A Linear-Time Visualization Algorithm for Drawing Planar Graphs Represented by Regions

Jeffery A. Stuart, Brian T. Westphal, Bei Yuan, and Frederick C. Harris, Jr.
Department of Computer Science, University of Nevada, Reno Nevada 89557, USA
Fred.Harris@cse.unr.edu

Abstract

Graphs can be represented using a variety of means, the most common of which are adjacency matrices and edge lists. For undirected graphs, one alternative to these is to use a region-based representation. Compared with adjacency matrices or edge lists, region lists are significantly more difficult for humans to visualize. Still, a region-based description of a graph can be a powerful tool for certain types of problems that require highly generalized graph descriptions. The algorithm for region-based graph visualization outlined in this paper, which focuses on the drawing of planar graphs, is simple to understand, can be easily implemented using only a minimal background in graph theory, and executes in linear time. The algorithm has been implemented in a graph drawing tool called CircleGraph, which is also described in this paper.

Key Words: crossing number, region.

1 Introduction

In general, the notion "visualization" refers to the process of mapping abstract data to a suitable graphical representation that simplifies data interpretation by the user. For many applications, especially where data sets are very large, the problem of visualizing data can be a numerically intensive task. New graph drawing algorithms are needed in order for many complex data sets to be successfully visualized.

Originating with the work of Tutte [16] and Knuth [10], the amount of research on graph drawing algorithms has grown tremendously, especially in the last decade over which the number of high-resolution graphical devices and graph drawing software tools has steadily increased. Development APIs, such as LEDA [13] and GraphBase [11], provide tools for implementing graph-drawing algorithms and tools such as EDGE [14] and DaVinci [17] concentrate on visualization of specific classification of graphs.

Graph drawing algorithms commonly use the most general graph attributes such as adjacency lists to generate drawings under various constraints. In many cases however, the typically specified graph attributes are insufficient for solving desired problems quickly. One of the benefits of using a region-based approach towards solving graph problems is that one can assume the planarity of the resulting graph. That is, according to Euler's Formula [2], a connected graph G is guaranteed to be planar if:

$$n - m + r = 2$$

where n, m, and r are the number of vertices, edges, and regions.

Significant efforts have been made towards solving the minimum crossing number (MCN) problem for complete graphs K_n. In one of the most recent solutions, non-planar graphs that contain crossings are converted into planar graphs by regarding crossing points as virtual vertices. For instance, an MCN solution for K6, requires three crossings as a non-planar graph, and therefore requires three virtual vertices as a planar graph. K_6 is a complete graph with six vertices, shown in Figure 1.

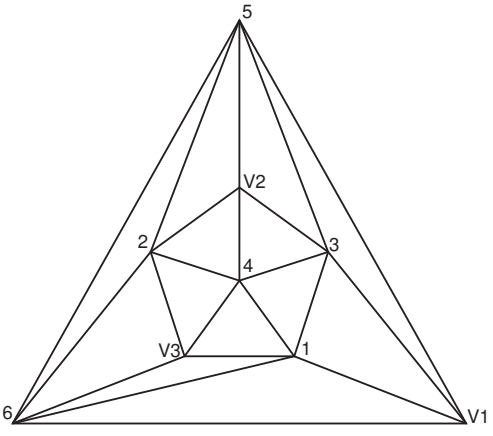

Figure 1: K_6 with three virtual vertices (V1, V2, V3)

An increasing number of algorithms [8] rely on region lists for graph representation. As scientific validation/verification of graphs is a driving force for the visualization of region-represented graphs, providing

tools and algorithms by which graphs can be drawn is essential. Though there are a good number of graph visualization algorithms and tools, most current methods are difficult to implement. This paper presents a simple algorithm, both to understand and to implement, for drawing graphs. Also presented is an interactive tool, used in conjunction with the algorithm, to draw graphs.

Many algorithms, dealing with NP-hard graph problems, create output graphs without assigning coordinates for specifically positioning vertices. Without visualization, one is left with the dilemma of either creating resulting graphs by hand (which is impractical for most cases) or taking the results on faith. In the case of the MCN problem, graph visualization is used to prove that the resulting data is correct. In other words, actual graphs, which have the minimal crossing number, can be presented. Errors, such as edge crossings, are readily apparent by briefly glancing at a visualized graph.

