Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1597–1602
DOI: 10.15439/2016F299
ACSIS, Vol. 8. ISSN 2300-5963

GRAD: A New Graph Drawing and Analysis Library

Renata Vaderna, Igor Dejanović, Gordana Milosavljević
Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića, Novi Sad, Serbia
Email: vrenata, igord, grist@uns.ac.rs

***Abstract*—Several important choices need to be made during the development of domain-specific languages, including the one regarding which concrete syntax to implement. There are several alternatives, with graphical and textual syntaxes being the most common ones. Having in mind that the developers and domain experts often have different preferences, supporting both is sometimes the best option. This means that models created using textual editors might need to be opened using separately developed graphical editors. Graphical elements corresponding to model elements must then be automatically created and positioned. Doing so in an aesthetically pleasing way requires usage of graph layout algorithms. Since implementing them is not an easy task, most developers have to rely on existing solutions. There are many Java libraries which have such capabilities, but they all have certain limitations and room for improvement, some of which are addressed in a new graph drawing and analysis library presented in this paper.**

I. Introduction

DOMAIN-SPECIFIC languages (DSLs) are computer languages specialized to a particular domain [1]. Development of such languages also includes the decision of how to present the concrete syntax to the users, who can be both developers and non-technical domain experts. There are several alternatives, with the most popular being the textual and graphical ones.

Each of these choices has its own advantages and disadvantages, so supporting both is the best solution at times. The textual concrete syntaxes can express any formal language, help in understanding all technical details of a DSL and are often preferred by the developers. On the other hand, end-users, who work in a non-technical domain, don't find them particularly appealing. These users generally prefer the graphical concrete syntax, which makes it possible to design DSL models using a completely functional graphical editor [2]. Graphical concrete syntaxes, if designed correctly, are intuitive and easy to understand. Having all of this in mind, it can be concluded that if a DSL needs to appeal to both developers and non-technical end-users, both textual and graphical concrete syntaxes should ideally be implemented. A similar conclusion was reached in [3], where the authors discussed the positive and negative experiences of using both the textual syntax, described in [4] and a graphical one to define the static structure of database applications.

Implementing both syntaxes leads to one problem: what happens if a part of the model is specified using the textual syntax and needs to be viewed and/or edited using the graphical editor? Graphical elements corresponding to previously described concepts need to be created automatically. These elements have additional visual properties, including positions which need to be calculated. This can be accomplished by applying a layout algorithm.

Implementing even the simplest of layout algorithms that would guarantee a somewhat pleasing arrangement of elements requires excessive knowledge of graph theory and can be rather time consuming. This is why the developers often rely on existing solutions. This paper focuses on libraries for the Java programming languages, but there are many similar ones for other languages like C/C++ and Python. The most popular open-source libraries offering the possibility of laying out elements of a diagram for Java projects include JUNG framework, JGrapX and Prefuse. All of these solutions put emphasis on visualization, providing their own visual components and thus strongly coupling layout capabilities with them. This makes the integration with separately developed graphical editors overly complex [5]. Furthermore, they only support a small number of different classes of layout algorithms, despite offering several algorithms belonging to the same classes. Even though the available algorithms can be used to lay out any diagram with acceptable results, it can be noticed that there is room for improvement. Certain classes of layout algorithms were designed with the goal of getting excellent results when applied to diagrams satisfying some special conditions (e.g. planar, straight-line, symmetric, rectangular). The mentioned libraries implement very few of them. Also, they don't offer a way of automatically choosing an appropriate algorithm based on properties of the diagram or on the wishes of the users regarding diagram aesthetics.

In order to address the mentioned issues, we are developing another graph drawing and analysis Java library, called GRAD (GRaph Analysis and Drawing) [6]. GRAD's main goals are to:

- offer a large number of different graph drawing algorithms, including some that haven't been implemented in Java yet
- provide a very quick and easy way to lay out elements of any existing graphical editor
- offer algorithms for graph analysis, which can later be used to automatically select a suitable layout algorithm
- enable the users to specify aesthetic criteria and automatically choose an appropriate layout algorithm based on their wishes

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.

