9

Circular Drawing Algorithms

Janet M. Six
Lone Star Interaction Design

Ioannis G. Tollis
University of Crete and Technology Hellas-FORTH

9.1 Introduction

A *circular drawing* of a graph (see Figure 9.1 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 that would be strengthened by an accompanying circular graph drawing. For example, the drawing techniques could be added to tools which manipulate telecommunication [Ker93], computer [Six00], and social networks [Kre96] 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 pinpoint strengths and weaknesses within that design. It is essential that the number of edge crossings within each cluster remains low in

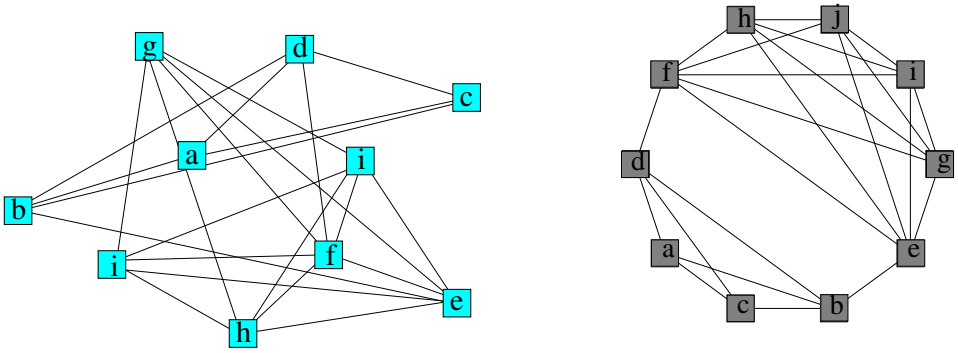

Figure 9.1 A graph with arbitrary coordinates for the nodes and a circular drawing of the same graph as produced by an implementation of Algorithm CIRCULAR. Figure taken from [ST99, ST06].

order to reduce the visual complexity of the resulting drawings. Researchers have produced several circular drawing techniques [Bra97, DMM97, KMG88, Kre96, TX95], 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 chapter, we present circular drawing techniques for simple biconnected and nonbiconnected graphs which are efficient and also produce drawings with a low number of edge crossings. The first technique produces single-circle drawings of biconnected graphs. The second technique produces single-circle drawings of nonbiconnected graphs. Finally, the third technique produces multiple-circle drawings of nonbiconnected graphs.

These techniques are very useful for many applications, however, with the exception of the Graph Layout Toolkit (GLT) technique [DMM97, KMG88], these techniques do not allow the user to define which nodes should be grouped together on an embedding circle. And in the GLT technique, the layouts of the user-defined groups are themselves placed on a single embedding circle. For some graph structures, this may not be ideal. In this chapter, we also present a circular drawing algorithm that allows the user to define the node groups, draws each group of nodes efficiently and effectively, and visualizes the superstructure well. We call this approach *user-grouped circular drawing*.

An example of an application in which user-grouped circular drawing would be useful is a computer network management system in which the user needs to know the current state of the network. It would be very helpful to allow the user to group the computers by department, floor, usage rates, or other criteria. See Figure 9.2. This graph drawing could also represent a telecommunications network, social network, or even the elements of a large software project. There are, of course, many other applications which would benefit from user-grouped circular drawing: e.g., biological networks, financial market modeling, HR management, and physical science models.

The remainder of this chapter is organized as follows: Section 9.1.1 discusses previous work in this area. In Section 9.2, we present an $O(m)$ time algorithm for the circular layout of biconnected graphs. The algorithm guarantees that if a zero-crossing circular drawing exists for a biconnected graph, then it will find it. In Section 9.2.1, we discuss properties of circular drawings created by the technique in Section 9.2. In Section 9.3, we discuss an approach for reducing the number of edge crossings in circular drawings. In Section 9.4, we present an $O(m)$ time algorithm for drawing nonbiconnected graphs on a single embedding circle. In Section 9.5, we present an $O(m)$ time algorithm for drawing nonbiconnected

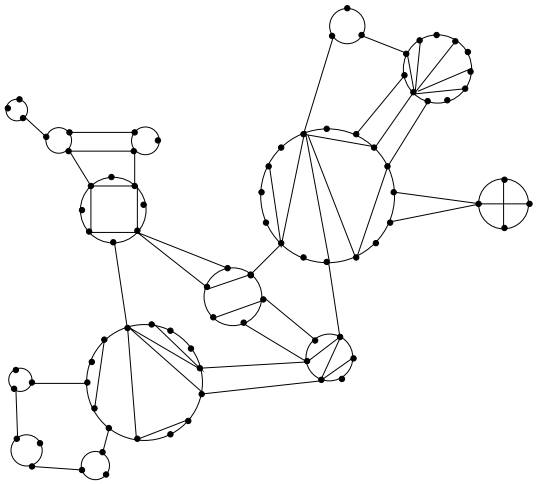

Figure 9.2 A user-grouped circular drawing. Figure taken from [ST03b].

graphs on multiple embedding circles. In Section 9.6, we introduce a framework for user-grouped circular drawing. In Section 9.7, we discuss implementation details and give results of experimental studies for these techniques. In Section 9.8, we present conclusions.

9.1.1 Other Circular Drawing Techniques

Kar, Madden, and Gilbert present a circular drawing technique and tool in [KMG88] 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 that 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 [DMM97], an advanced version of this $O(n^2)$ technique has been implemented as part of Tom Sawyer Software's successful Graph Layout Toolkit (GLT). An early heuristic on circular drawings was presented in [Ma88].

Tollis and Xia introduced several linear time algorithms for the visualization of survivable telecommunication networks in [TX95]. Given the ring covers of a network, these algorithms create circular drawings such that the survivability of the network is clearly 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 that 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 [Bra97]. 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 [Kre96] 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 [KW02], new ideas are presented that extend the framework for circular drawings described in this chapter, in order to make the framework suitable for user interaction. They introduce the concept of hicircular drawings, a hierarchical extension of the mentioned framework replacing the circles of single vertices by circles of circular or star-like structures. Various heuristic algorithms that find an ordering of vertices that reduce the number of crossings in the corresponding circular drawing are presented in [HS04]. A two-phase heuristic for crossing reduction in circular layout is proposed in [BB05]. Their extensive experimental results indicate that they yield few crossings. Three independent, complementary techniques for lowering the density and improving the readability of circular layouts are presented in [GK07]. First, an algorithm places the nodes on the circle such that edge lengths are reduced. Second, the circular drawing style is enhanced by allowing a set of carefully selected edges to be routed around the exterior of the circle. The third technique reduces density by coupling groups of edges as bundled splines that share part of their route.

Due to lack of space, we can not describe other techniques here, but refer the reader to other works such as [BB05, GK07, HS04, KW02, Ma88].

For more information on the algorithms presented in this chapter, see [ST06, ST03b].

9.1.2 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-hard *crossing number* problem. Furthermore, 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-hard as proven in [MKNF87]. The authors show the NP-hardness by giving a polynomial time transformation from the NP-complete *Modified Optimal Linear Arrangement* problem.

9.2 Circular Drawings of Biconnected Graphs

In order to produce circular drawings with few crossings, the algorithm tends to place edges toward the outside of the embedding circle. This characteristic is a result of placing a few edges in the middle of the drawing to be crossed. 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. However, the edge placement near the periphery may decrease the readability of the drawing. If this is an issue, an increase of scale will be helpful. An alternative approach where selected edges are drawn outside the embedding circle is described in [GK07].

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 Figure 9.3. A *wave center node* is adjacent to some other node that 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 Figure 9.4. Nodes v and w are said to be *paired* by u, and u is said to *establish* the pair edge (v, w).

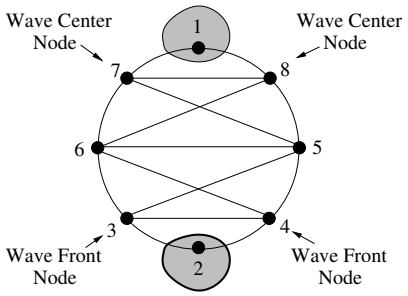

Figure 9.3 Examples of wave front and wave center nodes. The shaded region includes those nodes that have already been processed. The node labeled 2 is the most recently processed. Figure taken from [ST06].

