Chapter 11

Circular Drawing of Graphs

11.1 Introduction

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

A *circular graph drawing* (see Figure 11.1 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, circular drawing techniques could be added to tools which

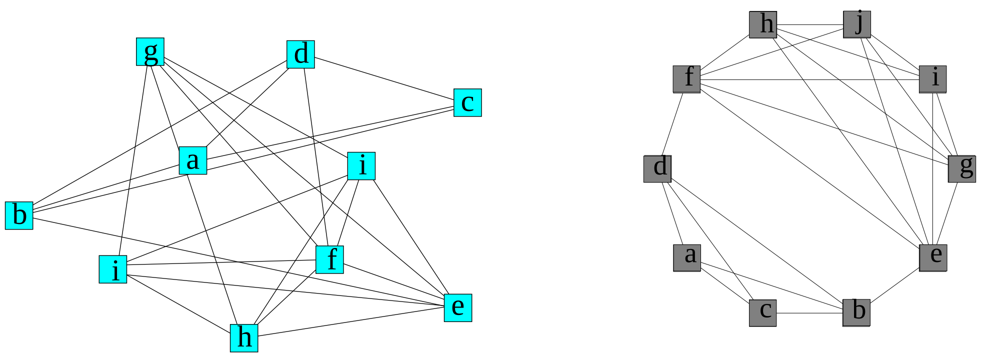


Figure 11.1: A graph with arbitrary coordinates for the nodes and a circular drawing of the same graph as produced by an implementation of our algorithm.

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. The remainder of this chapter is organized as follows: In section 2 we will discuss how to draw a graph on a single embedding circle with a low number of crossings. In section 3 and 4 algorithms for drawing graphs on multiple embedding circles will be discussed. In section 3 the circles will be determined by properties of the graph, while in section 4 we will consider user-defined groups of graphs.

11.2 Circular Drawing on a single embedding circle

In this chapter we will present algorithms that compute a permutation of vertices such that their placement on the periphery of a circle in this order will result in a low number of crossings. In general, the problem of minimizing the number of crossings is NP-complete. The algorithms we will describe guarantee that they produce a cross-free visualization if this is possible. In any other case, experimental results have shown a low number of crossings. We will divide the graphs into three categories each of which will be examined separately. The first class is trees, the second is biconnected graphs, and the third all other kinds of graphs.

11.2.1 Circular drawing of trees

When the graph $G(V, E)$ to visualize is a tree, there is always a way to place the vertices on the periphery of a circle in such a way that no two edges cross. The order of each vertex can simply be the discovery time of the vertex during a depth-first search traversal of the tree. The procedure takes $O(|V|+|E|) = (|V|)$ time since for every tree we have $|E| = |V|-1$. Figure 11.2 illustrates an example.

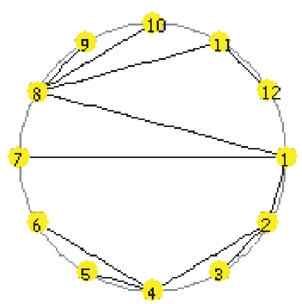


Figure 11.2: Circular drawing of a tree without crossings

11.2.2 Circular drawing of biconnected graphs

In this section, we are interested in biconnected graphs. The reason for this is that if a biconnected graph admits a visualization with no crossings, there exists an algorithm for finding this embedding. Slight modification of this algorithm leads in a general algorithm which gives a small number of crossings in the general case.

First we shall introduce the aspect of an outerplanar graph

Definition 11.1 *A graph is outerplanar iff it can be drawn on the plane such that all its vertices lie on the boundary of a single phase and no two edges cross.*

The requirement of placing all nodes on the periphery of some embedding circle is equivalent to placing all nodes on a single face (the external face) of some embedding. Thus, we would at least like to be able to recognize outerplanar graphs and draw them in a way that no crossings appear. The reason for concentrating on biconnected graphs, is that there is exactly one way to place the vertices of an outerplanar biconnected graph on the periphery of a circle without creating crossings.

Theorem 11.1 *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. It is obvious that any Hamilton circle, can be drawn on the periphery of an embedding circle in exactly one way. The ordering of the vertices has to be the same as in the hamilton circuit. In figure 11.3 we can see an example of a non-planar drawing of a hamilton circle.

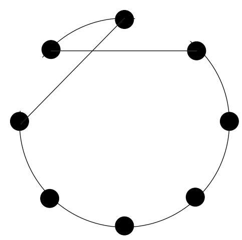


Figure 11.3: A non-planar drawing of a simple cycle.

