Laying out Diagrams Using Graph Analysis and Drawing Library - GRAD

Renata Vaderna, Gordana Milosavljević, Igor Dejanović

Faculty of Technical Sciences, University of Novi Sad
vrenata, grist, igord@uns.ac.rs

Abstract

Graph drawing is an area of computer science motivated by the relatively frequent need to visualize information as graphs. The importance of this field is evidenced by the fact that there is a large number of graph drawing algorithms in existence, with new ones still being developed. This paper gives an overview of the most important classes of these algorithms and showcases the most popular open-source Java libraries that offer a decent number of their implementations. All of these libraries strongly focus on visualization, making their integration with separately developed graphical editors overly complex. Furthermore, none of them provide a way of helping a user with little knowledge of graph drawing theory pick the right algorithm. In order to address the mentioned issues and provide additional graph analysis and drawing algorithms, we are developing another library called GRAD.

1 Introduction

Graph drawing is an area of mathematics and computer science concerned with the geometric representation of graphs and networks. It is motivated by those applications where it is crucial to visualize structural information as graphs [1]. Formally, a drawing of a graph maps each of its vertices to a distinct point of the plane and each of its edges to a simple Jordan curve connecting two vertices. The process of creating a drawing of a graph from the underlying structure is known as automatic graph layout. Every graph can be graph in a very large number of ways, but the resulting drawings are not always understandable and nice to look at. That is why there is a great number of graph layout algorithms which provide ways of creating aesthetically pleasing drawings of graphs. However, implementing even the simplest algorithms whose results are of satisfying quality is not an easy task and many developers in need for such features have to rely on existing solutions.

There are many libraries offering these features for various programming languages, including C/C++, Python and JavaScript, but this paper will focus the Java programming language. The most popular open-source Java graph drawing libraries include JGraphX [2], JUNG framework [3], and Prefuse [4].

In addition to implementing a number of layout algorithms, all of these libraries heavily focus on providing visualization components which enable interaction with edge-node graphs. Those looking to quickly visualize certain information would undoubtedly find these components very useful. On the other hand, using an algorithm to lay out diagrams of separately developed graphical editors is also a relatively common need. However, the mentioned libraries strongly couple layout algorithms with visualization components and simply calling a layout algorithm and retrieving its results is often overly complex.

Furthermore, the libraries provide a decent number of graph drawing algorithms of varying complexity, but certain classes of these algorithms are not present at all. These include planar straight-line, symmetric and more advance circular to name a few. In order to address the firstly mentioned issue of the complexity of separating an algorithm from the visualization and to provide a greater number of layout algorithms, we are developing a new graph drawing and analysis library called GRAD [5]. It offers implementations of certain algorithms belonging to the mentioned classes, not supported by other libraries, as well as a large number of graph analysis algorithms and a way of automatically selecting the best algorithm based on the graph's properties. Moreover, it also includes a way of letting the users descriptively specify how a drawing should look and automatically choosing the best algorithm or their combination.

The rest of the paper is structured as follows. Section 2 gives a short overview of different classes of layout algorithms. Section 3 describes the popular Java libraries in more detail. Section 4 presents GRAD, with the emphasis being put on its layout capabilities. Section 5 concludes the paper and outlines future work.

2 Overview of Graph Drawing Algorithms

There is a great number of graph layout algorithms, with plenty of researchers working on discovering new and enhancing existing ones. The quality of an algorithm is determined based on its computational efficiency as well as various aesthetic criteria. These algorithms can be grouped into several classes, which will be briefly described in the following paragraphs.

Tree drawing is one of the best studied areas of graph drawing with a great number of practical applications. There are various approaches to drawing trees, with the most frequently used ones being level-based, horizontalvertical (H-V) and ringed-circular approaches. A detailed overview and comparison of different tree drawing algorithms can be found in [6]. *Hierarchical drawing algorithms* are often used to draw directed graphs (or digraphs) which present hierarchies and are to complex to be drawn using the tree algorithms.

