TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 01/12 pp1-16
Volume 17, Number 4, August 2012

Simple Algorithms for Network Visualization: A Tutorial*

Michael J. McGuffin**

Department of Software and IT Engineering, École de technologie supérieure, Montréal, H3C 1K3, Canada

Abstract: The graph drawing and information visualization communities have developed many sophisticated techniques for visualizing network data, often involving complicated algorithms that are difficult for the uninitiated to learn. This article is intended for beginners who are interested in programming their own network visualizations, or for those curious about some of the basic mechanics of graph visualization. Four easy-to-program network layout techniques are discussed, with details given for implementing each one: force-directed node-link diagrams, arc diagrams, adjacency matrices, and circular layouts. A Java applet demonstrating these layouts, with open source code, is available at http://www.michaelmcguffin.com/research/simpleNetVis/. The end of this article also briefly surveys research topics in graph visualization, pointing readers to references for further reading.

Key words: network visualization; graph visualization; graph drawing; node-link diagram; force-directed layout; arc diagram; adjacency matrix; circular layout; tutorial

Introduction

Networks are increasingly encountered in numerous fields of study. A wide variety of situations can be modelled using networks (i.e., graphs), and many data sets are most naturally interpreted and depicted as networks. Comprehensive surveys of techniques for network visualization are available[1,2], and an entire discipline called *graph drawing* has matured, with its own annual conference and associated surveys[3,4]. Several feature-rich software packages for network visualization are freely available, including Tulip [5,6] (http://www.tulip-software.org/), Graphviz (http://www.graphviz.org/), Gephi (http://gephi.org/), Pajek [7] (http://pajek.imfm.si/), and Cytoscape [8] (http://www.cytoscape.org/).

Despite the availability of such software, researchers, students, and others who are competent at programming may wish to implement their own network visualizations. This may be to implement a visualization on a new computing platform, or to integrate a visualization within a larger software application. It may also be to learn the details of network visualizations, possibly as the first step of a research project. Finally, certain visualization techniques, such as adjacency matrix visualization, are poorly supported by existing packages, but may be implemented from the ground up in new software.

For those wishing to implement their own visualizations, the breadth of existing surveys of techniques[1–4], covering hundreds of references, may be daunting. Furthermore, most graph drawing algorithms that compute the positions of nodes in a visualization are non-trivial to implement, and some require that multiple papers be studied before the details of a single algorithm are understood.

Fortunately, there are some basic network visualization algorithms that are easy to understand and implement. This article discusses such algorithms, and gives sufficient detail for a competent programmer to implement them. Contrary to current textbooks on visualization, this article presents a synthesis of matrix and

Received: ; Accepted:
* Supported by the Natural Sciences and Engineering Research Council of Canada
** To whom correspondence should be addressed.
E-mail: michael.mcguffin@etsmtl.ca Tel: +1-514-685-6514

non-matrix approaches for visualizing networks, showing how they can be combined, and how an ordering algorithm (the barycenter heuristic) can be used for both.

After presenting simple algorithms for computing different graph layouts, Section 6 presents simple metrics for network analysis. Finally, Section 7 surveys research topics in graph visualization with references to examples in the literature, to serve as launching points for researchers and students.

1 Force-Directed Layout of Node-Link Diagrams

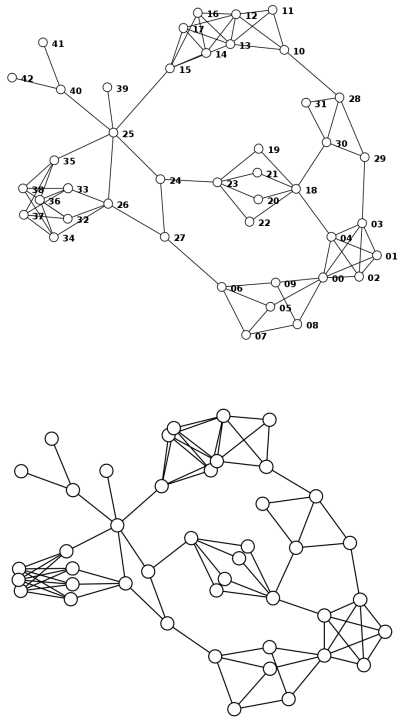

Fig. 1 Force-directed node-link diagrams of a 43-node, 80-edge network. Top: a low spring constant makes the edges more flexible. Bottom: a high spring constant makes them more stiff.

We use the term *network* as a synonym for *graph*, which can be defined as an ordered pair (N,E) of a set N of nodes and a set E of edges. In an *undirected graph*, each edge is an unordered pair of nodes, i.e., $E \subseteq \{\{x,y\}|x,y \in N\}$. Two nodes $n_1,n_2 \in N$ are *adjacent* if and only if there exists an edge $\{n_1,n_2\} \in E$, in which case n_1 and n_2 are *neighbors*. The *degree* of a node is the number of neighbors it has. In a *directed graph*, each edge is an ordered pair, i.e., $E \subseteq \{(x,y)|x,y \in N\}$, and the edge (x,y) is distinct from the edge (y,x). We will be concerned primarily with undirected graphs.

The most common graphical representation of a network is a node-link diagram, where each node is shown as a point, circle, polygon, or some other small graphical object, and each edge is shown as a line segment or curve connecting two nodes. Many sophisticated algorithms exist for computing the positions of nodes and edges in such diagrams, such as the Sugiyama-Tagawa-Toda algorithm [9], which positions nodes on the levels of a hierarchical layout. We will instead consider a class of algorithms based on *force-directed layout* [10,11] for positioning the nodes. We imagine the nodes as physical particles that are initialized with random positions, but are gradually displaced under the effect of various forces, until they arrive at a final position. The forces are defined by the chosen algorithm, and typically seek to position adjacent nodes near each other, but not too near.

Specifically, imagine that we simulate two forces: a repulsive force between *all* pairs of nodes, and a spring force between all pairs of *adjacent* nodes. Let d be the current distance between two nodes, and define the repulsive force between them to be $F_r = K_r/d^2$ (a definition inspired by inverse-square laws such as Coulomb's law), where K_r is some constant. If the nodes are adjacent, let the spring force between them be $F_s = K_s(d-L)$ (inspired by Hooke's law), where K_s is the spring constant and L is the rest length of the spring (i.e., the length "preferred" by the edge, ignoring the repulsive force).

To implement this force-directed layout, assume that the nodes are stored in an array `nodes[]`, where each element of the array contains a position `x`, `y` and the net force `force_x`, `force_y` acting on the node. The forces are simulated in a loop that computes the net forces at each time step and updates the positions of the nodes, hopefully until the layout converges to some usefully distributed positions. Fig. 1 shows the results of many iterations of such a loop. The inner body of the simulation loop could be implemented like this:

```
L = ... // spring rest length
K_r = ... // repulsive force constant
K_s = ... // spring constant
delta_t = ... // time step

N = nodes.length

// initialize net forces
for i = 0 to N-1
    nodes[i].force_x = 0
    nodes[i].force_y = 0

// repulsion between all pairs
```

```
14 for i1 = 0 to N-2
15    node1 = nodes[i1]
16    for i2 = i1+1 to N-1
17       node2 = nodes[i2]
18       dx = node2.x - node1.x
19       dy = node2.y - node1.y
20       if dx != 0 or dy != 0
21          distanceSquared = dx*dx + dy*dy
22          distance = sqrt( distanceSquared )
23          force = K_r / distanceSquared
24          fx = force * dx / distance
25          fy = force * dy / distance
26          node1.force_x = node1.force_x - fx
27          node1.force_y = node1.force_y - fy
28          node2.force_x = node2.force_x + fx
29          node2.force_y = node2.force_y + fy
30
31 // spring force between adjacent pairs
32 for i1 = 0 to N-1
33    node1 = nodes[i1]
34    for j = 0 to node1.neighbors.length-1
35       i2 = node1.neighbors[j]
36       node2 = nodes[i2]
37       if i1 < i2
38          dx = node2.x - node1.x
39          dy = node2.y - node1.y
40          if dx != 0 or dy != 0
41             distance = sqrt( dx*dx + dy*dy )
42             force = K_s * ( distance - L )
43             fx = force * dx / distance
44             fy = force * dy / distance
45             node1.force_x = node1.force_x + fx
46             node1.force_y = node1.force_y + fy
47             node2.force_x = node2.force_x - fx
48             node2.force_y = node2.force_y - fy
49
50 // update positions
51 for i = 0 to N-1
52    node = nodes[i]
53    dx = delta_t * node.force_x
54    dy = delta_t * node.force_y
55    displacementSquared = dx*dx + dy*dy
56    if ( displacementSquared
                  > MAX_DISPLACEMENT_SQUARED )
57       s = sqrt( MAX_DISPLACEMENT_SQUARED
                  / displacementSquared )
58       dx = dx * s
59       dy = dy * s
60    node.x = node.x + dx
61    node.y = node.y + dy
```

Lines 8 through 61 would be inside a loop that repeats hundreds or thousands of times, causing the nodes to move toward their final positions.

