CHR-Graph: A Platform for Animating Tree and Graph Algorithms

Nada Sharaf, Slim Abdennadher
Computer Science and Engineering Department
The German University in Cairo
Cairo, Egypt
{nada.hamed, slim.abdennadher@guc.edu.eg}

Thom Frühwirth
Institute of Software Engineering and Compiler Construction
University of Ulm
Ulm, Germany
thom.fruehwirth@uni-ulm.de

***Abstract*—Trees and graphs are two data structures that are commonly used in representing different kinds of data. They also have many associated algorithms taught in many courses. It is thus beneficial to have a tool that could be used by students, teachers and programmers to visually trace how their algorithms work. The work in this paper presents, CHR-Graph, an easy-to-use platform for animating trees and graphs and their correlated algorithms.**

***Index Terms*—Algorithm Animation, Tree, Graph**

I. Introduction

In [1], a meta-study of 24 experimental studies was performed. Through the analysis done it was found that algorithm visualizations are educationally effective. Algorithm animation could also help programmers to debug their code. Graphs and Trees are among the commonly used data structures. They have a lot of associated algorithms taught in different computer science courses around the world. It is thus important to have tools to visualize such data structures. The visualization should be generic. It should thus be able to adapt to any change in the data or the algorithm. Despite of the existence of libraries and tools (e.g. [2], [3]) to draw graphs and trees, most the tools require the user to know many details to be able to adjust the visualization and are thus not easy to use. In order to use a tool such as graphviz [3], the user has to learn how to express the graphs using the language "Dot". Most existing tools (such as draw.io) also do not show a step-by-step animation of the produced graph. It is just shown at once. Instead of building an animation suited to their needs, teachers would instead use available Youtube videos to demonstrate the effect of the different algorithms to their students. Instructors are sometimes provided with animations along with their textbooks. However, they cannot customize them.

The aim of the work presented in this paper is to thus provide a tool that could be easily used by teachers, programmers or students. The tool should not require users to have any special installations done. They also do not need to know of any details regarding how graphs and trees are drawn. Graphs and trees should be also described in an intuitive and declarative method.

The paper is organized as follows: Section II discusses the applied methodology. Section III introduces the features of the provided animation. Section IV discusses some implementation details. Finally, conclusions and directions to future work are given.