The rest of this paper is outlined as follows: Section 2 presents a review of the background literature for graphs and graph drawing. Section 3 introduces and discusses the Vertex Connection Scheme algorithm that we developed for defining the graphical structure of a graph. The results and effectiveness of the algorithm as well as an introduction to our interactive graph manipulation and visualization tool called CircleGraph are discussed in Section 4. This is followed by the future work and conclusion of this paper in Section 5.

2 Literature Review

2.1 Review of Graph Drawing

There are many types of graphs, each of which can be drawn in a multitude of *styles* with various *aesthetic criteria* and *algorithms*. Commonly defined graph types include: complete, bipartite, acyclic, directed, and undirected graphs, among others. In addition, one may limit the dimensionality of a graph, typically to $2D$ (planar) or $3D$ (spherical) coordinates. A drawing is *planar*, if there are no edge crossings. An *orthogonal* drawing maps each edge into a chain of horizontal and vertical segments [2]. A *grid* drawing is embedded in a rectilinear grid such that the vertices and edges have integer coordinates (a special case of orthogonal drawing). In addition to selecting a particular type, one drawing is considered better than other drawings if it meets certain aesthetic criteria such as number of crossings, size of drawing area, degree of the smallest angle, and number/type of symmetries. There will always be trade-offs for the output graph, since it is often impossible to simultaneously optimize two aesthetic criteria. For instance, Figure 2 is used to demonstrate that if one wants ensure the minimum crossings, one may sometimes have to sacrifice maximal symmetries.

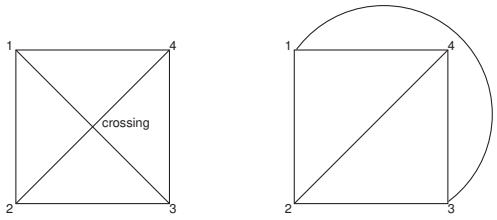

Figure 2: K_4 with/without crossings

2.2 Previous Work in Graph Drawing

Research on graph drawing focuses on creating models, developing algorithms, and building systems for the visualization of graphs and networks. This problem is more general than one might imagine as common applications include software engineering (class hierarchies), database systems (ER-diagrams), and circuit layout design.

Previous work in graph drawing is represented in [3, 4, 7, 15, 16]. Many of the experiments associated with the presented algorithms show that the current state of algorithm development is limited. In several cases, due to algorithm complexity, only small graphs can be constructed or the algorithms work in only a subset of the expected cases.

Because drawing a graph according to a pre-specified set of aesthetic criteria can be regarded as searching for an optimal layout within a virtually limitless graph layout space, many Genetic Algorithms have been applied to graph drawing. For example, [1] and [6] use Genetic Algorithms to draw undirected graphs and [12] bipartite graphs. Genetic Algorithms however, suffer from difficulty in achieving optimal configuration for crossover and mutation rates, randomization, and often, extensive processing requirements. Typically, results can only be gathered for moderate sized graphs, most of which may not be aesthetically pleasing. Furthermore, adding aesthetic criteria often diminishes the ability to find acceptable graph presentations while reducing criteria results in unusable graphs.

Consequently, researchers have begun to develop graph-drawing systems to allow direct interaction with graph layouts. A well-known drawing system, GraphEd [9], provided many experiments in evaluating different drawing algorithms. The system includes a set of parameterized layout algorithms that cover a wide range of modern drawing algorithms. One of the major issues with such systems however is one of

cross-platform compatibility. That is, most systems are developed to run on a single operating system / hardware configuration. This inevitably limits usage in the diverse world of scientific research. Most of these systems also require that graphs be input using specific formatting requirements, adding an often time-consuming process of conversion to ones work.

This paper demonstrates that it is possible to build a graph-drawing system that uses a fast (linear-time), automated process for approximating a final graph drawing followed by minimal human interaction to perfect the graph drawing in terms of human aesthetic criteria.

