Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Discrete Algorithms 4 (2006) 25–50
JOURNAL OF DISCRETE ALGORITHMS
www.elsevier.com/locate/jda

# A framework and algorithms for circular drawings of graphs ☆

Janet M. Six [a,*], Ioannis G. Tollis [b,c]

[a] *Lone Star Interaction Design, PO Box 1993, Wylie, TX 75098, USA*
[b] *Department of Computer Science, University of Crete, GR 714 09 Heraklion, Greece*
[c] *Institute of Computer Science, Foundation for Research and Technology Hellas-FORTH, PO Box 1385, 71110 Heraklion, Crete, Greece*

Available online 5 February 2005

**Abstract**

In this paper, we present a framework and two linear time algorithms for obtaining circular drawings of graphs. The first technique produces circular drawings of biconnected graphs and finds a zero crossing circular drawing if one exists. The second technique finds multiple embedding circle drawings. Techniques for the reduction of edge crossings are also discussed. Results of experimental studies are included.


*Keywords:* Circular graph drawing; Graph drawing; Information visualization; Network visualization; Clustered views; Minimization of crossings; Experimental studies

## 1. Introduction

*Graphs* are used to represent many kinds of information structures: computer, telecommunication, social networks, entity-relationship diagrams, data flow charts, resource allo-

☆ Most of this research was performed at The University of Texas at Dallas and was supported in part by NIST Advanced Technology Program grant number 70NANB5H1162 and the Texas Advanced Research Program under grant number 009741-040 and a graduate assistant stipend from the Provost's Office at The University of Texas at Dallas.
* Corresponding author.
*E-mail addresses:* jsix@lonestarinteractiondesign.com (J.M. Six), tollis@ics.forth.gr (I.G. Tollis).

1570-8667/$ – see front matter 
doi:10.1016/j.jda.2005.01.009

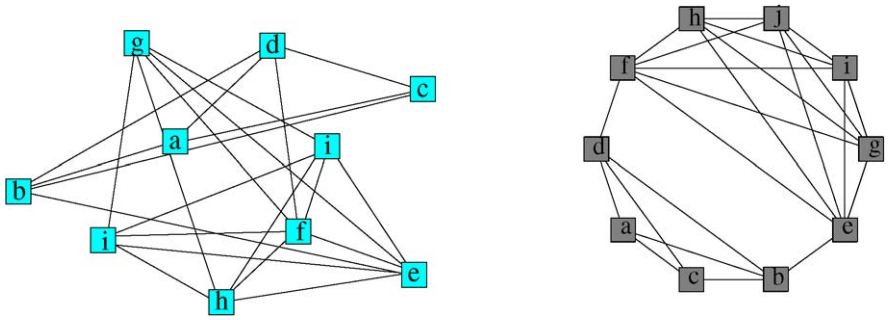


Fig. 1. A graph with arbitrary coordinates for the nodes and a circular drawing of the same graph as produced by an implementation of our algorithm.

cation maps, and much more. *Graph Visualization* is the study of techniques which produce drawings of graphs. These visualizations provide a snapshot of each graph and allow experts to be free from the work of organizing the nodes and edges and thereby allowing more time to interpret the composition of these structures. Much research has been done in the area of graph visualization: see [3,4] for discussion.

A *circular graph drawing* (see Fig. 1 for an example) is a visualization of a graph with the following characteristics:

- the graph is partitioned into clusters,
- the nodes of each cluster are placed onto the circumference of an embedding circle, and
- each edge is drawn as a straight line.

There are many applications which would be strengthened by an accompanying circular graph drawing. For example, our drawing techniques could be added to tools which manipulate telecommunication, computer, and social networks to show clustered views of those information structures. The partitioning of the graph into clusters can show structural information such as biconnectivity, or the clusters can highlight semantic qualities of the network such as sub-nets. Emphasizing natural group structures within the topology of the network is vital to pin-point strengths and weaknesses within that design. It is essential that the number of edge crossings within each cluster remain low in order to reduce the visual complexity of the resulting drawings. Researchers have produced several circular drawing techniques [2,6,9,10,18], some of which have been integrated into commercial tools. However, the resulting drawings are visually complex with respect to the number of crossings. In this paper, we introduce circular drawing techniques for simple graphs which are efficient and also produce drawings with a low number of edge crossings.

The remainder of this paper is organized as follows: Section 2 discusses previous work in this area. In Section 3, we present an $O(m)$ time algorithm for the circular layout of biconnected graphs. Our algorithm guarantees that if a zero crossing circular drawing exists for a biconnected graph, then it will find it. In Section 3.1, we discuss properties of circular drawings created by the technique in Section 3. In Section 4, we discuss an approach for reducing the number of edge crossings in circular drawings. In Section 5, we present an

$O(m)$ time algorithm for drawing nonbiconnected graphs on multiple embedding circles. In Section 6, we discuss implementation details and give results of an extensive experimental study over a set of 10,328 biconnected graphs. These results show our techniques to perform significantly better than the current technology. In Section 7, we present conclusions.

## 2. Previous work

### 2.1. Previous circular drawing techniques

Kar, Madden, and Gilbert present a circular drawing technique and tool in [9] for network management. Recognizing that a clustered view of a network can be quite helpful to its design and maintenance, the authors build a system which first partitions the network into clusters, places the clusters onto the main embedding circle, and then sets the coordinates of individual nodes. Finally a heuristic approach is used to minimize the number of crossings. As discussed in [6], an advanced version of this $O(n^2)$ technique has been implemented as part of Tom Sawyer Software's successful Graph Layout Toolkit (GLT).

Tollis and Xia introduced several linear time algorithms for the visualization of survivable telecommunication networks in [18]. Given the ring covers of a network, these algorithms create circular drawings such that the survivability of the network is very visible. Techniques were presented for outside (inside) drawings such that the rings are placed outside (inside) a root circle. An additional linear time algorithm produces drawings which are a combination of outside and inside drawings. This type of flexibility in a tool allows each network designer to choose the best technique given the exact application.

Citing a need for graph abstraction and reduction of today's large information structures, Brandenburg describes an approach to draw a path (or cycle) of cliques in [2]. This $O(n^3)$ algorithm creates a two-level abstraction of the given graph giving the ability to project a clique on each node of the abstracted graph.

Circular drawing techniques are not limited to telecommunication and computer network applications by any means. InFlow [10] is a tool to visualize human networks and produces diagrams and statistical summaries to pinpoint the strengths and weaknesses within an organization. The usually unvisualized characteristics of self-organization, emergent structures, knowledge exchange, and network dynamics can be seen in the drawings of InFlow. Resource bottlenecks, unexpected work flows, and gaps within the organization are clearly shown in these circular drawings.

In [14–17], we presented preliminary work on the algorithms in this paper.

### 2.2. Complexity of the circular graph drawing problem

Intuitively, the problem of creating circular graph drawings while minimizing the number of edge crossings seems very hard. The general problem of placing nodes such that the number of edge crossings is minimum is the well known NP-complete *crossing number* problem. However, the more restricted problem of finding a minimum crossing embedding such that all the nodes are placed onto the circumference of a circle and all edges are

represented with straight lines is also NP-complete as proven in [11]. The authors show the NP-completeness by giving a polynomial time transformation from the NP-complete *Modified Optimal Linear Arrangement* problem.

## 3. A technique for producing circular drawings of biconnected graphs

In order to produce circular drawings with fewer crossings than previous techniques, we have developed an algorithm which tends to place edges toward the outside of the embedding circle. Also, nodes are placed near their neighbors. In fact, this algorithm tries to maximize the number of edges appearing toward the periphery of the embedding circle. The algorithm achieves this improvement by selectively removing some edges and then building a depth first search (DFS) based node ordering of the resulting graph.

In order to selectively remove some edges, this technique visits the nodes in a wave-like fashion. Define a *wave front node* to be adjacent to the last node processed, see Fig. 2. A *wave center node* is adjacent to some other node which has already been processed. The algorithm starts at a lowest degree node and continues to visit wave front and wave center nodes if they are of lowest degree. If none of the current wave front or wave center nodes are of lowest degree, then some lowest degree node is chosen. The wave-like node traversal begins again from this newly chosen node and will continue from this node and the previous wave front and wave center nodes.