In the repulsive computation step (lines 13-29), we need to visit every pair of nodes once. Note, however, that the pair of nodes corresponding to `i1=3`, `i2=7` would be the same as that corresponding to `i1=7`, `i2=3`. Hence, to avoid visiting the same pairs twice, line 16 begins iterating at `i2=i1+1` rather than `i2=0`, to ensure `i1 < i2`.

Similarly, in the spring force step (lines 31-48), we avoid visiting the same adjacent pairs twice with line 37.

The computation of the repulsive and spring forces is inspired by physical forces (Coulomb's law and Hooke's law). However, for simplicity we do not store a velocity for each node, and the forces serve only to update the positions of nodes (lines 50-61) in a quasi-physical manner, without acceleration.

If the time step `delta_t` (used at lines 53, 54) is too small, many iterations will be needed to converge. On the other hand, if the time step is too large, or if the net forces generated are too large, the positions of nodes may oscillate and never converge. Line 56 imposes a limit on such movement. As a minor optimization, line 56 compares squares (i.e., `displacementSquared > MAX_DISPLACEMENT_SQUARED` rather than `displacement > MAX_DISPLACEMENT`), to avoid the cost of computing a square root (unless the `if` succeeds).

A minor improvement to the above pseudocode would be to detect if the distance between two nodes is zero (by adding an `else` clause to the `if` statement at line 20), and in that case to generate a small force between the two nodes in some random direction, to push them apart. Without this, if the two nodes happen to have the same neighbors, they may remain forever "stuck" to each other.

A user might interact with a force-directed layout by selecting and moving nodes with their mouse, or by using sliders to interactively adjust the values of `L`, `K_r`, `K_s`, or `delta_t`. It is not necessarily useful, however, to allow the user to adjust `K_r` and `K_s` independently. There are infinitely many pairs of (K_r, K_s) values that cause the layout to converge to the same final "shape" (i.e., the same angles between edges, differing only in edge lengths). A simpler user interface would allow the user to change a single parameter corresponding to a kind of ratio of the strength of the two forces. Taking K_r/K_s as this ratio is not ideal, however, because such a ratio is not dimensionless, and the final shape of the layout will depend on both K_r/K_s and L.

Fortunately, we can rewrite the force equations as $F_r = K_r/d^2 = K'_r/(d/L)^2$, and $F_s = K_s(d-L) = K'_s(d-L)/L$, yielding the constants K'_r and K'_s both in force units. Then, the ratio $R = K'_r/K'_s = K_r/(K_s L^3)$ is dimensionless, and can be controlled by the user with a single slider as a way of controlling the final shape of the layout. This final shape will depend only on R and be independent of L, which can also be controlled by the user to change the scale of the layout. So, given any values for

R (chosen by the user), L (possibly also chosen by the user), and K_r (having some hardcoded value), the software could compute $K_s = K_r/(RL^3)$ and simulate the forces using the updated constants to converge to a new layout. Fig. 1, top and bottom, show the result of a high and low R value, respectively. (For concreteness, Fig. 1, top, was produced with `L` = 50, `K_r` = 6250, `K_s` = 1, `delta_t` = 0.04, R = 0.05.)

Many variations on the forces used in the layout are possible. For example, rather than an inverse-square repulsion $F_r = K_r/d^2$, we could define $F_r = K_r/d^p$ with a variable exponent p. It could also be interesting to experiment with a tangential force that pushes apart the neighbors of each node n, to distribute them evenly around n (compare this idea to [12]). As another example, Noack[13] proposes a model depending on the degree of the nodes: nodes with high degree repel other nodes more strongly, helping to spread apart clusters of nodes.

In the pseudocode above, the computation of repulsive forces is a bottleneck, since it requires $O(N^2)$ time, where N is the number of nodes. This bottleneck can be eliminated by various means. For example, we could eliminate the repulsive force, and instead simulate springs of length L between all adjacent nodes, as well as springs of length $2L$ between all nodes that are two edges apart, and possibly springs of length $3L$ between nodes that are three edges apart, etc., up to some limit. (This is closely related to the approach of Kamada and Kawai[14] and Gansner et al.[15].) The extra springs would help to spread apart the network, as did the original repulsive forces. As long as the number of edges is not too high, and there aren't too many springs, the computation time may be much less than $O(N^2)$.

Also, in the above pseudocode, it is unclear how to choose the best value for `delta_t`. The GEM[16] algorithm speeds up convergence by decreasing a "temperature" parameter as the layout progresses, allowing nodes to move larger distances earlier in the process, and then constraining their movements progressively toward the end.

Fig. 2 shows a force-directed layout generated for a relatively small random graph. As can be seen, the multiple crossings of edges can make it unclear when certain edges pass close to a node or are connected to a node. Also, in such layouts where the nodes are rather closely packed, there isn't much room left to display labels or other information associated with each node. The following sections present alternative ways of depicting networks that address these problems.

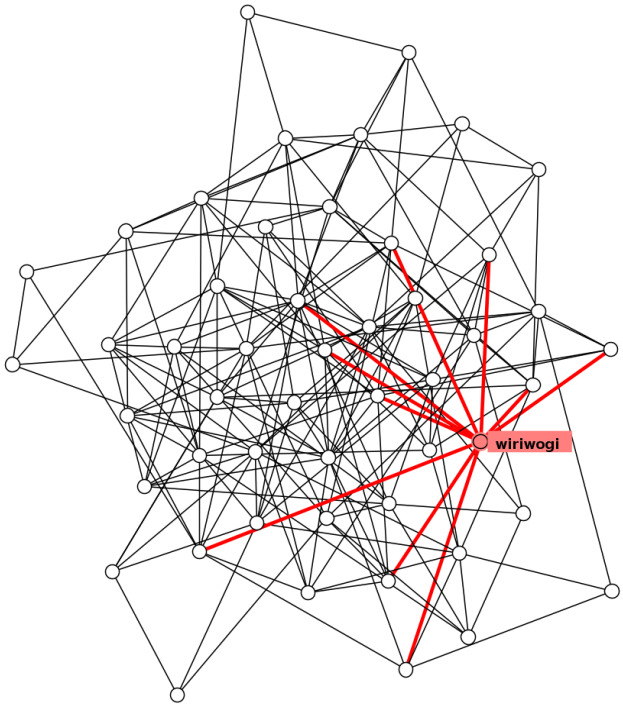

Fig. 2 Force-directed node-link diagram of a random 50-node, 200-edge graph.

2 Arc Diagrams and Barycenter Ordering

It is sometimes useful to layout the nodes of a network along a straight line, in what might be called *linearization*. With such a layout, edges can be drawn as circular arcs (Figure 3), yielding an *arc diagram*. This layout leaves much room to the right of the nodes, useful for long labels or other information to show for each node. The nodes may also be sorted in different ways.

Arc diagrams were independently discovered and proposed by Wattenberg [17] as a way of visualizing repeating substrings within a string of data, such as repeating phrases within a piece of music (http://www.bewitched.com/song.html). However, as with several other visualization techniques, an earlier example can be found in a single figure of Bertin's work [18], that shows a network with nodes on a linear layout and edges drawn as 180-degree arcs. (Interestingly, in Wattenberg's work, the thickness of arcs is varied, to show the length of substrings.)