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 that is inserted into the graph by the 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, the 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 Figure 9.5. The illustrations marked (a) show a 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 node u and 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 toward the periphery of the embedding circle.

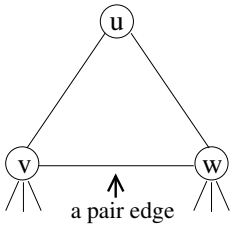

Figure 9.4 Example of a pair edge. Figure taken from [ST99, ST06].

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 whether (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 $O(deg(u))$ time. 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 last neighbors

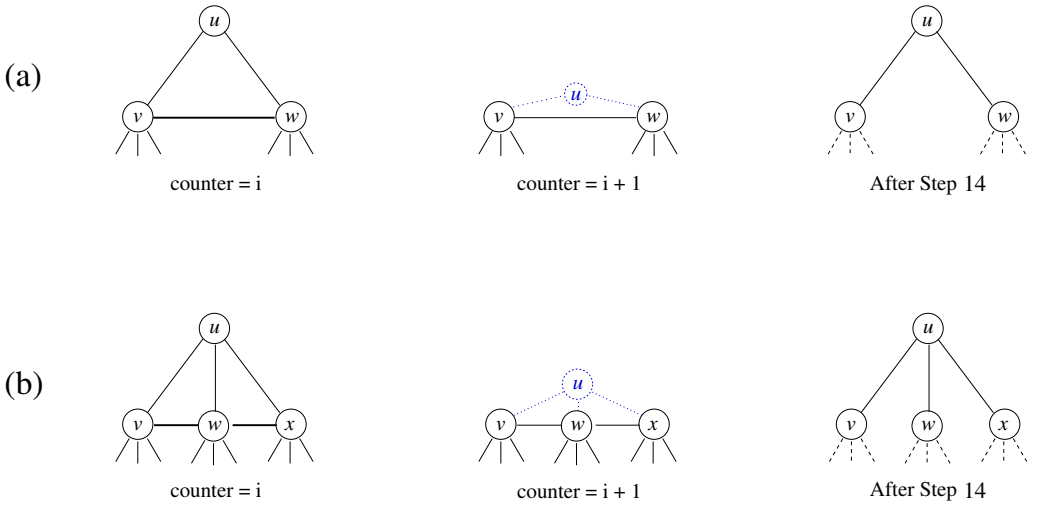

Figure 9.5 The node and edge absorption qualities of Algorithm CIRCULAR. Part (a) shows a degree-two node u and its neighbors v and w at three different points in the algorithm. First, the pair edge established by u, (v, w), is shown. Next, after node u is processed, node u and edges (u, v) and (u, w) are inherently represented by the edge (v, w). Finally, we see the same graph fragment after the pair edge (v, w) has been removed in Step 14. Part (b) shows a similar example with a degree-three node. Figure taken from [ST99, ST06].

visited cannot 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 ith iteration of the While loop. $minDeg_i \leq avgDeg$ before the ith iteration, $\forall i \geq 1$ and where $avgDeg$ is the average degree of the nodes in the original graph G. 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 $minDeg_i \leq avgDeg$ before the ith iteration, $\forall i \geq 1$, we also know that

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

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

Subsequent to the edge removal, the 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 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 no 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 CIRCULAR
Input: A biconnected graph, $G = (V, E)$.
Output: A circular drawing Γ 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* $\leq n - 3$
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 that 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.

Figure 9.6 Algorithm CIRCULAR.

Figure 9.6 shows the pseudocode for Algorithm CIRCULAR. The time complexity of Algorithm CIRCULAR 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 ith iteration. Notice that the number of triangulation edges added by Step 9 is $O(m)$, as shown above. 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.

9.2.1 Properties of Algorithm CIRCULAR

In this section, we give properties of Algorithm CIRCULAR. See [ST06, ST03b] for the detailed proofs. A biconnected 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 CIRCULAR is outerplanar, then the result will be a circular visualization such that no two edges cross. In fact, the technique has been inspired by the algorithm for recognizing outerplanar graphs presented in [Mit79].

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 that

is not outerplanar does not admit a plane circular drawing. In fact, the set of biconnected graphs that 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 CIRCULAR.

Therefore, we have the following theorem:

Theorem 9.1 *Given a biconnected graph G, if G admits a circular layout with zero crossings, then Algorithm CIRCULAR produces a circular drawing with zero crossings in $O(n)$ time.*

Also, as shown in the discussion of the time requirements for Algorithm CIRCULAR, we have:

Theorem 9.2 *Algorithm CIRCULAR produces a circular drawing of any biconnected graph in $O(m)$ time.*

9.3 Further Reduction of Edge Crossings

As will be shown in the experimental results of Section 7.1, Algorithm CIRCULAR produces drawings with a low number of edge crossings and works very well in practice. We can further reduce the number of edge crossings with the technique presented in this section. As discussed in Section 9.1.2, the problem of minimizing the number of edge crossings in a circular graph drawing is NP-hard. The configuration of the nodes as determined by Algorithm CIRCULAR 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 v and queries whether crossings can be reduced further by moving v next to one of its neighbors.

See Figure 9.7 for Algorithm CIRCULAR-Postprocessing. The time complexity of Algorithm CIRCULAR-Postprocessing 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 CIRCULAR-Postprocessing that the number of crossings be counted in an efficient fashion. As will be shown in Lemma 9.1, Step 1 of Algorithm CIRCULAR-Postprocessing requires $O(m + \chi)$ time to find the total number of crossings, where m is the number of edges and χ is the number of crossings. The experimental study presented in Section 9.7 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 12.3.2, we will show that it takes $O(m)$ time 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, χ is $O(\Sigma_{i=1}^{m} i)$, which is $O(m^2)$. Therefore, Algorithm CIRCULAR-Postprocessing has time complexity $O(m^2)$.

Algorithm CIRCULAR-Postprocessing
Input: A drawing Γ of biconnected graph $G = (V, E)$ produced by Algorithm CIRCULAR.
Output: A drawing Γ' 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.

Figure 9.7 Algorithm CIRCULAR-Postprocessing.

9.3.1 Counting All the Crossings in a Circular Drawing

Consider the straight edges e_i and e_j of Figure 9.8. 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 the problem is one dimensional, this technique has some similarities to the line segment intersection algorithm presented in [PS85].

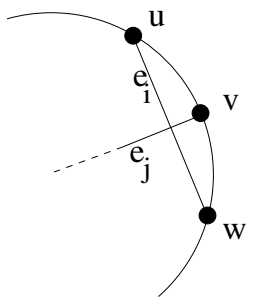

Figure 9.8 An open edge with respect to the arc uvw. Figure taken from [ST99, ST06].

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

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*.

Figure 9.9 Algorithm CountAllCrossings.

Algorithm CountAllCrossings 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 time

$$\sum_{i=1}^{2m} \chi_i = O(\chi),$$

where χ_i is the number of edge crossings caused by the edge e_i and χ is the total number of edge crossings in the embedding. We accomplish this time requirement by traversing *openEdgeList* backward from the end of the list to the element which contains e_i. Therefore, we have the following:

LEMMA 9.1 Algorithm CountAllCrossings counts the total number of edge crossings in a single circle embedding, where m is the number of edges and χ is the number of crossings in $O(m + \chi)$ time.

9.3.2 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:

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

where, χ_v = Number of crossings caused by v in the old location,
and χ'_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 χ_v and χ'_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 toward 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 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 Figure 9.10.