A *pair edge* is incident to two nodes which share at least one neighbor, see Fig. 3. Nodes $v$ and $w$ are said to be *paired* by $u$, and $u$ is said to *establish* the pair edge $(v, w)$. In other words, $u$, $v$, and $w$ form a triangle. Pair edges will be removed before the DFS step of the technique. A *triangulation* edge is a new pair edge which is placed into the graph by our technique. The triangulation edges are also removed from the graph before the DFS portion of the algorithm. Each time a node $u$ is visited, a list of pair edges is built. If there is an insufficient number of pair edges in the graph, our algorithm automatically inserts triangulation edges into the graph. With the ensuing removal of $u$, that node is inherently represented by the newly found pair edges, see Fig. 4. The illustrations marked (a) show a

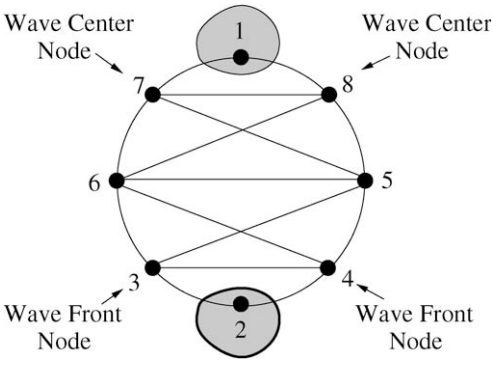


Fig. 2. Examples of wave front and wave center nodes. The shaded region includes those nodes which have already been processed. The node labeled 2 is the most recently processed.

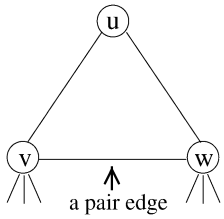


Fig. 3. Example of a pair edge.

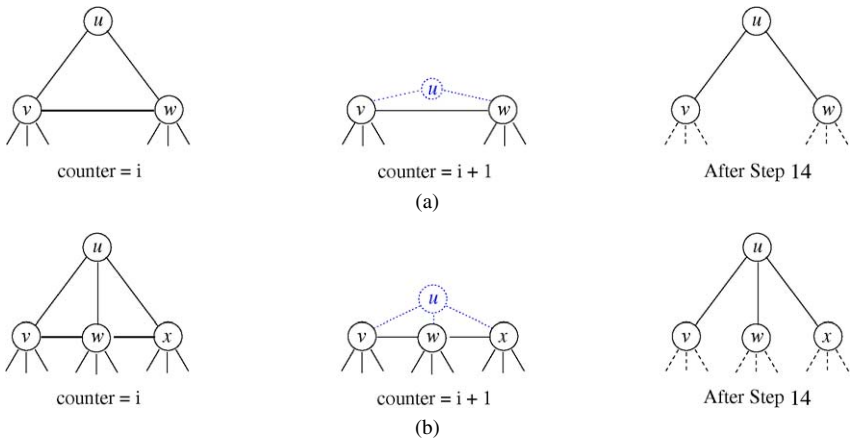


Fig. 4. The node and edge absorption qualities of Algorithm 1.

degree two node $u$ and its neighbors $v$ and $w$ at three different points in the algorithm. The pair edge established by $u$, $(v, w)$, is shown with a bold line in the first illustration. The illustration immediately to the right shows the same graph fragment when the next node is processed. Although node $u$ and edges $(u, v)$ and $(u, w)$ are not in the graph anymore, they are inherently represented by the edge $(v, w)$. The next illustration to the right shows the same graph fragment after the pair edge $(v, w)$ has been removed. At this point, the pair edge $(v, w)$ is inherently represented by the node $u$ and the edges $(u, v)$ and $(u, w)$. A similar example is shown in the illustrations labeled (b), where the current node being processed has degree three. It is this selective absorption that causes the behavior of edge placement towards the periphery of the embedding circle.

It is important to note that we do not find all pair edges. For each node $u$ we visit its neighbors $v_1, v_2, \ldots, v_k$ in some order, say the order in which they appear in the adjacency list. For example, we check to see if $(v_1, v_2)$ exists: if so, we add that edge to the removal list. If not, we add the triangulation edge $(v_1, v_2)$ to the graph and to the removal list. This part of the algorithm takes $deg(u)$ time, which is a significant improvement over the algorithm described in [15]. Notice that a new edge is added only between two nodes that are consecutive in the adjacency list of the current node (and of course if such an edge does not already exist). Also note that the first and the last neighbors visited can not experience an increase in degree. For each of those nodes, the edge incident to $u$ is removed while at

most one triangulation edge is added. Next, we show that the total number of triangulation edges added is O($m$).

The number of triangulation edges added to $G$ over the course of the algorithm is at most $\sum_{i=1}^{n-3} minDeg_i - 1$, where $minDeg_i$ is the minimum degree found in $G$ at the $i$th iteration of the While loop. We postulate that $minDeg_i \leqslant avgDeg$ before the $i$th iteration, $\forall i \geqslant 1$ and where $avgDeg$ is the average degree of the nodes in the original graph $G$.

**Lemma 3.1.** *$minDeg_i \leqslant avgDeg$ before the $i$th iteration, $\forall i \geqslant 1$.*

**Proof.** (By induction.) *Base* (for $i = 1$): Clearly true.

*Inductive hypothesis*: Assume that $minDeg_i \leqslant avgDeg$ before the $i$th iteration, $\forall i \leqslant k$.

*Inductive step*: Prove $minDeg_{k+1} \leqslant avgDeg$ before the $(k+1)$st iteration.

Let $v_{k+1}$ be the vertex that has $minDeg_{i+1}$ (and will be chosen at the $(k+1)$st iteration). Let vertex $v_k$ be the vertex chosen during the $k$th iteration (i.e., had $minDeg_i$). There are two cases:

(1) $v_{k+1}$ is not a neighbor of $v_k$. In this case its degree has not increased during the $k$th iteration. Hence the Inductive hypothesis guarantees that the degree of $v_{k+1} \leqslant avgDeg$.
(2) $v_{k+1}$ is a neighbor of $v_k$. In this case its degree may have increased during the $k$th iteration. However, there are two nodes (the first and last nodes in the chosen order) whose degree has not increased since we removed one edge and added to it at most one edge during the removal of $v_k$. We can choose $v_{k+1}$ to be one of those two nodes or another neighbor if it has lower degree. Hence the Inductive hypothesis guarantees that the degree of $v_{k+1} \leqslant avgDeg$. □

It is important to note that the visit of the neighbors starts from the lowest degree neighbor and proceeds cyclically around the adjacency list. Since we know that $minDeg_i \leqslant avgDeg$ before the $i$th iteration, $\forall i \geqslant 1$, we also know that

$$\sum_{i=1}^{n-3} minDeg_i - 1 < \sum_{i=1}^{n} minDeg_i \leqslant \sum_{i=1}^{n} avgDeg = 2m.$$

Therefore, the number of triangulation edges added is O($m$).

Subsequent to the edge removal, our algorithm proceeds to build an ordering of the nodes for the reduced graph. A traditional DFS is performed and then the nodes in a longest path of the DFS tree are placed around the embedding circle. Alternatively, a heuristic algorithm for finding a longest path in a graph can be used. Finally, the remaining nodes are nicely merged into the ordering. This can be accomplished by visiting each neighbor of $u$ and asking it if it is next to another neighbor of $u$ on the embedding circle. If two neighbors of $u$ are next to each other on the embedding circle, then we place $u$ between those two neighbors. (If there are multiple pairs of such neighbors, we arbitrarily pick one of those pairs). If there are not two neighbors of $u$ next to each other on the embedding circle, then we place $u$ next to some neighbor or $u$ or, if there are no neighbors of $u$ on the embedding circle yet, we pick an arbitrary position for $u$.

**Algorithm 1.** *CIRCULAR*.
**Input**: A biconnected graph, $G = (V, E)$.
**Output**: A circular drawing $\Gamma$ of $G$ such that each node in $V$ lies on the periphery of a single embedding circle.

(1) Bucket sort the nodes by ascending degree into a table $T$.
(2) Set *counter* to 1.
(3) While *counter* $\leqslant n - 3$
(4) If a wave front node $u$ has lowest degree then *currentNode* $= u$.
(5) Else If a wave center node $v$ has lowest degree then *currentNode* $= v$.
(6) Else set *currentNode* to be some node with lowest degree.
(7) Visit the adjacent nodes consecutively. For each two nodes,
(8) If a pair edge exists place the edge into *removalList*.
(9) Else place a triangulation edge between the current pair of neighbors and also into *removalList*.
(10) Update the location of *currentNode*'s neighbors in $T$.
(11) Remove *currentNode* and incident edges from $G$.
(12) Increment *counter* by 1.
(13) Restore $G$ to its original topology.
(14) Remove the edges in *removalList* from $G$.
(15) Perform a DFS (or a longest path heuristic) on $G$.
(16) Place the resulting longest path onto the embedding circle.
(17) If there are any nodes which have not been placed then place the remaining nodes into the embedding order with the following priority:
(i) between two neighbors, (ii) next to one neighbor, (iii) next to zero neighbors.

