

Integrating Edge Routing into Force-Directed Layout

Tim Dwyer, Kim Marriott, and Michael Wybrow

Clayton School of Information Technology,
Monash University, Clayton, Victoria 3800, Australia,
{tdwyer,marriott,mwybrow}@csse.monash.edu.au

Abstract. Force-directed layout is typically used to create organic-looking, straight-edge drawings of large graphs while combinatorial techniques are generally preferred for high-quality layout of small to medium sized graphs. In this paper we integrate edge-routing techniques into a force-directed layout method based on constrained stress majorisation. Our basic procedure takes an initial layout for the graph, including poly-line paths for the edges, and improves this layout by moving the nodes to reduce stress and moving edge bend points to straighten the edges and reduce their overall length. Separation constraints between nodes and edge bend points are used to ensure that node labels do not overlap edges or other nodes and that no additional edge crossings are introduced.

Keywords: graph layout, constrained optimisation, force-directed layout, edge routing

1 Introduction

Researchers and practitioners in various fields have been arranging diagrams automatically using physical “mass-and-spring” models since at least 1965 [1]. Typically, the objective of such *force-directed* techniques is to minimise the difference between actual and ideal separation of nodes [2]:

$$\text{stress}(X) = \sum_{i < j} w_{ij} (\|X_i - X_j\| - d_{ij})^2 \quad (1)$$

where w_{ij} is typically $\frac{1}{d_{ij}^2}$, X_i gives the placement in two or more dimensions of the i^{th} node and d_{ij} the ideal distance between nodes i and j .

More recent researchers have realised that one of the attractive qualities of such physical models is that the physical analogy can be easily extended to include additional aesthetic requirements by adding additional forces between objects in the drawing. For example, to preserve edge crossing properties in an initial layout Bertault [3] added a repulsive force between nodes and their projection points on edges, thus preventing nodes from passing through edges during layout. This method was only applicable to layouts with straight-line

edges and point-size nodes. Brandes et al. [4] and later Finkel and Tamassia [5] allowed the edges to bend by treating dummy nodes as control points for splines. The problem with these methods being that the splines could not easily be prevented from overlapping one another and creating new crossings.

Recently, the force-directed model has been extended to allow *separation constraints* of the form $u + g \leq v$ in each dimension between objects in the drawing [6]. The basic idea is to modify the iterative step in *functional majorization* [7] to solve a one-dimensional quadratic objective subject to the separation constraints for that dimension. Previously it has been shown that separation constraints allow aesthetic requirements such as placing nodes below other nodes in directed graphs or containment in clusters to be integrated into force-directed layout [6]. Here we show how they can be used to take into account aesthetic criteria involving edge routing.

Our basic procedure takes as input an initial layout for the graph including poly-line paths for the edges and rectangular bounding boxes for the node labels (or other text or graphics associated with nodes). This layout is then improved by moving nodes and edge bend points such that edges are straightened and made more uniform in length. Our approach is similar in spirit to that of Bertault, but instead of introducing repulsive forces between nodes and edges we introduce separation constraints between edge bend points and nodes and other edges. The result is drawings with no overlap between nodes and edges that preserve the edge crossing properties of the starting layout. Further, these drawings should have low stress with respect to the original goal function (1) while minimizing edge bends and overall edge length. Visually, the output is compact yet readable.

Our method—explored in detail in Section 2—has two main advantages over previous approaches. First, separation constraints allow us to guarantee that edge-edge crossings will not be introduced and that there will be no overlap between rectangular nodes or nodes and edges. Secondly, since our approach is based on functional majorization it has better convergence properties than the earlier approaches which were solved with iterative local-search methods similar to those introduced by Kamada and Kawai [2] or Fruchterman and Reingold [8].

A key question is how to obtain the initial layout. Bertault [3] considered input generated with a planarisation based technique [9]. Such methods seek to find a maximal planar subgraph, draw this subgraph with no edge crossings, and then use heuristics to reinsert edges whilst creating as few crossings as possible.¹ Typically, the resulting drawings have few edge crossings but are aesthetically displeasing, which is why techniques such as that of Bertault [3] or the earlier simulated annealing based beautification algorithm [11] have been suggested as a post-processing step to improve the layout. The procedure described here can also be used for this purpose. The great benefit of our approach over that of Bertault [3] is that it handles rectangular nodes which means that it must handle poly-line edges and non-overlap of nodes and non-overlap of nodes and edges.

