Using Graphviz to create aesthetically pleasing LaTeX drawings of graphs

Niels Benen
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
n.j.benen@student.utwente.nl

ABSTRACT

In this paper, the design of an application built to extend upon GraphViz is outlined. The current solutions relating to visualizations of graphs are discussed, after which a new architecture is proposed. In this architecture, an application allows a user to upload a graph, after which a 'first' drawing will be created. The user can then change this drawing by moving around nodes and edges, and finally export this drawing to LaTeX. This is not possible within any of the currently available programs. Features which will assist a user in creating a graph to their liking are discussed, and a prototype implementing these features was created to verify that the application works correctly.

Keywords

Graph Visualization, Graph Manipulation, GraphViz, TikZ

1. INTRODUCTION

A graph is an abstract mathematical object consisting of nodes and edges, used to depict relational information in a data set. A graph drawing is a 2D representations of these nodes and edges, where each node is shown as a circle and each edge as a line or a curve. By drawing a graph, the structure can be illustrated and useful properties such as symmetries and planarity can be uncovered. Graph visualization has always been a much-researched topic within the community of graph science. Countless algorithms exist which will all return a different layout for the same input (in the form of nodes and edges). In general, all algorithms try to fulfil the following aesthetical criteria [3]:

- Evenly distributed nodes, with a minimum distance between them, and should not overlap.
- Minimal number of edge crossings.
- Minimal number of edge bends.
- Minimal edge lengths.
- Maximize minimum edge angles within a node. This means that no two edges should come into a node at an (almost) equal angle. This avoids cluttering of edges.
- Maximize symmetry.
- Minimize graph area.

It is often the case that minimizing one aesthetical criteria will lead to the increase of another. Algorithms which create graph drawings are therefore trying to fulfil (some of) the aforementioned criteria, but no single algorithm can be fit for all different use cases.

33rd Twente Student Conference on IT July. 3rd, 2020, Enschede, The Netherlands.

2. CONTEXT

There are already many available algorithms which can accommodate a graph visualization. There are many different types of algorithms, of which Force-Based and Layered Drawings are more the most common type of algorithms that will work on many types of input graphs. For this reason, these lay outs will therefore be explained in more detail [14]. Besides these algorithms, there are also many algorithms which only work when the input graph has certain properties. These algorithms are also discussed.

2.1 Force-Based Layout

In a force based layout algorithm, each node has a repulsive force on all other nodes, but edges create an attractive force between the connected nodes. Then, based on real-life physics, a minimized energy state is pursued. In general, the closer two nodes are to each other, the stronger the repulsion will be between them. This force will get so big that all edges will come close to an 'ideal length', avoiding very short edge lengths. In this state, there should be as little forces as possible still acting on any node [1].

2.2 Layered Drawing (Sugiyama Drawing)

In a layered algorithm, all nodes are first assigned a layer using a separate algorithm (for example Coffman-Graham). It should be noted that this algorithm requires a DAG (Directed Acyclic Graph). Therefore, if the input graph has a directed cycle, the first step of this algorithm is to perform a cycle removal stage. When this is the case, one or more edges will be reversed to make sure the graph becomes acyclic. Then, each of the nodes are assigned a layer, such that edges only move downwards from layer to layer, thus disallowing edges on the same layer or moving upwards. Coffman-Graham, or similar algorithms, are designed to perform this task. Once all nodes have been assigned a layer in this way, an edge crossing minimization phase is entered. During this phase, the nodes get ordered from left to right among the layer they were assigned to in the order which yields the fewest number of crossings [1]. This problem is NP-hard and no practical algorithm is known to date [4]. Therefore, a heuristic function is used for this stage. Finally, the edges that were reversed will be returned back to their original direction. This means that

there still might be edges that move up the layers, which might be undesirable and should be taken into account when creating a Sugiyama Drawing of a directed cyclic graph.

2.3 Other Algorithms

There are also plenty of other algorithms, such as circular, radial and tree layouts. In a circular layout, nodes are divided into clusters. Each cluster gets placed along the circumference of an individual circle. This is useful for showing clustered information within graphs [12]. A radial drawing is an extension to this, with the additional constraint that each circle has the same center, and only the radii differing. As mentioned, these algorithms often only work properly on input graphs that have certain properties, such as being a (binary) tree.

Usually, it is up to the creator of the graph to determine which algorithm they find the best fit. Even if the graph can very cleanly be displayed using a circular layout, the creator might find that another algorithm produces a better result because it encapsulates his thought of the graph better. For the same reasons as described in section 1, there is no 'best' way to draw a graph. One person might find one aesthetical criterion to be the most important while another person might think this criterion is not as important.