The rest of the paper is structured as follows. Section 2 gives an overview of basic graph theory concepts, graph drawing aesthetic criteria, and different classes of graph drawing algorithms. Section 3 showcases some popular Java graph drawing and analysis libraries. Section 4 presents GRAD. Finally, section 5 concludes the paper and outlines future work.

II. Graph Drawing Aesthetics and an Overview of Graph Layout Algorithms

In the following section a short overview of the graph aesthetic criteria and the most popular classes of graph layout algorithms will be given. Firstly, the most important concepts which will be referenced later will be defined.

A. Basic graph drawing theory definitions

A graph (V, E) is an ordered pair consisting of a finite set V of vertices and a finite set E of edges, that is, pairs (u, v) of vertices [7]. If each edge is an unordered (ordered) pair of vertices, the graph is undirected (directed). A graph is simple if it doesn't contain any edges that join a vertex to itself or more than one edge connecting the same two vertices (multiple edges). A graph is said to be connected if there is a path from any vertex to any other vertex in the graph. A biconnected graph is a connected graph which has no vertices whose removal would disconnected it. Graphs which contain at least one cycle are called cyclic graphs, while the ones that do not are known as acyclic. A tree is a connected acyclic graph. Finally, a graph is planar if it can be drawn in a plane without graph edges crossing. A planar drawing partitions the plane into connected regions called faces.

The process of creating a drawing of a graph from the underlying structure is known as automatic graph layout. There is a great number of graph layout algorithms, with plenty of researchers still 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. The following sections will give an overview of the mentioned criteria and the most popular layout methods.

B. Aesthetic Criteria

Many different quality measures or aesthetic criteria have been defined for graph drawings. Authors often optimize certain aesthetics claiming that the resulting drawing is therefore more understandable and more visually pleasing to a human observer. The most common criteria includes the following: Minimization of the number of edge crosses, maximization of the minimal angle between edges extending from a node, minimization of the total number of bends in polyline edges, even distribution of edges within a bounding box, appropriate lengths of edges, neither too short nor too long, similar length of edges, same flow of edges in directed graphs (as much as possible),orthogonality, and symmetry [8].

Some layout methods put emphasis on one of the measures trying to produce a drawing which, for example, has no edge crosses (planar drawing) or is maximally symmetric, while the other ones attempt to optimize as many as possible. The desired aesthetic criterion or criteria can be the deciding factor in choosing an appropriate layout algorithm. However, certain layout algorithms which insist on a particular aesthetic criterion might not be applicable to all graphs. Obviously, it is not possible to produce a drawing with no edge intersections of a graph which is not planar. Therefore, properties of the graph which is to be laid out can also be an important indicator of the best choice of the algorithm.

C. An Overview of Graph Layout Algorithms

There is a very large number of different classes of graph layout algorithms and the following paragraphs will present the most popular ones.

Tree drawing is one of the best studied areas of graph drawing. That is not surprising since automatic generation of drawings of trees finds many practical applications. Namely, a tree whose vertices represent entities and whose edges represent relationships is a typical data structure for modeling hierarchical information. There are various approaches to drawing trees and their detailed overview and comparison can be found in [9].

A *circular drawing* of a graph is its visualization where it is partitioned into clusters whose nodes are placed onto the circumference of an embedding circle. Each edge is drawn as a straight line. Simply placing nodes on a circumference of a circle might result in a drawing which is not particularly aesthetically pleasing due to a very large number of edge crossings, which is why there are techniques which also minimize this number when determining positions of the nodes [10].

Symmetric graph drawing algorithms aim to draw a graph with nontrivial symmetry, or, more ambitiously, with as much symmetry as possible. Some consider symmetry as one of the most important aesthetic criteria which clearly reveals the structure and properties of a graph. For example, graphs in textbooks on graph theory are normally drawn symmetrically and a symmetric drawing is in some cases preferred over a planar one.