The time complexity of Algorithm 1 is $O(m)$, where $m$ is the number of edges in $G$. Step 1 takes $O(m)$ time. Step 3 takes $O(m)$ time over all iterations since the use of efficient data structures (as explained in Section 6.2) allows each iteration to take only $O(deg(v_i))$ time, where $v_i$ is the vertex chosen during the $i$th iteration. Notice that the number of triangulation edges added by step 9 is $O(m)$, as shown in Lemma 3.1. As discussed earlier, this is a significant improvement over our previous algorithm described in [15]. Clearly, steps 13–16 require $O(m)$ time. Finally, step 17 also requires $O(m)$ time since at most $\sum_{i=1}^{n} deg(v_i) = O(m)$ possible placements are reviewed.

### *3.1. Properties of Algorithm 1*

A graph, $G$, is *outerplanar* if and only if $G$ can be drawn on the plane such that all nodes lie on the boundary of a single face and no two edges cross. If the biconnected graph given to Algorithm 1 is outerplanar then the result will be a circular visualization such that no two edges cross. In fact, our technique has been inspired by the algorithm for recognizing outerplanar graphs presented in [12].

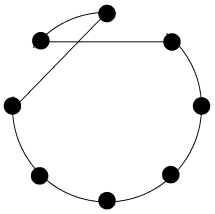

Fig. 5. A non-planar drawing of a simple cycle.

By the definition of outerplanar graphs, we know that there exists a plane circular drawing for any outerplanar graph. Also, by that same definition we know that a graph which is not outerplanar does not admit a plane circular drawing. In fact, the set of biconnected graphs which may be drawn in a circular fashion without any crossings is exactly the set of biconnected outerplanar graphs. The requirement of placing all nodes on the periphery of some embedding circle is equivalent to placing all nodes on a single face (say the external face) of some embedding. Furthermore, if a zero-crossing visualization exists for a biconnected graph, $G$, then that drawing can be found by Algorithm 1.

**Theorem 3.1.** *There exists only one clockwise ordering of the nodes in a biconnected outerplanar graph G such that the drawing of G with the nodes in that order around the embedding circle is plane.*

**Proof.** ($\Rightarrow$) By the definition of an outerplanar graph, any outerplanar graph must admit a circular drawing such that no two edges cross.

($\Leftarrow$) Given the simplest outerplanar biconnected graph, a simple cycle, we can not change the relative order of any two nodes without creating crossings. See Fig. 5. The edges of the simple cycle are also known as *outerface* edges. We know that for any biconnected outerplanar graph, all the outerface edges must exist since without them the graph would not be biconnected. By this fact, we can immediately extend the above observation to all outerplanar graphs. □

**Theorem 3.2.** [12] *A graph G with n nodes is outerplanar if and only if either G is a triangle or*

(1) *G has at most* $2n - 3$ *edges,*
(2) *G has at least two degree two nodes,*
(3) *no edge of G lies on more than two triangles, and*
(4) *for any degree two node u which is adjacent to nodes v and w, the graph G minus node u plus the edge* $(v, w)$ *(if not already in G) is also outerplanar.*

**Lemma 3.2.** *Let G be a biconnected outerplanar graph with at least four nodes. Algorithm 1 places an edge* $(v, w)$ *into the removal list if and only if it is an edge between two internal faces.*

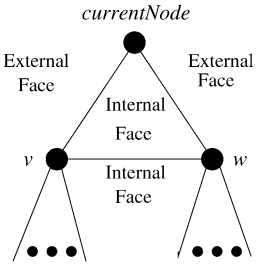


Fig. 6. Illustration for the proof of Lemma 3.2.

**Proof.** (⇒) We know the following facts at any execution of step 4 in Algorithm 1:

(1) $G$ is biconnected,
(2) *currentNode* has degree two, and
(3) the two nodes adjacent to *currentNode*, $v$ and $w$, must have degree at least three.

The first two attributes are included in Theorem 3.2. Proof by contradiction will prove the third. See Fig. 6. Without loss of generality, assume that node $v$ has degree one. The lone neighbor, *currentNode*, of $v$ must be an articulation point and this is a contradiction of attribute 1. Again, without loss of generality, assume that node $v$ has degree two. In this case $w$ must be an articulation point and we have another contradiction of attribute 1. Therefore the edge to be placed into *removalList* must be an edge between two internal faces.

(⇐) We have two categories of edges: *External/Internal* and *Internal/Internal*. External/Internal edges are those which appear between the external and some internal face. These are also known as outerface edges. Internal/Internal edges are those which appear between two internal faces. From Theorem 3.1, we know that there is only one clockwise circular ordering of the nodes such that the resulting layout is plane. And with this unique ordering, each edge can belong to only one of the two edge categories described above. Since there is only one clockwise circular ordering of the nodes such that the drawing is plane and that order is found by DFS on the reduced graph, we know that the reduced graph must consist of edges which exactly form a Hamiltonian cycle for $G$.

We prove by contradiction that all Internal/Internal edges are placed into the removal list. Assume that edge $(v, w)$ is an Internal/Internal edge that is not placed into the removal list. From the first part of this proof, we know that no External/Internal edges have been placed into the removal list, so when we remove the removal list edges from $G$, the reduced graph will contain all the External/Internal edges plus $(v, w)$, which is an Internal/Internal edge. When DFS is applied to this reduced graph, there will be one of two possible node orderings found: one will contain $(v, w)$ and one will not. Since more than one node ordering exists, we have a contradiction to Theorem 3.1. □

**Theorem 3.3.** *Algorithm* 1 *produces a plane circular drawing of any outerplanar graph in* $O(n)$ *time.*

**Proof.** By Lemma 3.2 we know that Algorithm 1 places all the Internal/Internal edges into *removalList*. This means that at the completion of step 14, the edges remaining in $G$ form a Hamiltonian cycle. Therefore the longest path of the DFS tree found in step 14 must contain all the nodes of the graph. Furthermore, the order formed by this longest path must be the unique order which produces a plane embedding. Therefore, the layout found by Algorithm 1, given an outerplanar graph, must contain zero crossings. By Theorem 3.2, we know that the number of edges in an outerplanar graph is at most $(2n - 3)$ so the time required for Algorithm 1 given an outerplanar graph is $O(n)$. □

Therefore we have:

**Theorem 3.4.** *Given a biconnected graph $G$, if $G$ admits a circular layout with zero crossings, then Algorithm 1 produces a circular drawing with zero crossings in $O(n)$ time.*

## 4. Further reducing the number of edge crossings

As will be shown in the experimental results of Section 6.1, Algorithm 1 produces drawings with a low number of edge crossings and significantly outperforms the current leading technology. We can further reduce the number of edge crossings with the technique presented in this section. As discussed in Section 2.2, the problem of minimizing the number of edge crossings in a circular graph drawing is NP-complete. The configuration of the nodes as determined by Algorithm 1 produces drawings with a low number of crossings, which can then be further reduced to some local minima with a monotonic crossing reduction technique. The postprocessing step visits each node and queries whether crossings can be reduced by the movement of that node next to one of its neighbors.

**Algorithm 2.** *CIRCULAR—Postprocessing*.
**Input**: A drawing $\Gamma$ of biconnected graph $G = (V, E)$ produced by Algorithm 1.
**Output**: A drawing $\Gamma'$ of $G$ with fewer or equal number of crossings.

(1) *currentCrossings* = current number of crossings in the drawing.
(2) For a fixed number of times
(3) For each node, $u$, in $G$
(4) Initialize $List_1$ to contain the embedding circle positions which lie between two nodes adjacent to $u$.
(5) If $List_1$ is empty
(a) Initialize $List_2$ to contain the embedding circle positions which lie next to one neighbor of $u$.
(b) *PositionList* = $List_2$.
(6) Else *PositionList* = $List_1$.
(7) For each location in *PositionList*
(8) Place $u$ at this location
(9) *newCrossings* = the new number of crossings.

(10) If $newCrossings < currentCrossings$ then
$currentCrossings = newCrossings$.
(11) Else Place $u$ back into its previous position.
(12) If no improvement was made during this iteration, stop.

The time complexity of Algorithm 2 is $O(m^2)$. This order is dominated by the required time for counting the number of crossings (steps 1 and 9). It is vitally important to the time efficiency of Algorithm 2 that the number of crossings be counted in an efficient fashion. As will be shown in Lemma 4.1, step 1 of Algorithm 2 requires $O(m + \chi)$ time to find the total number of crossings, where $m$ is the number of edges and $\chi$ is the number of crossings. The experimental study presented in Section 6 has shown that the loop of step 2 needs to be iterated at most 9 times. In fact, the vast majority of drawings converged within the first two iterations. In the worst case, step 2 requires a constant amount of time. Steps 3 and 6 require $O(n)$ time. Steps 4 and 5 require $O(m)$ time since we explore $\sum_{i=1}^{n} degree(i) = O(m)$ positions. Steps 7 and 8 require $O(m)$ time since we know there will be at most $\sum_{i=1}^{n} degree(i) = O(m)$ positions. In Section 4.2, we will prove that it takes $O(m)$ to find the new number of crossings in step 9. And since over the course of the algorithm, step 9 is repeated $O(m)$ times step 9 requires $O(m^2)$ time. Steps 10 and 11 require $O(m)$ time. So the time complexity of the entire algorithm is $O(m^2 + \chi)$. Since, each edge can cross any other edge in the drawing at most once in a circular visualization, $\chi$ is $O(\Sigma_{i=1}^{m} i)$ which is $O(m^2)$. Therefore, Algorithm 2 has time complexity $O(m^2)$.