It is important that the arcs in the diagram all cover the same angle, such as 180 degrees. This way, an arc between nodes n_1 and n_2 will extend outward by a distance proportional to the distance between n_1 and n_2,

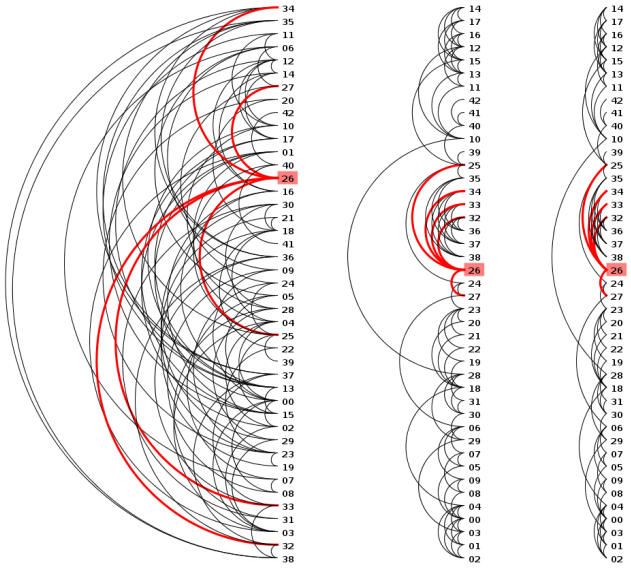

Fig. 3 Arc diagrams of a 43-node, 80-edge network. Left: with a random ordering and 180-degree arcs. Middle: after applying the barycenter heuristic to order the nodes. Right: after changing the angles of the arcs to 100 degrees.

making it easier to disambiguate the arcs. However, it is not necessary that the arcs cover a 180 degree angle. Figure 3, right, shows an arc diagram where all arcs cover 100 degrees.

To program a subroutine that draws an arc covering angle θ connecting points $A = (x, y_1)$ and $B = (x, y_2)$, we need to find the center C of the arc. Figure 4 shows a right triangle connecting A, C, and the midpoint between A and B. The length of one side of the triangle is $d = |y_1 - y_2|/2$, and we also have $\tan\theta/2 = d/e$, hence $C = (x+e, (y_1+y_2)/2)$ where $e = d/(\tan\theta/2)$.

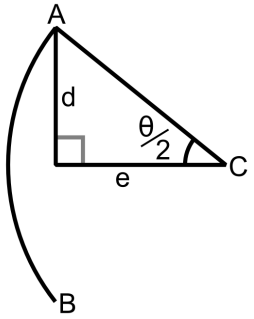

Fig. 4 An arc covering angle θ, with center C.

The nodes within an arc diagram might be sorted in various ways. For example, if each node has an associated label, and represents an object with a size, timestamp, or other attribute, the nodes in the arc diagram might be sorted alphabetically, or by size, time, etc., helping the user to analyze the network. Furthermore, every node has a degree, as well as additional metrics that can be computed (later we discuss how to compute the *clustering coefficient* and *coreness* of each node), and any of these might be used to sort the nodes within the linear ordering of an arc diagram.

We might also order the nodes to reduce the length of the arcs, making the topology of the network easier to understand. There are many algorithms for computing such an ordering (see Liiv [19] and section 4.2 of Henry [20]), however, we will discuss an easy-to-program technique called the barycenter heuristic [9,21]. The barycenter heuristic is an iterative technique where we compute the average position (or "barycenter") of the neighbors of each node, and then sort the nodes by this average position, and then repeat. Intuitively, this should move nodes closer to their neighbors, making the arcs shorter.

To implement a reordering algorithm, one approach might be to reorder the elements of the `nodes[]` array used in the previous section. However, this may not be convenient because the edges from nodes to their neighbors are typically stored as pointers, references, or indices (in the previous section, indices within `nodes[].neighbors[]`), and these would need to be updated if the nodes are relocated in memory. Furthermore, if each element of the `nodes[]` array contains additional data (like a name, color, or other metadata for the node), then reordering the array might involve moving a lot of data around the memory.

Instead, we will assume that the `nodes[]` array is fixed, and use a second data structure to store the current ordering of nodes to use for the arc diagram. Let this second data structure be an array `orderedNodes[]`, having one element for each node. We will use the term *index* to refer to a node's fixed location within `nodes[]`, and *position* to refer to the node's current location within `orderedNodes[]`. Each element of `orderedNodes[]` will store an `index` and an `average`. For example, if `orderedNodes[3].index == 7`, then `orderedNodes[3]` corresponds to `nodes[7]`, and `nodes[7]` is to be displayed at position 3 in the arc diagram. To find the index corresponding to a given position, we can simply perform a look-up in `orderedNodes[]`. To perform an inverse look-up, we define a function that computes the position `p` of a node given its index `i`:

```
function positionOfNode( i )
   for p = 0 to N-1
      if orderedNodes[p].index == i
         return p
```

Note that this function performs a linear-time search. A slightly more complicated, but much faster, implementation would cache the positions within the elements of `nodes[]` and lazily update them:

```
function positionOfNode( i )
   if orderedNodes[ nodes[i].position ].index != i
      // The cached position is not valid.
      // Update ALL the cached positions
      // so they will be valid next time.
      for p = 0 to N-1
         nodes[ orderedNodes[p].index ].position = p
   return nodes[i].position
```

Given either implementation above of `positionOfNode()`, we can implement the inner body of the barycenter heuristic like this:

```
 1 // compute average position of neighbors
 2 for i1 = 0 to N-1
 3    node1 = nodes[i1]
 4    p1 = positionOfNode(i1)
 5    sum = p1
 6    for j = 0 to node1.neighbors.length-1
 7       i2 = node1.neighbors[j]
 8       node2 = nodes[i2]
 9       p2 = positionOfNode(i2)
10       sum = sum + p2
11    orderedNodes[p1].average = sum
           / ( node1.neighbors.length + 1 )
12
13 // sort the array according to the values of average
14 sort( orderedNodes, comparator )
```

Lines 1 through 14 would be inside a loop that iterates several times, hopefully until convergence to a near-optimal ordering. Figure 3, middle, shows an arc diagram after several iterations of the barycenter heuristic to improve the ordering of nodes, thereby reducing the length of arcs with respect to Figure 3, left.

In practice, rather than converging, the algorithm sometimes enters a cycle. Thus, a limit on the number of iterations should be imposed, stopping the loop if the limit is reached (one rule of thumb is to limit the number of iterations to kN, where N is the number of nodes and k is a small positive constant). Simple ways to improve the algorithm would be to (1) detect if it has converged to an ordering that does not change with additional iterations, and in such a case stop the loop; (2) detect cycles, and similarly stop the loop.

Line 14 of the pseudocode sorts the contents of `orderedNodes[]` according to a `comparator` defined by the calling code. Typical programming environments provide an efficient $O(N \log N)$ implementation of `sort` (such as `qsort` in C, or `Arrays.sort()` in Java) that uses a client-defined comparator to determine which of a pair of array elements should appear before the other. In our case, our `comparator` should of course compare the values of `average` for any two given elements to determine their order.

The linear arrangement of nodes in an arc diagram has many advantages. As already mentioned, there is room to the right of each node for a long text label, if desired. The space to the right of nodes can also be used to display small graphics, such as line charts for each node, possibly to show a quantity associated with the node that evolves with time. TimeArcTrees [22] show changes in a graph over time by drawing multiple arc diagrams, each one at a different time, with the time axis progressing perpendicular to the layout axis of each arc diagram. Arc diagrams can also be incorporated as an axis within a larger graphic or visualization, as in [23].

Also, as mentioned, the nodes within an arc diagram can be sorted in different ways, which can be useful for seeing relationships between nodes with specific attribute values.

Despite the advantages of arc diagrams, and the room available to draw labels beside nodes, if there are too many edges that cross each other, it becomes difficult to read the edges. The next section presents an alternative visualization technique that eliminates edge crossings.

3 Adjacency Matrix Representations

An adjacency matrix (Figure 5, top) contains one row and one column for each node of a network. Given two nodes i and j, the cells located at (i, j) and (j, i) in the matrix contain information about the edge(s) between the two nodes. Typically, each cell contains a boolean value indicating if an edge exists between the two nodes. (In the figures in this article, a true boolean value is shown as a black, filled-in cell.) If the graph is undirected, the matrix is symmetric, i.e., the two cells (i, j) and (j, i) correspond to the same edge. If the graph is directed, however, the matrix is not symmetric.