Planar straight-line drawing algorithms rely on the fact that if a graph can be drawn with no crossings using edges of an arbitrary shape, then it can be drawn in the same way using only straight-line segments. Convex drawings are planar straight-line drawings where all faces are drawn as convex polygons. In [11] the author claims that the convex drawings of planar graphs make it possible for readers to easily and rapidly recognize structures of the graphs, such as adjacency of vertices.

Planar orthogonal and polyline drawing algorithms focus on angular resolution as the most important aesthetic criterion. Orthogonal drawings only use horizontal and vertical line segments for edges and are, therefore, often quite visually pleasing. A more specific type of orthogonal drawings are rectangular drawings, which also make sure that each face is drawn as a rectangle. However, they have a pretty serious limitation of only being applicable to graphs which don't contain a vertex whose degree is higher than four. Polyline drawing are more general and don't have the mentioned disadvantage. They usually focus directly on sizes of the angles (which should not be smaller than some fixed threshold) and not on the type of edges.

Force-directed algorithms are among the most important and most flexible algorithms. Unlike many previously mentioned ones, which can only be applied if a graph is planar or satisfy some other specific conditions, force-directed algorithms can be used to calculate layouts of all simple undirected graphs. They only need the information contained within the structure of the graph itself.

Graphs drawn with these algorithms tend to be aesthetically pleasing, exhibit symmetries, and tend to produce crossing-free layouts for planar graphs. There are many force-driven algorithms, the most popular of which include the spring layout method of Eades [12], Kamada-Kawai [13] and Fruchterman-Reingold [14] methods.

Hierarchical drawing algorithms can be used when dealing with directed graphs (or digraphs) which represent hierarchies. These algorithms name uniform "flow" of edges as one of their main goals. More precisely, the edges should either go from left to right or top to bottom, depending on a particular application.

III. Related Work

There are quite a few libraries for graph analysis and visualization for Java. The next section will present the most popular ones, primarily focusing on their layout capabilities and mentioning implemented graph analysis algorithms which could be used to determine the best choice of the drawing algorithm.

It is important to mention that visualization 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 considered since our focus is on open-source ones.

A. JUNG Framework

JUNG — the Java Universal Network/Graph Framework [15] is an open-source software library that provides a common and extendible language for modeling, analysis, and visualization of data that can be represented as a graph or network. JUNG framework is licensed under the permissive BSD license.

The current distribution of JUNG includes implementations of a number algorithms from graph theory, data mining, and social network analysis. However, most of them are of little importance to this research. Only Dijktra's shortest path and decomposition of a graph into biconnected components can be singled as potentially useful in the mentioned case.

JUNG framework offers implementations of several layout algorithms, some of which are quite complex. Most importantly, these include three tree layout algorithms and a number of force-directed ones. The tree layout algorithms include the following: an implementation of a level-based approach, radial tree method, and the balloon method. The radial tree method displays a tree structure in a way that expands outwards, radially. The balloon method positions vertices using associations with nested circles or "balloons". The force-directed algorithms are the already mentioned popular and flexible spring method, Kamada-Kawai and Fruchterman-Reingold, as well as an algorithm based on Bernd Meyer's self-organizing graph methods [16]. JUNG framework also contains a relatively basic circle layout drawing algorithm, which simply places vertices on a circumference of a circle of a given radius.

B. JGrapX

JGraphX is a Java Swing graph visualization library which is also licensed under the BSD license. JGraphX provides visualization and interaction with node-edge graphs, as well as a decent number of algorithms for graph analysis, such as graph traversal, forming the minimum spanning tree and Dijkstra's shortest path [17]. The minimum spanning tree is defined as the set of all vertices with minimal lengths that forms no cycles. Graph traversal includes deep-first search and bread-first search, both of which construct spanning trees with certain properties useful in other graph algorithms.