### 4.1. Counting all the crossings in a circular drawing

Consider the straight edges $e_i$ and $e_j$ of Fig. 7. The edge $e_i$ can cross $e_j$ if and only if one endpoint $v$ of $e_j$ appears between the two endpoints $u$ and $w$ of $e_i$. In this case, $e_j$ is called an *open edge with respect to the arc uvw*. If both endpoints of $e_j$ appear between $u$ and $w$ on the perimeter of the embedding circle, then $e_i$ and $e_j$ do not cross. So, if we order the edges as they are encountered around the embedding circle and visit their endpoints in that order, we can determine the total number of edge crossings by counting the number of open edges. Although our problem is one dimensional, this technique has some similarities to the line segment intersection algorithm presented in [13].

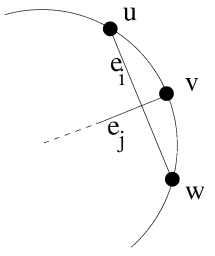


Fig. 7. An open edge with respect to the arc $uvw$.

**Algorithm 3.** *CountAllCrossings*.
**Input**: A single circle drawing $\Gamma$ of a biconnected graph $G = (V, E)$.
**Output**: The number of edge crossings in $\Gamma$.

(1) Order the edges as they are encountered around the circle in a clockwise order.
(2) *numberOfCrossings* $= 0$.
(3) For each edge endpoint, $p_i$, of edge $e_i$, do
(4) If $p_i$ is the first endpoint of edge $e_i$ append $e_i$ to *openEdgeList*.
(5) Else
(a) Increase *numberOfCrossings* by the number of open edges with respect to the arc $p_g p_h p_i$, where $p_g$ and $p_i$ are the endpoints of $e_i$ and $p_h$ is some endpoint which was visited after $p_g$ and before $p_i$.
(b) Remove $e_i$ from *openEdgeList*.

Algorithm 3 requires $O(m + \chi)$ time. Step 1 takes $O(m)$ time. This step can be accomplished in $O(m)$ time by visiting the incident edges of each node as they appear around the embedding circle. Steps 3, 4, and 5(b) require $O(m)$ time. Step 5(a) requires $\sum_{i=1}^{2m} \chi_i = O(\chi)$ time, where $\chi_i$ is the number of edge crossings caused by the edge $e_i$ and $\chi$ is the total number of edge crossings in the embedding. We accomplish this time requirement by traversing *openEdgeList* backwards from the end of the list to the element which contains $e_i$. Therefore, we have the following:

**Lemma 4.1.** *Algorithm CountAllCrossings counts the total number of edge crossings in a single circle embedding, where m is the number of edges and* $\chi$ *is the number of crossings in* $O(m + \chi)$ *time.*

### *4.2. Determining the new number of crossings after moving a node*

Since we can determine the overall number of crossings at the beginning of the algorithm and then move one node at a time, it is necessary to count only the number of crossings caused by the incident edges of the current node, $v$, to update the number of crossings in the drawing. During each iteration of the crossing reduction, the number of crossings in the entire drawing is equal to the following formula:

$$\textit{New Number of Crossings} = \textit{Old Number of Crossings} - \chi_v + \chi'_v$$

where, $\chi_v$ = Number of crossings caused by $v$ in the old location, and $\chi'_v$ = Number of crossings caused by $v$ in the new location.

Because we already know the old number of crossings, finding the new number of crossings is dominated by the time to find $\chi_v$ and $\chi'_v$. Any change in the edge crossings will occur between edges incident to $v$ and edges that have exactly one endpoint in the arc between the old and new positions of $v$. These *pertinent edges* are visited in order from the old towards the new position of $v$. A counter, *ctr*, holds the number of open edges in the arc (not including the open edges incident to $v$). Each time that an endpoint of an edge incident to $v$ is encountered, the number of crossings is increased by the value in *ctr*. At

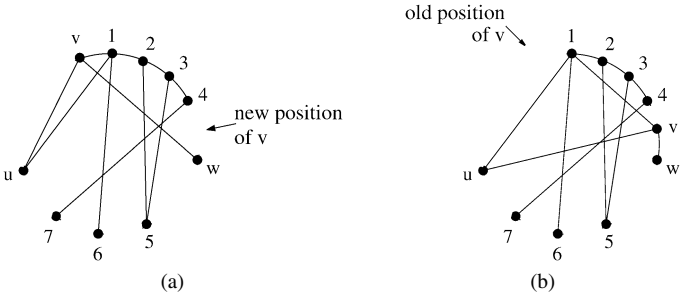


Fig. 8. The arc created by moving node $v$ to the position denoted with the arrow. The pertinent edges of the arc are shown.

the conclusion of this process, the number of crossings caused by $v$ in the old position is known. The number of crossings caused by $v$ in its new position is found by repeating this process from the new towards the old position of $v$ after moving $v$ to its new position.

See Fig. 8.

Therefore we have:

**Lemma 4.2.** *An* $O(m)$ *time algorithm exists to count the number of edge crossings gained or lost by moving a node* $v$ *within a single circle embedding.*

**Algorithm 4.** *CountSingleNodeCrossings*.
**Input**: A single circle drawing of a graph $G = (V, E)$,
a node $v \in V$, and
a new position $\alpha$ for $v$.
**Output**: The change in the number of edge crossings caused by moving $v$ to $\alpha$.

(1) $ctr = 0$.
(2) $numberOfCrossings = 0$.
(3) Order the pertinent edge endpoints as they are encountered around the embedding circle.
(4) Mark the pertinent edges as not seen.
(5) For each pertinent edge endpoint $p_i$ of edge $e_i$ do
(6) If $e_i$ is incident to $v$ increment the *numberOfCrossings* by *ctr*.
(7) Else If $e_i$ has been seen decrement *ctr* by 1.
(8) Else increment *ctr* by 1 and mark $e_i$ as seen.
(9) $OldNumberSingleNodeCrossings = numberOfCrossings$.
(10) $ctr = 0$.
(11) $numberOfCrossings = 0$.
(12) Move $v$ to its new position, $\alpha$.
(13) Mark the pertinent edges as not seen.
(14) Repeat steps 5–8 in the opposite direction.
(15) $NewNumberSingleNodeCrossings = numberOfCrossings$.
(16) $changeInCrossings = NewNumberSingleNodeCrossings - OldNumberSingleNodeCrossings$.

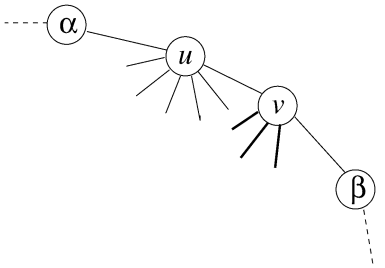


Fig. 9. The pertinent edges for Algorithm 4 if the two adjacent nodes $u$ and $v$ are being swapped.

Algorithm 4 requires $O(m)$ time. Steps 3–8 require $O(m)$ time since the number of pertinent edges is $O(m)$ as described above. Step 13 requires $O(m)$ time. Finally, step 14 requires $O(m)$ time since it is a repetition of steps 5–8.

If Algorithm 4 is swapping the placement of two nodes which are next to each other, $u$ and $v$, on the embedding circle, then Algorithm 4 only takes $O(maxDegree)$ time, where *maxDegree* is the maximum degree of all nodes in $V$. This is because the number of pertinent edges is the smaller degree of $u$ and $v$. See Fig. 9. Since a swap of these two nodes can be accomplished by moving $u$ between $v$ and $\beta$ or moving $v$ between $\alpha$ and $u$, we choose the movement such that the number of pertinent edges (i.e., the degree of the node which is not moved) is smaller. Both of the movements produce the same node ordering, so we perform the movement which requires less time. In the specific case of Fig. 9, we choose to move node $u$.

Given the time analysis of Algorithm 4, Lemmas 4.1 and 4.2, Algorithm 2 produces a visualization with a reduced number of edge crossings in $O(m^2)$ time.