3. REQUIREMENTS

It is important that the application adheres to a number of requirements to ensure that it works correctly and intuitively. An important general requirement is that the application should be free-to-use, so that every researcher around the world could make use of it without the need for any financial investments.

3.1 Importing of Graphs

As the program does not facilitate the creation of graphs, nodes or edges, it is important that the user can import a graph using an easy-to-create format. The easiest way to allow for this is by having the user upload a graph in a format in which only the edges are specified. This information alone is enough to create a drawing of the graph and thereby requires the least effort by the user. It would also be preferable if the format used is already widely used. By conforming to already an available specification, it will be much easier for a (new) user to use the application.

3.2 Provide a lay-out for the graph

Once the user has uploaded a graph, they should be able to choose from a variety of different lay-out algorithms to facilitate the type of graph they uploaded. Once an algorithm has been selected, the program should automatically create a drawing of the provided graph. This should be done via an already available API, specifically designed for the purpose of graph drawing. This ensures that the first drawing presented to the user is as good as possible. It might happen that an edge goes through another node. The user should be notified about these collisions by a message informing them about which edge(s) collide(s) with which node(s).

3.3 Allow for customization

It is important that a user can customize the graph to their liking. This includes the toggling of labels, toggling directed edges and the specification of node size. By allowing a user to set these parameters, they can easily customize the graph to their liking. Furthermore, it should be possible to move nodes or curve edges simply by clicking and dragging them to another location. By allowing a user to do this, (minor) errors by the aforementioned API can be mitigated by the user.

3.4 Export to LaTeX

Finally, the application should ensure that the graph can easily be exported to LaTeX. A user should be able to simply copy paste the generated result into the document, which should then place the graph they created there. In this process, it is important that the graph is scaled correctly, such that the relative distance between any two nodes remains the same, and that the node size is also scaled accordingly. If the node size would not be scaled, it can be possible that an edge will go through another node in the exported drawing, but not in the application, because the relative sizes differ.

4. EXISTING SOLUTIONS

There are already available applications which can generate an initial drawing of a given graph. Furthermore, there is also an extension to LaTeX which allows a user to generate and neatly display a given graph in a document. We will look at two often-used graph drawing programs: GraphViz and yEd. Their APIs will also be examined to determine which program is the best to extend upon. We will also look at TikZ, the LaTeX extension which provides features to draw graphs.

4.1 Graph Drawings Programs

4.1.1 yEd

yEd is a closed source program that allows a user to upload a graph in a specific format [15]. A selection can be made from a wide variety of lay-out algorithms, among which the most common ones such as a force-based layout. It is possible to customize a graph to a certain extent. A user can change the color of the nodes or the size of a node. It is not possible for a user to edit the topology of a created graph; a user has to take this graph for what it is and is not able to manipulate this topology any further. This is a major shortcoming, as the lay-out algorithms often, as described before, rely on heuristics to come up with the best graph. These heuristics are not fail-proof, and it may happen that a node is placed in a non-optimal location. A created graph can be exported into a variety of common formats, such as .PDF, .JPG and .PNG, but there is no option to directly export to a LaTeX compatible format.
The API of yEd is well-documented and offers all of the lay out algorithms that are also offered by the official GUI [16]. A license is needed to be able to use the API. This license means that creating an application that runs using this API has to be closed-source, or requires users to buy their own API key. This is of course not desirable and directly counters the requirement that the application should be free-to-use.

4.1.2 GraphViz

GraphViz is an open source program that does not itself come with any sort of GUI. [8] It allows a user to upload a graph in the .DOT format. This format requires a user to list the edges in any graph and will, using this information, generate the complete graph. GraphViz also offers a variety of common force-based and layered layouts, as well as some other lesser-used layout algorithms [6]. As GraphViz purely applies a lay-out algorithm to a graph, it is not possible within GraphViz to change the created graph. But, just like with yEd and any other graph drawing software, the generated drawing might not be com-