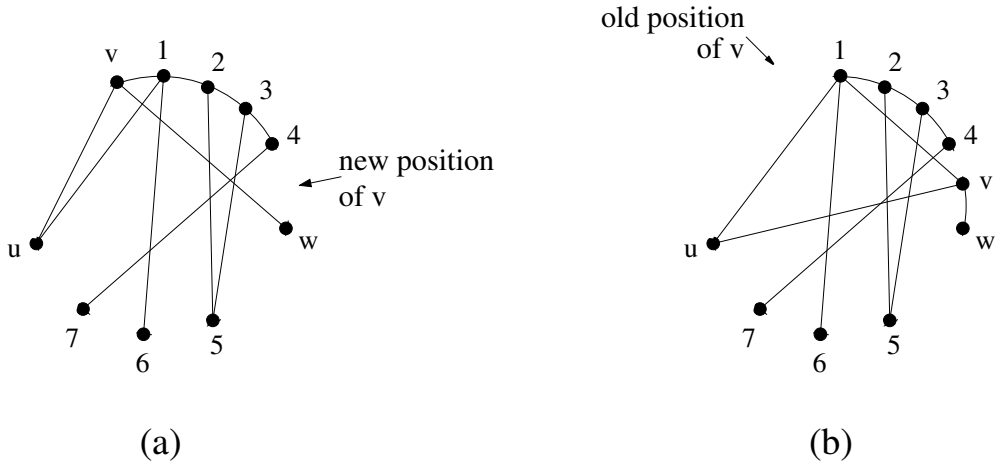

Figure 9.10 The arc created by moving node v to the position denoted with the arrow. The pertinent edges of the arc are shown. Figure taken from [ST06].

Therefore, we have the following result:

LEMMA 9.2 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.

The pseudocode for Algorithm CountSingleNodeCrossings is shown in Figure 9.11. This algorithm requires $O(m)$ time. Steps 3, 4, 5, 6, 7, and 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 CountSingleNodeCrossings is swapping the placement of two nodes which are next to each other, u and v, on the embedding circle, then Algorithm CountSingleNodeCrossings 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 Figure 9.12. Since a swap of these two nodes can be accomplished by moving u between v and β or moving v between α and u, we choose the move such that the number of pertinent edges (i.e., the degree of the node which is not moved) is smaller. Both of the moves produce the same node ordering, so we perform the move which requires less time. In the specific case of Figure 9.12, we choose to move node u.

Given Lemma 9.1, and Lemma 9.2, Algorithm CIRCULAR-Postprocessing produces a visualization with a reduced number of edge crossings in $O(m^2)$ time.

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

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, α.
13. Mark the pertinent edges as not seen.
14. Repeat Steps 5–8 in the opposite direction.
15. $NewNumberSingleNodeCrossings = numberOfCrossings$.
16. $changeInCrossings = NewNumberSingleNodeCrossings - OldNumberSingleNodeCrossings$.

Figure 9.11 Algorithm CountSingleNodeCrossings.

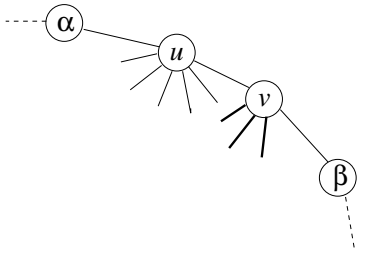

Figure 9.12 The pertinent edges for Algorithm CountSingleNodeCrossings if the two adjacent nodes u and v are being swapped. Figure taken from [ST99, ST06].

9.4 Nonbiconnected Graphs on a Single Circle

Most networks are not biconnected. Therefore, it is important for a circular drawing tool to provide a component that visualizes nonbiconnected graphs. An algorithm for producing circular drawings of nonbiconnected graphs on a single embedding circle is presented in [Six00, ST06]. Given G, a nonbiconnected graph, it can be decomposed into its biconnected components. The algorithm layouts the resulting block-cutpoint tree on a circle and then it layouts each biconnected component with a variant of Algorithm CIRCULAR.

First, we consider how to obtain a circular drawing of a tree. A DFS produces a numbering that we can use to order the nodes around the embedding circle in a crossing-free manner. From this result, we know how to order the biconnected components around the embedding circle. Next, we need to consider articulation points which are not adjacent to a bridge (*strict*

articulation points). Strict articulation points appear in multiple biconnected components. In which biconnected component should a strict articulation point appear in the circular drawing? Multiple approaches to this issue are discussed in [Six00, ST99]. Due to space restrictions, we do not discuss these solutions here. A third issue to consider is how to transform the layout of each biconnected component to fit onto an arc of the embedding circle. This transformation is called *breaking*. The resulting breaks occur at an articulation point within the biconnected component.

The worst-case time requirement for the above algorithm is $O(m)$ if we use Algorithm CIRCULAR to layout each biconnected component. The resulting drawings have the property that the nodes of each biconnected component (with the exception of some strict articulation points) appear consecutively. Furthermore, the order of the biconnected components on the embedding circle are placed according to a layout of the accompanying block-cutpoint tree. Therefore, the biconnectivity structure of a graph is displayed even though all of the nodes appear on a single circle. An example drawing is shown in Figure 9.13. More details on this algorithm can be found in [Six00, ST06].

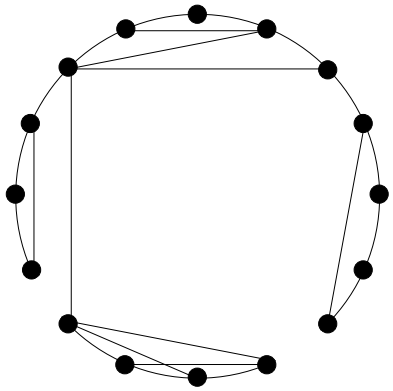

Figure 9.13 An example drawing produced by Algorithm CIRCULAR-Nonbiconnected.

9.5 Nonbiconnected Graphs on Multiple Circles

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 block-cutpoint tree using a radial layout technique similar to [Ber81, Ead92, Esp88], then we layout each biconnected component of the graph with a variant of Algorithm CIRCULAR. See Figure 9.14.

The 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 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 [DETT99]. Alternatively, we can pick the root dependent on some important metric: e.g., size of the biconnected component. Next we address the second issue. *Strict articulation points* (i.e., articulation points that are not adjacent to a bridge) 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 Figure 9.15(a). The second approach assigns the articulation point to the biconnected component which contains the most neighbors of that articulation point, see Figure 9.15(b). The third approach assigns the articulation point to a position between its biconnected components, see Figure 9.15(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 while performing the layout of the block-cutpoint tree is 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 [Ber81, Ead92, Esp88] 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 the block-cutpoint trees, the nodes of the trees laid out with [Ber81, Ead92, Esp88] are all the same size. The technique in [YFDH01] 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 [Ber81, Ead92, Esp88]:

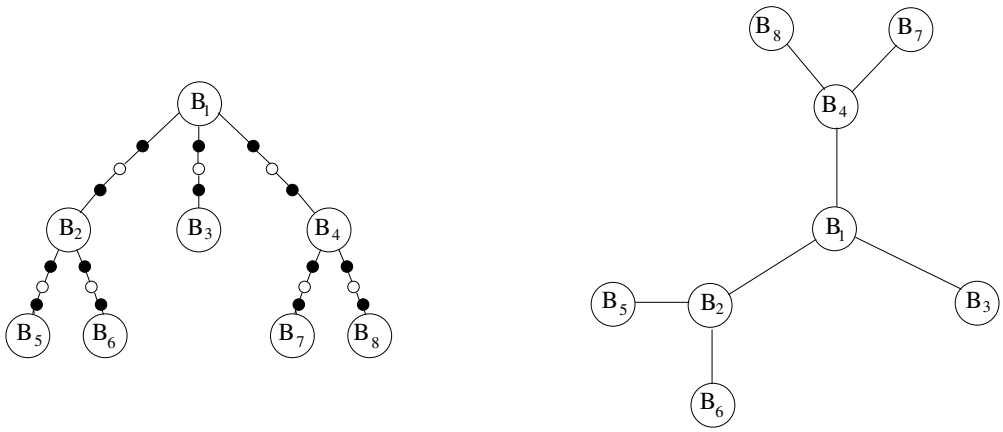

Figure 9.14 The illustration on the left shows the block-cutpoint tree of a nonbiconnected graph. The small black tree nodes represent articulation points and the small white tree nodes represent bridges. The right illustration is a drawing of the same graph where the block-cutpoint tree is laid out with a radial tree layout technique. Figure taken from [ST06].