## 5. Circular drawings of nonbiconnected graphs on multiple embedding circles

Most networks are not biconnected. Therefore it is important for a circular drawing tool to provide a component which visualizes nonbiconnected graphs. In [16], we present $O(m)$ time algorithms which produce circular drawings of trees and other nonbiconnected networks on a single embedding circle. In this section, we will present a technique for producing circular drawings of graphs on multiple embedding circles. Given a nonbiconnected graph $G$ we can decompose the structure into biconnected components in $O(m)$ time. Taking advantage of this inherent structure, we first layout the biconnected components of the block-cutpoint tree with a radial layout technique similar to [1,7,8], then we layout each biconnected component of the network with a variant of Algorithm 1. See Fig. 10.

Our algorithm addresses several issues in order to produce good quality circular drawings: (1) which biconnected component is considered to be the root of the block-cutpoint tree, (2) articulation points can appear in multiple biconnected components of the block-cutpoint tree and need to be assigned to a unique biconnected component, (3) the nodes of the block-cutpoint tree can represent biconnected components of differing size, and (4) the nodes of each biconnected component should be visualized such that the articulation

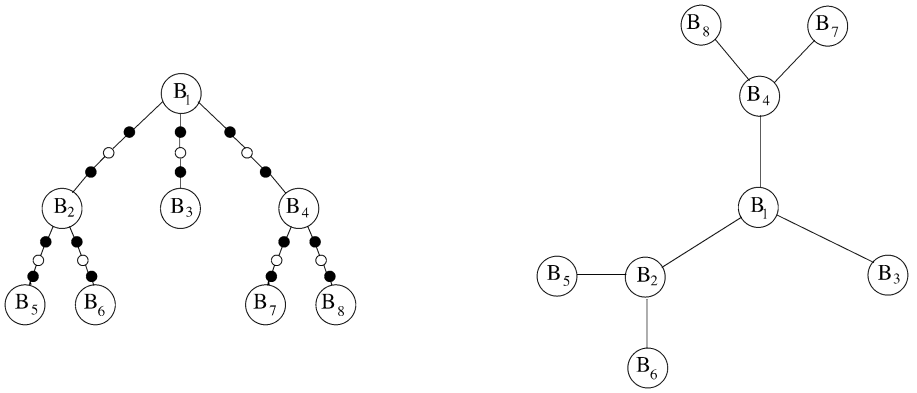


Fig. 10. The illustration on the left shows the block-cutpoint tree of a non-biconnected graph. The small black tree nodes represent articulation points and the small white tree nodes represent bridges. The right illustration is a drawing of the same graph where the block-cutpoint tree is laid out with a radial tree layout technique.

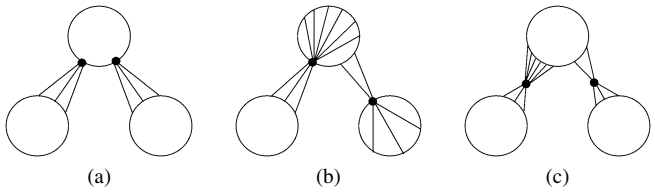


Fig. 11. Examples of three approaches for the assignment of strict articulation points to biconnected components. The black nodes are strict articulation points.

points appear in good positions and also there is a low number of edge crossings. We will address each of these issues in turn.

In order to address the first issue, we can choose the root with a recursive leaf-pruning algorithm to find the "center" of the tree [4]. Alternatively, we can pick the root dependent on some important metric: e.g., size of the biconnected component. Next we address the second issue. Define a *strict articulation point* as an articulation point which is not adjacent to a bridge. Strict articulation points are duplicated in more than one biconnected component of the block-cutpoint tree, but of course each node should appear only once in a drawing of that graph. Therefore, we offer three approaches in which each articulation point will appear only once in the drawing. The first approach assigns each strict articulation point, $u$, to the biconnected component which contains $u$ and is also closest to the root in the block-cutpoint tree. This biconnected component is the parent of the other biconnected components which contain $u$. See Fig. 11(a). The second approach assigns the articulation point to the biconnected component which contains the most neighbors of that articulation point, see Fig. 11(b). The third approach assigns the articulation point to a position between its biconnected components, see Fig. 11(c). Placing a node in this manner will highlight the fact that this node is an important articulation point. Following the assignment step, the duplicates of a strict articulation point are removed from the blocks in the block-cutpoint tree. We refer to the nodes adjacent to a removed strict articulation point

in a biconnected component as *inter-block nodes*. In order to maintain biconnectivity for the method which will layout this component, a thread of edges is run through the inter-block nodes. These edges will be removed from the graph after the layout of the cluster is determined.

The third issue to be addressed is that while performing the layout of the block-cutpoint tree we must consider that the biconnected components may be of differing sizes. The node sizes are proportional to the number of nodes contained in the current block. The radial layout algorithms presented in [1,7,8] place the root at $(0, 0)$ and the subtrees on concentric circles around the origin. These algorithms require linear time and produce plane drawings. However, unlike our block-cutpoint trees, the nodes of the trees laid out with [1,7,8] are all the same size. The technique in [19] handles graphs with different node sizes, however node overlap is allowed. In order to produce radial drawings of trees with differing node sizes, we present a modification of the classical radial layout technique [1, 7,8].

*RADIAL—with different node sizes*: For each node, we must assign a $\rho$ coordinate, which is the distance from point $(0, 0)$ to the placement of that node and a $\theta$ coordinate which is the angle between the line from $(0, 0)$ to $(\infty, 0)$ and the line from $(0, 0)$ to the placement of that node. The $\rho$ coordinate of node $v$, $\rho(v)$, is defined to be $\rho(u) + \delta + \frac{d_u}{2} + \frac{\max(d_1, d_2, \ldots, d_k)}{2}$, where $\rho(u)$ is the $\rho$ coordinate of the parent $u$ of $v$, $\delta$ is the minimum distance allowed between two nodes, $d_u$ is the diameter of $u$, and $\max(d_1, d_2, \ldots, d_k)$ is the maximum of the diameters of all the children of $u$. It is important to note that while all descendants of a node $i$ are placed on the same concentric circle, not all nodes in the same level of the block-cutpoint tree are placed on the same concentric circle.

In order to prevent edge crossings, each subtree must be placed inside an annulus wedge, and the width of each wedge must be restricted such that it does not overlap a wedge of any other subtree. The $\theta$ coordinate of node $v$ depends on the widths of the descendants of $v$, not just the number of leaves as in [1,7,8]. This assignment of coordinates leads to a layout of the form shown in Fig. 12.

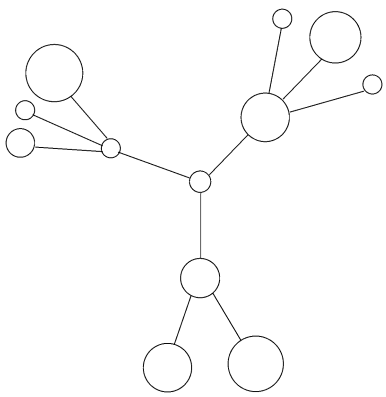

Fig. 12. A radial layout of a tree with differing size nodes.

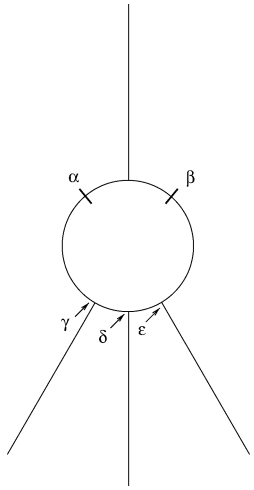


Fig. 13. The relation between the layout of the block-cutpoint tree and the layout of an individual biconnected component.

The fourth issue to be addressed by our circular drawing technique is the visualization of each component. After performing *RADIAL—with Different Node Sizes* we have a layout of the block-cutpoint tree, and need to visualize the nodes and edges of each biconnected component. The radial layout of the block-cutpoint tree should be considered while drawing each biconnected component. See Fig. 13. Define *ancestor nodes* to be adjacent to nodes in the parent biconnected component in the block-cutpoint tree. Likewise, define *descendant nodes* to be adjacent to nodes in child biconnected components. In order to reduce the number of crossings caused by inter-biconnected component edges, our technique tries to place ancestor nodes in the arc between the points $\alpha$ and $\beta$. The size of the arc from $\alpha$ to $\beta$ is dependent on the distance between the placement of a biconnected component to the placement of its parent in the radial layout of the block-cutpoint tree. Descendant nodes are placed uniformly in the bottom half of the biconnected component layout. For example, if there are three descendant nodes, they would be placed at points $\gamma$, $\delta$, and $\varepsilon$ in Fig. 13. These special positions for the ancestor and descendant nodes are called *ideal positions*. Due to a high number of ancestor and descendant nodes, it may not be possible to place all ancestor and descendant nodes in an ideal position, however the algorithm places as many as possible in ideal positions.