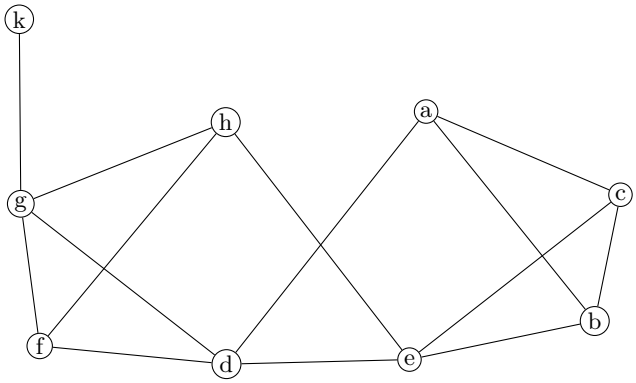

Figure 1. A graph as created by GraphViz using the FDP algorithm

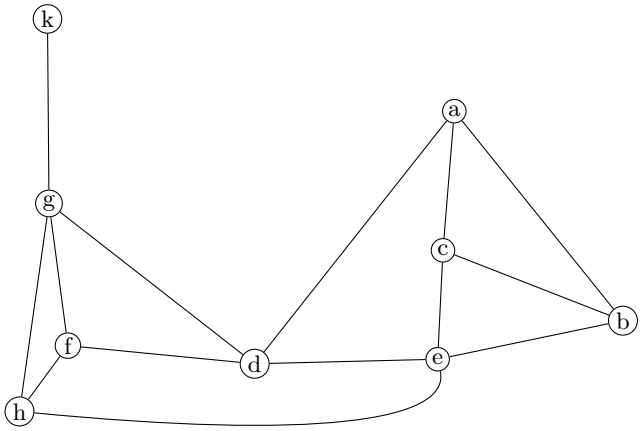

Figure 2. The same graph in planar form

pletely to the users liking. An example of such a graph is given in Figure 1. This graph is generated by GraphViz and has not been altered. Generally, planar graphs are easier to look at and understand. A planar graph is a graph with no overlapping edges. It is not possible to create a planar drawing for every graph and they are hard for an AI to automatically create, which is why an AI will try to approach a planar drawing. A user might then manipulate the drawing to create a planar drawing, such as the one in Figure 2, and think that this drawing is a better fit for their purpose.

The GraphViz API offers quick and easy access to all of the features it offers. As mentioned before, GraphViz is open source and this means that their API is free to use. Because GraphViz is written in C, the API is also in C [7]. But there are also APIs available that allow the use of GraphViz in both Python and Java [11].

4.2 Graphs in LaTeX

Many researchers use LaTeX to write their reports, papers or other documents. The general way of getting a drawing of a graph in a LaTeX document now is as follows: first, the software generates an image of the graph, which can then be inserted into the document using. This means that additional steps have to be taken to load a graph into a document, which slows the process down. Furthermore, inserting images into a LaTeX document can make this image pixelated, and thereby not readable. There is also a LaTeX extension to generate graphs, called TikZ [13]. This program allows a user to neatly insert a graph in a document, just like a table or a list would be inserted. TikZ does not provide any inbuilt lay-out algorithms which will generate a graph based on a list of nodes and vertices. Graphs can only be generated by providing the (relative) positions of each of the nodes This makes it so an author cannot generate a first guess of any graph. On top of this, if a small change is made to the graph, the whole document has to be reloaded for the change to be loaded into the document. Creating a curved edge is possible by providing the angle at which an edge has to leave a node, or by giving an edge a specific parameter. It is not easy for a human to work out at what angle the edge should leave the node when the result does not automatically update, and the parameters do not yield full flexibility but rather only specific angles.

dot2tex

It should be noted that there is already an application to convert GraphViz drawings to a LaTeX compatible format, called dot2tex [5]. This application does not preview the created graph to the user, which means that they would have to paste the generated into a LaTeX document before they can see the result. This makes it hard to compare different lay out algorithms. On top of this, there is again no option for customization of the positions of the nodes and edges, unless the returned code is edited manually by the user. It is then a blind guess as there is no direct feedback to what the user edited, which makes the application unfit for the purpose of this research.

5. NEW ARCHITECTURE

An application was designed to fulfil all of the requirements. In this section, the design of this application is discussed.

5.1 Libraries used

As described before, the application should extend an already available graph drawing API to make sure that all the requirements are met. In order to be able to meet the requirement that the application should be free-to-use and available to anyone, the preference lied at an open-source graph drawing software. This is why the GraphViz Java API was chosen as the API to use for generating the first drawing of the graph. This decision was taken in spite of the relatively few lay-out algorithms that are offered by GraphViz (five). This fact was not seen as a major problem, though, as the nature of the application allowing people to change the graph to their liking means that there is no need for a lot of different algorithms. The five algorithms offered are therefore sufficient, as the combination of these manage to evaluate many of the input graphs quite well.