So in order to prove the theorem it suffices to show that every outerplanar biconnected graph is hamiltonian. We will prove this by contradiction. So let us say that an outerplanar biconnected graph is not hamiltonian. Let us consider the cross-free embedding E of the graph on a circle. Then there exists at least one pair of vertices that are placed successively and are not neighbors,say u,v (figure 11.4). If we visit all vertices adjacent to u in a clockwise order starting from u, let w be the vertex we visit last. If we repeat the same procedure for vertex v but in counterclockwise order this time, the vertex we visit last has to be a vertex z appearing not later than w. If we meet z after w, it would mean that we would have at least one crossing, as the red line in figure ilustrates.But the drawing is outerplanar so this is impossible. This means that either w=z, or z appears before w, in a counterclockwise traverse of the periphery starting from v. Furthermore no edge with an endpoint in arc $\widehat{u,v}$ can have its other endpoint in $u\check{,}v$ otherwise we would have crossings. This means that if $z \neq w$ the graph is not connected, since no path connects u and v. On the other hand, if $z = w$ the removal of this vertex would result in a disconnected graph for the same reason. This means the the graph is not biconnected. Both of these cases

contradict the fact that we have a biconnected outerplanar graph.
So every outerplanar biconnected graph is hamiltonian. We know that it admits an outerplanar drawing by definition of an outerplanar graph, and this is unique because of the existence of a hamilton circuit. In addition, there is exactly one Hamilton circuit, otherwise, one of the Hamilton circuits would not be on the periphery, which according to the previous, would result in crossings. □

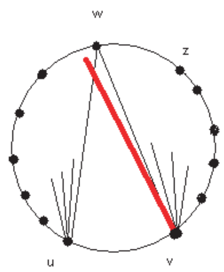


Figure 11.4: Any biconnected outerplanar graph is hamiltonian.

From the above we can see that in case of a biconnected outerplanar graph, the hamilton path has to appear on the periphery of the circle. The edges that are on the outer face are called *External/Internal* since one of the faces they belong to is the external face and the other internal. All other edges are *Internal/Internal*. This means that in order to find an outerplanar drawing it suffices to find all the *External/Internal* edges, since they build the hamilton circuit. Equivalently, we can find all *Internal/Internal* edges.

Biconnected outerplanar graphs have a series of properties that can help towards this direction.

Theorem 11.2 *[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.*

Proof
We will only prove one direction of the theorem, that for every outerplanar graph the previous criteria hold.
1. *For every outerplanar graph G,* $|E| \leqq 2|V|-3$

Consider a n-polygon. This is a hamilton circle of n vertices so it has n edges. If we select an arbitrary vertex u and draw edges towards all other vertices we will add n-3 more edges, edges towards all vertices except u and its two adjacent vertices in the hamilton circuit (figure 11.5). So we have an obviously outerplanar graph with $2|V|-3$ edges. Now

it suffices to show that no outerplanar graph can have more than $2|V| - 3$ edges. So let such a graph $G(V, E)$, $|E| > 2|V| - 3$ exist. If we add a vertex on the outer face and draw edges towards all other vertices we obtain a planar graph G'. This appears in figure 11.6 Then for the new graph G' we have:
$|V'| = |V| + 1$
$|E'| = E| + |V| > 3|V| - 3 = 3|V'| - 6$
But G' is planar, so Euler formula holds. Then $|V'| + |F'| = |E'| + 2$. We know that $|F'| < 2|E'|/3$, so $|E'| - |V'| + 2 < 2|E'|/3 - 2 \Rightarrow |E'| < 3|V'| - 6$. But this is a contradiction, so there exists no outerplanar graph with more than $2|V| - 3$ edges.

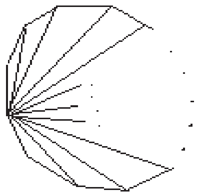


Figure 11.5: An outerplanar graph with 2n-3 edges.

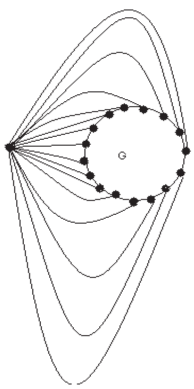


Figure 11.6: No outerplanar graph has more than 2n-3 edges.

2. *There are at least two vertices with degree at most two.*

Consider the weak dual of an outerplanar graph. This is the dual of the graph without the vertex that corresponds to the external face. The weak dual of an outerplanar graph is a tree. If it was not it would contain at least one circle. In the case of a circle appearing in the weak dual graph, the original graph can not be outerplanar. Figure 11.7 shows the case for a minimum circle (triangle). The fact that the weak dual is a tree means that it has at least two leaves. These leaves correspond to faces in the original graph with only one internal edge. So there are at least two faces which have only one internal edge each, consequently at least two external edges. This appears in figure 11.8. At least one vertex exists between u_i and u_j which has degree 2. Since we have at least two leaves in the weak dual graph, we have at least two such cases, so at least two vertices of degree at most 2.