Visualizing a network as a matrix has the advantage of eliminating all edge crossings, since the edges correspond to non-overlapping cells. However, in such a visualization, the ordering of rows and columns greatly influences how easy it is to interpret the matrix. Figure 5, top, has a random ordering, whereas Figure 5, bottom, has had its rows and columns ordered according

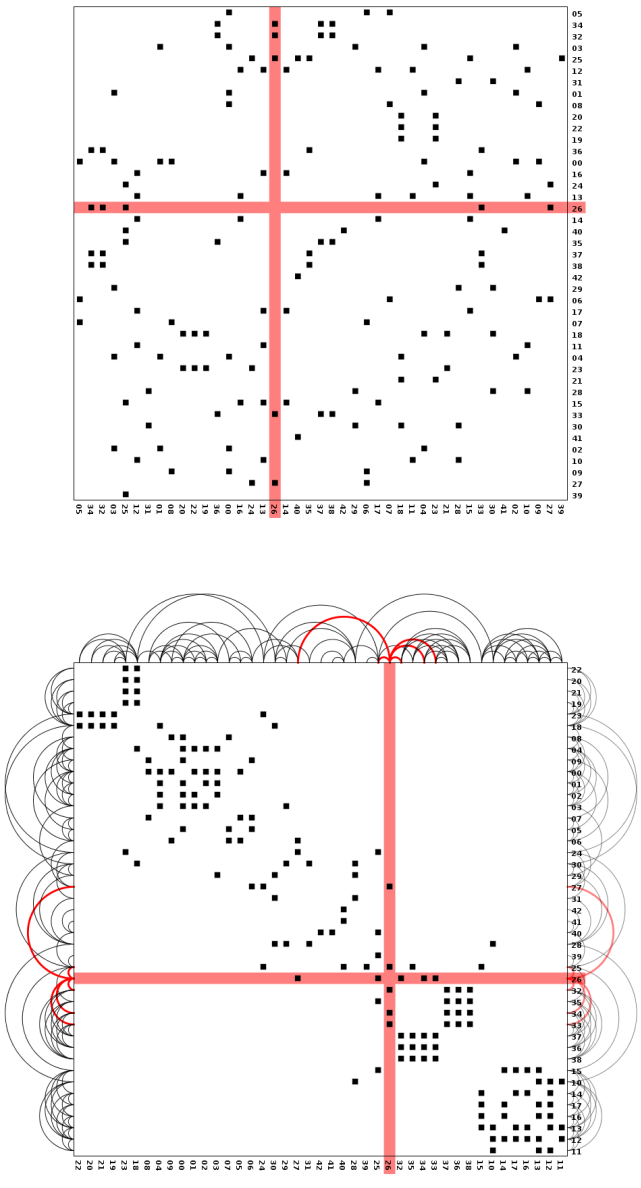

Fig. 5 Adjacency matrix visualizations of a 43-node, 80-edge network. Top: with a random ordering of rows and columns. Bottom: after barycenter ordering and adding arc diagrams. The multiple arc diagrams are redundant, but reduce the distance of eye movements from the inside of the matrix to the nearest arcs.

to the same barycenter heuristic presented in the previous section. Interestingly, by bringing nodes "closer" to their neighbors with the barycenter heuristic, this pushes the edges (filled-in matrix cells) closer to the diagonal of the matrix, making certain patterns appear in the positions of the cells.

Certain subgraphs (subsets of nodes and edges in the graph) correspond to easy-to-recognize patterns in the adjacency matrix, given an appropriate ordering of rows and columns. Figure 6 shows that *cliques* (subgraphs with all possible edges connecting the nodes) correspond to square "blocks" of filled-in cells along the matrix diagonal (with only the cells on the diagonal not filled in, since edges do not connect a node to itself). Furthermore, each *biclique* (pair of subsets of nodes with edges connecting each node in one subset with each node in the other subset) corresponds to two filled-in rectangular blocks of cells, and each cluster (subset of nodes interconnected by many edges) is recognizable as a set of filled cells along the matrix diagonal. Finally, the degree of a node is shown by the number of filled cells within the column or row corresponding to the node, as shown in Figure 5 where the degree of the highlighted node "26" is 5.

Note that the ordering in Figure 5, bottom, was generated with the barycenter heuristic, whereas that in Figure 6 was chosen manually, to make all the desired patterns visible. The visibility of patterns is very sensitive to ordering, and the barycenter heuristic does not necessarily make all such patterns visible. Worse, there may occur cases where no single ordering makes all the patterns in a network visible at the same time.

Despite these problems, Ghoniem et al. [24] demonstrated experimentally that adjacency matrices allow certain graph analysis tasks to be performed better than with node-link diagrams. However, they also found that tasks related to finding paths between nodes were more difficult with adjacency matrices. Subsequently, Henry and Fekete [25] and Shen and Ma [26] proposed visual ways to make paths within a matrix easier to see. Figure 5, bottom, and Figure 7, show Henry and Fekete's approach, called MatLink: the matrix is augmented with arc diagrams drawn along the edges of the matrix.

Henry and Fekete's MatLink visualization also allows users to select a node, and then roll their cursor over other nodes, causing the shortest path between the two nodes to be highlighted in response.

Like arc diagrams, adjacency matrices can have infor-

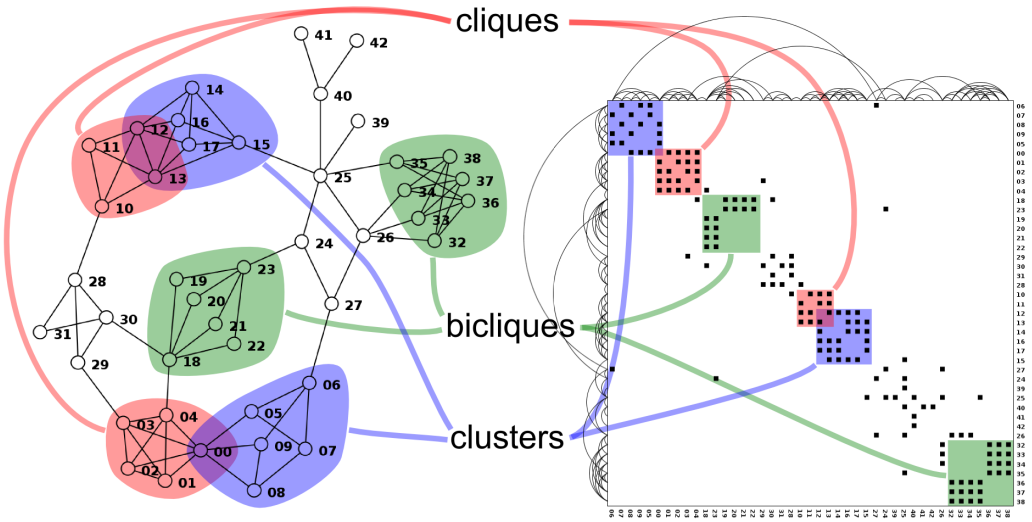

Fig. 6 Patterns corresponding to interesting subgraphs appear along the diagonal of an appropriately ordered adjacency matrix.

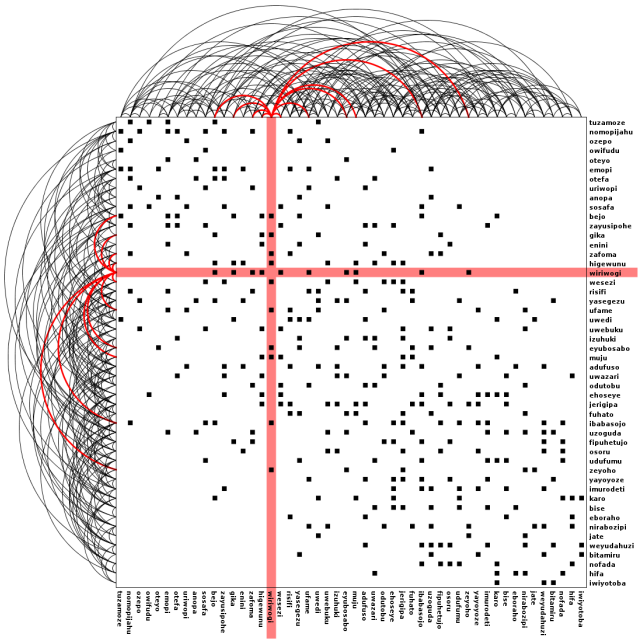

Fig. 7 MatLink visualization of a random 50-node, 200-edge graph, after barycenter ordering.

mation (such as labels) drawn beside each row or column. Matrices have the added advantage of also being able to display information related to each edge within the cells of the matrix. For example, if the edges are weighted, this weight can be shown in the color of the cell. Cells can also contain small graphics or glyphs, as in Brandes and Nick's "gestaltmatrix" [27] where each cell contains a glyph showing the evolution of the edge over time.

An important disadvantage of using adjacency matrices, however, is that the space they require is $O(N^2)$ where N is the number of nodes, as pointed out by Henry and Fekete [25]. We next present a technique that allows the labels of nodes to be drawn larger than with arc diagrams or adjacency matrices, when constrained to a window of the same size.