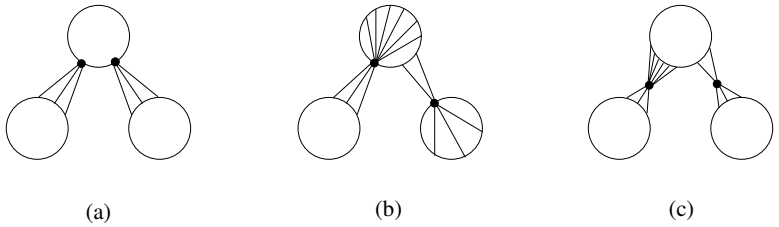

Figure 9.15 Examples of three approaches for the assignment of strict articulation points to biconnected components. The black nodes are strict articulation points. Figure taken from [ST06].

RADIAL – with Different Node Sizes: For each node, we must assign a ρ coordinate, which is the distance from point $(0, 0)$ to the placement of that node and a θ 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 ρ 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 ρ coordinate of the parent u of v, δ is the minimum distance allowed between two nodes, d_u is the diameter of u, and $max(d_1, d_2, ..., d_k)$ 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 θ coordinate of node v depends on the widths of the descendants of v, not just the number of leaves as in [Ber81, Ead92, Esp88]. This assignment of coordinates leads to a layout of the form shown in Figure 9.16.

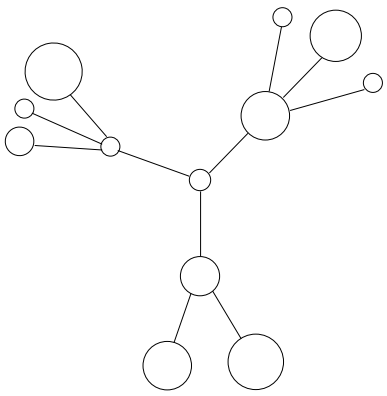

Figure 9.16 A radial layout of a tree with differing size nodes. Figure taken from [ST06].

The fourth issue to be addressed by the 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 Figure 9.17. 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, the technique tries to place ancestor nodes in the arc between the points α and β. The size of the arc from α to β 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 γ, δ, and ϵ, as shown in Figure 9.17. These special positions for the ancestor and descendant nodes are called *ideal positions*. Because of 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.

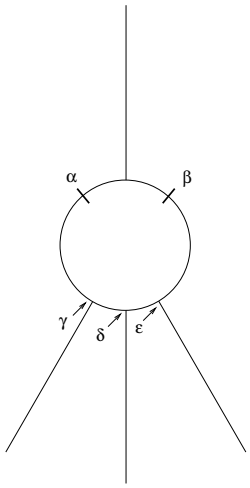

Figure 9.17 The relation between the layout of the block-cutpoint tree and the layout of an individual biconnected component. Figure taken from [ST06].

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 $\frac{n_i}{2}$, 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 technique is to perform Algorithm CIRCULAR 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. Next, this drawing is updated 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 CIRCULAR 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. See Figure 9.18 for Algorithm LayoutCluster1. This algorithm requires $O(m_i)$ time. See Figure 9.19(b) for an example.

Algorithm 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 CIRCULAR on B_i and save the results in Γ_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 Γ_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 that 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.

Figure 9.18 Algorithm LayoutCluster1.

The second technique *LayoutCluster2* has a higher time complexity but may lead to layouts with fewer edge crossings. The first seven steps are the same as that of Algorithm LayoutCluster1. During the placement of ancestor and descendant nodes that 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 Figure 9.19(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 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 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, called Algorithm CIRCULAR-with Radial, see Figure 9.20 for the pseudocode of the algorithm. The time complexity of Algorithm CIRCULAR-with Radial 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. Figure 9.21 shows an example produced by this algorithm.

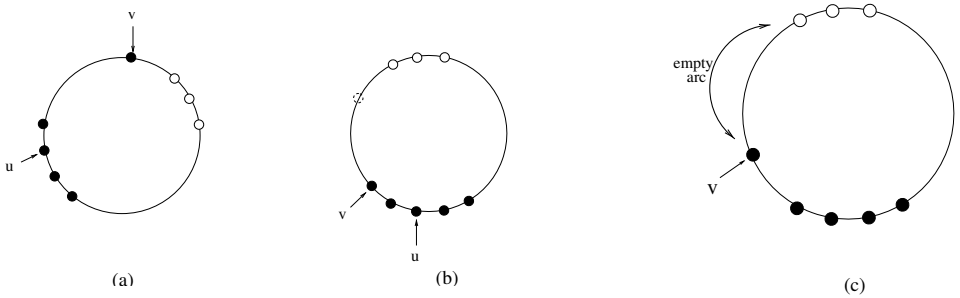

Figure 9.19 This figure demonstrates Algorithms *LayoutCluster1* and *LayoutCluster2*. The black nodes are descendant nodes and the white nodes are ancestor nodes. (a) Drawing produced by Algorithm CIRCULAR; (b) the rotated drawing of part (a) produced by Algorithm LayoutCluster1; (c) the resulting drawing of part (a) produced by Algorithm LayoutCluster2. Figure taken from [ST06].

Algorithm CIRCULAR-with Radial
Input: Any graph G.
Output: A circular drawing Γ of G.

1. Decompose G into a block-cutpoint tree T.
2. If G has only one biconnected component, perform Algorithm CIRCULAR on G.
3. Else
4. Assign the strict articulation points to a biconnected component.
5. Layout the root cluster of T with Algorithm CIRCULAR.
6. For each subtree S of the root cluster
7. Perform the ρ 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 θ 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.

Figure 9.20 Algorithm CIRCULAR-with Radial.

An extension of Algorithm CIRCULAR-with Radial to include interactive schemes has been presented by Kaufmann and Wiese in [KW02].

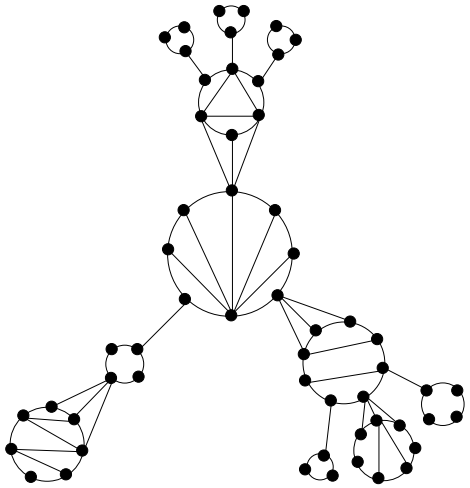

Figure 9.21 A sample drawing as produced by Algorithm CIRCULAR-with Radial. Figure taken from [ST06].

9.6 A Framework for User-Grouped Circular Drawing

The problem of producing circular drawings of graphs grouped by biconnectivity is quite different from the problem of drawing a graph whose grouping is user-defined. In the latter case, there is no known structure of either the groups or the relationship between the groups. Therefore, we must use a general method for producing this type of visualization. The four goals of a user-grouped circular drawing technique should be:

1. the user-defined groupings are highly visible,
2. each group is laid out with a low number of edge crossings,
3. the number of crossings between intra-group and inter-group edges is low, and
4. the layout technique is fast.

We know from previous work in clustered graph drawing [EFL97, EF97, EFN99, HE98] that the relationship between groups is often not very complex. We take advantage of this knowledge in this framework. Define the *superstructure* G_s of a given graph $G = (V, E, P)$, where P is the node group partition, as follows: the nodes in G_s represent the elements of P. For each edge $e \in E$ which is incident to nodes in two different node groups, place an edge between nodes representing the respective groups in G_s. The type of structure that we expect G_s to have should be visualized well with a force-directed [DETT99, Ead84] technique; therefore, the superstructure G_s will be drawn with this approach. Additionally, since G_s will likely not be a very complicated graph, it should not take much time to achieve a good drawing with a force-directed technique.