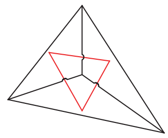


Figure 11.7: The weak dual of an outerplanar graph can not contain a circle

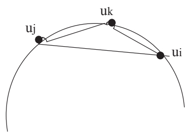


Figure 11.8: At least one vertex with degree 2

3. *No edge belongs to more than two triangles.*

We will try to create an outerplanar embedding of a graph with an edge adjacent to three triangles and find out that this is impossible. Consider a triangle T and one edge of it, namely e. We can not add a triangle with its third point in arc (u,v) or (v,w) without creating one crossing (figure 11.9). So we can only add a vertex z in (u,w) (figure 11.10). But then for the same reason as earlier, we can not add a vertex in arc(u,z) or (z,w). This means that we can add no more vertex on the periphery of the circle in order to build a triangle with e as an edge in such a way that the graph remains outerplanar.

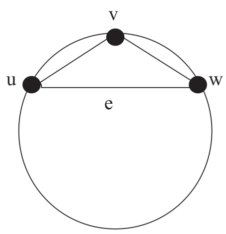


Figure 11.9: An edge belonging to one triangle

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*

Since the graph is outerplanar, there can be no edges going from a vertex in $arc(v, w)$ to some other vertex outside it, since it would create a crossing. Thus the addition of (v, w) can not create a crossing as seen in figure 11.11. Furthermore, the removal of u can not create a crossing. $\square$

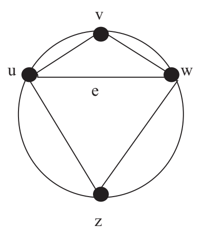


Figure 11.10: An edge belonging to two triangles

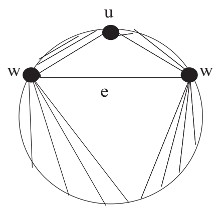


Figure 11.11: (u, v) cannot create crossings

In order to produce an outerplanar drawing of a biconnected outerlanar graph, we have to find its Hamilton circuit. Placing this on the periphery of a circle will give the required drawing. We know that both of the edges of a degree-2 vertex belong in any Hamilton circuit of the graph (if any exists). Based on this idea and the fact that any biconnected outerplanar graph has at least two degree-2 vertices, according to the previous theorem, we can start decomposing the graph removing such vertices sequentially. As stated earlier, the removal of a degree-2 vertex u followed by the addition of an edge between its neighbors (if it does not already exist), leads to a biconnected graph and the procedure can go on. If we add such an edge, we call it triangulation edge, otherwise it is called a pair edge. The edge between the neighbors of a removed vertex (added or already existing) now appears on the periphery of the circle, without being initially external. Since it is either an internal, or a triangulation (added) edge, we mark it. We repeat this procedure, until the graph is decomposed into a triangle. At this point all the internal edges have been stored and all external edges have been removed when deleting some vertex, without having been stored. The only exception can be the edges in the triangle left. If any edge (u, v) in the triangle left is internal at some point during the decomposition there was one vertex between u and v. Its removal implies that the edge between its neighbors, namely (u, v) has been marked. So all internal edges have been stored. Additionally, we have stored any triangulation edges that have been added. If we take the original graph and remove all stored edges, we obtain the Hamilton circuit. Applying a depth first search traversal will give us the required ordering of the vertices on the periphery of the embedding circle. In addition, in every iteration we prefer the degree-2 vertex we choose to be a neighbor of a previously removed vertex, if possible of the vertex removed last. We call all vertices adjacent to the last removed vertex $wavefront$ nodes, and all vertices adjacent to some previously removed vertex $wavecenter$ nodes. The names are inspired by the wave-like fashion a graph is decomposed using this procedure.

In the general case of a biconnected graph, not necessarily outerplanar, we face some problems.

