context of time required to understand concept, which visualization presents. Also review authors mark that effectiveness of visualizations is better if students are involved into a process of construction of algorithm visualization. If students have opportunity to interact with visualization then it produces more denotable effect in education.

This paper introduces the approach to construction of algorithm visualization constrained for graph algorithms. It's noted in paper [2] that constrains with domains allow to make corresponding assumption which lead to improvement of expressiveness and effectiveness of algorithm visualizations. We consider only graph algorithms. It means that a text of an algorithm is written on terms of the graph theory.

RELATED WORK

There are many tools to create algorithm visualization. Most of them focus on creation one-time visualization of particular algorithm which can be stored as video record or number of pictures, and later can be demonstrated in education class. Also there are several tools of software visualization field which show work of some program written with programming language. In this approach state of memory is been visualized [5,6,8,9,11]. Also there are approaches using static images in order to show dynamic process [4]. Also there are works which proceed with limited fix set of algorithms [12].

This does not allow user to change input algorithms what would be high degree of freedom in interaction with algorithm visualization. According to reviews [2] it would be useful to have opportunity to interact with visualization process for education purposes.

VISUALIZATION

In this work we consider approach in which we limit only graph algorithms instead of general algorithms. It means that we consider algorithm working only with graphs. It allows us to select events expressed in terms of graphs. For example adding of vertex, remove edge from subgraph or change attribute value of tree node or enumerate nodes incident to given one. And we can describe graph

 ISBN 978-80-86943-67-1

algorithms in such terms with using some programming language [10]. In this form algorithm can be compiled into executable program. 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. Here is we have the first aspect of visualization interactivity. We change algorithm text and get a visualization with another properties without additional efforts. 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 a traversal of a graph.

```
VertexQueue.Enqueue(Graph.Vertices[0]);
while (VertexQueue.Count > 0)
{
   Vertex v = VertexQueue.Dequeue();
   Set(v, "state", "visited");
   foreach(Edge e in v.Edges)
  {
      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 [3]. 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 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(4);
   Set(5, v.ID, "state", "visited");
   foreach(Edge e in ForCollection(6, v.InEdges))
  {
      Vertex t = e.PortFrom.Owner;
      string c = Get(9, t, "state");
      if(IfCondition(10, c != "visited"))
     {
         Set(12, t, "state", "visited");
         VertexQueue.Enqueue(13, 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 history of operations executed over graph elements. This operation history contains the detailed information on the state of the graph model during the algorithm running. Further the history of operations, the input graph and the original text of the algorithm are used to generate the algorithm visualization. Each operation history element 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. Here is the second aspect of visualization interactivity. We changes an assignment any particular visual effect and get changed.

EXPERIMENTS

A system of graph algorithm visualization has been constructed based on suggested approach. 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 history. The algorithm running is separated from its visualization. This allows performing the algorithm once and after that the operation history can be used to visualize and refine the visualization as many times as needed. This can be useful when computationally-intensive algorithms are visualized since the second cycle of execution of the algorithm is complex in such cases. To provide correct work of the algorithm execution module, it is necessary to meet a significant condition related to the algorithmic complexity. It is reasonable to visualize only efficient algorithms, because it will take much time to build the operation history of execution of an inefficient algorithm. Efficient algorithm means that an algorithm of polynomial complexity. We can use a small input graph for this case. This assumption allows constructing of visualization for a reasonable time.

The algorithm execution module takes the given algorithm text written down with a 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 other events related to input graph elements during the execution. Further this information is used to generate graphical representation of process.

Another component of the visualization system is the visualizer itself. This component receives the algorithm text, the graph, the history of operations and additional graphical options. A history 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 making 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 history 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 history 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 history element 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 has not been visited yet.

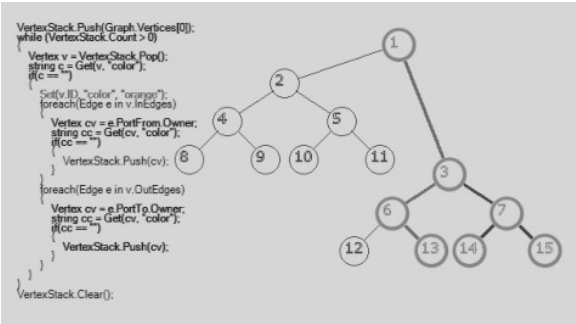

Fig. 1. Visualization of the deep-first search algorithm. This is one of intermediate images.

Displaying of additional data structures can 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.

ACKNOWLEDGMENT

This work was supported in part by the Russian Foundation for Basic Research under grant N 15-07-02029.

CONCLUSIONS AND FUTURE WORK

This paper describes the approach of visualization of graph algorithms, providing the capability to build visualization with the help of a flexible system of visual effects and using the algorithm as an input parameter.

The system of graph algorithm visualization has been created in order to test proposed approach. The system uses runtime information of algorithm execution in order to construct corresponding visual effects. Visual effects are used to reflect intermediate states of graph model during algorithm execution.

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 after restart of execution. Simple hierarchical graphs are considered to be a parameter for the visualization system. The class of algorithms admissible for the system is a subject for further research.

At present, the described method is used for construction of a visualization subsystem of a cloud parallel programming system being under development at Institute of Informatics Systems in Novosibirsk.

REFERENCES

[1] Christopher D. Hundhausen, Sarah A. Douglas, John T. Stasko, A Meta-Study of Algorithm Visualization Effectiveness // Journal of Visual Languages & Computing, - Volume 13, Issue 3, - June 2002, - pp. 259-290.

[2] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. 2010. // Algorithm Visualization: The State of the Field.*Trans. Comput. Educ.* - 10, 3, Article 9, - 22 pp.

[3] 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.

[4] Sorting algorithm visualizations [electronic source]. Available from http://sortvis.org/visualisations.html (accessed 01.05.2015).

[5] 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).

[6] Higres graph drawing system [electronic source]. Available from http://pcosrv.iis.nsk.su/higres/ (accessed 01.05.2015).

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

[8] C. Demetrescu and I.Finocchi, A general-purpose logic-based visualization framework, Proceedings of the 7th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99), pp. 55-62, Plzen, Czech Republic, February 1999.

[9] Leonardo programming environment [electronic source]. Available from http://www.dis.uniroma1.it/~demetres/Leonardo/ (accessed 01.05.2015).

[10] Gordeev D.S. Model of interactive visualization of graph algorithms. // Works of KIS 2011 /

Workshop: The knowledge-intensive software. - Novosibirsk: IIS SB RAS, 2011. - pp. 58-62.
[11] Moreno A., Sutinen E., Joy M. Defining and evaluating conflictive animations for programming education: the case of jeliot ConAn. // Proceedings of the 45th ACM technical symposium on Computer science education. - 2014. - pp. 629-634.
[12] Naps, T. L., Rößling G JHAVÉ–More Visualizers (and Visualizations) Needed. // Electronic Notes in Theoretical Computer Science. - 2007. - V.178, I. 0 – pp. 33-41.