A *circular drawing* of a graph is its visualization which partitions it into clusters whose nodes are placed onto the circumference of an embedding circle. Also, each edge is drawn as a straight line. Circular drawings can be combined with techniques which set the order of nodes in order to minimize the number of edge crossings.

Symmetric graph drawing algorithms aim to draw a graph with a nontrivial symmetry, or, more ambitiously, with as much symmetry as possible.

Planar straight-line drawing algorithms focus on creating drawing of graphs without any edge crossings (planar) where all edges are drawn using only straight-line segments. Not all graph have a planar drawing however.

Force-directed algorithms are among the most important and flexible graph drawing algorithms. They can be used to calculate layouts of all simple undirected graphs and only need the information contained within the structure of the graph itself. There are many force-driven algorithms, with Tutte's 1963 barycentric method [7] being considered to be the first one. The most popular ones include the spring layout method of Eades [8], Kamada - Kawai [9] and Fruchterman - Reingold [10] methods.

3 Related Work

There are quite a few libraries for graph analysis and visualization for Java, the most popular of which will be presented in this section. The focus will be put on the layout algorithms they provide and the complexity of using them within separately developed graphical editors. Generally, the libraries that will be mentioned put emphasis on visualization and strongly couple layout algorithms with it, thus making the secondly mentioned task overly complex. It should be noted that tools which generate static drawings of graphs in a variety of output formats will not be taken into consideration since they are not suitable for this particular purpose. Furthermore, commercial solutions will not be discussed since our focus is on open-source ones.

JUNG — the Java Universal Network/Graph Framework is an open-source software library that provides a common and extendable language for modeling, analysis, and visualization of data that can be represented as a graph or network.

Among others, JUNG framework offers several graph layout algorithms, with some of them being quite complex. Summarily, there are three tree drawing algorithms, a number of force-directed, and a simple implementation of a circular layout algorithm. The tree layout algorithms include the following: an implementation of a level-based approach, radial tree method, and the balloon method. The balloon method positions vertices using associations with nested circles or "balloons". The force-directed algorithms are the popular and flexible spring method, Kamada-Kawai and Fruchterman-Reingold, as well as an algorithm based on Bernd Meyer's self - organizing graph methods [11], referred to as ISOM layout.

However, JUNG framework's layout algorithms are mainly meant to be used with its visualization tools. For example, in order to trigger the execution of any of these algorithms, it is necessary to instantiate the visualization model. Furthermore, setting the sizes of the vertices before calling a layout algorithm requires extensive knowledge of the framework.

JGraphX is a Java Swing library which provides visualization and interaction with node-edge graphs, as well as a decent number of algorithms for graph analysis. JGraphX offers various usable implementations of graph drawing algorithms: a tree, hierarchical, and two algorithms based on the force-directed layout paradigm. The tree layout in question is compact tree layout, which improves the standard level-based approaches by trying to make the resulting drawing as compact as possible.

In order to use one of JGraphX's algorithms it is necessary to create an instance of its graph class. Same can be said about JUNG framework, however, unlike the previously described library, JGraphX graphs are not parameterized. This means that that existing sets of objects representing vertices and edges of a diagram need to be transformed into JGraphX's. More importantly, one must be quite familiar with how JGraphX works in order to get positions of the vertices once layout algorithms have finished calculating them. More detailed description of the difficulties of using JUNG framework's and JGraphX's algorithms can be found in [12].

Prefuse is a software framework for creating dynamic visualization of both structured and unstructured data, that provides theoretically-motivated abstractions for the design of a wide range of visualization application. Prefuse is bundled with a library which, among other actions, provides a host of layout and distortion techniques. Available layout algorithms include random, grid-based, circular, a highly configurable forced-directed, and several tree ones.

Although Prefuse is an excellent visualization tool, integrating its layout algorithms into already existing editors is quite challenging. The difficulty lays in the fact that Prefuse uses schedulers, precisely defining when certain actions should be performed. So, determining the point in time when it is possible to retrieve the calculated positions of graph elements can be problematic.