4 Circular Layouts

Figures 8 and 9 depict networks by positioning nodes on the circumference of a circle. As illustrated in Figure 8, drawing edges as curves rather than straight lines increases the readability of the drawings. Once again, the order chosen for the nodes greatly influences how clear the visualization is. The barycenter heuristic can again be applied to this layout, with a slight modification to account for the "wrap around" of the circular layout.

Let C be the center of the circular layout. To draw a curved arc between points A and B on the circumference, we draw a circular arc that is tangent to the lines

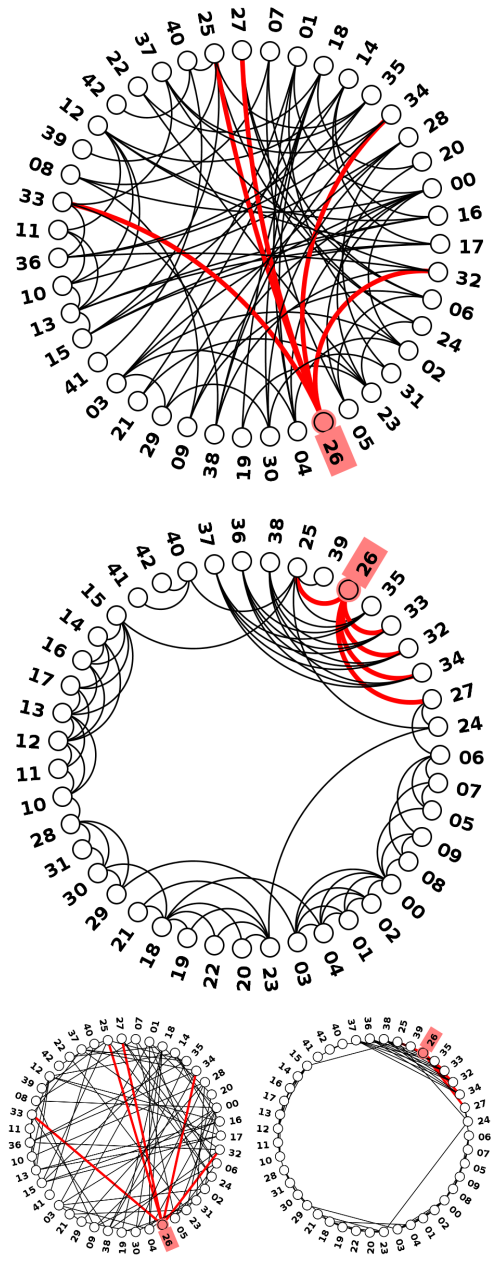

Fig. 8 Circular layouts of a 43-node, 80-edge network, before (top, and bottom left) and after (middle, and bottom right) barycenter ordering, with curved (top, and middle) and straight (bottom) edges.

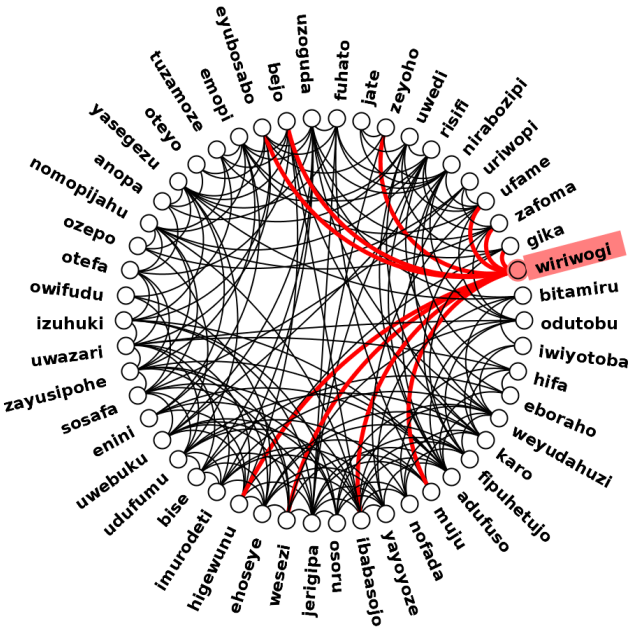

Fig. 9 Circular layout of a random 50-node, 200-edge graph, after barycenter ordering.

AC and *BC* (Figure 10). The center C' of the arc can be found by finding the intersection between a line through *A* that is perpendicular to *AC*, and a line through *B* that is perpendicular to *BC*.

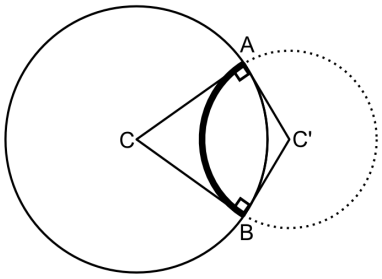

Fig. 10 *A* and *B* are two nodes connected by the arc drawn in bold. *AC* and *AC′* are perpendicular, as are *BC* and *BC′*.

To correctly adapt the barycenter heuristic to this layout, consider how to compute the "average position" of the neighbors of a node. As an example, if one neighbor is positioned at an angle of 10 degrees, and another is at an angle of 350 degrees, simply taking the numerical average yields $(10+350)/2 = 180$ degrees, whereas the intuitively correct barycenter is at 0 degrees (or, equivalently, 360 degrees). So, to correctly compute the barycenter, we do not compute averages of angles. Instead, we convert each node to a unit vector in the appropriate direction, add these unit vectors together, and find the angle of the vector sum. Define the function

`angle(p) = p*2*pi/N` giving the angle of a node at position p. Then, the pseudocode for the barycenter heuristic becomes

```
 1  // compute average position of neighbors
 2  for i1 = 0 to N-1
 3     node1 = nodes[i1]
 4     p1 = positionOfNode(i1)
 5*    sum_x = cos(angle(p1))
 6*    sum_y = sin(angle(p1))
 7     for j = 0 to node1.neighbors.length-1
 8        i2 = node1.neighbors[j]
 9        node2 = nodes[i2]
10        p2 = positionOfNode(i2)
11*       sum_x = sum_x + cos(angle(p2))
12*       sum_y = sum_y + sin(angle(p2))
13*    orderedNodes[p1].average
14*         = angleOfVector(sum_x,sum_y)
15
16  // sort the array according to the values of average
17  sort( orderedNodes, comparator )
```

The above pseudocode is very similar to the pseudocode given earlier for the barycenter heuristic. The only differences appear at lines 5-6 and 11-14, marked with stars after their line numbers. Line 14 calls a function `angleOfVector()` which simply computes the angle of a vector relative to the positive x axis, and can be implemented as:

```
function angleOfVector( x, y )
   hypotenuse = sqrt( x*x + y*y )
   theta = arcsin( y / hypotenuse )
   if x < 0
      theta = pi - theta
   // Now theta is in [-pi/2,3*pi/2]
   if theta < 0
      theta = theta + 2*pi
   // Now theta is in [0,2*pi]
   return theta
```

Many improvements to the basic circular layout are proposed by Gansner and Koren [28]. In addition, Circos [29] (http://circos.ca) is another visualization technique that uses a circular layout and curved arcs, though not specifically for visualizing network data.

5 Comparison of Layout Techniques

The following table contrasts the layout techniques according to several criteria:

	node-link diagram	circular layout	arc diagram	adjacency matrix	MatLink
Height of each node's label	$O(1/\sqrt{N})$ (best)	$O(\pi/N)$	$O(1/N)$	$O(k_1/N)$	$O(k_2/N)$ (worst)
Easy to perceive paths	yes	somewhat	somewhat	no	somewhat
Avoids edge crossings	no	no	no	yes	yes
Avoids ambiguity from edges passing close to nodes	no	yes	yes	yes	yes
Can depict an ordering of nodes	no	yes	yes	yes	yes
Can depict information about each edge	somewhat	somewhat	somewhat	yes	yes
Node labels all have the same orientation, for easier reading	yes	no	yes	yes	yes

The first row of the table quantifies the space efficiency of each layout. This is done by assuming that each layout is confined to fill the same 1×1 square, and by calculating the height of the labels on the nodes as a function of the number N of nodes. For example, in a node-link diagram, if we assume the nodes are distributed uniformly, then each node should be surrounded by an area of roughly $(1/\sqrt{N})\times(1/\sqrt{N})$ within which a label can be displayed without overlapping neighboring nodes (although such labels will, generally, overlap edges). The height of such a label, therefore, will be proportional to $1/\sqrt{N}$. The height of the labels in the other layouts is always $O(1/N)$, but with different hidden constants. In particular, in an adjacency matrix, the hidden constant is $k_1 < 1$ because margins must be reserved for the row and column labels; and with MatLink, the hidden constant is $k_2 < k_1$ since even larger margins must be reserved to display the arcs. Thus, the columns of the above table are ordered left-to-right, from best to worst space efficiency in terms of label height.