Placing the ancestor and descendant nodes in this manner reduces the number of crossings caused by inter-biconnected component edges going through a biconnected component. In fact, the only times that these edges do cause crossings are when the number of ancestor (descendant) nodes in the biconnected component $B_i$ is more than about $n_i/2$, where $n_i$ is the number of nodes in $B_i$. In those cases, the set of ideal positions includes all the positions in the upper (respectively lower) half of the embedding circle and also positions in the lower (upper) half which are as close as possible to the upper (lower) half.

We present two algorithms for the layout of each biconnected component such that ancestor and descendant nodes are placed near their ideal positions. The first step of each

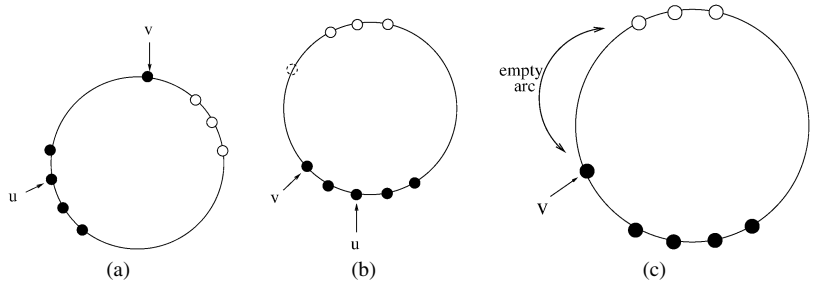


Fig. 14. This figure demonstrates Algorithms *LayoutCluster1* and *LayoutCluster2*. The black nodes are descendant nodes and the white nodes are ancestor nodes. (a) Drawing produced by Algorithm *CIRCULAR*; (b) the rotated drawing of part (a) produced by *Algorithm LayoutCluster1*; (c) the resulting drawing of part (a) produced by Algorithm *LayoutCluster2*.

technique is to perform Algorithm 1 on the current biconnected component, $B_i$. This requires $O(m_i)$ time, where $m_i$ is the number of edges in biconnected component $B_i$. Then we update this drawing so that the ancestor and descendant nodes appear near their ideal positions.

The first technique rotates the layout of the biconnected component as found by Algorithm 1 such that many ancestor and descendant nodes are placed close to their ideal positions. Then, the remaining ancestor and descendant nodes are moved to their closest ideal position. This algorithm requires $O(m_i)$ time. See Fig. 14(b) for an example.

**Algorithm 5.** *LayoutCluster1*.
**Input**: A biconnected component, $B_i$.
**Output**: An circular layout of $B_i$ such that the positions of the articulation points are placed well with respect to the ideal positions.

(1) Perform Algorithm 1 on $B_i$ and save the results in $\Gamma_1$.
(2) If the number of ancestor nodes in $B_i$ is less than the number of descendant nodes set the block type to be descendant, otherwise set the block type to be ancestor.
(3) Loop through the nodes of $B_i$ as they appear around the embedding circle in $\Gamma_1$ and for each node which is the same type as the block type, record the clockwise distance to the last node of that type.
(4) Find the nodes which have the smallest value of the distances recorded in step 3 and determine the median node, $u$, of this set.
(5) If the block type is descendant rotate the layout of $B_i$ found in step 1 such that $u$ is in the middle of the lower half of the embedding circle.
(6) Else rotate the layout of $B_i$ found in step 1 such that $u$ is in the middle of the upper half of the embedding circle.
(7) Place the remaining ancestor and descendant nodes in their closest ideal position.

The second technique *LayoutCluster2* has a higher time complexity, but may lead to layouts with fewer edge crossings. The first eight steps are the same as that of Algorithm 5.

During the placement of ancestor and descendant nodes which are not in ideal positions, each such node $v$ is placed in an ideal position and if the number of edge crossings added exceeds a threshold $T_1$ or the movement of $v$ exceeds a threshold $T_2$, then the size of the embedding circle is increased such that node $v$ can be placed in an ideal position without changing the relative order between $v$ and its neighbors on the embedding circle. See Fig. 14(c) for an example. The thresholds are determined on a per application basis. If increasing component edge crossings or node movement is undesirable for an application, the thresholds are adjusted accordingly. The time required for Algorithm *LayoutCluster2* is $O(m_i)$ if the threshold $T_2$ (based on node movement) is used or $O(m_i * k)$, where $k$ is the number of ancestor and descendant nodes in the cluster, if the threshold $T_1$ (based on the number of crossings) is used.

Another technique for drawing a biconnected component would rotate the embedding circle through many positions to find a good solution.

Now that we have addressed the subproblems, we present a comprehensive technique for obtaining circular layouts of nonbiconnected graphs.

**Algorithm 6.** *CIRCULAR—with Radial*.
**Input**: Any graph $G$.
**Output**: A circular drawing $\Gamma$ of $G$.

(1) Decompose $G$ into a block-cutpoint tree $T$.
(2) If $G$ has only one biconnected component perform Algorithm 1 on $G$.
(3) Else
(4) Assign the strict articulation points to a biconnected component.
(5) Layout the root cluster of $T$ with Algorithm 1.
(6) For each subtree $S$ of the root cluster
(7) Perform the $\rho$ coordinate assignment phase of *RADIAL—with Different Node Sizes* on $S$.
(8) For each biconnected component, $B_i$, of $S$
(9) Layout $B_i$ with Algorithm *LayoutCluster1*, or *LayoutCluster2* taking into account the radii defined for the superstructure tree in step 7.
(10) Considering the order of the subtrees defined during the layout of biconnected components in step 9, perform the $\theta$ coordinate assignment phase of *RADIAL—with Different Node Sizes* on $S$.
(11) Translate and rotate the clusters of $S$ according to the radial layout of $S$.

The time complexity of Algorithm 6 is $O(m)$ if the biconnected components are laid out with Algorithm *LayoutCluster1* or $O(m * k)$, where $k$ is the total number of ancestor and descendant nodes in the graph if Algorithm *LayoutCluster2* is used. See Fig. 15 for an example.

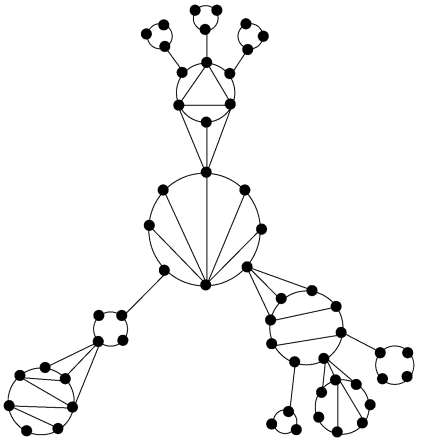

Fig. 15. A sample drawing as produced by Algorithm 6.

## 6. Implementation and experiments

### 6.1. Experimental analysis of Algorithm 1

We have implemented Algorithm 1 in C++ (GNU C++ version 2.7.2.1) on a SPARC 5 running SunOS 4.1.3. The code runs on top of the Tom Sawyer Software Graph Layout Toolkit (GLT) version 2.3.1. We also performed an extensive experimental study to compare Algorithms 1 and 2 with the circular layout component of the GLT. The circular layout technique in the GLT requires $O(n^2)$ time [6,9]. The results of the study show that the drawings of Algorithm 1 have about 15% fewer crossings on average than those produced by the GLT. Furthermore, the worst case time requirement for Algorithm 1 is $O(m)$ versus the $O(n^2)$ worst case time requirement for the GLT technique. Algorithm 2 is able to significantly further reduce the number of edge crossings.

The set of input graphs for our experiments included 10,328 biconnected components of minimum size 10 extracted from the 11,399 Rome graphs [5] which have between 10 and 80 nodes. The number of edge crossings is measured for Algorithm 1, Algorithm 2, and the circular drawing component of the GLT. As shown in the plot of Fig. 16, our techniques produce significantly fewer crossings on average than the GLT. Specifically the drawings of Algorithm 1 have significantly fewer crossings. And as the plot shows, Algorithm 2 effectively reduces the number of edge crossings even further. The percentage improvement between Algorithm 2 and GLT averages is a very good 30%. Sample drawings as produced by both GLT and our techniques are shown in Figs. 17–19.