The node groups themselves will be either biconnected or not. Since Algorithms CIRCULAR and CIRCULAR-Nonbiconnected can layout biconnected and nonbiconnected graphs on a single embedding circle in linear time and have been shown to perform well in practice, we also will use those techniques here.

We have now addressed how to achieve Goals 1 and 2 with good speed. However, in order to produce good user-grouped circular graph drawings, we must successfully merge these two techniques so that we can simultaneously reach Goals 1,2, and 3. And, of course, we need a fast technique in order to achieve Goal 4. Attaining Goal 3 is very important to

the quality of drawings produced by a user-grouped circular drawing technique. As shown in [Pur97], a drawing with fewer crossings is more readable. It is especially important to reduce the number of intra-group and inter-group edge crossings as those can particularly cause confusion while interpreting a drawing. See Figure 9.22. How can we achieve this low number of crossings? We must place nodes that are adjacent to nodes in other groups (called *outnodes* in [DMM97, KMG88]) close to the placement of those other nodes. A force-directed approach is a good way to attain this goal since it would encourage outnodes to be closer to their neighbors. Traditional force-directed approaches [DETT99, Ead84] will not work here though, because we need to constrain the placement of nodes to circles. In Section 9.6.1, we present a force-directed approach in which the nodes are restricted to appear on circular tracks. With the use of this technique we will reach Goal 3. As will be discussed, we can do this in a reasonable amount of time.

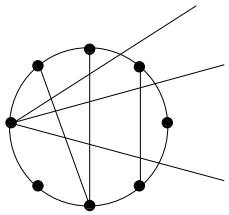

Figure 9.22 Example of intra-group and inter-group edge crossings. Figure taken from [ST03b].

As with most force-directed techniques, the initial placement of nodes has a very significant impact on the final drawing [DETT99, Ead84]. Therefore, it is important to have a good initial placement. This is why we should layout the superstructure and each node group first. At the completion of those steps, we should have the almost-final drawing. It will then be a matter of fine-tuning the drawing with the circular-track force-directed technique. And as shown in [ST01] (see extended version in [ST03a]), once you have an almost-final drawing, it does not take much time for a force-directed technique to converge.

9.6.1 Circular-Track Force-Directed

In order to adapt the force-directed paradigm for circular drawing, we need a way to guarantee that the nodes of a group appear on the circumference of an embedding circle, *the circular track*. The nodes are restricted to appear on the circular track, but are allowed to jump over each other and appear in any order, see Figure 9.23. And as in the force-directed approach, we want to minimize the potential energy in the spring system which is modelling the graph. In this section, we describe how this circular-track adaptation is achieved.

First, we need to look at node coordinates in a different way. Node i belongs to group α and is located at position (x_i, y_i). Given that the center of the embedding circle on which α is located is at (x_α, y_α) and the radius of that circle is r_α, we can restate the coordinates of i in the following way:

$$x_i = x_\alpha + r_\alpha * cos(\theta_i) \tag{9.1}$$

$$y_i = y_\alpha + r_\alpha * sin(\theta_i) \tag{9.2}$$

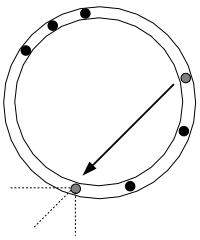

Figure 9.23 Circular-track force-directed technique. Figure taken from [ST03b].

Remember that Hooke's Law [HR88] gives us the following equation for the potential energy V in a spring system:

$$V = \sum_{ij} k_{ij}[(x_i - x_j)^2 + (y_i - y_j)^2] \tag{9.3}$$

where k_{ij} is the spring constant for the spring between nodes i and j. Equation (12.3) can be rewritten using (12.1) and (12.2):

$$\begin{aligned} V \quad = \textstyle\sum_{(i,j)\in E} k_{ij} \quad & [((x_\alpha + r_\alpha * cos(\theta_i)) - (x_\beta + r_\beta * cos(\theta_j)))^2 + \\ & ((y_\alpha + r_\alpha * sin(\theta_i)) - (y_\beta + r_\beta * sin(\theta_j)))^2] \end{aligned} \tag{9.4}$$

where node j belongs to group β, (x_β, y_β) is the center and r_β is the radius of the embedding circle on which β appears. Thus, we have:

$$\begin{aligned} V \quad = \textstyle\sum_{(i,j)\in E} k_{ij} \quad & [(x_\alpha + r_\alpha * cos(\theta_i) - x_\beta - r_\beta * cos(\theta_j))^2 + \\ & (y_\alpha + r_\alpha * sin(\theta_i) - y_\beta - r_\beta * sin(\theta_j))^2] \end{aligned} \tag{9.5}$$

We can find a minimal energy solution on variables x, y, and θ. It is interesting to note that if i and j are on the same circle, then x_α and x_β are equivalent as are y_α and y_β. And, of course, $r_\alpha = r_\beta$. Now we rewrite equation (5):

$$V = \sum_{(i,j)\in E} k_{ij}[r_\alpha(cos(\theta_i) - cos(\theta_j))^2 + r_\alpha(sin(\theta_i) - sin(\theta_j))^2)] \tag{9.6}$$

We can calculate r_α from the number of nodes in α so that means that finding the minimum V is now a one-dimensional problem based on finding the right set of θs. When we combine (12.5) or (12.6) with equations for magnetic repulsion to prevent node occlusion, we have a force-directed equation for which the nodes of a group lie on the circumference of a circle. Now we extend equation (12.5) to include repulsive forces.

$$\begin{aligned} \rho_{ij} \quad = \quad & [(x_\alpha + r_\alpha * cos(\theta_i) - x_\beta - r_\beta * cos(\theta_j))^2 + \\ & (y_\alpha + r_\alpha * sin(\theta_i) - y_\beta - r_\beta * sin(\theta_j))^2] \end{aligned} \tag{9.7}$$

$$V = \sum_{(i,j)\in E} k_{ij}\rho_{ij} + \sum_{(i,j)\in V\times V} g_{ij}\frac{1}{\rho_{ij}} \tag{9.8}$$

where g_{ij} is the repulsive constant between nodes i and j.

Another important consideration is the set of spring constants used in the above equations. It is not necessary for the spring constant to be the same for each pair of nodes. It is also possible for these constants to change during different phases of execution.

9.6.2 A Technique for Creating User-Grouped Circular Drawings

Now that we have a force-directed technique in which the nodes are placed on circular tracks, we need to show how we will successfully merge the force-directed approach and the circular drawing techniques discussed earlier in this chapter. We present a technique for creating user-grouped circular drawings in Figure 9.24.

Algorithm CIRCULAR-with Forces
Input: A graph $G = (V, E, P)$.
Output: User-grouped circular drawing of G Γ.

1. Determine the superstructure G_s of G.
2. Layout G_s with a basic force-directed technique.
3. For each group p_i in P
   (a) If the subgraph induced by p_i, G_i, is biconnected
   layout G_i with *CIRCULAR*.
   (b) Else layout G_i with *CIRCULAR-Nonbiconnected*.
4. Place the layout of each group p_i at the respective location found in Step 2.
5. For each group p_i
   (a) rotate the layout circle and keep the position which has the lowest local potential energy.
   (b) reverse the order of the nodes around the embedding circle and repeat Step 5a.
   (c) if the result of Step 5a had a lower local potential energy than that of Step 5b revert to the result of Step 5a.
6. Apply a force-directed technique using the equations of Section 9.6.1 to G.

Figure 9.24 Algorithm CIRCULAR-with Forces.

Going back to the four goals discussed in Section 9.6, we will attain Goal 1 by using a basic force-directed technique to layout the superstructure. We will attain Goal 2 by laying out each group with either Algorithms CIRCULAR or CIRCULAR-Nonbiconnected. Attaining Goal 3 means successfully merging the results of the force-directed and circular techniques.