As mentioned previously in section 4, there is already an available extension which can display a graph in a very neat manner in a LaTeX document, called TikZ. As this extension offers a great variety in options to accommodate the lay-out of a graph within a document, TikZ will be used to display the graphs within LaTeX.

5.2 Implementation

A prototype was created implementing the most important features. The code can be found on the GitHub page of the project [2].

5.2.1 Importing of Graphs and generating lay-out

Because GraphViz was designed to work with .DOT files, it was only logical to also use that as the way for users to import a graph into the application. In the DOT language,

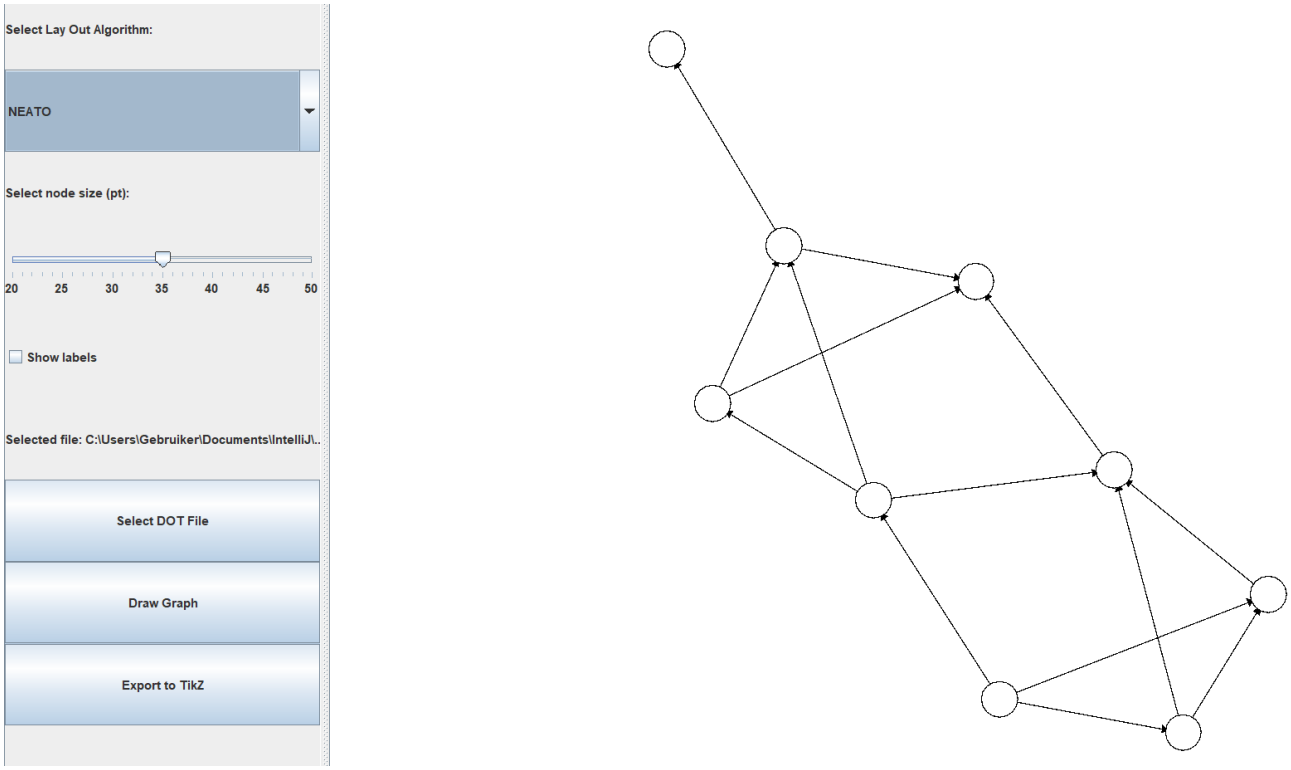

Figure 3. The GUI of the prototype

a user first specifies whether the graph is directed or not. Optionally, the user can then define all the nodes with an ID and define some properties that define their layout, such as shape, color and label. For this prototype, only the label feature has been implemented. Finally, all edges are defined by: *nodeFrom* – *nodeTo* for undirected graphs or *nodeFrom* -> *nodeTo* for directed graphs. If a node ID was not defined before, it will be automatically created at this stage with the default lay out properties. An example DOT file is given in Figure 4. Once the user uploaded the file and selected the layout method, the selected file will be parsed and the lay-out will be generated by GraphViz. It is possible for GraphViz to return a drawing with splines (curved edges), but the application requests GraphViz explicitly not to do this. There are a few advantages to this, the main one being that it creates a more intuitive environment for the user. This will be explained in more detail in section 5.2.2.