¹ Of course there is no guarantee that this is the minimum since the general crossing minimisation problem is NP-complete [10].

In Section 3 we introduce an alternative approach for obtaining the initial layout based on initially positioning the nodes by performing a force-directed layout on the graph, ignoring edge routing. Then poly-line connectors between the nodes are routed to minimize edge length, amount of bend, and edge crossings. To do this we first use the incremental poly-line connector routing library described in [12] to find the connector routes that minimize edge length and amount of bend and then iteratively improve this using a simple greedy heuristic in which the edges with the largest number of crossings are re-routed to reduce the overall number of edge crossings as long as this does not lead to very long edges or a large number of bends, as seen in Figure 6. The advantage of our technique is that the heuristic considers number of edge crossings, edge length, number of bends and degree of bendiness. Clearly, all of these measures are important [13, 14] and since there is a trade-off between them it is important to consider them together. As far as we know this is the first approach to do so since previous approaches have either been planarization based and only considered the number of crossings [9], or not considered crossings at all [15, 16, 12].

2 Modelling Edge Routing in Force-directed Layout

We assume as input a graph $G = (V, E)$ and a layout for the graph. Each node $v \in V$ has a bounding box of dimensions given by $width(v)$ and $height(v)$, and each edge e has a minimum width $width(e)$. The routing for each edge consists of a curved or piece-wise linear path between and around node bounding boxes. We use the functions $xpos(e, h)$ which returns a list of intersection points between the edge e and the line $y = h$; $top(e)$ and $bottom(e)$ which return the topmost and bottom-most coordinate, respectively, through which e passes. The functions $ypos(e, h)$, $leftlimit(e)$ and $rightlimit(e)$ are defined symmetrically.

Recall from [7] that in functional majorisation the value of the stress function (1) is reduced by alternately minimising quadratic forms in the horizontal and vertical axes that bound the stress functions:

$$x^T Lx - l_x^T x \quad , \quad y^T Ly - l_y^T y \tag{2}$$

where: x and y are $|V|$ dimensional vectors of node positions in each axis; the $|V| \times |V|$ Hessian matrix L is taken as the graph Laplacian; and the linear arguments $l_{x,y}$ are computed before processing each axis based on the difference between ideal separation of nodes and their actual separation at the current placement (for details see [7]).

The input to our new layout problem includes bend points for some edges. Ideally we would like such edges to be straightened out while still satisfying the original node separation objectives of the goal function and without creating any node/edge, node/node intersections. The following Lemma gives us quadratic terms that can be added to the objective functions (2) at each stress-majorisation iteration to achieve this.

Lemma 1. *Given points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ and $b = (x_b, y_b)$, the angle between lines $\overrightarrow{p_1b}$ and $\overrightarrow{bp_2}$ is maximised by adjusting the x coordinates of these three points to minimise:*

$$f(x_1, x_2, x_b) = (x_b - (1 - t_b)x_1 - t_b x_2)^2 \quad (3)$$

where the constant parameter t_b gives an ideal position for b along the straight line from p_1 to p_2 and can be determined from the (fixed) y positions of these points:

$$t_b = \frac{y_b - y_1}{y_2 - y_1} \quad (4)$$

where we require that $0 \leq t_b \leq 1$.

Proof. (sketch) These results are trivially derived from the parameterisation of the straight line from p_1 to p_2 : $p_1 + t_b(p_2 - p_1)$. The smaller the value of expression (3) the closer b is to this line. \square

So for an edge $(v_i, v_j) \in E$ with one bend at position b we can straighten the edge when moving vertices and bend points horizontally by minimising $f(x_i, x_j, x_b) = (x_b - (1 - t_b)x_i - t_b x_j)^2$, provided the initial bend angle is obtuse. If an edge is routed through multiple bend points $b[1], b[2], \dots, b[n]$ we can straighten all bends by minimising:

$$f(x_i, x_{b[1]}, x_{b[2]}) + \sum_{k=2}^{n-1} f(x_{b[k-1]}, x_{b[k]}, x_{b[k+1]}) + f(x_{b[n-1]}, x_{b[n]}, x_j)$$

and similarly, when placing points vertically, we will straighten edges by minimising an equivalent set of expressions over y . The condition that for each bend b , $0 \leq t_b \leq 1$ is guaranteed by the fact that all edges are routed around rectangular nodes. Note that a side effect of straightening edges is to shorten their edge.