To explain the second to last row in the above table, we point out that information such as edge type or edge weight can be depicted in node-link diagrams and other non-matrix layouts by varying the color, thickness, or opacity of edges. However, this has a limited ability to convey information. Matrix-based layouts, on the other hand, can display richer information (such as a glyph) for each edge, because an entire cell is available for each edge.

Designers may thus choose a layout from the above table based on whatever criteria are most important to them. Generally speaking, node-link diagrams may often be best for showing the topology of the network in a clear and simple manner, so long as the network is not too dense. Matrix-based layouts are potentially best for dense networks, since they eliminate all inter-edge occlusion. Arc diagrams may be best for integration with other visual information, since they can be laid out along a single axis of a larger diagram. Circular layouts may be best for making labels on the nodes larger than is possible with arc diagrams.

A Java applet demonstrating these layouts, with open source code, is available at http://www.michaelmcguffin.com/research/simpleNetVis/.

6 Elementary Node Metrics

Visualizations are often enriched by computing automatic analyses of the data. For example, nodes in a layout might be colored and/or ordered according to metrics of how important or central each node is. One such metric is simply the degree $\deg(n)$ of the node n, i.e., the number of neighbors n has. Nodes with a high degree can be expected to be more important. In this section, we explain two other metrics that can be computed rather easily for each node.

The first is the *clustering coefficient* of the node, which is a measure of how interconnected the neighbors of a node are. If a node n has $k = \deg(n)$ neighbors, then there are up to $k(k-1)/2$ edges between those neighbors. The clustering coefficient of n is the fraction of such edges actually present. A clustering coefficient of zero means none are present (i.e., none of n's neighbors are neighbors of each other), and a coefficient of 1.0 means that n and its neighbors form a *complete subgraph*. If m is the number of edges present between the neighbors, the clustering coefficient is $m/(k(k-1)/2)$, and can be computed with the following algorithm:

```
function clusteringCoefficient( i )
   node = nodes[i]
   deg = node.neighbors.length
   if deg == 0
      return 0 // this is arbitrary
   if deg == 1
      return 1 // this is arbitrary
   count = 0 // num. edges present between neighbors
   for j = 0 to deg-2
      i2 = node.neighbors[j]
      node2 = nodes[i2]
      for k = j+1 to deg-1
         i3 = node.neighbors[k]
         node3 = nodes[i3]
         if edgeExistsBetween(node2,node3)
            count = count + 1
   return count / (deg * (deg-1) / 2)
```

The last metric we discuss is related to the *k-core decomposition* of a graph [30,31]. To obtain the k-core of a graph, we remove all nodes with degree lower than k, updating the degrees of remaining nodes as we remove lower-degree nodes. Figure 11 shows an example. Because all nodes in that figure initially have a degree of 1 or greater, the 1-core is the entire graph. Imagine then removing all nodes of degree 1 (such as nodes "41" and "42"), as well as all nodes whose degree has been reduced to 1 by the removal of other nodes (specifically, node "40"). When nodes can no longer be removed, we are left with the 2-core. We then remove nodes of degree 2 to obtain the 3-core, etc. Notice that node "18" initially has degree 6, but it is not part of a 6-core or even the 3-core, because its neighbors "19" through "22" are removed for having a degree of only 2, causing the degree of "18" to drop to 2 and requiring it to also be removed and excluded from the 3-core.

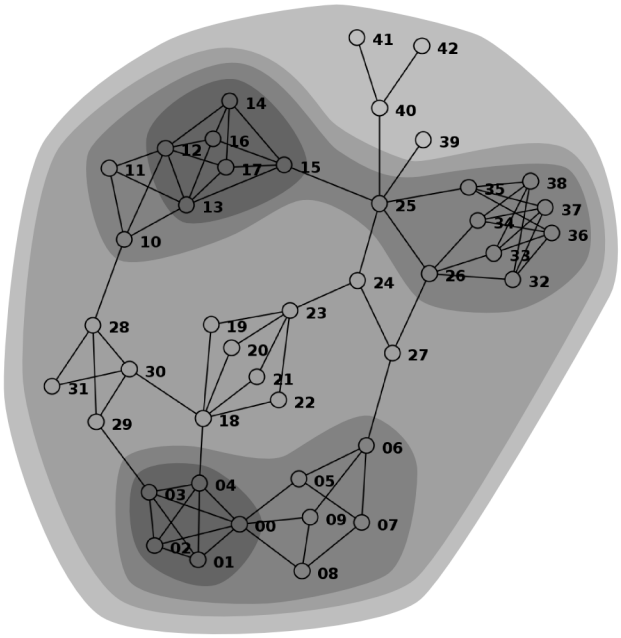

Fig. 11 *k*-**core decomposition of a graph. Shaded regions show the 1-core, 2-core, 3-core, and 4-core.**

We then define the *coreness* of a node as the highest integer k for which it is a member of the k-core. For example, the coreness of node "18" is 2, whereas the nodes "00" through "04" as well as "12" through "17" have a coreness of 4.

The three metrics mentioned, degree, clustering coefficient, and coreness, are just three of the many metrics that have been proposed in the literature. (Another often used metric is betweenness, which is more complicated to compute. An efficient algorithm for it is given by Brandes[32].) Any of these metrics can be used to classify nodes within a graph, and can be indicated using colors, or using a numerical label beside each node, or can be used to sort the nodes within an arc diagram or other ordered layout. We can also position nodes on a 2D plane by mapping one metric to the x-axis and another metric to the y-axis. Such an approach is sometimes called an attribute-driven layout, an excellent example of which is found in GraphDice [33], a system where the user may choose to map any metric or attribute to either the x or y axes.

Much more information about network analysis, es-

pecially analysis of social networks, can be found in Wasserman and Faust [34].

7 Further Reading

Readers interested in learning more are encouraged to consult von Landesberger et al. [2], which is the most recent and comprehensive survey of research on network visualization. Nevertheless, in this section we provide a brief, and necessarily incomplete, sampling of some interesting topics for research, at times citing examples of recent work.

7.1 Alternative Layouts and Visual Representations

We have already seen that the barycenter heuristic can be used to reorder arc diagrams, matrices, and circular layouts. The barycenter heuristic is only one of many reordering algorithms (Liiv [19] and section 4.2 of Henry [20]). A comprehensive analysis and comparison of different reordering algorithms, in terms of performance, convergence, quality, etc., is still lacking in the literature.

Another useful network layout not discussed in the previous sections is based on concentric circles, e.g., where the user selects some focal node(s) for the center, and other nodes are assigned to progressively larger circles based on a breadth-first traversal. Yee et al. [35] present an example of this with cleverly designed animated transitions when the focal node changes.

Attribute-driven layouts position nodes based on computed metrics and/or associated attributes of the nodes. Two recent examples include [33,36], which plot nodes within scatterplots and parallel coordinates. A closely-related way of visually representing a network is presented by Kairam et al. [37], who generate heatmaps as a function of network topology.

Many metrics can be computed on nodes to quantify the "importance" of different nodes. Examples of metrics are discussed in [2,33,34,36–38]. A comprehensive survey and comparison of graph metrics is still lacking in the literature.

Hive plots [38] are an interesting variant of the linear arc diagrams already discussed, and might inspire further variants in layout.

Researchers have also proposed hybrid approaches. For example, some hybrid approaches are used to apply the most appropriate representation to different subsets of the data. TopoLayout [39] detects subgraphs with specific characteristics and applies an appropriate node-link layout algorithm to each subgraph. Other hybrid approaches mix matrix and node-link representations [40,41] or display variants of the standard adjacency matrix [42,43].

7.2 Simplified Visual Representations

Edge bundling has received much recent attention in the literature; a few examples are [28,44–47]. Edge bundling involves routing curved edges so that they overlap and share some of their length, to reduce visual clutter. The results can be aesthetically pleasing, but often introduce ambiguity, since each edge entering a bundle may exit in many different ways.

There are also related methods for visual simplification that do not introduce any ambiguity. These include edge concentration [48], edge compression [49], the "tracks" in confluent drawings [50], and power graphs [51]. A key idea for many of these techniques is identifying bicliques in the graph and replacing each biclique with a visually simpler representation.