The lay out algorithms offered by GraphViz are described in [6], and consist of two force-based lay outs (FDP, NEATO), a layered drawing (DOT), a radial lay out (TWOPI) and a circular lay out (CIRCO). There is also the OSAGE algorithm, but this algorithm is built for displaying larger, clustered graphs. As this application focuses on smaller graphs which can be displayed in documents, this algorithm will not be discussed further.

```
digraph {
    a [label="Foo"];
    a -> { b c };
    b -> c;
}
```

Figure 4. The DOT file for a triangular graph

This layout is returned in the plain format [9], which in turn will be parsed again by the program. The program will store the generated location and label of each of the nodes as well as the (from, to) tuple of all edges. Based on the screen size of the user, the graph will be scaled to fill as much of the screen as possible and is moved towards the middle of the screen. In this step, special care was taken to ensure that the *relative* positions of the nodes do not change.

5.2.2 Graph Customization

Now that the graph is presented to the user, it is possible to manipulate the graph. One of the main issues that massively decreases the readability of a graph is edges which go through another node. To inform the user of such events, the colliding nodes and edges will be colored red. The user can then decide to fix these collisions by moving nodes or curving edges. The GUI of the prototype is shown in Figure 3.

Moving a node is simple, the user can drag and drop any node to any other location on the screen. The edges that are connected to that node will automatically move along with the node. This happens instantly when the node is moved, even if the user has not released the node. This immediate feedback makes it so a user can see what effect moving one node has on the total picture.

The decision was made to also allow users to curve edges. A user can click on an edge and drag around his mouse, just like with the movement of nodes. The line then becomes a Bézier Curve with a control point at the position of the mouse. Doing so allows for users to move the edges in an intuitive manner. This is also the reason why it was decided that the initial drawings should be straight-lined. The splines that are produced by GraphViz are Bézier Curves with n control points. This would mean that curving an edge would no longer result in an edge that would

predictably follow a complete trajectory according to the position of the mouse, but rather one that only changes a part of the curve. The splines that GraphViz automatically create can have many control points, which would mean that the user would have to move each and every one of these points by hand in order to change the trajectory of a curve completely. Alongside this, when many control points get placed close together, it becomes very hard to determine which control point the user meant to click. This problem is amplified by the fact that control points lie away from the actual curve and are invisible to the user. For this reason, it was decided that a user can only create a curve using one control point.

More features were added that allow the user to customize the look of the graphs, such as changing the size of the nodes and toggling whether labels are shown.

5.2.3 Export to / Import into LaTeX

The application also comes with the option to export the created graph into a LaTeX document. Because use cases will differ across different graphs, the maximum width and height can be defined. The application will then scale the graph down to fit these new dimensions, again ensuring the preservation of *relative* positions. Because TikZ uses a coordinate system starting from the bottom left, whereas the application uses a coordinate system starting from the top left, all node positions are amended to ensure that that the graph will not be upside down when displayed in LaTeX. Then, the graph is finally exported to a TikZ compatible format such as the example in Figure 5 as follows:

First, the picture is initialized by setting some properties that control the lay-out of the nodes. Then, each node is defined with their ID, position and label. If no label is present, the value between the curly brackets will be empty. Finally, all the edges are drawn and the picture is created.

Because TikZ does not draw the quadratic Bézier curve produced by the application in the same way, a conversion has to take place. Within TikZ, using only one control point will result in the creation of an edge that is a cubic Bézier curve with two control points in the same spot, rather than a quadratic curve. This means that the edge is much sharper than originally shown in the application. To convert the quadratic Bézier curve produced by the application to an equivalent cubic one, the following theorem is used:

THEOREM 1. *Let $p1, p2$ be the start, respectively end point of a quadratic Bézier curve, and $q1$ be the control point. The two control points $c1, c2$ for a cubic Bézier curve can then be defined as [10]*

$$c1 = p1 * \frac{2}{3}(q1 - p1)$$

$$c2 = p2 * \frac{2}{3}(q1 - p2)$$

This is equivalent to setting the 2 new control points at exactly one third of the distance between the old control points and the start and end points. Therefore, this is exactly what happens in the generated TikZ code.