Now the Hessian matrix of the second-order partial derivatives of f is:

$$H(f(u, v, b)) = \nabla^2 f(u, v, b) = 2 \cdot \begin{pmatrix} (1 - t_b)^2 & t_b(1 - t_b) & t_b - 1 \\ t_b(1 - t_b) & t_b^2 & -t_b \\ t_b - 1 & -t_b & 1 \end{pmatrix} \quad (5)$$

Thus, if B is the full set of bends (or potential bends, as will be discussed later) we have to define $|B|$ new variables and to redefine the goal functions (2) in terms of $m = |V| + |B|$ variables. We define a new Hessian matrix A that encompasses the quadratic terms for all bends as well as ideal node separation terms of the graph Laplacian:

$$A = L' + \sum_{b \in B} A_b \quad (6)$$

where L' is an $m \times m$ matrix with the top-left $|V| \times |V|$ cells set to L and the remaining cells 0; for each bend $b \in B$, A_b is a symmetric matrix with the 9

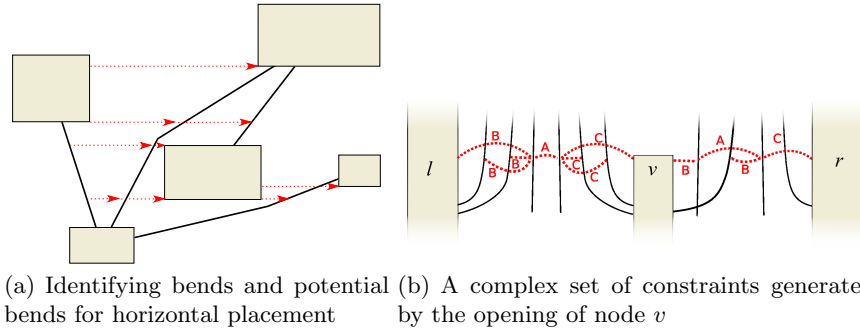


Fig. 1. Examples showing the separation constraints (dashed arrows) required to prevent the creation of new node/node and node/edge crossings when placing nodes horizontally.

cells corresponding to the variables u, v, b set to the entries in $H(f(u, v, b))$ as above and all other cells 0. This gives us a new goal function to minimise in each dimension. For example in the x dimension we have:

$$x^T Ax + l'^T x \tag{7}$$

Where the linear terms in (2) are unaffected by the new quadratic terms so l' is simply an m -vector s.t. $l' = [l|0, \dots]$. It is simple to prove that A is symmetric and positive semi-definite meaning that efficient convex optimisation methods such as the gradient-projection method described in [6] are applicable. Further, since each bend point requires only a small number of entries in the A matrix the complexity of the optimisation process will increase only linearly in the number of additional bend variables.

To find a placement in each dimension that minimises bend angles as well as the usual separation based on connectivity criteria, we solve this quadratic objective subject to a set of separation constraints. These constraints prevent:

- bend points and hence edges from overlapping any node’s bounding box;
- nodes overlapping one another;
- bend points passing through another edge and so increasing the number of edge crossings;
- and guarantee a minimum separation between parallel edges.

Figure 1 shows the constraints generated for horizontal placement. Note that we introduce bends in some straight edges. We create a new bend-point in an edge wherever the top or bottom of a node is visible from that edge where nodes restrict visibility but edges do not. In our implementation we can give different weighting for the penalty for dummy bends and that for non-dummy bends, meaning that we can strongly penalise bending a previously straight edge.

The number of separation constraints is clearly linear in the number of nodes and bend-points. However, since we may introduce up to $|V|$ dummy bends in each edge, the worst-case number of constraints is $O(|V| \cdot |E|)$.

Figure 2 gives an algorithm for generating these constraints for the x -direction: the code for the y -direction is symmetric. We generate the constraints using a line-sweep algorithm related to standard rectangle overlap detection methods [17] and the non-overlap constraint generation algorithm [18]. We use a vertical sweep through the nodes and edges, keeping a horizontal “scan line” list of open nodes sorted by horizontal position and an unsorted list of open edges. When the scan line reaches the top of a new node, v , this is added to the list and its left and right node neighbors, l and r , computed. The function *neighbourhood_x_constraints* generates bend variables and constraints for all edges between l and r for the opening or closing of v . Note that deciding whether to create a constraint between a particular pair of variables requires considering several cases. These are illuminated in Figure 1(b).