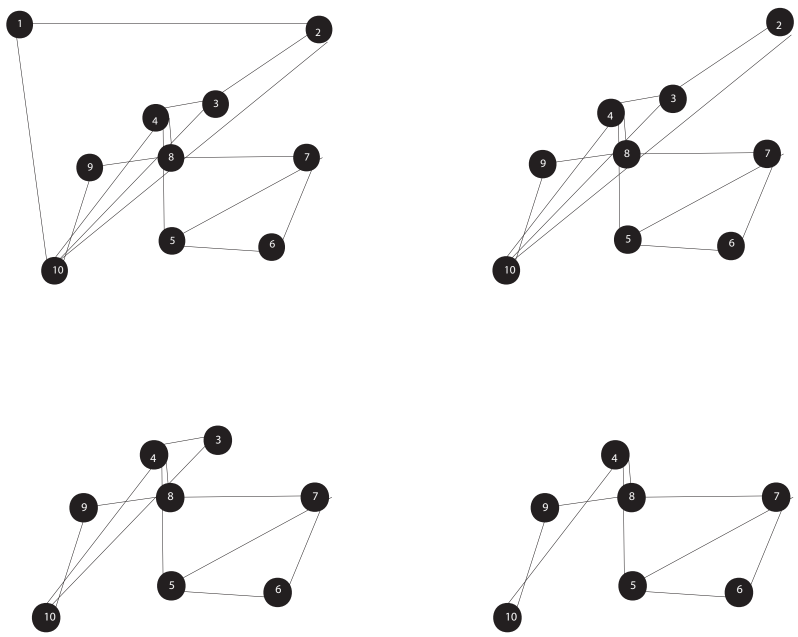
1. It is not always possible to have a degree-2 vertex. In this case we choose a lowest degree node with the following priority. A $wavefront$ node, a *wavecenter* node, any lowest degree node. Furthermore, we do not have only two neighbors. Then, in order to add and mark triangulation edges, or mark pair edges, we check the neighbors of the node to remove consecutively, rather than checking all possible combinations. In other words, the neighbors are ordered e.g. in increasing degree order, and we only check the first with the second, the second with the third and so on. Note that we do not check first with the last.

2. It is not always the case that after the decomposition and edge removal steps are completed we will the Hamilton circuit. We do not even know if one exists. But this decomposition and edge removal has given a sparser graph. We can apply some longest path heuristic, such as finding the longest path of a depth first tree, and place it on the embedding circle. Any vertices that have not been placed yet, are placed next to as many neighbors as possible (2,1,0), so as to guarantee that at least a number of edges will not produce crossings.

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 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.

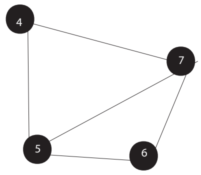
In the following figures appears an example of the execution of the algorithm.

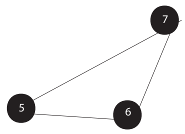


First we choose vertex 1. We check for edge(2,10), which exists. We store it and remove vertex 1. Next, we choose a lowest-degree neighbor of the removed vertex 1, which is 2. Then we check for edge (3,10) which exists. We store it and remove vertex 2.

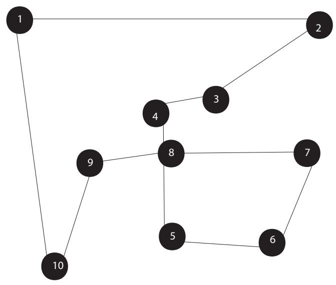
Next we select a lowest degree neighbor of vertex 2. This is vertex 3. We check for edge (4,10). It exists so we store it and remove vertex 3. Similarly we can select vertex 10 and check for edge (4,9). It does not exist. So we add edge (4,9) which is a triangulation edge, store it and remove vertex 10

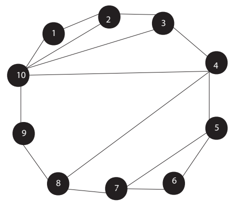
We continue choosing vertex 9, and check for edge (4,8). It exists so we store it and remove vertex 9. Next, for vertex 8 we check for edge(4,7) which does not exist. We add it to the graph and store it. After this, we remove vertex 8.





In the same way, we select vertex 4 and check for edge (5,7), which exists. So we mark. Now we have only three vertices left, so this phase of the algorithm is completed.





Now we restore the graph and remove all stored edges. Since the graph is outerplanar, we have the Hamilton circle left. A depth first search gives a hamilton path and placing the longest path (which is the path itself) gives the final result.

11.2.3 Computational Complexity

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. We postulate that $minDeg_i \leq avgDeg$ before the ith iteration, $\forall i \geq 1$ and where $avgDeg$ is the average degree of the nodes in the original graph G.

Lemma 11.1 *$minDeg_i \leq avgDeg$ before the ith iteration, $\forall i \geq 1$.*

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

Inductive hypothesis: Assume that $minDeg_i \leq avgDeg$ before the ith iteration, $\forall i \leq k$.

Inductive step: Prove $minDeg_{k+1} \leq avgDeg$ 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 kth 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 kth iteration. Hence the Inductive hypothesis guarantees that the degree of $v_{k+1} \leq avgDeg$.

2. v_{k+1} is a neighbor of v_k. In this case its degree may have increased during the kth 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 $v_{k+1} \leq avgDeg$.