4 Graph Analysis and Drawing Library - GRAD

The following section will present GRAD, our graph analysis and drawing library for the Java programming language. GRAD is currently being used in our open-source Kroki mockup tool [13] for laying out imported class diagrams created by other modeling tools. These diagrams can contain over 600 classes. More details can be found in [12], where the emphasis is put on this integration.

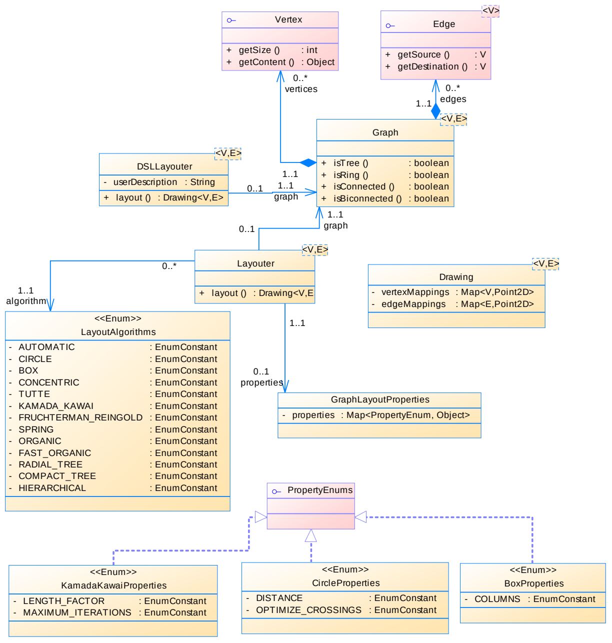
The main motivation for GRAD's development is to make the process of laying out diagrams of any graphical editor as simple as possible, find and port the best layout algorithms offered by other open-source libraries and provide drawing and analysis algorithms that none of them implement. So, GRAD strives to make calling any available layout algorithm and retrieving its results a very easy task. Furthermore, it offers a way of automatically selecting the most suitable layout algorithm or algorithms based on the properties of a graph. For example, if the graph in question is a tree, one of the tree drawing algorithm is selected; if it is a ring, a circular layout is used etc. This can be of help to the users with little or no knowledge of graph drawing theory and algorithms. On top of that, GRAD offers the users the possibility of descriptively specifying how the drawing of their graph should look. Based on that description, the algorithms whose results are closest to the user's wishes is picked. GRAD is not intended to be used as a visualization tool, but it also provides a simple graphical editor which can be used for familiarization with different algorithms.

4.1 Implementation of GRAD's Layout Features

The following paragraphs will present the core concepts of GRAD's implementation of layout functionalities. The focus will not be on how certain algorithms were implemented, but on what is involved in the process of calling them and retrieving the results. A class diagram containing the key elements of this feature is shown in fig. 1.

Firstly, it can be noticed that the class representing a graph is parameterized. So, it is safe to use any two classes as types of vertices and edges, as long as they implement GRAD's *Vertex* and *Edge* interfaces. This makes it possible to easily specify properties of the vertices and edges (e.g. sizes of the vertices). The central class that handles a layout request is *Layouter*. It accepts lists of vertices and edges, an enumerated value corresponding to the algorithm of choice and, optionally, values of configurable parameters of the algorithm. An instance of GRAD's *Graph* class will be automatically created later using the two provided lists. Similarly, the appropriated layout algorithm will be instantiated based on the secondly mentioned parameter. If the value "AUTOMATIC" is chosen, GRAD picks the best algorithm by analyzing the graph's properties. With the help of the class *GraphLayoutPropeties*, certain parameters of the chosen algorithm can be specified. This class maps enumerated values naming properties of the algorithms to their desired values. In order to avoid the escalation of the diagram's size, only a few of the properties enumerations were shown. This method of implementing the specification of properties was chosen in order to achieve as much genericness as possible. Finally, an object containing mappings of vertices and edges to their positions, *Drawing*, is returned. Therefore, all information needed to position the diagram's elements is obtained with no additional effort. It should also be mentioned that once the algorithm of choice has finished executing, parallel and multiple edges are detected and positioned before the result is returned.