Once we have the layout of the superstructure and each group, we place the layout of each group at the respective location found during the layout of the superstucture. Now we have an almost-final layout: it is a matter of rotating the layouts of the groups and maybe adjusting the order of nodes around the embedding circle. Since we know that Algorithms CIRCULAR and CIRCULAR-Nonbiconnected produce good visualizations, we should change these layouts as little as possible. So first, we will fine-tune the almost-final drawing by rotating each layout and keeping the rotation that has the least local potential energy. We rotate each embedding circle through n_α positions, where n_α is the number of nodes in group α. With respect to determining local potential energy, we need to determine

the lengths of inter-group edges that are incident to the nodes of α. The rotation of choice should minimize the lengths of those edges. In other words, we choose the rotation in which as many nodes as possible are close to their other-group neighbors. Since for each embedding circle we try n_α positions and examine the length of α's incident inter-group edges at each position, then the rotation step will take $O(n*m_{inter-group})$ time for the entire graph, where $m_{inter-group}$ is the number of inter-group edges. As discussed in Section 9.6, we expect $m_{inter-group} \ll m$. Then we will "flip" each layout and again rotate. We keep the rotation which has the least local potential energy. After these steps, it is still possible that some nodes will be badly placed with respect to their relationships with nodes in other groups. In other words, those placements cause intra-group and inter-group edges to cross. In order to address this problem, we will apply the force-directed technique described in Section 9.6.1. The result of this step will be the reduction of intra-group and inter-group edge crossings since nodes will be pulled to the side of the embedding circle which is closer to their other-group relatives.

Because Algorithm CIRCULAR-with Forces makes use of a force-directed technique, the worst-case time requirement is unknown. However, in practice, we expect the time requirement to be $O(n^2)$ for the following reasons: Step 1 requires $O(m)$ time. Step 2 will be on a small graph and should not require much time to reach convergence. Step 3 requires $O(m)$ time. Step 4 requires $O(n)$ time. Step 5 require $O(n * m_{inter-group})$ time. Since Step 6 is a force-directed technique, it could take $O(n^3)$ time in practice; however, the result of the previous steps will be an almost-final layout and thus should not need much time to converge. It was evidenced in [ST01] (see extended version in [ST03a]) that when a force-directed technique is applied to an almost-final layout, it does not take much more time for convergence to occur. Therefore, in practice we expect this step to require $O(n^2)$ time. Thus, we have attained Goal 4 from Section 9.6.

9.7 Implementation and Experiments

9.7.1 Experimental Analysis of Algorithm CIRCULAR

We have implemented Algorithm CIRCULAR in C++. 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 CIRCULAR and CIRCULAR-Postprocessing with the circular layout component of the GLT. The circular layout technique in the GLT requires $O(n^2)$ time [DMM97, KMG88]. The results of the study show that the drawings of Algorithm CIRCULAR have about 15% fewer crossings on average than those produced by the GLT. Furthermore, the worst-case time requirement for Algorithm CIRCULAR is $O(m)$ versus the $O(n^2)$ worst-case time requirement for the GLT technique. Algorithm CIRCULAR-Postprocessing is able to significantly further reduce the number of edge crossings.

The set of input graphs for the experiments included 10,328 biconnected components of minimum size 10 extracted from the 11,399 Rome graphs [DGL+97], which have between 10 and 80 nodes. The number of edge crossings is measured for Algorithm CIRCULAR, Algorithm CIRCULAR-Postprocessing, and the circular drawing component of the GLT. As shown in the plot of Figure 9.25, the techniques produce significantly fewer crossings on average than the GLT. Specifically the drawings of Algorithm CIRCULAR have significantly fewer crossings. And as the plot shows, Algorithm CIRCULAR-Postprocessing effectively reduces the number of edge crossings even further. The percentage improvement between Algorithm CIRCULAR-Postprocessing and GLT averages is 30%. Sample drawings as produced by both GLT and the techniques are shown in Figures 9.26–9.28.

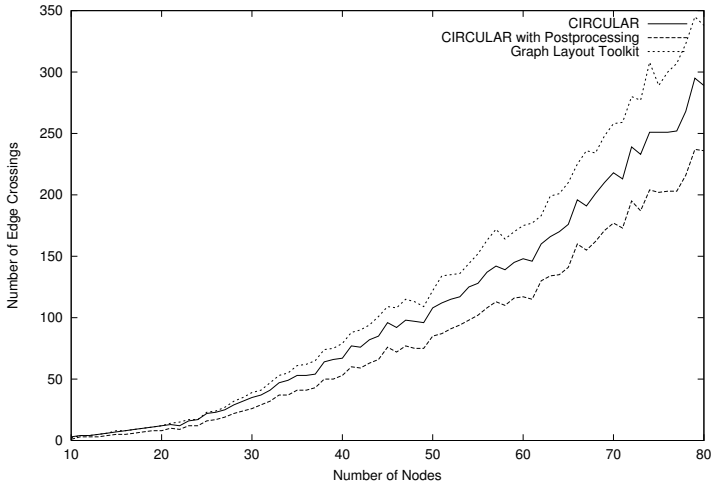

Figure 9.25 The average number of edge crossings produced by Algorithm CIRCULAR, Algorithm CIRCULAR-Postprocessing, and the Graph Layout Toolkit over 10,328 biconnected graphs. Figure taken from [ST99, ST06].

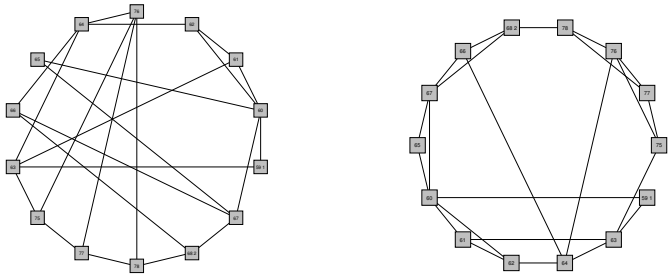

Figure 9.26 The drawing on the left is produced by the GLT. The drawing on the right is of the same graph and is produced by Algorithm CIRCULAR-Postprocessing. The drawing produced by Algorithm CIRCULAR-Postprocessing has 75% fewer crossings than the GLT drawing. Figure taken from [ST99, ST06].

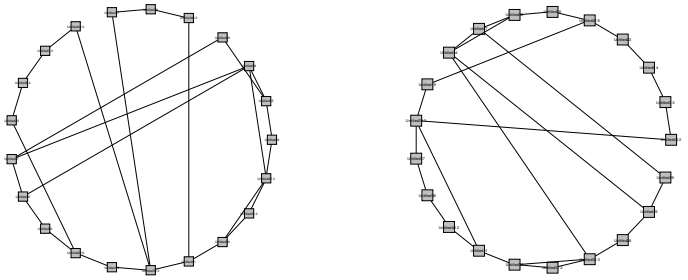

Figure 9.27 The drawing on the left is produced by the GLT. The drawing on the right is of the same graph and is produced by Algorithm CIRCULAR-Postprocessing. The drawing produced by Algorithm CIRCULAR-Postprocessing has 53% fewer crossings. Figure taken from [ST99, ST06].

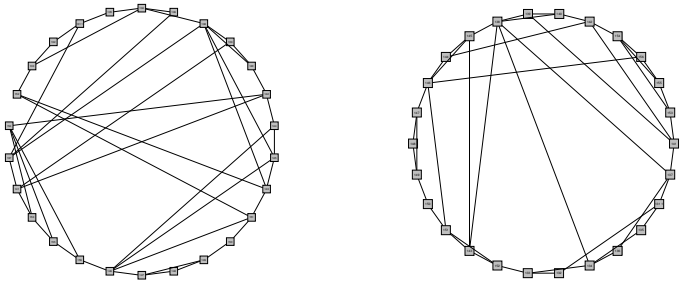

Figure 9.28 The drawing on the left is produced by the GLT. The drawing on the right is of the same graph and is produced by Algorithm CIRCULAR-Postprocessing. The drawing produced by Algorithm CIRCULAR-Postprocessing has 55% fewer crossings. Figure taken from [ST99, ST06].

9.7.2 Implementation Issues

During Step 4 of Algorithm CIRCULAR, 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 that reflect those categories. The table is updated as nodes and edges are removed from the graph. A bucket sort is initially used to place each node into its respective category. In order to keep the table updated, when 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 Figure 9.29.