### 6.2. Implementation issues

During step 4 of Algorithm 1, the technique chooses a node of lowest degree with the following priority: a wave front node, a wave center node, or some lowest degree node. An efficient way to execute this is to initially sort the nodes by degree into a table of lists which reflect those categories. The table is updated as nodes and edges are removed from

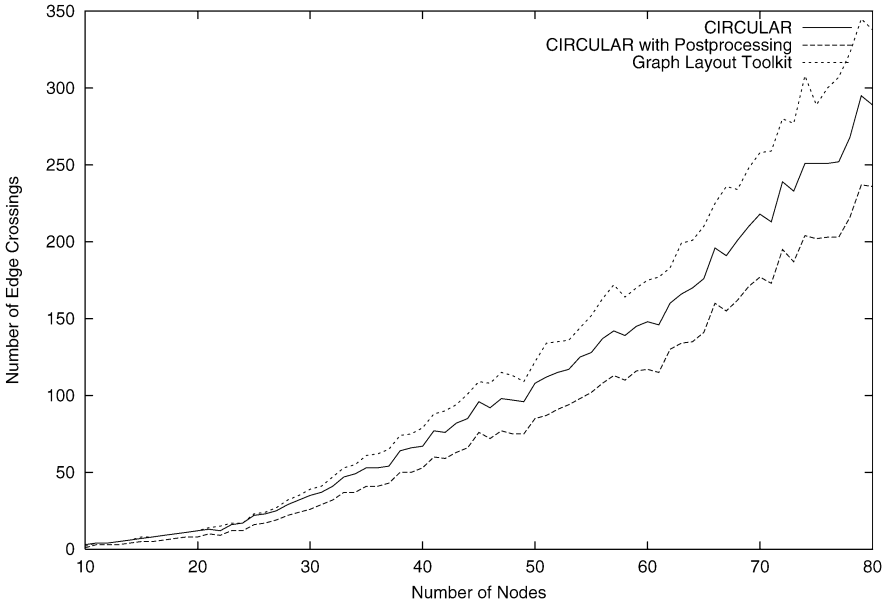


Fig. 16. The average number of edge crossings produced by Algorithm 1, Algorithm 2, and the Graph Layout Toolkit over 10,328 biconnected graphs.

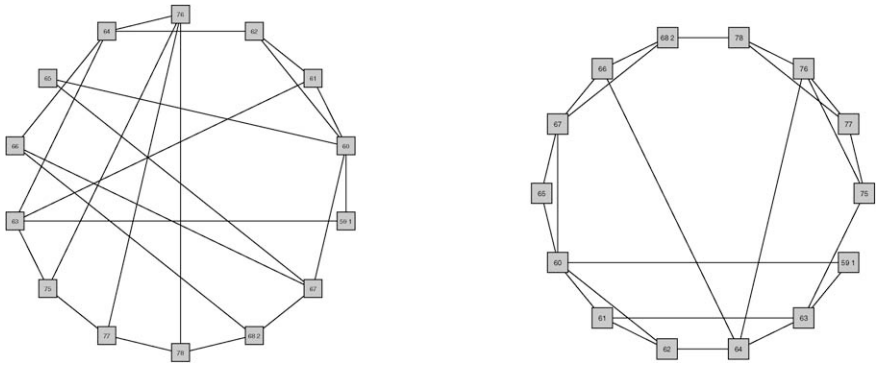

Fig. 17. The drawing on the left is produced by the GLT. The drawing on the right is of the same graph and is produced by Algorithm 2. The drawing produced by Algorithm 2 has 75% fewer crossings than the GLT drawing.

the graph. A bucket sort is initially used to place each node into its respective category. In order to keep the table updated, when a node, $v$, is processed, we simply move each neighbor of $v$ into the front of its respective degree list during each iteration (similar to self-adjusting lists). This way the nodes are retrieved in the desired priority: neighbor, previous neighbor, and lowest degree node. See Fig. 20.

During step 15, the algorithm performs a DFS which will result in a DFS tree. Then we place the nodes from the longest path within that DFS tree onto the embedding circle and

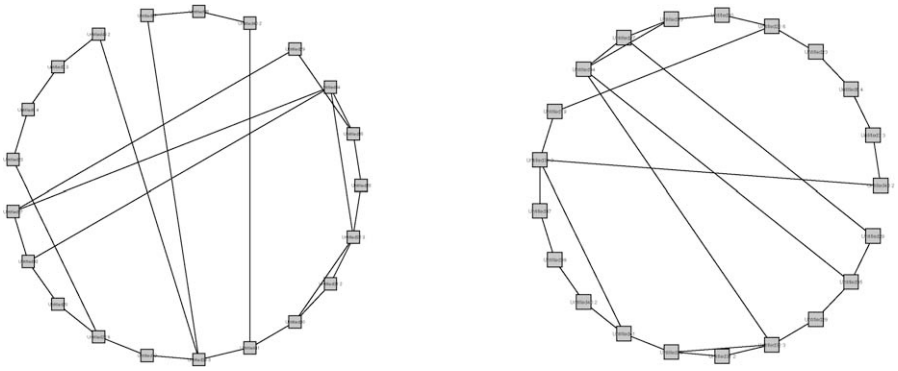

Fig. 18. The drawing on the left is produced by the GLT. The drawing on the right is of the same graph and is produced by Algorithm 2. The drawing produced by Algorithm 2 has 53% fewer crossings.

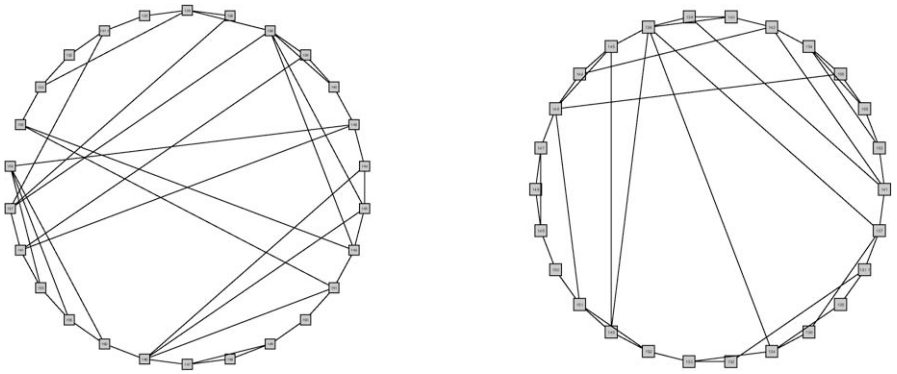

Fig. 19. The drawing on the left is produced by the GLT. The drawing on the right is of the same graph and is produced by Algorithm 2. The drawing produced by Algorithm 2 has 55% fewer crossings.

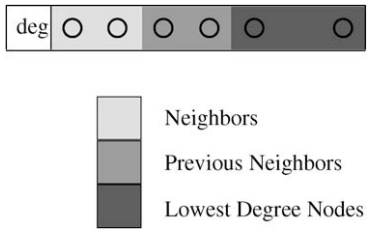


Fig. 20. The construction of each degree list within the node table.

we merge in the nodes of the remaining DFS tree branches. See Fig. 21. The longest path does not necessarily go through the root of the DFS tree as it does in this example.

If the input graph is outerplanar, the drawing produced by Algorithm 1 will always be plane. But if there are crossings then it may be possible to further reduce the number of crossings by moving nodes to a better position on the embedding circle. As noted in the

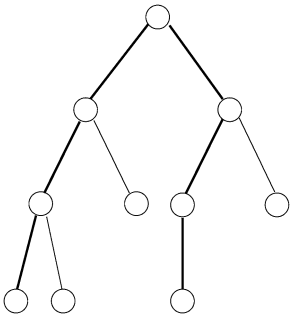

Fig. 21. A DFS tree with the edges of the longest path designated by thick lines.

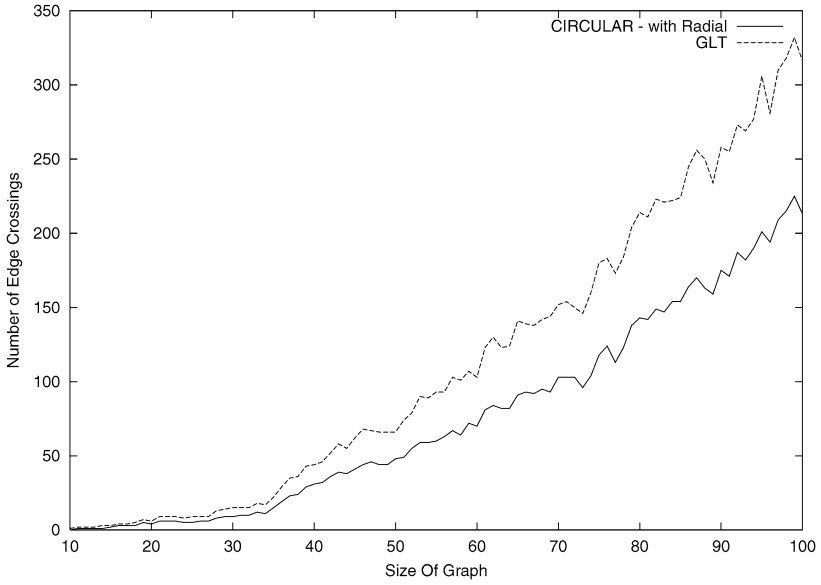


Fig. 22. This plot shows the average number of edge crossings produced by Algorithm 6 and the Graph Layout Toolkit over 11,399 graphs from [5].

time complexity analysis of Algorithm 2, the order is dominated by the time required for counting the number of crossings. Therefore it is vitally important to the time efficiency of the implementation of this algorithm that the number of crossings be counted in an effective manner. In order to lower the average time cost of counting crossings in the drawing, we ignore all edges which lie on the periphery of the embedding circle. These edges cannot possibly cause crossings. Also, in the step which determines the number of crossings caused by a single node, either the clockwise or counter-clockwise direction is first chosen dependent on which has the shorter arc.