The current algorithm is reasonably straightforward but has some obvious inefficiencies. The worst case complexity could be improved by delaying generation of constraints on a line until all nodes have been inserted, and keeping a sorted list of line segments and adding line segment crossings as a new kind of event.

The procedure *generate_x_constraints*(V, E) has worst-case complexity in $O(|V|(|V| + |E| \log |E|))$ time. Furthermore, it will generate $O(|V| \cdot (|V| + |E|))$ constraints.

3 Computing the Initial Layout

The algorithm given in the previous section requires an initial layout including node positions and poly-line routes for edges. One approach would be the use of a planarisation based technique [9]. These seek to find a maximal planar subgraph, draw this in a planar way, and then reinsert edges whilst producing few crossings. However, algorithms for drawing strictly planar graphs (or subgraphs) generally require further refinement, an example is shown in Figure ???. Our approach to computing an initial edge routing with minimal crossings is novel, so we describe it in more detail. It has four main steps.

The first step is to position the nodes by performing a force-directed layout on the graph, ignoring edge routing. A method such as [18] can be used to remove node overlap, allowing edges to be routed between neighbouring nodes.

Step two is to use the incremental poly-line connector routing library described in [12] to compute poly-line routes for each edge, which minimise edge length and amount of bend. This is done by constructing a visibility graph for the nodes, itself containing a node for each vertex of the bounding box of each node in the original graph. The visibility graph contains an edge between two nodes iff they are mutually visible, i.e., there is no intervening obstacle. Next, the edge paths are routed using an A^* based-search to find the best route for each edge. The cost of routing each edge is $O((|E| + |V|) \log |V|)$ where $|E|$ is the number of edges in the visibility graph.

Step three uses a simple greedy heuristic to reduce the number of edge crossings. Each edge with crossings is considered once, in decreasing order of crossings. Again we use an A^* based-search to find the best route for each edge, though this

```

procedure generate_x_constraints( $V, E$ )
   $C \leftarrow \emptyset$ 
   $Events \leftarrow \{(top(v), Open, v), (bottom(v), Close, v) | v \in V\}$ 
     $\cup \{(top(e), Open, e), (bottom(e), Close, e) | e \in E\}$ 
  sort  $Events$  by decreasing  $y$  position
  #  $OpenNodes$  is maintained in order of each node's  $x$  position
   $OpenNodes \leftarrow \emptyset$ 
   $OpenEdges \leftarrow \emptyset$  %  $OpenEdges$  is unordered
  for each  $(y, type, obj) \in Events$  do
    if  $obj$  is a node then
       $v \leftarrow obj$ 
       $l \leftarrow$  first node to left of  $v$  in  $OpenNodes$ 
       $r \leftarrow$  first node to right of  $v$  in  $OpenNodes$ 
       $Z \leftarrow Z \cup neighbourhood\_x\_constraints(v, l, r, y, OpenEdges)$ 
    if  $type = Open$  then
      if  $obj$  is a node then
        insert( $obj, OpenNodes, xpos(obj)$ )
      else if  $obj$  is an edge then
        add  $obj$  to  $OpenEdges$ 
      endif
    else if  $type = Close$  then
      delete  $obj$  from  $OpenNodes$  /  $OpenEdges$ 
    endif
  endfor
return  $Z$ 

procedure neighbourhood_x_constraints( $v, l, r, y, OpenEdges$ )
  # If  $l$  (or  $r$ ) is unspecified we say  $l$ (or  $r$ ) = 0 and include all edges to the left (or right) of  $v$ .
   $L \leftarrow \emptyset, minx = -\infty$ 
  if  $l \neq 0$  then
     $L \leftarrow \{l\}, minx = xpos(l)$ 
  endif
   $L \leftarrow L \cup \{dummyNode(e, x, y) | e \in OpenEdges \wedge x \in xpos(e, y)$ 
     $\wedge minx < x < xpos(v) \wedge e$  is not connected to  $l$  or  $v\} \cup \{v\} \cup \dots$ 
  ... similarly add dummy nodes for edges between  $v$  and  $r$  including  $r$  (if  $r \neq 0$ )
   $u_A \leftarrow u_B \leftarrow u_C \leftarrow 0$ 
  for each  $w \in L$  in ascending  $xpos$  order do
    A: if  $edge(w) \wedge \neg end(w) \in \{v, l, r\}$  then
      if  $u_A \neq 0$  then
         $Z \leftarrow Z \cup \{xpos(u_A) + (width(u_A) + width(w))/2 \leq xpos(w)\}$ 
       $u_A \leftarrow w$ 
      else if  $node(w)$  then
         $u_A \leftarrow 0$ 
      endif
    B: if  $edge(w)$  then
      if isend( $u_B, edge(w)$ ) then
         $skipList \leftarrow skipList \cup \{w\}$ 
      else
        for each  $s \in skipList$  do
           $Z \leftarrow Z \cup \{xpos(s) + (width(s) + width(w))/2 \leq xpos(w)\}$ 
         $skipList \leftarrow \emptyset$ 
      endif
      else if  $node(w)$  then
         $skipList \leftarrow \{w\}$ 
         $u_B \leftarrow w$ 
      endif
    for each  $w \in L$  in descending  $xpos$  order do
      C: # symmetrical to B
      # May need to also generate constraints  $l, v$  and  $v, r$  if necessary
  return  $Z$ 