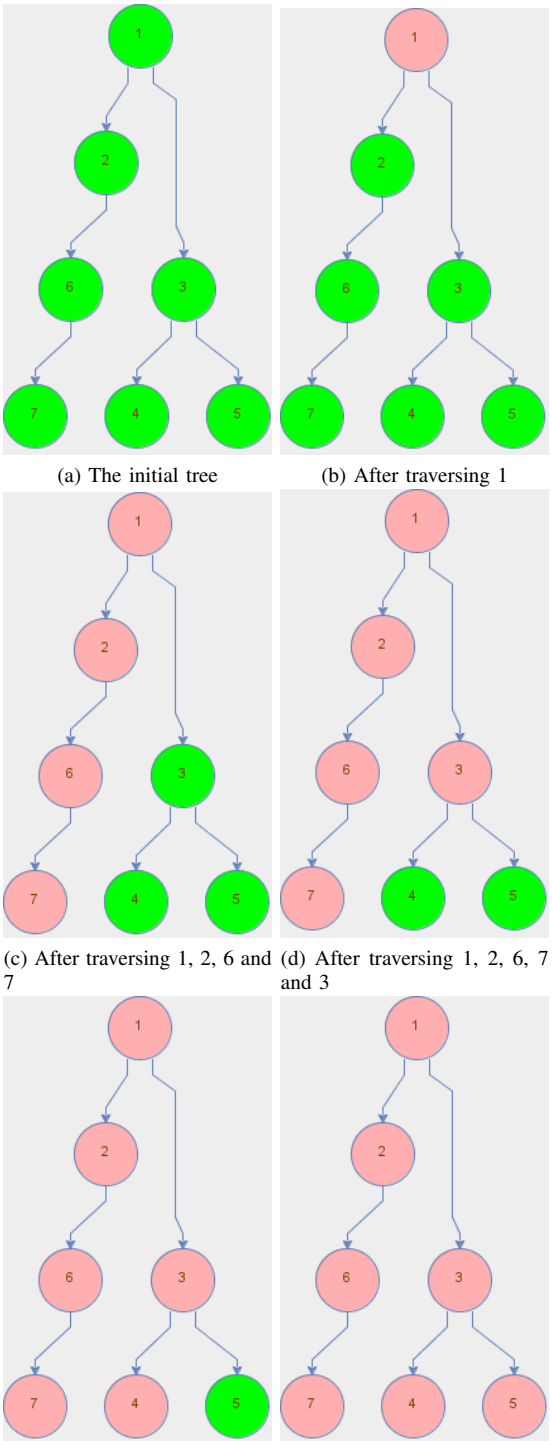
II. Methodology

In order to keep the tool as generic as possible, details of the underlying (graph) drawing utility and algorithms were outsourced to jgraph [4]. The aim is to waive the user from the need to know/implement any of the drawing related tasks. The goal is to have the user use a descriptive methodology to specify the needed graph. The drawing utility is then able to draw the required graph. The user is thus not exposed to any implementation details. To use the animation system, they have to provide a file containing the commands of the animation. The commands are abstract showing the basic information needed. The architecture is similar to that used in the Jawaa [5] animation library where the user has to write the required animation file only.

A. Commands

The commands that could be provided in the animation file are described in this section. Each command has to be written in a separate line.

- *layout:*
  To specify the required layout, the command `layout` is used. The user has to specify one parameter which corresponds to the requested layout type. Another optional parameter could be provided to state whether a specific orientation is needed. If the extra parameter was not provided, the default orientation is used.
  For example the line
  `layout mxCompactTreeLayout`
  makes sure that the graph is built according to the Moen placement algorithm [6].
- *node:*
  Each node is specified using the command `node`. The command has three required parameters corresponding to the name/id, text and color respectively. Two extra parameters could be added to specify the x and y coordinates. If they were not provided, a default value of `0` is used.
  For example
  `node 1 A green`



(a) The initial tree (b) After traversing 1

(c) After traversing 1, 2, 6 and 7 (d) After traversing 1, 2, 6, 7 and 3

(e) After traversing 1, 2, 6, 7, 3 and 4 (f) After traversing 1, 2, 6, 7, 3, 4 and 5

Fig. 1. Visualizing Depth-First Traversal of a Tree

adds a new `green` node with the id `1`. The text inside this node is `A`

- *edge:*
  In order to be able to add an edge, the command `edge` is used. An edge has two parameters corresponding to the identifiers of the source and target nodes respectively. An additional parameter could be added to specify a label for the edge. For example
  `edge 1 2 edge1-2`
  adds a new edge from node `1` to node `2`. The edge is labeled with `edge1-2`.
- *updateNode:*
  The command `updateNode` is used to change a property of an existing node. It thus requires three parameters: the name/id of the target node, the parameter that is to be changed and its new value. For example
  `updateNode 1 color red`
  updates the color of the node with id `1` to be `red`.

The user can use the system in different ways:

1) They can write the animation files using the commands stated in this section.
2) The user can also implement their algorithm (such as depth-first traversal). They can then annotate their programs using the methodology presented in [7] in case the implementation is done through Java or using the one presented in [8] in case of using Constraint Handling Rules (CHR). The idea is to associate every method call/constraint with a graphical object/action. This thus results in a visual animation of the algorithm execution. For example, a Java program implemented to perform depth first tree traversal includes the methods :
   a) `addNode(String text)` to add a node to the graph with the given `text`.
   b) `addEdge(String node1, String node2)` to add an edge from the node named `node1` to the node named `node2`.
   c) `traverse(String n)` to traverse the node named `n`.

   A snippet of the program is given below:

```
public void constructTree() {
        MyNode node1 = addNode("1");
        nodes.add(node1);
        MyNode node2 = addNode("2");
        nodes.add(node2);
        MyNode node3 = addNode("3");
        nodes.add(node3);
        MyNode node4 = addNode("4");
        nodes.add(node4);
        MyNode node5 = addNode("5");
        nodes.add(node5);
        MyNode node6 = addNode("6");
        nodes.add(node6);
        MyNode node7 = addNode("7");
        nodes.add(node7);
        addEdge("1", "2");
        addEdge("1", "3");
        addEdge("3", "4");
        addEdge("3", "5");
        addEdge("2", "6");
        addEdge("6", "7");
        myRecurive(node1);}

public void myRecurive(MyNode n) {
        if (n == (null))
                return;
        taverse(n.data);
        myRecurive(n.left);
        myRecurive(n.right);}
```