□

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 $minDeg_i \leq avgDeg$ before the ith iteration, $\forall i \geq 1$, we also know that

$$\sum_{i=1}^{n-3} minDeg_i - 1 < \sum_{i=1}^{n} minDeg_i \leq \sum_{i=1}^{n} avgDeg = 2m. \qquad (11.1)$$

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

This observation means that steps 3 - 13 require $O(E)$ time, since this is where all possible pair and triangulation edges are manipulated. The bucket sorting takes O(n)time, while steps 13-16 obviously require $O(E)$ time. Finally, Step 17 also requires $O(E)$ time since at most $\sum_{i=1}^{n} deg(v_i) = O(m)$ possible placements are reviewed. Thus, the whole algorithm requires $O(E)$ time. Additionally, if we consider that in any outerplanar graph we have $|E| < 2|V| - 3$ and the fact that the algorithm draws biconnected outerplanar graphs so that no two edges cross, we conclude with the following theorem

Theorem 11.3 *Algorithm Circular produces a plane circular drawing of any outerplanar graph in $O(V)$ time.*

We can slightly modify the algorithm so that it runs faster for all non-outerplanar biconnected graphs. If a biconnected graph is not outerplanar, we can not find a cross-free embedding. We can determine at run time that a graph is not outerplanar with no additional cost if the lowest degree is greater than 2 at some time. If this is the case, we can stop adding triangulation edges, since their purpose was to find the hamilton circuit, which is of no help now. This means that during steps 3-13 we will be adding at most one edge per iteration and stop adding edges after the lowest degree becomes greater than 3. So at most $|V| - 3$ edges will be added rather than $O(|E|)$.
A similar approach is the following: We add triangulation edges whenever the vertex we choose has degree 2. The difference with the previous proposal is that if at some point the lowest degree is greater than 2, it is not sure that we will add no more edges. If at some later point due to the decomposition the lowest degree becomes 2 we will start again adding edges. Notice that in both cases pair edges are stored always, regardless of the lowest degree. Furthermore, it is important to say that in both cases outerplanar biconnected graphs are drawn in a cross-free manner since we in such graphs we always have a degree-2 vertex. Experiments have shown that the time required decreases dramatically compared to the basic algorithm, while the number of crossings is similar.

11.2.4 Circular Drawing of non-tree non-biconnected graphs on a single embedding circle

We have discussed algorithms for embedding trees and biconnected graphs on the periphery of a circle. What is left is to present an algorithm for the rest types of graphs, namely, non-tree, non-biconnected graphs. In any such case we decompose the graph into biconnected-components. From now on, we consider the graph to be at least connected. If the graph is not connected, the procedures described are repeated for each connected component and all the components are placed successively on the embedding circle.

In case of a connected graph, we can find the vertices that are responsible for non-biconnectivity using a modification of the depth first search algorithm. These vertices are called articulation or cut points. Additionally, a similar procedure can give us the biconnected components of a connected graph. If we represent each biconnected component using a pseudo-vertex, and link this vertex with the articulation point(s) that belong in the component, we will obtain a *blockcutpoint* tree. This structure is indeed a tree since the graph is connected, and cannot contain circles because this would imply the existence of two distinct paths between articulation points. Then the whole circle is a biconnected component which is against the construction procedure followed. The block cut point tree can be placed on the periphery of the circle using the algorithm described earlier. Next we replace the pseudo -vertices by the biconnected components they represent. For determining the order of the vertices of each component, we apply algorithm *Circular* in each of the components. Notice that the placement of the articulation points has already been accomplished, so we do not draw them again. Additionally since every biconnected component appears next to at least one of its articulation points in the block cut point tree, we start by placing the vertex whose order is determined by Circular to be immediately after the articulation point.

11.2.5 Computational Complexity

The algorithms for the discovery of the biconnected components and the articulation points are based on the depth first search algorithm and require $O(|E|)$ time. The cost for performing a depth first search on the block-cut point tree requires $O(|E|)$ time and the cost for applying circular on all biconnected components is $O(E_1) + O(E_2) + ... + O(E_k)$, where E_i is the number of edges of biconnected component i. Clearly $\sum(E_i) = E$, thus the total time required by the algorithm is $O(E)$

11.3 Circular Drawings of Nonbiconnected Graphs on Multiple Embedding Circles

Up to now, we have seen algorithms for drawing graphs on a single circle. In this section, we will present some techniques 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, the biconnected components and articulation points of the block-cutpoint tree can be layout with some radial drawing technique. Then, each biconnected component of the network can be drawn with a variant of Algorithm *Circular*. See Figure 11.12.