The main benefits of being able to perform all these tasks within one application are ease of use and the fact that they can be performed more quickly. This will save users a lot of time in the long run, as they will no longer be required to export the result of one application into another application to perform the next step in the process of creating a graph, but rather they can directly go from the .DOT specification to the LaTeX compatible format.

```
\begin{tikzpicture} [every node/.style={draw,
    circle,minimum size=0.45cm,inner sep=1}]
\node (a) at  (3.71,8.34) {Foo};
\node (b) at  (3.71,1.12) {b};
\node (c) at  (9.97,4.73) {c};
\draw[-{Latex[length=2mm]}](a) -- (b);
\draw[-{Latex[length=2mm]}] (a) ..
     controls($(10.19,8.83)!1/3!(a)$)
     and ($(10.19,8.83)!1/3!(c)$) .. (c);
\draw[-{Latex[length=2mm]}](b) -- (c);
\end{tikzpicture}
```

Figure 5. Generated TikZ code for the example DOT file in Figure 4, with a curved edge from a to c

6. VERIFICATION

To verify whether the new architecture adheres to the requirements as specified in section 3, a number of tricky graphs were tested together with the supervisor of the project, prof. dr. ir. Hajo Broersma. The tested graphs were a cube, a dodecahedron and the Petersen graph. In all cases, the graph could easily be imported into the prototype using the .DOT description of the graph. These dot files can also be found on the GitHub page of the research.[2] Some of the results produced during this verification can be found in Appendix A.

Cube

First, it was tested how the available lay-out algorithms presented the cube to the user. The NEATO and FDP algorithms were found to present the user with a very good first guess for what the graph should look like if a 3d representation of a cube was desired. The CIRCO algorithm would return a graph that could easily be manipulated to create a planar graph.

Dodecahedron

The dodecahedron is a more complex graph consisting of 20 nodes. Despite this, it can still be displayed in a planar form. None of the provided algorithms will instantly display this planar form, but the NEATO algorithm comes quite close. The result could be manipulated to create a planar representation of the graph by moving the middle circle to the outside. Furthermore, the FDP algorithm displayed the best 3d result.

Petersen Graph

As a final test, the Petersen graph was examined. This graph is commonly displayed in a variety of different lay-outs, but none of the provided lay-out algorithms would properly capture this. Despite this, the desired representation of the graph could still be created by using the available manipulation tools when using the FDP algorithm as a basis. This is, in this case, hard to do without prior knowledge of what a user wants a graph to look like in the end, but even without this knowledge a user can move the nodes and edges until they find a lay out that they like.

The final test concerned testing the correctness of the function which exports the graph to LaTeX. The created graphs were exported to a TikZ compatible format and put in a

LaTeX document. This feature is mostly functional, although there are still some minor bugs, such as node sizes differing if a node has a longer label than the others. All of the graphs in this paper have been created using the prototype and its export function.

7. CONCLUSION

This research set out to find a solution to help researchers produce nice LaTeX drawings of graphs. By defining a number of requirements for a new architecture, the process of creating visualizations can be sped up drastically. All these requirements have been fulfilled by means of a prototype. The prototype was verified to be working correctly in a testing session together with the supervisor of the project. It was also found that, indeed, it was easy to manipulate any graph to the user's liking. The most useful lay-out algorithms to start with for the tested graphs turned out to be the CIRCO, NEATO and FDP algorithms. The TWOPI and DOT did not produce nice drawings for any of the tested graphs, but rather returned unusable results. A few examples of such unusable graphs are shown at the end of Appendix A, Figures 13-15. As stated before, TWOPI is a lay-out algorithm that only will work nicely on graphs which can nicely be visualized by a radial layout. DOT is a layered drawing, which does not work well for geometrical figures, but will work on many other types of input graphs. There is definitely potential in creating an application that helps researchers create aesthetically pleasing graphs as it will speed up the process of creating (smaller) graphs, and available libraries do not always return the desired result.

8. DISCUSSION AND FUTURE WORK

During the verification of the program, it became apparent that more features could be added to assist a researcher in creating even better graphs. Firstly, more manipulation options could be added. This includes features such as adding labels to edges, being able to create dotted edges or changing the color of the nodes. Furthermore, it would be very handy of researcher to be able to 'snap' to the x- or y-coordinate of other nodes. This would make it so symmetries can much more easily be created, as the user would not have to rely on their own eyes to see whether two nodes are aligned or not. This feature would also assist a user who would like to use the application with a bigger graph, although there will always be a maximum number of nodes that can fit because of the limited available space and lack of zooming capabilities on a physical paper.

9. REFERENCES

[1] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. 1999.