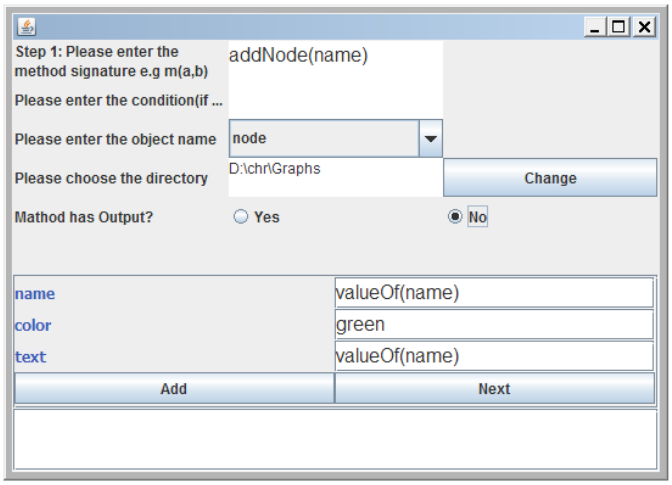


Fig. 2. Annotating a Java method

In this case, annotation could be done as follows [1]:

1) Each `addNode(String name)` method is annotated with a graphical `node` whose `id` is set to `valueOf(name)`. The color of the node could be set to any color e.g. `green`. The content is set to `valueOf(name)`. Thus every time the method `addNode` is called, a new `node` is added to the graph.
2) Similarly each `addEdge(String node1, String node2)` is annotated with an `edge` command. The source is `valueOf(node1)`. The target is `valueOf(node2)`.
3) On the other hand, `traverse(String node1)` is annotated with the command `updateNode`. The source is `valueOf(node1)`. The parameter to be changed is `color`. The new value is `pink`.

The resulting annotation rules are thus:[2]

```
addNode(name)==>node#true#id=valueOf(name)#text=
    valueOf(name)#color=green
addEdge(node1,node2)==>edge#true#src=valueOf(node1)#
    target=valueOf(node2)
traverse(node1)==>updateNode#true#src=valueOf(node1)
    #parameter=color#value=pink
```

The engine implemented through [7], [8] is able to use the above annotation rules to automatically produce the corresponding animation file. An example for annotating the Java method `addNode` is shown if Figure 2. Users could thus use the provided graphical user interface to produce the required annotation rules. In this example, whenever the method `addNode("1")` is called, the following line is added to the animation file: `node 1 green 1`.

The result of the animation of the previously shown code is shown in Figure 1. The user can thus visually see how depth-first tree traversal works. Every time a node is traversed, its color changes to `pink`. Another example of using the engine is shown in the appendix.

III. Features

The tool provides its users with different features to be able to animate algorithms.

1) **Drawing a New Node**
   In order to draw a new node the user has to specify its content and color. The user should give each node a name/id. By default, each node has the value `0` for the x and the y coordinates since the actual values should be computed by the placement algorithm once it is linked to any other node. However, the user can also specify an initial x and y coordinates values.
2) **Defining an Edge**
   An edge links two nodes together. An edge can also have a text label. To define an edge, the user has to define the source and target nodes. The node is identified by its name/id.
3) **Changing a Property of a Node**
   To provide a library that could be used for animating algorithms, it was important to provide the possibly to change the appearance of a node. Users are thus able to change the color and the text inside an existing node.
4) **Step-By-Step Viewing**
   The aim of the interface is not only to be able to view trees and graphs. However, it is also to be able to visualize the changes happening to them while executing an algorithm. That is why it is important to have the option to view the visualization in a step-by-step manner showing one change at each step.
5) **Customizing the Layout**
   The tool is built using the open-source jgraph library [4]. JGraph provides different layouts. Users are thus able to choose the layout they would like to use. The available options include:
   a) `mxHierarchicalLayout`: suitable for trees (default layout).
   b) `mxHierarchicalLayout` with a `left` orientation.
   c) `mxCircleLayout` positioning nodes in a circle like structure.
   d) `mxCompactTreeLayout`. It implements the Moen placement algorithm [6]. It is suitable for trees/graphs with no cycles [3].

IV. Implementation

To have a generic platform as much as possible, animation files (either written manually or produced through annotations) are processed using a CHR program. CHR [9]–[11] was initially introduced for writing constraint solvers. However, it developed into a general purpose language. It consists of different rules that rewrite constraints in the store until a fixed

[1] As introduced in [7], the built_in function $valueOf(Arg)$ returns the value of the argument named Arg.