In addition to either directly selecting an algorithm or stating that the graph should be laid out automatically, the users can also provide a description of how the graph should look and GRAD selects the layout algorithm based on it. This feature was implemented by developing an external domain-specific language (DSL) [14] and its interpreter. In order to use it, a string conforming to the DSL, as well as the lists of vertices and edges of the graph should be passed to an instance of the *DSLLayouter* class.



Vertex
+ getSize () : int
+ getContent () : Object
Edge
<V>
+ getSource () : V
+ getDestination () : V
0..*
vertices
0..*
edges
1..1
<V,E>
Graph
+ isTree () : boolean
+ isRing () : boolean
+ isConnected () : boolean
+ isBiconnected () : boolean
<V,E>
DSLLayouter
- userDescription : String
+ layout () : Drawing<V,E>
0..1
1..1
1..1
graph
1..1
graph
0..1
<V,E>
Layouter
+ layout () : Drawing<V,E
0..*
1..1
algorithm
<V,E>
Drawing
- vertexMappings : Map<V,Point2D>
- edgeMappings : Map<E,Point2D>
<<Enum>>
LayoutAlgorithms
- AUTOMATIC : EnumConstant
- CIRCLE : EnumConstant
- BOX : EnumConstant
- CONCENTRIC : EnumConstant
- TUTTE : EnumConstant
- KAMADA_KAWAI : EnumConstant
- FRUCHTERMAN_REINGOLD : EnumConstant
- SPRING : EnumConstant
- ORGANIC : EnumConstant
- FAST_ORGANIC : EnumConstant
- RADIAL_TREE : EnumConstant
- COMPACT_TREE : EnumConstant
- HIERARCHICAL : EnumConstant
1..1
0..1
properties
GraphLayoutProperties
- properties : Map<PropertyEnum, Object>
PropertyEnums
<<Enum>>
KamadaKawaiProperties
- LENGTH_FACTOR : EnumConstant
- MAXIMUM_ITERATIONS : EnumConstant
<<Enum>>
CircleProperties
- DISTANCE : EnumConstant
- OPTIMIZE_CROSSINGS : EnumConstant
<<Enum>>
BoxProperties
- COLUMNS : EnumConstant

Figure 1: The core of GRAD's layout model

4.2 GRAD's Layout Algorithms and Their Computational Efficiency

GRAD ports the best algorithms from the JUNG framework, JGraphX and Prefuse, and adds a number of new implementations. The current version of GRAD includes several tree and force-directed drawing algorithms, a hierarchical, Tutte's straight-line, a circular which minimizes the number of edge crossings, a symmetric based on the work of Carr and Kocay [15], and a so-called box layout, which places elements in a table-like structure. Apart from the algorithms belonging to the first three classes (tree, force-directed and hierarchical), the other ones are GRAD's original implementations. A more detailed description of the algorithms and aesthetic criteria they focus on, as well as several examples of laid out graphs can be found in [16]. Graph layout algorithm are rated based on their computational efficiency (amount of computational resources used by the algorithm) and the aesthetics of the resulting drawing. In this section, the emphasis will be put on the first criterion.

Table 1 shows how well the supported algorithms perform when laying out bigger graphs. The first test was performed on a randomly generated graph with 1000 vertices and twice as many edges, and the second one on a tree with 1000 vertices. Naturally, the tree drawing algorithms can only be performed on trees. If there is a large number of algorithms from the same class (tree and force-directed), a few were chosen as representatives. Algorithms only meant to be performed on smaller graphs, like Tutte embedding, were omitted from the tests. Tests were performed on a computer with 8GB RAM and Intel i5 2.50 GHz processor.