JGraphX provides various usable implementations of graph drawing algorithms. Similarly to JUNG framework, these include a tree and several force-directed layouts, but also a hierarchical one meant to be used if a graph is too complex to be laid out using the tree drawing algorithm. The tree layout in question is the compact tree layout, which improves the standard level-based approaches by trying to make the resulting drawing as compact as possible. Furthermore, JGraphX provides two force-directed layout algorithms: fast organic and organic. The fast organic method is best applied to smaller graphs with a more regular structure, but is supposed to be one of the faster force-directed layouts. The organic layout is one of the most complex algorithms implemented in JGraphX and is based on Davidson and Harel's simulated annealing layout [18].

JGraphX doesn't stop there and offers a very nicely implemented hierarchical layout. This implementation not only positions the vertices, but also routes the edges.

C. Prefuse

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 applications [19]. Like other mentioned libraries, Prefuse is also licensed under the BSD license.

Prefuse is bundled with a library which, among other actions, provides a host of layout and distortion techniques.

Available layout algorithms include random, circular, grid-based, forced-directed, and several tree ones.

The force-directed layout positions graph elements based on a physical simulation of interactive forces acting on bodies. The force simulator used to drive this layout can be set explicitly, allowing custom force-directed layouts to be created.

Tree layouts provided by Prefuse are previously mentioned balloon and radial algorithms, as well as an additional node-link tree layout, which lays out a rooted tree so that each depth level of the tree is on a shared line.

Based on the previous overview, it can be noticed that the mentioned libraries offer many layout algorithms which would be a good addition to any graphical editor in need for such features. However, all of them put heavy emphasis on data visualization and thus strongly couple it with graph drawing algorithms. Therefore, simply calling the desired algorithm and retrieving the results (positions of vertices and edges determined during the execution of the algorithm) requires an understanding of how a library works. GRAD aims to offer a solution to this problem.

Moreover, certain classes of graph drawing algorithms are substantially represented in these libraries (like tree and force-directed ones), while the other classes are barely or not present at all. For example, no symmetric, straight-line or orthogonal algorithms are available. Out library aims to remedy this and offer implementations of certain algorithms, that, according to our knowledge, have not been implemented in Java yet.

Finally, none of these libraries offer a way of automatically choosing an appropriate layout algorithm based on properties of the graph or according to the desired aesthetic criteria specified by the users. Implementing both of these features is among the goals of our solution. It can also be pointed out that none of the libraries provide many graph analysis algorithms which are of great significance to graph drawing.

IV. GRAD (Graph Analysis and Drawing Library)

In the following section different algorithms supported by our graph drawing and analysis library-GRAD will be shown. Additionally, possible ways of choosing appropriate drawing algorithms based on the properties of graphs will be discussed. It can be noted that GRAD can be used both to transform and existing drawing and to form a completely new one when nothing is known about the positions of the graph's vertices. The later is used in our open-source Kroki tool [21] for laying out imported class diagrams created by other modeling tools.

Like it was already mentioned, GRAD also provides a simple graphical editor which can be used to draw graphs we want to experiment with. This editor was used to create all examples of laid out graphs which will be shown in the upcoming sections.

A. *Supported graph drawing algorithms*

The most important objective of GRAD is to provide a large number of different graph drawing algorithms, both those which can only be applied if a graph has certain properties (e.g. is planar) and those that can be applied to all graphs