7.3 Interaction Techniques

Several rapid interaction techniques have been proposed for navigating through networks [52] or performing other operations via popup widgets [42,53–55], some of which involve interaction using both hands simultaneously.

Interaction techniques involving curved edges have also been explored, such as EdgeLens [56], and Edge Plucking [57]. A survey of these and related techniques is given in [58].

Multitouch interaction has become a popular topic in human-computer interaction, and a few multitouch techniques for interacting with networks have been proposed [59,60].

To date, there is no consistent and unifying user interface that combines the best of all these interaction techniques.

7.4 Dealing with Large Networks

The force-directed layout paradigm presented earlier in this article does not scale well to large graphs. To scale better with larger graphs, a common approach is to compute a hierarchical clustering of the graph and perform a multilevel layout. Archambault et al.[39] briefly survey such techniques, and propose their own hybrid layout algorithm.

Topological fisheye views [61] exploit hierarchical clustering to render a view of the network that depends interactively on the user's current focus: parts of the network further from the focus are rendered more

coarsely.

GPU programming has been used to significantly accelerate layout computations, as described by Frishman and Tal [62], allowing larger networks to be visualized.

It has also been proposed that, in some situations, the user may not be interested in seeing an overview of an entire large network. Instead, the user can be shown a single initial node, from which the user can expand outward, toward neighbors of interest, expanding the context as desired [63].

7.5 Graphs with Auxiliary Information

Graphs may be associated with additional information that can be visualized beside, or on top of, the graph. For example, one recent approach for visualizing subsets of nodes is given in [64], which also surveys other approaches for visualizing subsets. This is related to the challenge of visualizing a *hypergraph*, i.e., a graph with *hyperedges* that can be incident on more than 2 nodes each.

7.6 Graphs from Multidimensional Data

Imagine a relational data table listing sales of products, with one column for the client identifier, one column for the product identifier, and other columns for price, date, etc. Each row of the table corresponds to a sale. From such a table, we might want to generate a bipartite network of links between clients and products sold to them, or we might instead want to generate a network of clients with edges between two clients if they bought the same product. Orion [65] and Ploceus [66] are prototype software systems that allow such graphs to be easily generated from a relational database and subsequently transformed.

Work in this vein can be thought of as a complement to attribute-driven layouts: GraphDice [33] and other attribute-driven layouts [36] display a network using multidimensional visualization techniques (namely, scatterplots and parallel coordinates), whereas Orion and Ploceus visualize multidimensional data as networks.

7.7 Dynamic Graphs

Other recent work [22, 67–70] visualizes networks that change over time, using small multiples, animation, or by showing the differences between two instances of the graph (e.g., at times t and $t+1$). Some visualizations of dynamic graphs extend the adjacency matrix to show the temporal dimension [27, 71]. Others identify clusters within the graph and show overall changes of the clusters over time (e.g., merging and splitting) [72, 73], rather than emphasizing detailed topological information — an appropriate approach for large dynamic graphs.

7.8 Evaluation

Lee et al. [74] identify several tasks related to visualization and analysis of graphs. Such a reference list of tasks can help make the design and evaluation of user interfaces more systematic.

Many perceptual questions surrounding graph visualization have been investigated, and many still remain to be explored. One recent example is by Holten et al. [75], who compared different ways of depicting directed edges.

Another recent user study [76] compared layouts of node-link diagrams generated algorithmically versus those arranged manually by users.

8 Conclusion

We have presented some elementary layout and analysis algorithms, and given a brief sampling of research topics within network visualization. Hopefully, this will whet the appetite of readers, encouraging them to track down some interesting references and try their hand at programming some of the algorithms given. The graph visualization literature is already quite large and still growing, but many challenges remain, and real-world applications are found wherever graphs can be used to model relationships or data.

References

[1] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 6(1):24–43, 2000.

[2] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J.-D. Fekete, and D. W. Fellner. Visual analysis of large graphs. In *EuroGraphics: State of the Art Report*, 2010.

[3] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[4] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*. Springer, 2001.

[5] David Auber. Tulip: A huge graph visualization framework, 2004. A chapter (pp. 105–126) in Michael Jünger and Petra Mutzel, editors, Graph Drawing Software, Springer.

[6] D. Auber, D. Archambault, R. Bourqui, A. Lambert, M. Mathiaut, P. Mary, M. Delest, J. Dubois, and G. Melançon. The Tulip 3 framework: A scalable software library for information visualization applications based on relational data. Technical Report RR-7860, INRIA Bordeaux Sud-Ouest, 2012.

[7] Vladimir Batagelj and Andrej Mrvar. Pajek – program for large network analysis. *Connections*, 21(2), 1998.

[8] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13:2498–2504, 2003.

[9] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, February 1981.

[10] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[11] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[12] Chun-Cheng Lin and Hsu-Chun Yen. A new force-directed graph drawing method based on edge-edge repulsion. *Journal of Visual Languages and Computing*, 29(1):29–42, 2012.

[13] Andreas Noack. Energy-based clustering of graphs with nonuniform degrees. In *Proceedings of Symposium on Graph Drawing (GD)*, 2005.

[14] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[15] Emden R. Gansner, Yehuda Koren, and Stephen North. Graph drawing by stress majorization. In *Proceedings of Symposium on Graph Drawing (GD)*, 2004.

[16] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of Symposium on Graph Drawing (GD)*, 1994.

[17] Martin Wattenberg. Arc diagrams: Visualizing structure in strings. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis)*, pages 110–116, 2002.

[18] Jacques Bertin. *Sémiologie graphique: Les diagrammes, Les réseaux, Les cartes*. Éditions Gauthier-Villars, Paris, 1967. (2nd edition 1973, English translation 1983).

[19] Innar Liiv. Seriation and matrix reordering methods: An historical overview. *Statistical Analysis and Data Mining*, 3(2):70–91, April 2010.

[20] Nathalie Henry. *Exploring Social Networks with Matrix-based Representations*. PhD thesis, Université Paris Sud, France, and University of Sydney, Australia, 2008.

[21] Erkki Mäkinen and Harri Siirtola. The barycenter heuristic and the reorderable matrix. *Informatica*, 29(3):357–363, 2005.

[22] Martin Greilich, Michael Burch, and Stephan Diehl. Visualizing the evolution of compound digraphs with TimeArcTrees. *Computer Graphics Forum*, 28(3):975–982, 2009.

[23] A. Johannes Pretorius and Jarke J. van Wijk. Visual analysis of multivariate state transition graphs. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 12(5):685–692, 2006.

[24] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. On the readability of graphs using node-link and matrix-based representations: Controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[25] Nathalie Henry and Jean-Daniel Fekete. MatLink: Enhanced matrix visualization for analyzing social networks. In *Proceedings of IFIP TC13 International Conference on Human-Computer Interaction (INTERACT)*, pages 288–302, 2007.

[26] Zeqian Shen and Kwan-Liu Ma. Path visualization for adjacency matrices. In *Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis)*, pages 83–90, 2007.

[27] Ulrik Brandes and Bobo Nick. Asymmetric relations in longitudinal social networks. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17(12):2283–2290, 2011.

[28] Emden Gansner and Yehuda Koren. Improved circular layouts. In *Proceedings of Symposium on Graph Drawing (GD)*, pages 386–398, 2006.

[29] Martin I. Krzywinski, Jacqueline E. Schein, Inanc Birol, Joseph Connors, Randy Gascoyne, Doug Horsman, Steven J. Jones, and Marco A. Marra. Circos: An information aesthetic for comparative genomics. *Genome Research*, 2009.

[30] Vladimir Batagelj and Matjaž Zaveršnik. An O(m) algorithm for cores decomposition of networks, 2003. http://arxiv.org/abs/cs/0310049v1.

[31] Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. k-core decomposition: a tool for the visualization of large scale networks, 2005. http://arxiv.org/abs/cs/0504107v2.

[32] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[33] Anastasia Bezerianos, Fanny Chevalier, Pierre Dragicevic, Niklas Elmqvist, and Jean-Daniel Fekete. GraphDice: A system for exploring multivariate social networks. *Computer Graphics Forum*, 29(3):863–872, 2010.

[34] Stanley Wasserman and Katherine Faust. *Social Network Analysis*. Cambridge University Press, 1994.

[35] Ka-Ping Yee, Danyel Fisher, Rachna Dhamija, and Marti Hearst. Animated exploration of dynamic graphs with radial layout. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis)*, pages 43–50, 2001.