3 Algorithm

Current graph drawing algorithms, besides rendering disorganized graphs, tend to be complicated to understand and difficult to implement. The algorithm presented in this paper, Vertex Connection Scheme (VCS), is simple, and useful in producing aesthetically pleasing graphs. VCS makes use of several key concepts, each of which help to reduce the abstraction involved in choosing coordinates for graph vertices. Layering is a method used for organizing the vertices of a graph. All vertices in VCS lie on the perimeter of a concentric circle, where there is one concentric circle per layer (or virtual layer).

To define the layers of a graph one must first choose an outside region. The outside region used in this paper is the largest region of a graph. In the case of ties, the region whose sum of vertex degrees (the number of edges in which the vertex is involved) is highest is used. Alternative outer layers may be useful as well, but it has been found that space usage can be optimized by moving larger and higher-degree regions close to the outside. Inner layers are defined by performing a multi-source BFS starting from the outside layer. Vertices that are connected (via an edge) to the vertices of the outside layer are in layer 1; vertices that are connected to the vertices in layer 1 are in layer 2, and so on. After each vertex has been associated with a layer, one must also orient the vertices on each layer. The algorithm presented in this paper uses an inductive method to determine the (clockwise or counterclockwise) orientation of vertices.

Besides using standard layers as defined above, virtual layers are also incorporated for convenience. A virtual layer is an additional layer onto which vertices from adjacent layers are moved for the purpose of organizational convenience. Using virtual layers for specific cases allows for simplification in the vertex organization process. Upon calculating the vertex layers, layer orientations, and virtual layers, one may then begin defining absolute positions for vertices. The results section describes an interactive tool used for graph manipulation allowing one to specify the exact appearance of the graph while helping to maintain the constraints yielded by VCS.

Vertex Connection Scheme Structures: In order to complete the Vertex Connection Scheme, one must keep track of the following structures: region list, adjacency matrix, adjacency list, layers and virtual layers, the layer number associated with each vertex, and for each vertex a list of regions containing the vertex (which are referred to as the vertex-region maps). Some of this data is redundant, however this redundancy is used to achieve the linear-time complexity of VCS.

Initialization: The VCS algorithm is given the region list. From the region list, one initializes the adjacency matrix, adjacency list, vertex-region maps, and the outside layer. The algorithms for building the adjacency matrix and adjacency list can be found in [8] and will not be discussed in this paper.

Layer Labeling: Layers are fundamental to VCS as they describe the general order in which the vertices are placed. Layers define a topological ordering of the graph. This ordering is found using a multi-source breadth-first search. The search starts simultaneously with all the vertices in the outside layer (the outside layer was found in the initialization phase). All the vertices in the outside layer are labeled as being in layer 0; all other vertices are unlabeled. For all newly discovered vertices, any unlabeled neighbors are put into a new layer. For example, all unlabeled neighbors of layer x would become labeled as being in layer $x + 1$. This process continues until every vertex has been discovered.

Layer Ordering: After the vertices are labeled with a particular layer number they need to be correctly ordered within their layer. Edmonds Rotational Embedding Scheme [5] already imposes a cyclical ordering on the outside layer, so no extra work needs to be done for the outside layer. Edmonds Rotational Embedding Scheme enforces a cyclical ordering on the neighbors of every vertex.

Each layer's vertices can be ordered based on the order of the vertices in the previous layer. Since the outside layer is automatically ordered, according to the embedding scheme one can start with the layer one-in from the outside. Before ordering a layer, two

data structures should be setup: *nextLayer* (which represents the layer being ordered) and *ict* (which is the inward connection table). *nextLayer* is initialized as an empty list, *ict* as an empty table.

To order a layer, one traverses the vertices of the previous layer. With the first layer to be ordered, for example, this would be the vertices of the outside layer. For each of these vertices v from the previous layer, one finds and counts the number of vertices on the current layer to which v connects. The counted vertices are stored in *ict*.

If *ict* is empty, no more processing occurs for v. If there is one element in *ict*, the element is appended on *nextLayer*. If more than one element is in *ict*, one must determine the cyclical ordering o of the previous layer's vertex (as described in section 3). One must remove elements of o that are not in *ict*.