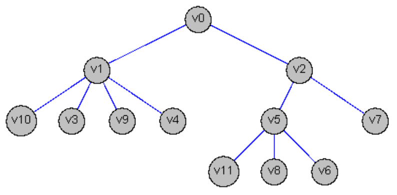

Fig. 1. Resulting drawing of applying the level-based tree drawing algorithm

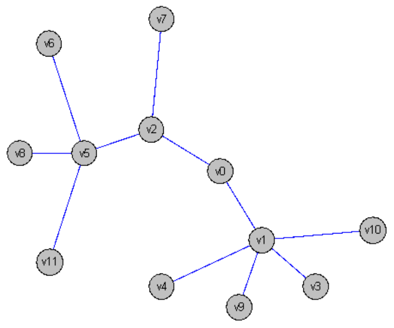

Fig. 2. Resulting drawing of applying the radial tree drawing algorithm

with acceptable results. GRAD ports the best algorithms from the JUNG framework, JGraphX and Prefuse, and adds a number of new implementations of various graph drawing algorithms, not offered by any of the mentioned libraries. Summarily, the current version of GRAD includes several tree and force-directed drawing algorithms, a hierarchical, two straight-line, a circular which minimizes the number of edge crossings, symmetric, and a so-called box layout, which places elements in a table-like structure. The last four algorithms are GRAD's original implementations. The box layout positions a predefined number of vertices in one row, before continuing to the next row. Due to its simplicity, it will not be discussed in more detail.

1) Tree and hierarchical drawing algorithms: Tree drawing algorithms included in GRAD consist of a level-based, radial, balloon and compact tree drawing algorithms, ported from the previously mentioned libraries. The best available implementation of a specific algorithm was selected. Fig. 1 shows the result of applying the level-based tree drawing algorithm to lay out a graph, while fig. 2 show the same graph laid out using the radial algorithm.

If a graph is too complex to be laid out using a tree layout and if it is important to emphasize the overall flow, a hierarchical layout can be used. GRAD ports this layout from JGraphX.

2) Force-directed graph drawing algorithms: Similarly to tree drawing algorithms, the force-directed ones were ported from the three mentioned libraries and the best one they had to offer were selected. They include spring, Fruchterman-Reingold, Kamada-Kawai and organic, and fast organic algorithms ported from JGraphX. Since the final results are relatively similar, only one of them will be shown. Fig. 3 show drawing of a graph laid out using the organic force-directed

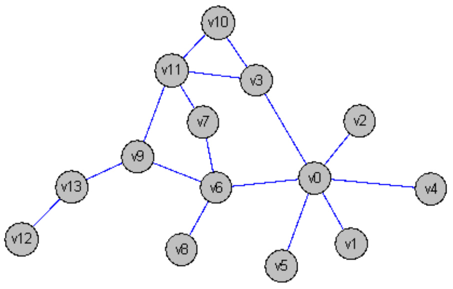

Fig. 3. Resulting drawing of applying organic force-directed drawing algorithm

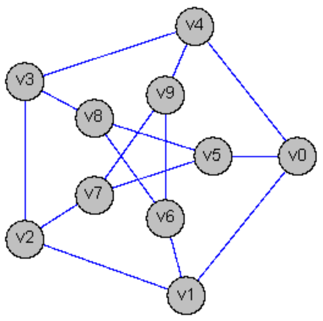

Fig. 4. Resulting drawing of applying the symmetric drawing algorithm

drawing algorithm.

Like it was already mentioned, force-directed algorithms tend to produce satisfactory drawings in most cases. However, the users might be looking for some specific aesthetic criteria, so GRAD doesn't stop here.

3) Symmetric graph drawing: Symmetric drawing algorithms are among the classes of drawing algorithms that the popular graph drawing libraries do not support. GRAD currently offers one such algorithm, with another one being implemented. The available symmetric layout algorithm is based on the work of Carr and Kocay [22], which, given a permutation (automorphism) and a graph, produces a drawing which displays the desired symmetry. The permutations can previously be discovered using an implementation of McKay's canonical graph labeling algorithm [23]. An example of a drawing computed by GRAD's symmetric graph drawing algorithm is shown in fig. 4. This drawing shows a very well-known view of the famous Peterson graph.

4) Straight-line drawings: Straight-line drawings are another class of drawing algorithms which are not provided by the most popular libraries. While not applicable to all graphs (they have to be planar, and in some cases, 2 or 3-connected), they can guarantee a crossing-free drawing.