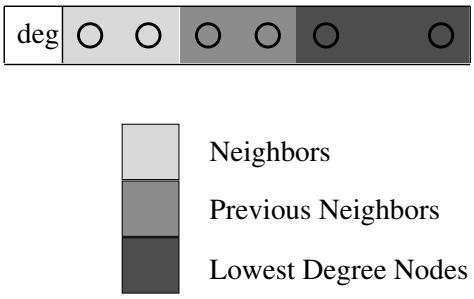

Figure 9.29 The construction of each degree list within the node table. Figure taken from [ST99, ST06].

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 we merge in the nodes of the remaining DFS tree branches. See Figure 9.30. 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 CIRCULAR will always be plane; if not, then there might be crossings. In this case, it may be possible to further reduce the number of crossings by moving nodes to a better position on the

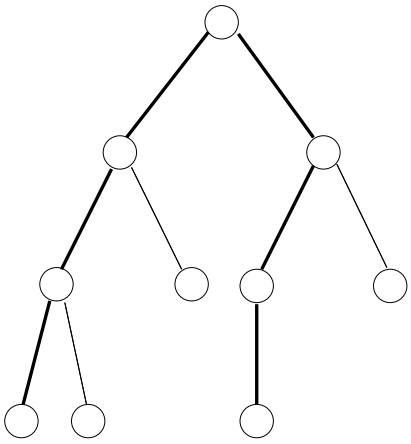

Figure 9.30 A DFS tree with the edges of the longest path designated by thick lines. Figure taken from [ST99, ST06].

embedding circle. As noted in the time complexity analysis of Algorithm CIRCULAR-Postprocessing, 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 that lie on the periphery of the embedding circle. These edges cannot possibly cause crossings. Also, in the step that 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.

9.7.3 Experimental Analysis of Algorithm CIRCULAR-with Radial

We have implemented Algorithm CIRCULAR-with Radial using Algorithm LayoutCluster1 and edge reduction postprocessing in C++ and run experiments with 11,399 graphs from [DGL+97]. The plot in Figure 9.31 shows the average number of edge crossings produced by the circular layout component of the GLT and Algorithm CIRCULAR-with Radial. As is shown by these results, the average number of crossings in the drawings produced by the technique is about 35% less than that of the GLT technique [DMM97, KMG88]. Sample drawings from both the GLT and Algorithm CIRCULAR-with Radial are shown in Figures 9.32 and 9.33.

The drawings produced by Algorithm CIRCULAR-with Radial 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 CIRCULAR-with Radial 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 Figure 9.34 for an example. Alternatively, the levels of visualization could be combined and some clusters shown in detail while others are shown with a single node.

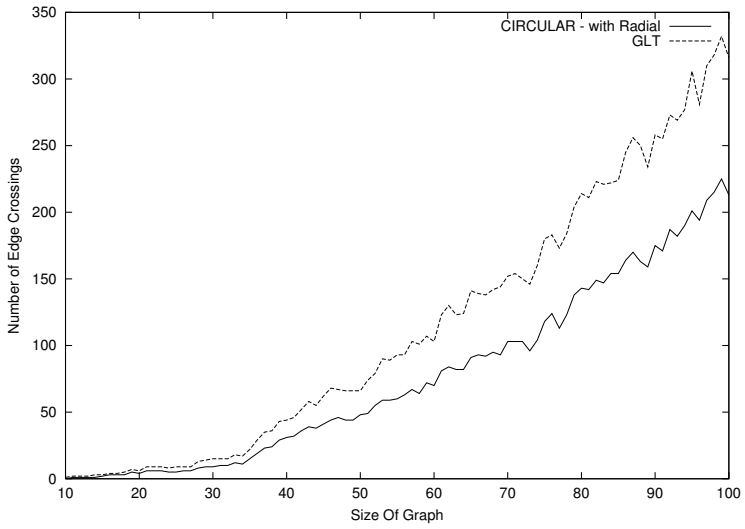

Figure 9.31 This plot shows the average number of edge crossings produced by Algorithm CIRCULAR-with Radial and the Graph Layout Toolkit when executed on 11,399 graphs from [DGL+97]. Figure taken from [ST06].

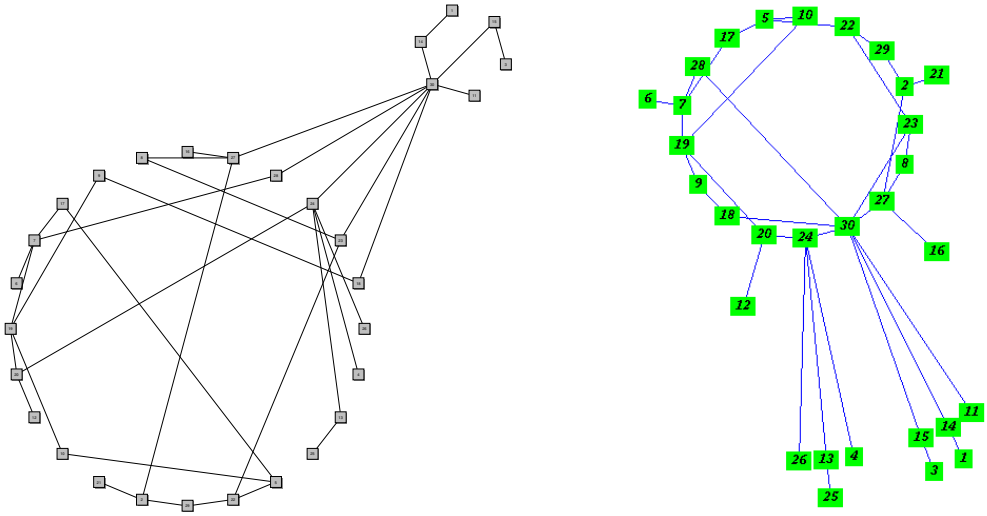

Figure 9.32 The drawing on the left is produced by the GLT and the drawing on the right is of the same graph and is produced by Algorithm CIRCULAR-with Radial. Figure taken from [ST06].

9.7.4 Implementation of Algorithm CIRCULAR-with Forces

We have implemented Algorithm CIRCULAR-with Forces so that all nodes and embedding circles are given an arbitrary initial placement. Then the force-directed equations of Section 9.6.1 are applied to the graph with the placement of group embedding circles frozen. See Figure 9.35 for a sample drawing.

An interesting behavior we noticed is that the drawing with minimal energy is not necessarily the best circular drawing. In circular drawing, a major goal is to reduce edge

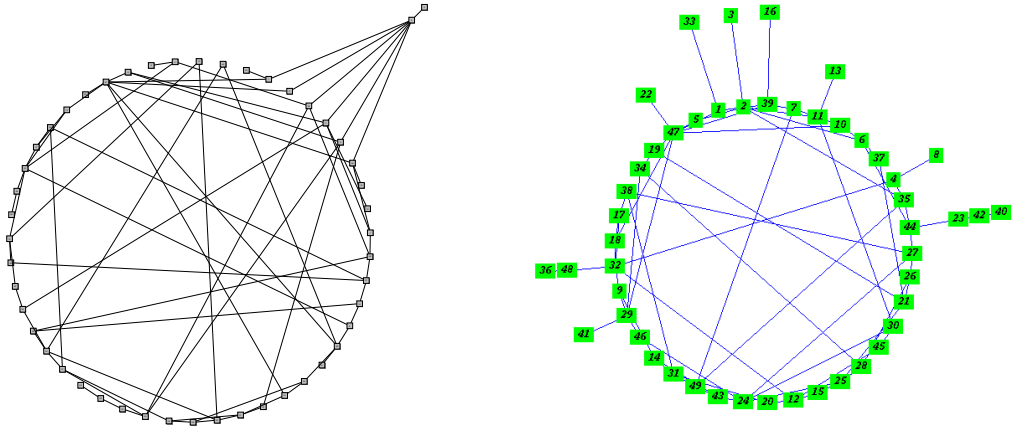

Figure 9.33 The drawing on the left is produced by the GLT and the drawing on the right is of the same graph and is produced by Algorithm CIRCULAR-with Radial. Figure taken from [ST06].