Using o, one can find $v's$ clockwise successor u. Traversing o in reverse order, if one finds a vertex w that connects v and u, then o should be rotated (with wrap around) so that w is the last vertex in the list. The elements of o are then appended on *nextLayer*. Duplicate vertices sometimes appear because of multiple options available in vertex ordering. These extraneous vertices need to be removed after each layer is completely ordered.

Determining the Cyclical Ordering of a Vertex's Neighbors: Figure 3 shows a region-based graph with seven vertices. This graph is used to demonstrate, through the following example, how one can determine the cyclical order on the neighbors of a vertex. The key to this algorithm is using the vertex-region map to orient the neighbors of a vertex. For this example, we have chosen an arbitrary vertex, 6, whose neighbors (2, 3, 5, and 7) will be oriented.

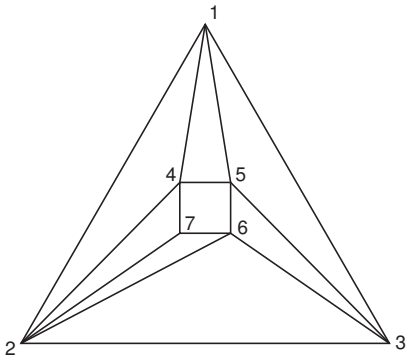

Figure 3: A graph with seven vertices

Figure 4a shows all the regions that contain our example vertex 6. The first step of this algorithm is to rotate each region so vertex 6 is the first vertex listed

2-6-3	6-3-2	3-2	3-2
2-7-6	6-2-7	2-7	2-7
3-6-5	6-5-3	5-3	7-4-5
4-5-6-7	6-7-4-5	7-4-5	5-3
(a)	(b)	(c)	(d)

Figure 4: The regions (from Figure 3 arranged)

in the region. Figure 4b shows the list of manipulated regions. The next step is to eliminate vertex 6 from each region. The resulting region list is shown in Figure 4c. The final step of this algorithm is to reorder the list of regions. The first region is chosen arbitrarily. The second region is chosen such that its first vertex is the same as the first region's last vertex. This procedure is continued using the second and third regions, the third and fourth regions, and so on until all regions have been ordered. One should notice that the list of regions is cyclical, being that the first vertex of the first region is the same as the last vertex of the last region. The region list generated by this algorithm for vertex 6 is shown in Figure 4d.

4 Results

Thus far, the authors have been given region-based data for solutions of all complete graphs from K_5 to K_{17}. These have been successfully visualized using the algorithm and tool (discussed below). Prior to discussing the tool used to perform visualization, it may be beneficial to view several examples. The following examples are of the completed visualizations for a K_{15} graph. The progression shows the level of detail as one zooms-in on the graph layers. Figure 5 shows the entire graph for K_{15} and the graph can be zoomed in on to show increasing levels of detail for the inner layers.

The results for K_{15}, shown in Figure 5 is demonstrative of the types of graphs generated by the algorithm and tool. One can clearly notice the layering and how useful it is in keeping the graph organized.

The Tool: Not all aspects of VCS-based graph drawing are currently automated. As such, we have developed an interactive tool that allows one to perform the non-automated portions of the graph drawing process. From a region list, VCS is able to give the vertices associated with each layer and the orientations of the vertices on each layer. However, exact positioning within the layers is not yet automatically determinable. CircleGraph is a program that allows one to visualize and manipulate graphs in real-time. The most important operations are that users can

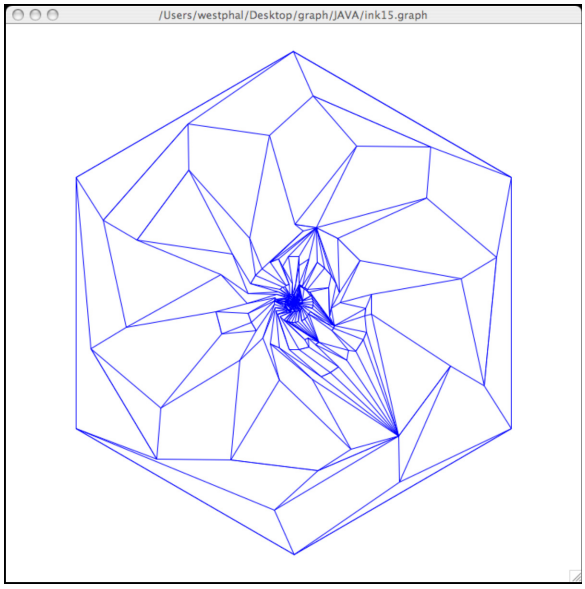

Figure 5: Visualized K_{15} (all layers)