The first of the provided methods is the one based on Tutte's or barycentric embedding [24]. Given a simple 3-connected planar graph, Tutte's theorem produces a crossing-free straight-line embedding whose outer face is a convex polygon. It is considered to be the first force-directed algorithm at the same time. This method is not difficult to implement, but is only recommended to be used on smaller graphs, with 100 or less vertices.

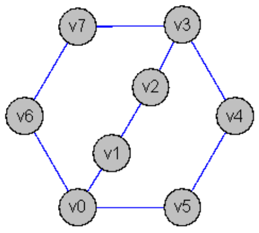

Fig. 5. Drawing of a graph on which Chiba's algorithm was applied

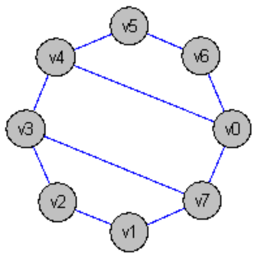

Fig. 6. A circular drawing of a graph

The second straight-line drawing algorithm is a much more complex one. The implementation is based on Chiba's linear algorithm for convex drawing of planar graphs [11], which firstly determines if a graph has a convex drawing and then draws one if the mentioned condition is satisfied. An example of this algorithm's application is shown in fig. 5.

5) Circular drawing: Practically all libraries which deal with graph drawing in some form, provide a circular drawing algorithm. Most of them, however, simply position vertices on a circumference of a circle. In addition to doing so as well, GRAD also implements an algorithm, described in [10], which determines the order in which vertices are placed so that the number of edge-crossings is as small as possible. An example is shown in 6.

It can be noticed that if, for example, vertices v_0 and v_7 switched places, the drawing wouldn't be planar.

Finally, it should be stressed that GRAD allows execution of drawing algorithms which were designed to be applied on simple graphs even if the given one is not simple. Upon execution of the desired algorithm in such case, GRAD detects multiple edges and loops and routes them in order to avoid overlapping of edges and correctly show those which connect one vertex to itself. Furthermore, GRAD provides a very simple way of calling any desired algorithm from a separately developed graphical editor, thoroughly explained in [5].

B. Graph analysis algorithms and choosing the appropriate graph drawing algorithm

Being a graph analysis library as well, GRAD provides a wide array of different algorithms of this sort. They can be used to reveal useful information regarding properties of a graph, which can later influence the decision of which drawing algorithm to apply in the given situation. Some of these algorithms were of great importance to the implementations of the existing drawing algorithms and can be used to help the implementation of additional ones. Among others, GRAD provides several algorithms for planarity testing, algorithms for splitting of graph into biconnected components based on depth-first search, Hopcroft-Tarjan splitting into triconnected components [25], different algorithms for finding cycles of graphs and previously mentioned McKay's graph labeling algorithm.

By applying appropriate algorithms it can be determined if a graph is, for example, a tree, if it has a planar straight-line drawing or non-trivial automorphisms. Taking advantage of this fact, GRAD provides a way of automatically invoking an algorithm which might be best suited for the graph in question. If a graph is a tree or a forest, a tree drawing algorithm is used. If it is planar, convex drawing is performed. If no special properties are detected, a force-directed layout is applied. Also, if the graph is disjoint, an algorithm is chosen for each component independently and these drawings are later combined to get the final one. An example of this is a class diagram with several disjoint groups of classes where some represent hierarchies, while the other ones do not. This offers users who have no special preferences a simple way to lay out their graphs, even if they don't know anything about graphs and graph drawing.

Furthermore, every somewhat sophisticated drawing algorithm puts emphasis on one or more aesthetic criteria. By naming the criteria, the users can basically choose the algorithm without even knowing its name. In order to accomplish this, a DSL for describing the desired properties of the drawing is currently being developed.

V. Conclusion

This paper explained the need to automatically lay out diagram elements, gave and overview of different classes of graph drawing algorithms and the most popular Java libraries offering some of them. Some difficulties one might encounter when using layout capabilities of the mentioned libraries within a separately developed graphical editor, as well as certain areas of improvement were pointed out. For example, a developer of a DSL which needs to support both textual and graphical syntaxes might run into these issues. They were addressed in our new graph drawing and analysis library called GRAD.