[2] N. Benen. Graph visualization tool. `https://github.com/nielsbenen/GraphVisualization`.

[3] C. Bennett, J. Ryall, L. Spalteholz, and A. Gooch. The aesthetics of graph visualization. *Proceedings of Computational Aesthetics in Graphics, Visualization, and Imaging*, pages 57–64, 01 2007.

[4] C. Buchheim, M. Chimani, D. Ebner, C. Gutwenger, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. A branch-and-cut approach to the crossing number problem. *Discrete Optimization*, pages 373–388, 05 2008.

[5] K. M. Fauske. dot2tex - a graphviz to latex converter. `http://ftp.cvut.cz/mirrors/ctan.org/graphics/dot2tex/dot2tex.pdf`. [accessed 28-May-2020].

[6] E. R. Gansner. Drawing graphs with Graphviz. `http://www.ammd.ch/1.pdf`, 2011. [Accessed 1-June-2020].

[7] E. R. Gansner. Using Graphviz as a Library. `https://www.graphviz.org/pdf/libguide.pdf`, 2014. [Accessed 25-April-2020].

[8] Graphviz. `https://www.graphviz.org/`.

[9] Graphviz. `https://www.graphviz.org/doc/info/output.html#d:plain`. [Accessed 29-April-2020].

[10] D. Morgan. Degree Reduction of Bézier Curves. `https://yamnuska.ca/geek/degreeReductionBezier.pdf`. [Accessed 15-June-2020].

[11] S. Niederhauser. graphviz-java. `https://github.com/nidi3/graphviz-java`.

[12] J. Six and I. Tollis. A framework for circular drawings of networks. *Graph Drawing*, 1731, 07 1999.

[13] TikZ. `https://www.overleaf.com/learn/latex/TikZ_package`.

[14] R. Vaderna, G. Milosavljevic, and I. Dejanovic. Graph layout algorithms and libraries: Overview and improvements. 03 2015.

[15] yEd. `https://www.yworks.com/products/yed`.

[16] yEd. `https://www.yworks.com/products/yfiles/documentation`.