[2] Since no precondition is needed for applying any of the annotation rules, the keyword true is used.

[3] According to: jgraph.github.io/mxgraph/docs/js-api/files/layout/mxCompactTreeLayout-js.html#mxCompactTreeLayout.

point is reached where no more rules are applicable. There are two types of constraints: built-in constraints handled by the host language and user-defined constraints provided by the user. A CHR rule has a head and a body and an optional guard. A rule is applicable if the constraint store contains constraints matching the head constraints. The guard of the rule, which acts as pre-condition, has to be satisfied for the rule to be applicable. The constraints store is initialized by the constraints in the query of the user. For space reasons, two types of CHR rules are introduced: simplification and propagation rules. A simplification rule has the format:
$name \ @ \ Head \ \Leftrightarrow \ Guard \mid Body.$
When a simplification rule is fired, the constraints matching the $Head$ constraints are removed from the store. The $Body$ constraints are added to the constraint store. A propagation rule, on the other hand, has the format:
$name \ @ \ Head \ \Rightarrow \ Guard \mid Body.$
When a propagation rule is fired, the constraints matching the $Head$ constraints are kept in the constraint store. The $Body$ constraints are also added to the constraint store.

The implemented animation library processes the produced animation files. Every time a command is read, it adds the corresponding CHR constraint to fire the matching rule. The aim of using the library through CHR is to make use of its declarativity. The processing is thus done in a generic way independent of the actual used tool for drawing the graph (jgraph in this case). Thus in case a different library should be used, the back end of the application remains unchanged. The developer needs to only provide mappings to the calls $(draw\_node, \ldots)$ that change the graphical layout accordingly. The below code shows an abstraction of the code used to annotate constraints with drawing events.

```
layout(LayoutName, ExtraSpecs)==> call(set_layout(
    LayoutName, ExtraSpecs)).
object(node, Name, Color, Text,X,Y) ==> call(
    draw_node(Node, Name, Color, Text)).
object(edge, Node1, Node2, Label) ==> call(draw_edge
    (Node1, Node2, Label)).
change_parameter(Name, Type, NewValue) <=> call(
    update_object(Name, Type, NewValue)).
```

For example, whenever the animation line `node 1 green first_node`, is read, the constraint `object(node, 1, green, first_node,0,0)` is called. The constraint triggers `draw_node` which adds the required graphical node.

V. Conclusions & Future Work

In the paper we presented a new engine for animating graph and tree algorithms in an easy-to-use way. The engine uses an animation file to provide the properties of the objects to be drawn. Using annotation rules, such animation files could be produced automatically. In the future, more graphical properties could be provided to the output trees and graphs such as zooming in and out. Minimizing parts of trees could be also addressed.

References

[1] C. Hundhausen, S. Douglas, and J. Stasko, "A Meta-Study of Algorithm Visualization Effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002.

[2] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia, "Animating geometric algorithms over the web," in *Proceedings of the Twelfth Annual Symposium on Computational Geometry, Philadelphia, PA, USA, May 24-26, 1996*, S. Whitesides, Ed. ACM, 1996, pp. C–3–C–4. [Online]. Available: http://doi.acm.org/10.1145/237218

[3] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, *Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 127–148.

[4] G. Alder, *Design and Implementation of the JGraph Swing Component*, 1st ed., Feb. 2003, available at: http://jgraph.sourceforge.net/doc/paper/.

[5] W. C. Pierson and S. H. Rodger, "Web-based animation of data structures using JAWAA," in *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, 1998, Atlanta, Georgia, USA, February 26 - March 1, 1998*, J. Lewis, J. Prey, D. Joyce, and J. Impagliazzo, Eds. ACM, 1998, pp. 267–271.

[6] S. Moen, "Drawing dynamic trees," *IEEE Software*, vol. 7, no. 4, pp. 21–28, 1990.

[7] N. Sharaf, S. Abdennadher, and T. W. Frühwirth, "A rule-based approach for animating java algorithms," in *20th International Conference Information Visualisation, IV 2016, Lisbon, Portugal, July 19-22, 2016*, E. Banissi, M. W. M. Bannatyne, F. Bouali, R. Burkhard, J. Counsell, U. Cvek, M. J. Eppler, G. G. Grinstein, W. Huang, S. Kernbach, C. Lin, F. Lin, F. T. Marchese, C. M. Pun, M. Sarfraz, M. Trutschl, A. Ursyn, G. Venturini, T. G. Wyeld, and J. J. Zhang, Eds. IEEE Computer Society, 2016, pp. 141–145. [Online]. Available: http://dx.doi.org/10.1109/IV.2016.55

[8] ——, "CHRAnimation: An Animation Tool for Constraint Handling Rules," in *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014.*, ser. Lecture Notes in Computer Science, vol. 8981. Springer, 2014, pp. 92–110.