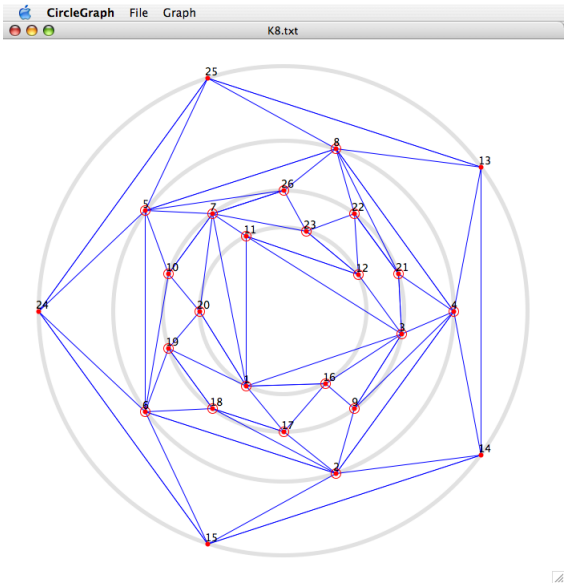

Figure 6: A screenshot of the CircleGraph program

resize and rotate layers and slide vertices within a layer. Using only these two operations one can correctly determine absolute positions of vertices for graphs. Still, additional features allow the tool to be both more powerful and more quickly used.

The features of CircleGraph revolve around visualizing graphs for the purposes of presenting work in research documentation. As such, the list of features is small but all are critical. Figure 6 is a screenshot of the CircleGraph program. Figure 7 shows a complete list of supported operations and features.

Besides being able to visualize and manipulate graphs, for the purposes of research it is essential to support output formats such as JPEG and X-Y coordinate lists for scientific verification. CircleGraph also supports options for zero or one based vertex indices, enabling/disabling anti-aliasing, enabling/disabling auto-zoom, and allowing for curved lines or strictly drawing rectilinearly. In addition, the CircleGraph program is implemented using Java and as such is platform independent. This allows for maximum flexibility in heterogeneous research environments.

Strategies: Because not every aspect of the algorithm is automated, we have developed the following strategies to assist you in developing aesthetically pleasing graphs using the CircleGraph interactive tool.

(1) Use "Auto-Rotate Layers" before manually adjusting any layers or vertices, but some additional layer rotation may help cleanup the graph.

(2) Expand each layer as much as possible (while looking out for crossing edges), but keep in mind that it is sometimes necessary to shrink a layer.

(3) Try to move vertices as little as possible, to help maintain uniformity.

(4) Expand virtual layers as much as possible and place vertices as closely as possible to connecting vertices in surrounding layer.

(5) When two adjacent vertices are not next to each other in layer orientation, move the vertices and their neighbors to get them as close to $\frac{\pi}{2}$ radians apart as possible.

(6) When it appears that one vertex on the same layer as another vertex must be "underneath" an edge from that other vertex, making the layer bigger will help.

(7) Use the highlight crossing edges tool to verify your graphs (this does not currently work when allowing curved lines).