GRAD provides a number of different graph layout algorithms and a quick and easy way of using them to position elements of a diagram in any graphical editor. In addition to porting the best layout algorithms provided by other open-source Java graph drawing libraries, it implements various other ones, not offered by the other mentioned libraries. These include symmetric and two straight-line algorithms, as well as an enhanced version of a circular one. Additionally, GRAD offers ways of automatically choosing appropriate algorithm or their combination to get the best possible result. GRAD is currently being used in our open-source Kroki tool for laying out imported class diagrams created by other modeling tools. These diagrams can contain over 600 classes.

Plans for future improvements of GRAD include:

- implementation of additional drawing algorithms, including a better symmetric and one or more orthogonal ones
- labeling algorithms which address automatic placement of text symbol labels
- a better way of letting user specify desired aesthetic criteria by developing a DSL.

References

[1] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[2] U. Zdun and M. Strembeck, "Architectural decisions for dsl design: Foundational decisions in dsl development," in *Proceedings of 14th European Conference on Pattern Languages of Programs*, Germany, 2009, pp. 1–37.

[3] I.Dejanović, M. Tumbas-Živanov, G. Milosavljević, and B. Perišić, "Comparison of textual and visual notations of dommlite domain-specific language," in *Proceedings of the Advances in Databases and Information Systems*, 2010, pp. 20–24.

[4] I.Dejanović, G. Milosavljević, B. Perišić, and M. Tumbas-Živanov, "Domain-specific language for defining static structure of database applications," *Computer Science and Information Systems*, vol. 7, p. 409–440, 2010. doi: 10.2298/CSIS090203002D

[5] R. Vaderna, G. Milosavljvić, and I. Dejanović, "Graph layout algorithms and libraries: Overview and improvement," in *ICIST 2015 5th International Conference on Information Society and Technology Proceedings*, 2015.

[6] "Graph analysis and drawing library," https://github.com/renatav/GraphDrawing, accessed: 2016-4-4.

[7] M. Patrignani, *Handbook of Graph Drawing and Visualization*. Chapman and Hall/CRC, 2007, ch. 1, pp. 1–42.

[8] H. Purchase, *Computer Graphics and Multimedia: Applications, Problems and Solutions*. Idea Group Publishing, 2004, ch. 8, pp. 110–144.

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

[10] J. Six and I. Tollis, *Handbook of Graph Drawing and Visualization*. Chapman and Hall/CRC, 2007, ch. 9, pp. 155–192.

[11] N. Chiba, T. Yamanouchi, and T. Nishizeki, *Progress in graph theory*. Academic Press, 1984, ch. 5, pp. 153–173.

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

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

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

[15] "Jung framework," http://jung.sourceforge.net, accessed: 2016-4-4.

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

[17] "Jgraphx," https://github.com/jgraph/jgraphx, accessed: 2016-4-4.

[18] R. Davidson and D. Harel, "Drawing graphs nicely using simulated annealing," *ACM Transactions on Graphics*, vol. 15, pp. 301–331, 1996.

[19] "Prefuse," http://prefuse.org, accessed: 2016-4-4.

[20] C. Buchheim, M. Juenge, and S. Leipert, "Improving walker's algorithm to run in linear time graph drawing," in *Proceedings of 10th International Graph Drawing Symposium*, Irvine, CA, USA, 2002.

[21] "Kroki mockup tool," http://www.kroki-mde.net, accessed: 2016-4-4.

[22] 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.

[23] B. McKay, "Practical graph isomorphism," in *Proceedings of 10th. Manitoba Conference on Numerical Mathematics and Computing*, 1980, pp. 45–87.

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

[25] J. Hopcroft and R. Tarjan, "Dividing a graph into triconnected components," *SIAM J. Computing*, vol. 2, p. 135–158, 1973.