[9] T. W. Frühwirth, "Theory and practice of constraint handling rules," *J. Log. Program.*, vol. 37, no. 1-3, pp. 95–138, 1998. [Online]. Available: http://dx.doi.org/10.1016/S0743-1066(98)10005-5

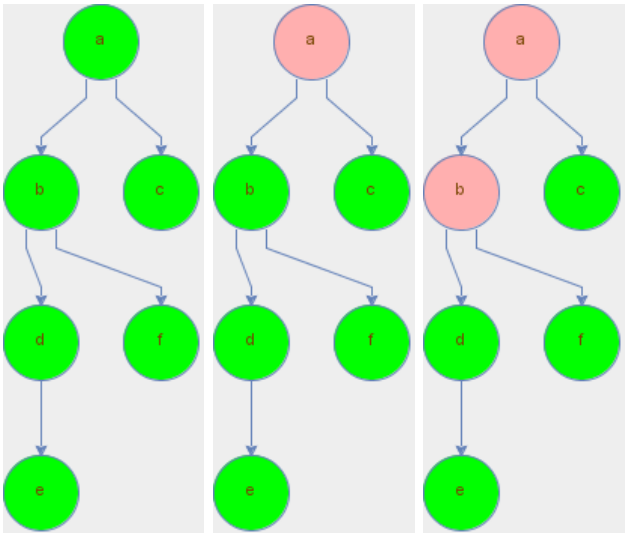
[10] ——, *Constraint Handling Rules*. Cambridge University Press, 2009.

[11] ——, "Constraint handling rules - what else?" in *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings*, ser. Lecture Notes in Computer Science, N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, and D. Roman, Eds., vol. 9202. Springer, 2015, pp. 13–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21542-6_2

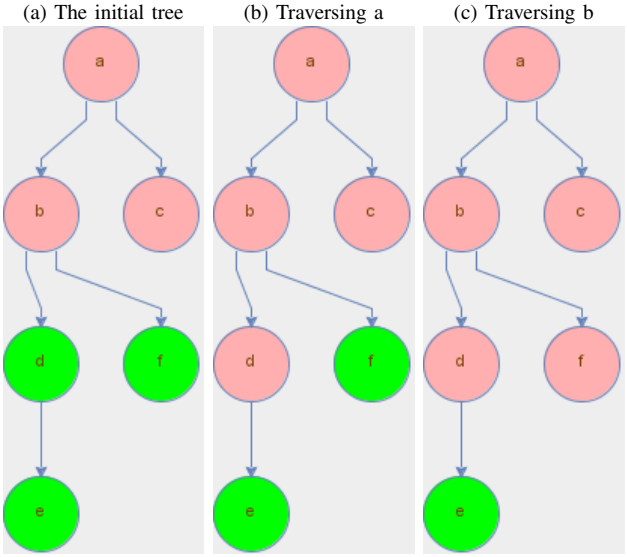
APPENDIX

Figure 3 shows the result of animating a breadth-first tree traversal algorithm. In this case, the animation file contains the following lines (The animation file could be manually written by the programmer/produced through annotation rules of the algorithm).

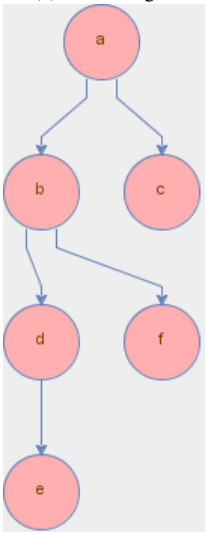
```
node 1 green a
node 2 green b
node 3 green c
node 4 green d
node 5 green e
node 6 green f
edge 1 2
edge 1 3
edge 2 4
edge 2 6
edge 4 5
updateNode 1 color pink
updateNode 2 color pink
updateNode 3 color pink
updateNode 4 color pink
updateNode 6 color pink
updateNode 5 color pink
```



(a) The initial tree (b) Traversing a (c) Traversing b



(d) Traversing c (e) Traversing d (f) Traversing f



(g) Traversing e

Fig. 3. Visualizing Breath-First Traversal of a Tree