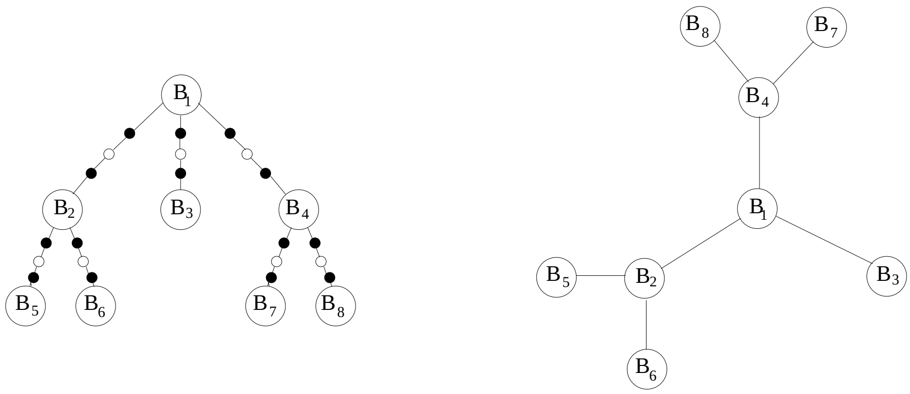


Figure 11.12: The illustration on the left shows the block-cutpoint tree of a non-biconnected graph. The small black tree nodes represent articulation points and the small white tree nodes represent bridges. The right illustration is a drawing of the same graph where the block-cutpoint tree is laid out with a radial tree layout technique.

There are several issues that need to be addressed 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

Now, we will discuss 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. 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, many approaches can be follwed 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 11.13(a). The second approach assigns the articulation point to the biconnected component which contains the most neighbors of that articulation point, see Figure 11.13(b). The third approach assigns the articulation point to a position between its biconnected components, see Figure 11.13(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.

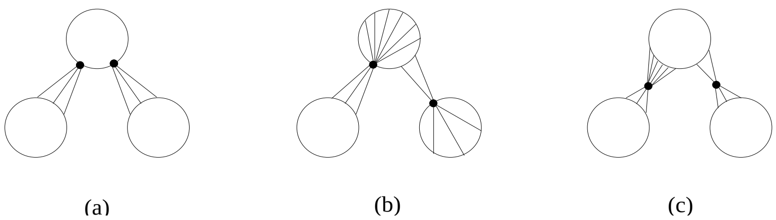


Figure 11.13: Examples of three approaches for the assignment of strict articulation points to biconnected components. The black nodes are strict articulation points.

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, a modification of the classical radial layout technique [1, 7, 8] is required:

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 $\rho(u) + \delta + \frac{d_u}{2} + \frac{max(d_1, d_2, ..., d_k)}{2}$, 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 [1, 7, 8]. This assignment of coordinates leads to a layout of the form shown in Figure 11.14.

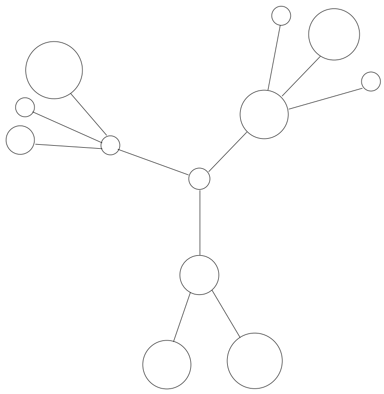


Figure 11.14: A radial layout of a tree with differing size nodes.

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 Figure 11.15. 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, we can try 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 in Figure 11.15. 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 $\frac{n_i}{2}$, where n_i is the number of nodes in B_i. In those cases, the set of ideal positions includes all the positions in

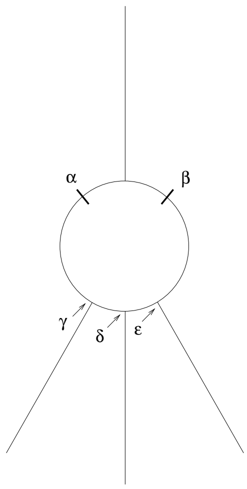


Figure 11.15: The relation between the layout of the block-cutpoint tree and the layout of an individual biconnected component.

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.

Here, we will 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. Then this drawing can be 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. This algorithm requires $O(m_i)$ time. See Figure 11.16(b) for an example.

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 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 *LayoutCluster*1. 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 Figure 11.16(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 *LayoutCluster*2 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.

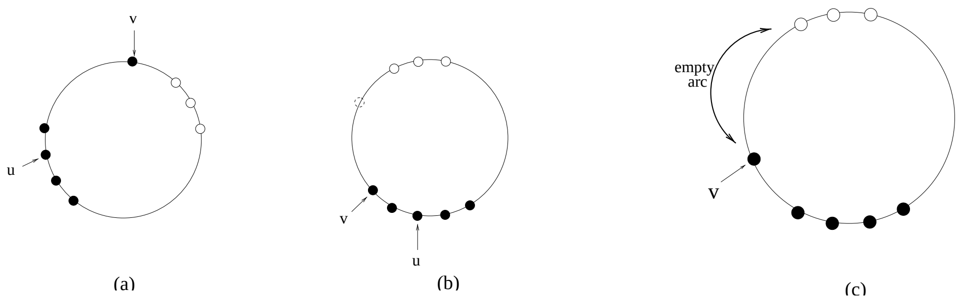


Figure 11.16: This figure demonstrates Algorithms *LayoutCluster1* and *LayoutCluster2*. The black nodes are descendant nodes and the white nodes are ancestor nodes. (a) drawing produced by Algorithm CIRCULAR; (b) the rotated drawing of part (a) produced by Algorithm LayoutCluster1. (c) the resulting drawing of part (a) produced by Algorithm LayoutCluster2.