APPENDIX

A. RESULTS OF THE VERIFICATION

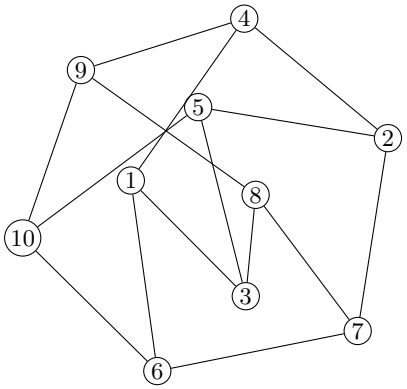

Figure 6. The Petersen Graph as returned by the FDP algorithm

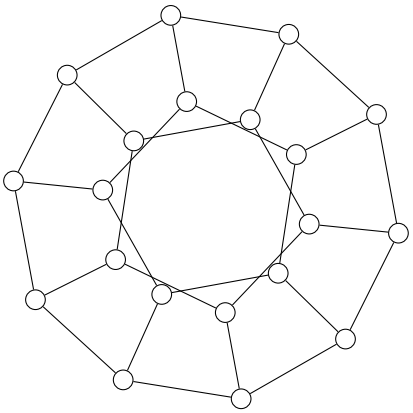

Figure 9. The Dodecahedron as returned by the NEATO algorithm

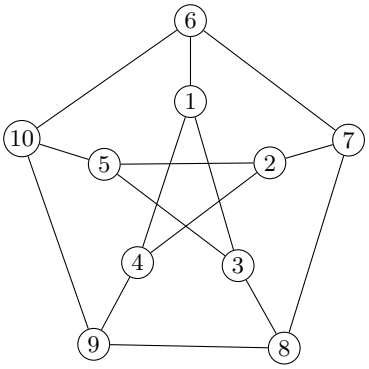

Figure 7. The Petersen Graph in a more readable form after manipulation

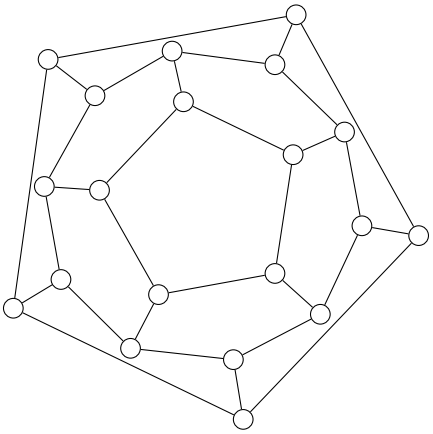

Figure 10. The Dodecahedron in planar form after manipulation

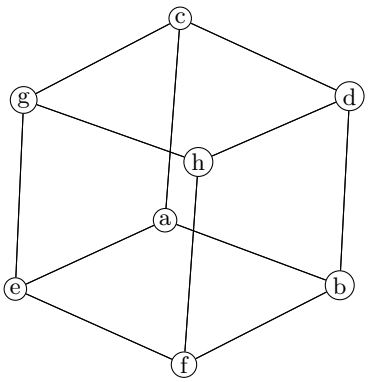

Figure 8. The Cube as returned by the NEATO algorithm

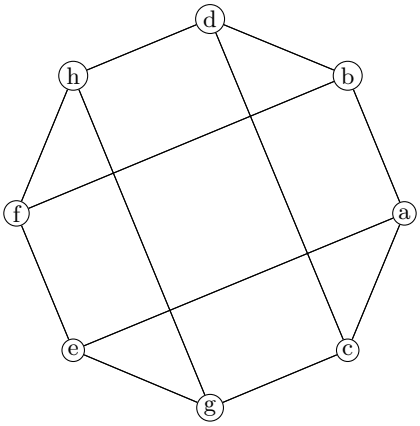

Figure 11. The Cube as returned by the CIRCO algorithm

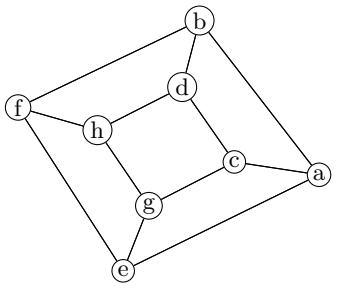

Figure 12. The Cube in planar form after manipulation

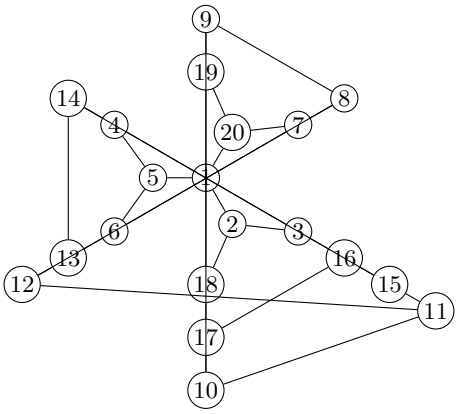

Figure 14. The Petersen Graph as returned by the TWOPI algorithm

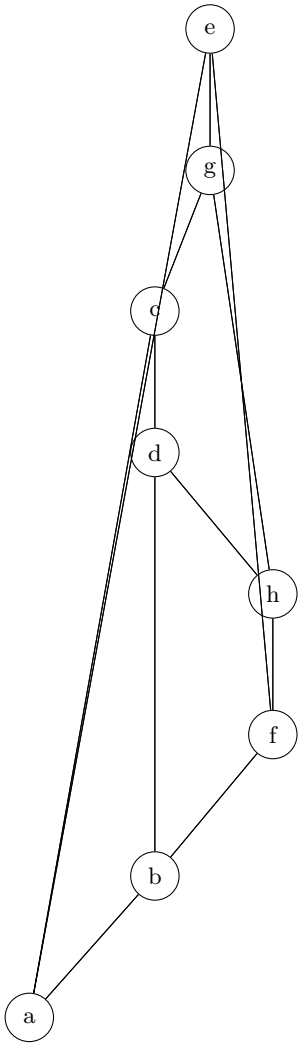

Figure 13. The Cube as returned by the DOT algorithm

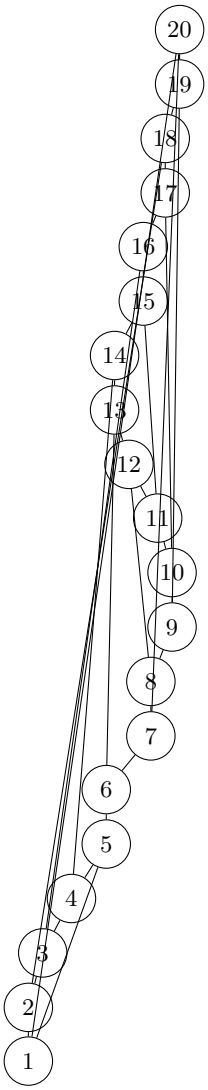

Figure 15. The dodecahedron as returned by the DOT algorithm