### 6.3. Experimental analysis of Algorithm 6

We have implemented Algorithm 6 using Algorithm 5 and edge reduction postprocessing in C++ and run experiments with 11,399 graphs from [5]. The plot in Fig. 22 shows the average number of edge crossings produced by the circular layout component of the GLT and Algorithm 6. As is shown by these results, the average number of crossings in the drawings produced by our technique is about 35% less than that of the $O(n^2)$ GLT technique [6,9]. Sample drawings from both the GLT and Algorithm 6 are shown in Figs. 23 and 24.

The drawings produced by Algorithm 6 clearly show the biconnectivity characteristics of networks. And although these drawings have a low number of edge crossings, they may show more details than a user would wish to see at one time. Therefore, we suggest that Algorithm 6 can be used in an interactive environment in which the superstructure would be shown and the user would click on a node to see the details of the cluster. See Fig. 25 for

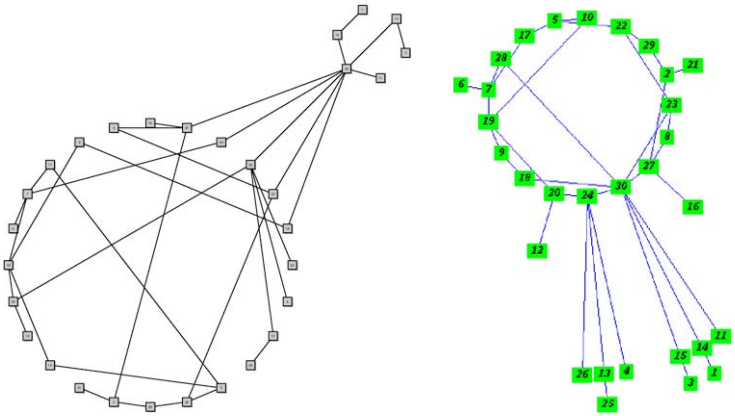

Fig. 23. The drawing on the left is produced by the GLT and the drawing on the right is of the same graph and is produced by Algorithm 6.

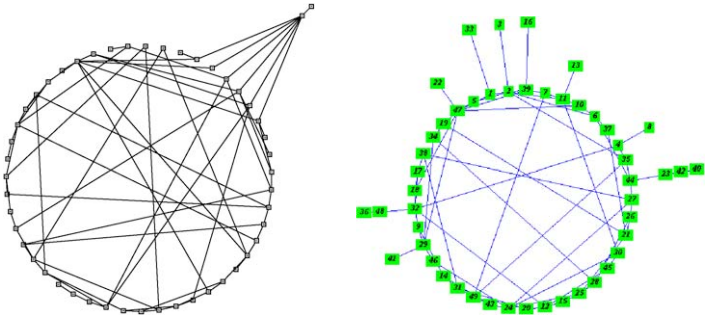

Fig. 24. The drawing on the left is produced by the GLT and the drawing on the right is of the same graph and is produced by Algorithm 6.

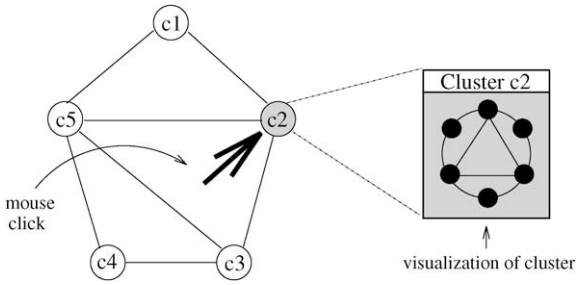


Fig. 25. Example interactive circular visualization.

an example. Alternatively, the levels of visualization could be combined and some clusters shown in detail while others are shown with a single node.

## 7. Conclusions and future work

Circular visualizations of networks which show the inherent strengths and weaknesses of structures with clustered views would be advantageous additions to many design tools. Some techniques for circular graph drawing have been previously presented, but the resulting drawings are visually complicated by the number of crossings.

We have introduced an $O(m)$ time algorithm for drawing circular visualizations of biconnected graphs on a single embedding circle. Not only is this technique efficient, but it also produces a plane drawing of the biconnected graph if such exists. Extensive experiments show that our technique significantly outperforms the current state of technology. We have also discussed an $O(m)$ time technique which decomposes the given graph into biconnected components and visualizes each cluster on a separate embedding circle. This technique has been implemented and results of an experimental study also show this algorithm to perform better than the current state of technology. Both techniques produce drawings which clearly show the biconnectivity structure of the given graphs and also have a low number of crossings. In the future, it would be interesting to study nonbiconnected graph drawing techniques in which the clusters are not necessarily biconnected and also that the superstructure is not a tree.

## References

[1] M.A. Bernard, On the automated drawing of graphs, in: Proc. 3rd Caribbean Conf. on Combinatorics and Computing, 1994, pp. 43–55.
[2] F. Brandenburg, Graph clustering 1: cycles of cliques, in: Proc. GD'97, in: Lecture Notes in Comput. Sci., vol. 1353, Springer, Berlin, 1997, pp. 158–168.
[3] G. Di Battista, P. Eades, R. Tamassia, I. Tollis, Algorithms for drawing graphs: an annotated bibliography, Computational Geometry 4 (5) (1994) 235–282.
[4] G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis, Graph Drawing: Algorithms for the Visualization of Graphs, Prentice-Hall, Englewood Cliffs, NJ, 1999.
[5] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, F. Vargiu, L. Vismara, An experimental comparison of four graph drawing algorithms, Computational Geometry 7 (5–6) (1997) 303–326.

[6] U. Doğrusöz, B. Madden, P. Madden, Circular layout in the Graph Layout Toolkit, in: Proc. GD'96, in: Lecture Notes in Comput. Sci., vol. 1190, Springer, Berlin, 1997, pp. 92–100.
[7] P. Eades, Drawing free trees, Bull. Inst. Combin. Appl. 5 (1992) 10–36.
[8] C. Esposito, Graph graphics: theory and practice, Comput. Math. Appl. 15 (4) (1988) 247–253.
[9] G. Kar, B. Madden, R. Gilbert, Heuristic layout algorithms for network presentation services, IEEE Network 11 (1988) 29–36.
[10] V. Krebs, Visualizing human networks, Release 1.0: Esther Dyson's Monthly Report, February 12, 1996, pp. 1–25.
[11] S. Masuda, T. Kashiwabara, K. Nakajima, T. Fujisawa, On the NP-completeness of a computer network layout problem, in: Proc. IEEE 1987 International Symposium on Circuits and Systems, Philadelphia, PA, 1987, pp. 292–295.
[12] S. Mitchell, Linear algorithms to recognize outerplanar and maximal outerplanar graphs, Inform. Process. Lett. 9 (5) (1979) 229–232.
[13] F.P. Preparata, M.I. Shamos, Computational Geometry: An Introduction, Springer, Berlin, 1985.
[14] J.M. Six (Urquhart), Vistool: a tool for visualizing graphs, Ph.D. Thesis, The University of Texas at Dallas, 2000.
[15] J.M. Six, I.G. Tollis, Circular drawings of biconnected graphs, in: Proc. of ALENEX'99, in: Lecture Notes in Comput. Sci., vol. 1619, Springer, Berlin, 1999, pp. 57–73.
[16] J.M. Six, I.G. Tollis, Circular drawings of telecommunication networks, in: D.I. Fotiadis, S.D. Nikolopoulos (Eds.), Advances in Informatics, Selected Papers from HCI'99, World Scientific, Singapore, 2000, pp. 313–323.
[17] J.M. Six, I.G. Tollis, A framework for circular drawings of networks, in: Proc. GD'99, in: Lecture Notes in Comput. Sci., vol. 1731, Springer, Berlin, 1999, pp. 107–116.
[18] I.G. Tollis, C. Xia, Drawing telecommunication networks, in: Proc. GD'94, in: Lecture Notes in Comput. Sci., vol. 894, Springer, Berlin, 1994, pp. 206–217.
[19] K. Yee, D. Fisher, R. Dhamija, M. Hearst, Animated exploration of dynamic graphs with radial layout, in: Proc. InfoVis 2001, IEEE, 2001, pp. 43–50.