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.

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 *LayoutCluster*1, or *LayoutCluster*2 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.

The time complexity of Algorithm $CIRCULAR{-}withRadial$ is $O(m)$ if the biconnected components are laid out with Algorithm *LayoutCluster*1 or $O(m * k)$, where k is the total number of ancestor and descendant nodes in the graph if Algorithm *LayoutCluster*2 is used. See Figure 11.17 for an example.

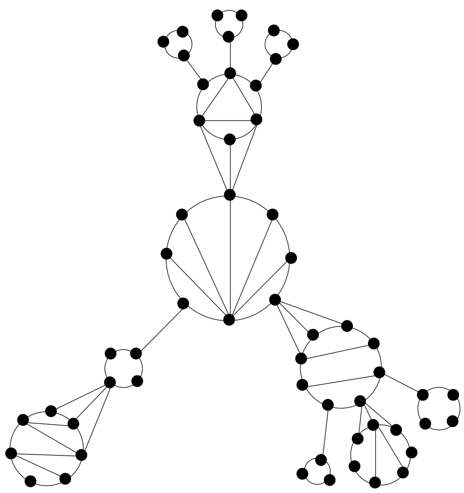


Figure 11.17: A sample drawing as produced by Algorithm $CIRCULAR - withRadial$.

11.4 Circular Drawing of user defined groups on multiple embedding circles

Regardless of the structure of a graph, there may exist interesting sets of vertices that have some important properties. Thus, it would be useful to draw graphs taking into consideration a partition of the vertices in some user-defined groups. In this case there is a number of issues that need to be addressed:

1. Each group should be clearly distinguishable and highly visible
2. A low number of edge crossings is wanted in order to have a comprehensive visualization
3. An overall aesthetic result is required
4. the overall layout technique should be fast

11.4.1 Description of the algorithm

Using one embedding circle for each group, guarantees that recognizing a group will be easy. The circular drawing techniques described in section 11.2 promise a low number of intra-group crossings and a low running time. What remains is to place each circle- group on the plane in some nice way and try to minimize the inter-group crossings in an effective way.

If we use a super-node to represent each group and add one edge between two such super-nodes whenever there exists one edge between a vertex u in the first group and another vertex v in the second group, we can obtain a super-graph $S_g = (V_g, E_g)$. This super-graph has the user-defined groups as vertices and the inter-group edges of the original graph as its set of edges. The goal now is to draw this graph in such a way that it satisfies the criteria set. It is important to notice here that each super-node must have its own size, which differs according to the number of vertices the group it represents has. So each super-node is assigned a size (radius) proportional to the number of vertices it has. Since we have no idea about the structure of the super-graph a force-directed technique can be adopted in order to place each super-vertex (group) on the plane. A simple yet effective force directed method is the simulation of a spring system. In this case, each vertex is assigned an electric load q and each edge is considered as a spring, with some strength constant k, and natural length l. The next step is to assign random coordinates to the vertices, and move them to the direction the physical forces denote until the system becomes stable.

After the termination of the algorithm, we have the coordinates of the center of each super-vertex, which is the center of each of the groups. Next we determine the radius of each group, to be proportional to the number of its vertices, and apply the suitable algorithm from those presented in 11.2. At this point, we have an aesthetic placement of the groups due to the force-directed technique. If we apply the appropriate circular drawing algorithm for each group, we will also have a drawing with a low number of intra-group crossings. What is left is some further processing that tries to minimize the number of inter-group crossings.

The problem of minimizing the inter-group crossings remains NP-complete. Thus, some heuristic approach has to be adopted. Trying to bring adjacent vertices close is the main idea behind the algorithm we will discuss. This way most of the inter-group edges will not pass through circles, and their length will become small decreasing the probability of other crossings. We can start rotating the groups one by one, since we do not want the ordering to alter, and keep the positioning that gives the lowest sum of edge lengths. Since this is only a heuristic approach with no guarantees we want it to be fast so the rotation of the groups is done sequentially rather than checking all possible combinations. If we reverse the ordering in a group (one can imagine this as putting a group in the mirror), there is no change in the group, considering edge crossings and symmetry. This means that we can follow the above described procedure for this ordering and rotate the groups once again. The sum of edge lengths is compared to the minimum found and if it is lower we can keep the ordering just produced. A request for some distance between different vertices can also be integrated. Then some force directed technique can be again used. In this case, for each different positioning produced by rotating, we measure the energy of the system and keep the positioning that gives the lowest energy. Notice that the result is different only in the case of assigning different electric loads on the vertices. Otherwise, we will always have the same amount of electric load in the same coordinates, thus the electric energy will never alter.