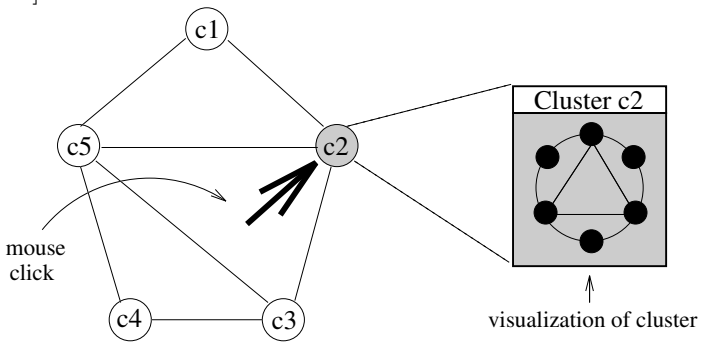

Figure 9.34 Example of interactive circular visualization. Figure taken from [ST06].

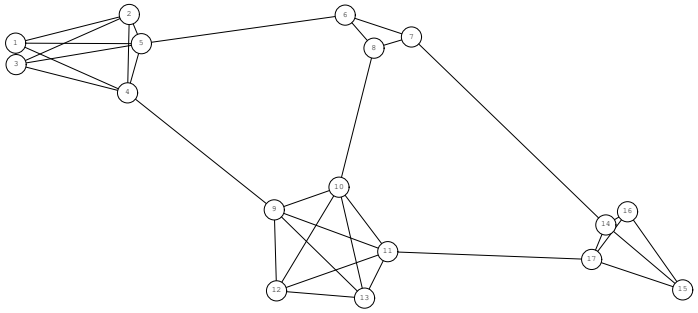

Figure 9.35 Sample user-grouped circular drawing from the preliminary implementation. Figure taken from [ST03b].

crossings. However, it is well known [DETT99] that reducing crossings sometimes means the compromise of other aesthetics, especially area, and area is related to minimum energy in spring systems. We propose adding springs from each node to its initial placement on the plane with the spring constants for these springs being high. This should keep these nodes from gravitating toward each other too much and causing extra crossings. We also suggest creating dummy nodes which are placed in the center of each embedding circle and attaching strong springs from them to every node in their respective group.

9.8 Conclusions

Circular visualizations of networks which show the inherent strengths and weaknesses of structures with clustered views are advantageous additions to many design tools.

We have presented 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 the technique works very well in practice. We have also presented 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 very well in practice. Both techniques produce drawings that clearly show the biconnectivity structure of the given graphs and also have a low number of crossings. We have also discussed a framework for creating circular graph drawings in which the grouping is defined by the user. This framework includes the successful merging of the force-directed and circular graph drawing paradigms. Algorithm CIRCULAR-with Forces is fast and produces drawings in which the user-defined groupings are highly visible, each group is laid out with a low number of edge crossings, and the number of crossings between intra-group and inter-group edges is low.

References

[BB05] M. Baur and U. Brandes. Crossing Reduction in Circular Layouts In *Proc. WG '04, LNCS 3353*, pages 332–343, 2005.

[Ber81] M. A. Bernard. On the automated drawing of graphs. In *Proc. 3rd Caribbean Conf. on Combinatorics and Computing*, pages 43–55, 1981.

[Bra97] F. J. Brandenburg. Graph clustering I: Cycles of cliques. In *Proc. GD '97, LNCS 1353*, pages 158–168, 1997.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing.* Prentice Hall, Upper Saddle River, NJ, 1999.

[DGL+97] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7:303–325, 1997.

[DMM97] U. Dogrusoz, B. Madden, and P. Madden. Circular layout in the graph layout toolkit. In *Proc. GD '96, LNCS 1190*, pages 92–100, 1997.

[Ead84] P. Eades. A heuristic for graph drawing. *Congr. Numer.*, 42:149–160, 1984.

[Ead92] P. D. Eades. Drawing free trees. *Bulletin of the Institute for Combinatorics and its Applications*, 5:10–36, 1992.

[EF97] P. Eades and Q. Feng. Multilevel visualization of clustered graphs. In *Proc. GD '96, LNCS 1190*, pages 101–112, 1997.

[EFL97] P. Eades, Q. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In *Proc. GD '96, LNCS 1190*, pages 113–128, 1997.

[EFN99] P. Eades, Q. W. Feng, and H. Nagamochi. Drawing clustered graphs on an orthogonal grid. *Jrnl. of Graph Algorithms and Applications*, pages 3–29, 1999.

[Esp88] C. Esposito. Graph graphics: Theory and practice. *Comput. Math. Appl.*, 15(4):247–253, 1988.

[GK07] E. Gansner and Y. Koren. Improved Circular Layouts. In *Proc. GD '06, LNCS 4372*, pages 386–398, 2007.

[HS04] H. He and O. Sýkora. New Circular Drawing Algorithms. Unpublished manuscript, Creative Commons License, 2004.

[HE98] M. L. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In *Proc. GD '98, LNCS 1547*, pages 107–116, 1998.

[HR88] D. Halliday and R. Resnick. *Fundamentals of Physics: 3rd Edition Extended.* Wiley, New York, NY, 1988.

[KMG88] G. Kar, B. Madden, and R. Gilbert. Heuristic layout algorithms for network presentation services. *IEEE Network*, pages 29–36, 11 1988.

[KW02] M. Kaufmann and R. Wiese. Maintaining the Mental Map for Circular Drawings. In *Proc. GD 2002, LNCS 2528*, Pages 12–22, 2002.

[Ker93] A. Kershenbaum. *Telecommunications Network Design Algorithms.* McGraw-Hill, 1993.

[Kre96] V. Krebs. Visualizing human networks. *Release 1.0: Esther Dyson's Monthly Report*, pages 1–25, February 12 1996.

[Ma88] E. Mäkinen. On Circular Layouts. In *Intl. Jrnl of Computer Mathematics*, pages 29–37, 24(1988).

[Mit79] S. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Information Processing Letters*, pages 229–232, 9(5) 1979.

[MKNF87] S. Masuda, T. Kashiwabara, K. Nakajima, and T. Fujisawa. On the NP-completeness of a computer network layout problem. In *Proc. IEEE 1987 International Symposium on Circuits and Systems, Philadelphia, PA*, pages 292–295, 1987.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[Pur97] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *GD '97, LNCS 1353*, pages 248–261, 1997.

[Six00] J. M. Six(Urquhart). *Vistool: A Tool For Visualizing Graphs*. PhD thesis, The University of Texas at Dallas, 2000.

[ST99] J. M. Six and I. G. Tollis. Circular drawings of biconnected graphs. In *Proc. of ALENEX '99, LNCS 1619*, pages 57–73, 1999.

[ST01] J. M. Six and I. G. Tollis. Effective graph visualization via node grouping. In *Proc. of IEEE InfoVis 2001*, pages 51–58, 2001. (see extended version in [ST03a])

[ST03a] J. M. Six and I. G. Tollis. Effective graph visualization via node grouping. In K. Zhang Ed., editor, *Software Visualization: From Theory to Practice, The Kluwer Intl. Series in Engineering and Computer Science Vol. 734*. Kluwer Academic Publishers, 2003.

[ST06] J. M. Six and I. G. Tollis. A framework and algorithms for circular drawings of graphs. *Jrnl. of Discrete Algorithms*, 4(1), pages 25–50, 2006.

[ST03b] J. M. Six and I. G. Tollis. A framework for user-grouped circular drawings. In *Proc of GD 2003, LNCS 2912*, pages 135–146, 2003.

[TX95] I. G. Tollis and C. Xia. Drawing telecommunication networks. In *Proc. GD '94, LNCS 894*, pages 206–217, 1995.

[YFDH01] K. Yee, D. Fisher, R. Dhamija, and M. Hearst. Animated exploration of dynamic graphs with radial layout. In *Proc. of InfoVis 2001*, pages 43–50. IEEE, 2001.