5 Future Work and Conclusions

Though our tool shows that one can display well-organized, large graphs, some experiments have been done to further clean up the results. Virtual layers, layers that exist between layers, can be used in certain circumstances to reduce the number of vertices per layer, and to reduce the interactions between layers. Another direction of research currently underway is implementing a better method for removing duplicate vertices in layers. As mentioned at the end of section 3,

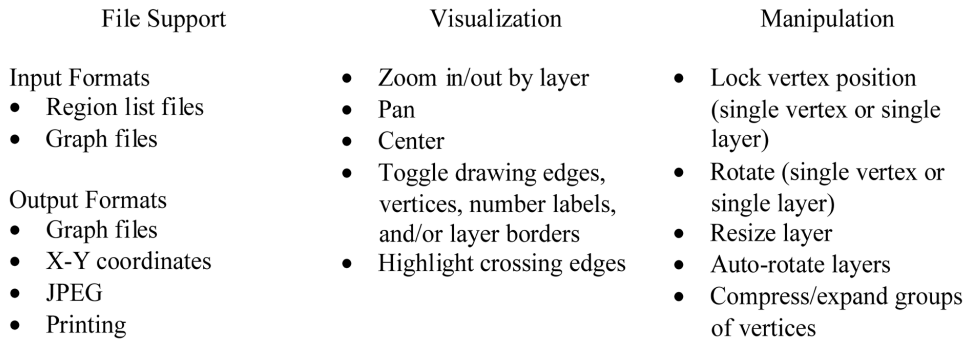

Figure 7: Operations and features of Circlegraph

the results of our algorithm may include vertices multiple times in a single region. The current method simply removes duplicate instances of vertices after the first instance. This may not be ideal as a few crossings, which require human attention or interaction to fix, are generated in some graphs, due to this issue.

The most important work that needs to be done is to run the algorithm to visualize graphs of higher orders. Visualizing graphs on the order of ten thousand vertices or more should demonstrate the scalability of this algorithm. To perform such visualizations however, more data needs to be collected from region based graph algorithms.

References

[1] J. Branke, F. Bucher, and H. Schmeck. Using genetic algorithms for drawing undirected graphs. Technical Report 347, D-76128 Karlsruhe, Germany, 1996.

[2] G. Chartrand and L. Lesniak. *Graphs and Digraphs*. Chapman & Hall/CRC, Boca Raton, FL, 3rd. edition, 1996.

[3] N. Chiba, K. Onoguchi, and T. Nishizeki. Drawing planar graphs nicely. *Acta Inform.*, **22**:187–201, 1985.

[4] M. Chrobak and T. Payne. A linear-time algorithm for drawing planar graphs. *Information Processing Letters*, **54**:241–246, 1995.

[5] J. Edmonds. A combinatorial representation for polyhedral surfaces. *Notices Amer. Math. Soc.*, **7**:646, 1960.

[6] T. Eloranta and E. Makinen. Timga - a genetic algorithm for drawing undirected graphs, 1996.

[7] H. de Fraysseix, J. Pach, and R. Pollack. Small sets supporting fary embeddings of planar graphs. In *In Proc.* 20^{th} *Annu. ACM Sympos. Theory Comput.*, pages 426–433, 1988.

[8] J. R. Fredrickson, B. Yuan, and F. C. Harris, Jr. A time-saving region restriction for calculating the crossing number of k_n. *Submitted*, May 2004.

[9] M. Himsolt. Graphed: A graphical platform for the implementation of graph algorithms. *Lecture Notes in Computer Science*, pages 182–193, 1995.

[10] D. E. Knuth. Computer drawn flowcharts. *Commun. ACM*, 6, 1963.

[11] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Algorithms*. New York: ACM Press, 1993.

[12] E. Makinen and M. Sieranta. Genetic algorithms for drawing bipartite graphs. Technical report, Dept. of Computer Science, University of Tampere, Finland, 1994. Report A-1994-1.

[13] S. Nåher. *LEDA Manual*. Max-Planck Institute für Informatik, Saarbrücken, 1993.

[14] F. Newberry Paulisch. The design of an extendible graph editor. *LNCS*, 704, 1993.

[15] R. Read. New methods for drawing a planar graph given the cyclic order of the edges at each vertex. *Congr. Number.*, **56**:31–44, 1987.

[16] W. T. Tutte. How to draw a graph. In *Proceedings of London Maths Society*, volume 3, 1963.

[17] M. Werner and M. Fröhlich. The interactive graph visualization system davince v1.2. Tech Report, Universität Bremen, 1993.