11.4.2 Computational Complexity

The force directed method applied on the supergraph,has no guarantee about the time limit. It depends on the number of iterations required. Each iteration takes $O(|V_g|^2 + |E_g|)$ time. Since the size of the supergraph is small, we can expect only a small number of iterations to be required until converge. Setting a limit MAX to the number of iterations provides a close - to - the - final result since only local refinement takes place after the first few iterations. The time required is $O(MAX * (|V_g|^2| + |E_e|))$. Notice that the number of vertices $|V_g|$ refers to the vertices of the super-graph, the user-defined groups of the original graph, which is expected to be much smaller than the number of actual vertices. Also $|E_g|$ refers only to the inter-group edges, which is again smaller than the total number of edges.

Applying Circular on all groups takes $O(E)$ time, as discussed in 11.2. And we rotate each group twice the number of its vertices (using the ordering found and its reverse). If we only try to minimize the edge lengths the cost is $O|V| * |E_{inter-group}|$. Otherwise, if the approach that minimizes the total energy is followed, the total time required is $O(|V| * (|V_g| + |E_g|))$

Bibliography

[1] M. A. Bernard, On the Automated Drawing of Graphs, *Proc. 3rd Caribbean Conf. on Combinatorics and Computing*, pp. 43-55, 1994.

[2] F. Brandenburg, Graph Clustering 1: Cycles of Cliques, *Proc. GD '97, LNCS 1353*, Springer-Verlag, pp. 158-168, 1997.

[3] G. Di Battista, P. Eades, R. Tamassia and I. Tollis, Algorithms for Drawing Graphs: An Annotated Bibliography, *Computational Geometry: Theory and Applications*, 4(5), pp. 235-282, 1994.

[4] G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice-Hall, 1999.

[5] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, F. Vargiu and L. Vismara, An Experimental Comparison of Four Graph Drawing Algorithms, *Computational Geometry: Theory and Applications*, 7(5-6), pp.303-26, 1997.

[6] U. Doğrusöz, B. Madden and P. Madden, Circular Layout in the Graph Layout Toolkit, *Proc. GD '96, LNCS 1190*, Springer-Verlag, pp. 92-100, 1997.

[7] P. Eades, Drawing Free Trees, *Bulletin of Inst. for Combinatorics and its Applications*, 5, 10-36, 1992.

[8] C. Esposito, Graph Graphics: Theory and Practice, *Comput. Math. Appl.*, 15(4), pp. 247-53, 1988.

[9] G. Kar, B. Madden and R. Gilbert, Heuristic Layout Algorithms for Network Presentation Services, *IEEE Network*, 11, pp. 29-36, 1988.

[10] V. Krebs, Visualizing Human Networks, *Release 1.0: Esther Dyson's Monthly Report*, pp. 1-25, February 12, 1996.

[11] S. Masuda, T. Kashiwabara, K. Nakajima and T. Fujisawa, On the NP-Completeness of a Computer Network Layout Problem, *Proc. IEEE 1987 International Symposium on Circuits and Systems, Philadelphia, PA*, pp.292-295, 1987.

[12] S. Mitchell, Linear Algorithms to Recognize Outerplanar and Maximal Outerplanar Graphs, *Information Processing Letters*, 9(5), pp. 229-232, 1979.

[13] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 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 and I. G. Tollis, Circular Drawings of Biconnected Graphs, *Proc. of ALENEX '99, LNCS 1619*, Springer-Verlag, pp. 57-73, 1999.

[16] J. M. Six and I. G. Tollis, Circular Drawings of Telecommunication Networks, *Advances in Informatics, Selected Papers from HCI '99*, D. I. Fotiadis and S. D. Nikolopoulos, Eds., World Scientific, May 2000, pp. 313-323.

[17] J. M. Six and I. G. Tollis, A Framework for Circular Drawings of Networks, *Proc. of GD '99, LNCS 1731*, Springer-Verlag, pp. 107-116, 1999.

[18] I. G. Tollis and C. Xia, Drawing Telecommunication Networks, *Proc. GD '94, LNCS 894*, Springer-Verlag, pp. 206-217, 1994.

[19] K. Yee, D. Fisher, R. Dhamija, and M. Hearst, Animated Exploration of Dynamic Graphs with Radial Layout, *Proc. of InfoVis 2001*, IEEE, pp. 43-50, 2001.