Table 1: Time in milliseconds needed to lay out graphs

Algorithm	Graph 1	Tree
Spring	1026	703
Fruchterman-Reingold	878	614
Kamada-Kawai	5283	4139
ISOM	479	403
Circular optimizing crossings	73099	71943
Circular not optimizing crossings	419	321
Symmetric	523	467
Hierarchical	52505	1059
Box	114	81
Compact tree	x	643
Radial tree	x	115
Balloon tree	x	1652

In [17] some highly effective force-directed algorithms are presented and their performance is measured. It is stated that a graph of the similar size to the one used here can be laid out in under 1 second. Some of the algorithms offered by GRAD live up to that standard, while also producing understandable drawings. The ISOM algorithms is the best example. However, it should be noted that the slower algorithms put more emphasis on aesthetics. That is why GRAD offers a variety of different algorithms to choose from.

5 Conclusion

This paper gave an overview of different classes of graph layout algorithms and presented some of the most popular Java libraries focusing on graph drawing and visualization. Due to the libraries strongly coupling their layout algorithms with visualization, using the provided algorithms within a separately developed graphical editor is often too complex. Furthermore, none of them have any features that would make the process of selecting a layout algorithm an easy task even for those with little or no knowledge of graph drawing theory. To address the mentioned issues and provide additional graph layout and analysis algorithms, we are developing a new graph drawing and analysis library called GRAD.

GRAD ports the best algorithm provided by other Java open-source libraries and adds its own set of layout algorithms. These include a symmetric and a straight-line algorithm, as well as an enhanced version of a circular one. Additionally, GRAD offers a way of automatically choosing the best algorithm based on the properties of the graph and of selecting the algorithm based on the user's description of the desired characteristics of the drawing. Plans for future improvements of GRAD include the implementation of additional drawing algorithms, including a better symmetric and one or more orthogonal ones as well as the enhancement of the current ones to make them more efficient.

References

[1] "International symposium on graph drawing and network visualization," http://graphdrawing.org/index.html.

[2] "Jgraphx," https://github.com/jgraph/jgraphx.

[3] "Jung framework," http://jung.sourceforge.net.

[4] "Prefuse," http://prefuse.org.

[5] "Graph analysis and drawing library," https://github.com/renatav/GraphDrawing.

[6] A. Rusu, *Handbook of Graph Drawing and Visualization*. Chapman and Hall/CRC, 2007, ch. 5, pp. 155–192.

[7] W. Tutte, "How to draw a graph," in *Proceedings of the London Mathematical Society 13*, 1963, p. 743–767.

[8] P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, p. 149–160, 1984.

[9] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information Processing Letters*, vol. 31, pp. 7–15, April 1989.

[10] T. Fruchterman and E. Reingold, "Graph drawing by force-directed placement," *Software Practice and Experience*, vol. 21, p. 1129 – 1164, November 1991.

[11] B. Meyer, "Self-organizing graphs - a neural network perspective of graph layout," in *In Neural Computers, 393–406, ECKMILLER*. Springer, 1998, pp. 246–262.

[12] R. Vaderna, G. Milosavljvić, and I. Dejanović, "Graph layout algorithms and libraries: Overview and improvement," in *ICIST 2015 Proceedings*, 8-11 March 2015.

[13] "Kroki mockup tool," http://www.kroki-mde.net.

[14] T. Kosar, S. Bohra, and M. Mernik, "Domain-specific languages: A systematic mapping study," *Information and Software Technology*, vol. 71, pp. 77–91, March 2016.

[15] H. Carr and W. Kocay, "An algorithm for drawing a graph symmetrically," *Bulleting of the Institute of Combinatorics and its Applications*, vol. 27, pp. 19–25, 1997.

[16] R. Vaderna, I. Dejanović, and G. Milosavljević, "Grad: A new graph drawing and analysis library," in *MDASD 2016 Proceedings*, in press.

[17] Y. Hu, "Efficient and high quality force-directed graph drawing," *The Mathematica Journa*, vol. 10, pp. 37–71, 2005.