[36] Christophe Viau, Michael J. McGuffin, Yves Chiricota, and Igor Jurisica. The FlowVizMenu and parallel scatterplot matrix: Hybrid multidimensional visualizations for network exploration. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 16(6):1100–1108, 2010.

[37] Sanjay Kairam, Diana MacLean, Manolis Savva, and Jeffrey Heer. GraphPrism: Compact visualization of network structure. In *Proceedings of ACM Advanced Visual Interfaces (AVI)*, 2012.

[38] Martin Krzywinski, Inanc Birol, Steven J. M. Jones, and Marco A. Marra. Hive plots – rational approach to visualizing networks. *Briefings in Bioinformatics*, 2011. http://www.hiveplot.com/.

[39] Daniel Archambault, Tamara Munzner, and David Auber. TopoLayout: Multilevel graph layout by topological features. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 13(2):305–317, 2007.

[40] Nathalie Henry, Jean-Daniel Fekete, and Michael J. McGuffin. NodeTrix: A hybrid visualization of social networks. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 13(6):1302–1309, 2007.

[41] Sébastien Rufiange, Michael J. McGuffin, and Christopher P. Fuhrman. TreeMatrix: A hybrid visualization of compound graphs. *Computer Graphics Forum*, 31(1):89–101, 2012.

[42] Anastasia Bezerianos, Pierre Dragicevic, Jean-Daniel Fekete, Juhee Bae, and Ben Watson. GeneaQuilts: A system for exploring large genealogies. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 16(6):1073–1081, 2010.

[43] Juhee Bae and Benjamin A. Watson. Developing and evaluating quilts for the depiction of large layered graphs. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17(12):2268–2275, 2011.

[44] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 12(5):741–748, 2006.

[45] Danny Holten and Jarke J. van Wijk. Force-directed edge bundling for graph visualization. *Computer Graphics Forum (EuroVis 2009)*, 28(3):983–990, 2009.

[46] Sergey Pupyrev, Lev Nachmanson, and Michael Kaufmann. Improving layered graph layouts with edge bundling. In *International Symposium on Graph Drawing (GD)*, 2010.

[47] Ozan Ersoy, Christophe Hurter, Fernando V. Paulovich, Gabriel Cantareira, and Alexandru Telea. Skeleton-based edge bundling for graph visualization. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17(12):2364–2373, 2011.

[48] Frances J. Newbery. Edge concentration: A method for clustering directed graphs. In *Proceedings International Workshop on Software Configuration Management (SCM)*, pages 76–85, 1989.

[49] Frank van Ham, Martin Wattenberg, and Fernanda B. Viégas. Mapping text with phrase nets. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 15(6):1169–1176, 2009.

[50] Matthew Dickerson, David Eppstein, Michael T. Goodrich, and Jeremy Yu Meng. Confluent drawings: Visualizing non-planar diagrams in a planar way. In *International Symposium on Graph Drawing (GD)*, 2003.

[51] Loïc Royer, Matthias Reimann, Bill Andreopoulos, and Michael Schroeder. Unraveling protein networks with power graph analysis. *PLoS Computational Biology*, 4(7), 2008.

[52] Tomer Moscovich, Fanny Chevalier, Nathalie Henry, Emmanuel Pietriga, and Jean-Daniel Fekete. Topology-aware navigation in large networks. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI)*, 2009.

[53] Michel Beaudouin-Lafon, Wendy E. Mackay, Peter Andersen, Paul Janecek, Mads Jensen, Michael Lassen, Kasper Lund, Kjeld Mortensen, Stephanie Munck, Anne Ratzer, Katrine Ravn, Søren Christensen, and Kurt Jensen. CPN/Tools: A post-WIMP interface for editing and simulating coloured petri nets. In *Proc. International Conference on Application and Theory of Petri Nets (ICATPN)*, pages 71–80, 2001.

[54] Michael J. McGuffin and Ravin Balakrishnan. Interactive visualization of genealogical graphs. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis)*, pages 17–24, 2005.

[55] Michael J. McGuffin and Igor Jurisica. Interaction techniques for selecting and manipulating subgraphs in network visualizations. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 15(6):937–944, 2009.

[56] Nelson Wong, Sheelagh Carpendale, and Saul Greenberg. EdgeLens: An interactive method for managing edge congestion in graphs. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis)*, pages 51–58, 2003.

[57] Nelson Wong and Sheelagh Carpendale. Supporting interactive graph exploration using edge plucking. In *Proceedings of Visualization and Data Analysis (VDA)*, 2007.

[58] Nathalie Henry Riche, Tim Dwyer, Bongshin Lee, and Sheelagh Carpendale. Exploring the design space of interactive link curvature in network diagrams. In *Proceedings of ACM Advanced Visual Interfaces (AVI)*, 2012.

[59] Mathias Frisch, Jens Heydekorn, and Raimund Dachselt. Investigating multi-touch and pen gestures for diagram editing on interactive surfaces. In *Proceedings of ACM International Conference on Interactive Tabletops and Surfaces (ITS)*, 2009.

[60] Sebastian Schmidt, Miguel A. Nacenta, Raimund Dachselt, and Sheelagh Carpendale. A set of multi-touch graph interaction techniques. In *Proceedings of ACM International Conference on Interactive Tabletops and Surfaces (ITS)*, 2010.

[61] Emden R. Gansner, Yehuda Koren, and Stephen C. North. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 11(4):457–468, 2005.

[62] Yaniv Frishman and Ayellet Tal. Multi-level graph layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 13(6):1310–1319, 2007.

[63] Frank van Ham and Adam Perer. "search, show context, expand on demand": Supporting large graph exploration with degree-of-interest. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 15(6):953–960, 2009.

[64] Basak Alper, Nathalie Henry Riche, Gonzalo Ramos, and Mary Czerwinski. Design study of LineSets, a novel set visualization technique. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17(12):2259–2267, 2011.

[65] Jeffrey Heer and Adam Perer. Orion: A system for modeling, transformation and visualization of multidimensional heterogeneous networks. In *Proceedings of IEEE Visual Analytics Science and Technology (VAST)*, 2011.

[66] Zhicheng Liu, Shamkant B. Navathe, and John T. Stasko. Network-based visual analysis of tabular data. In *Proceedings of IEEE Visual Analytics Science and Technology (VAST)*, 2011.

[67] Daniel Archambault, Helen C. Purchase, and Bruno Pinaud. Difference map readability for dynamic graphs. In *Proceedings of Symposium on Graph Drawing (GD)*, pages 50–61, 2010.

[68] Daniel Archambault, Helen C. Purchase, and Bruno Pinaud. Animation, small multiples, and the effect of mental map preservation in dynamic graphs. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17(4):539–552, 2011.

[69] Loutfouz Zaman, Ashish Kalra, and Wolfgang Stuerzlinger. The effect of animation, dual view, difference layers, and relative re-layout in hierarchical diagram differencing. In *Proc. Graphics Interface (GI)*, pages 183–190, 2011.

[70] Steffen Hadlak, Hans-Jörg Schulz, and Heidrun Schumann. In situ exploration of large dynamic networks. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17(12):2334–2343, 2011.

[71] Ji Soo Yi, Niklas Elmqvist, and Seungyoon Lee. TimeMatrix: Analyzing temporal social networks using interactive matrix-based visualizations. *International Journal of Human-Computer Interaction*, 26:1031–1051, 2010.

[72] Martin Rosvall and Carl T. Bergstrom. Mapping change in large networks. *PLoS ONE*, 5(1), 2010.

[73] Khairi Reda, Chayant Tantipathananandh, Andrew Johnson, Jason Leigh, and Tanya Berger-Wolf. Visualizing the evolution of community structures in dynamic social networks. *Computer Graphics Forum*, 30(3):1061–1070, 2011.

[74] Bongshin Lee, Catherine Plaisant, Cynthia Sims Parr, Jean-Daniel Fekete, and Nathalie Henry. Task taxonomy for graph visualization. In *Proceedings of AVI workshop BEyond time and errors: novel evaLuation methods for Information Visualization (BELIV)*, 2006.

[75] Danny Holten, Petra Isenberg, Jarke J. van Wijk, and Jean-Daniel Fekete. An extended evaluation of the readability of tapered, animated, and textured directed-edge representations in node-link graphs. In *Proceedings of IEEE Pacific Visualization (PacificVis)*, pages 195–202, 2011.

[76] Tim Dwyer, Bongshin Lee, Danyel Fisher, Kori Inkpen Quinn, Petra Isenberg, George Robertson, and Chris North. A comparison of user-generated and automatic graph layouts. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 15(6):961–968, 2009.