```

Fig. 2. Vertical scan algorithm to create a set Z of constraints to prevent node/edge or node/node overlap when moving nodes horizontally. The procedures *generate_y_constraints* and *neighbourhood_y_constraints* for the horizontal scan are symmetrical. Constraints generated for the three distinct cases A , B and C are illustrated in Figure 1(b).

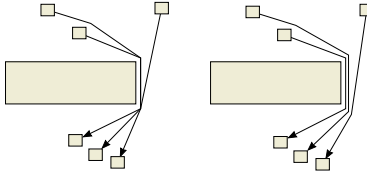


Fig. 3. An exaggerated example of the nudging we perform on shared edge bend points.

time the cost includes a penalty for each crossing. Setting a very high penalty produces routings with few crossings. We have the ability to adjust this penalty to create more pleasing routings such as the one shown in Figure 6(b). In this case, the four crossings at the perimeter are avoided but the one in the middle is allowed since it would otherwise result in a path of significantly greater length and amount of bendiness. The cost of routing each edge taking into account edge-crossings is currently $O(|S|(|E| + |V|) \log |V|)$ where $|E|$ is the number of edges in the visibility graph and $|S|$ the number of segments in the routed edges.

The penalty for an edge route is simply the sum of the penalties for its segments where the penalty for an edge segment is given by:

$$cost = l + \alpha s + \frac{\beta a \log(a + 1)}{10} + \gamma scc \quad (8)$$

where: α, β, γ are user specifiable penalties. They represent, respectively, the segment penalty, the angle penalty and the crossing penalty. l is the length of the segment. a is the angle away from a straight line that this segment makes with the previous segment, scaled to the range $0 \leq a \leq 10$, therefore $a = 0$ means the two segments make a straight line. If this is not the first segment and $a > 0$, then $s = 1$, otherwise $s = 0$. Finally, scc is the number of crossings for the current segment.

The use of a routing method of this type allows us to adjust the initial routing produced by setting penalties that equate to our desired combination of aesthetic criteria. The layout step is then able to preserve and improve these characteristics of the diagram, that have been shown to be of importance for user comprehension of graphs [14].

Finding poly-line edge crossings for a graph is not as simple as determining the crossings between all segments of all edge paths. There are several special cases that must be considered. For example, two edges will appear to cross if they share a common bend point. In this case we want to differentiate between the case where they cross and the case where they only touch. Similarly, it is common for poly-line edges routed using a visibility graph to share paths for part of their route, i.e., running along the same paths in the visibility graph. Again, we wish to penalise only the edges with crossings on their shared paths. Dealing with both of these cases requires comparing the order of edges entering and leaving shared paths or common bend points. The algorithm used for this purpose is described in Figure 4.


```

procedure crossingCount( $A, B$ )
   $crossings \leftarrow 0$ 
  for each segment ( $a1, a2$ ) in edge  $A$  do
    for each segment ( $b1, b2$ ) in edge  $B$  do
      if ( $a1, a2$ ) and ( $b1, b2$ ) are the same line segment then
        # We notice crossings for shared paths when they diverge, so skip.
        continue
      else if ( $a2 = b2$ ) or ( $a2 = b1$ ) or ( $a1 = b2$ ) then
        # The segments can share endpoints in four ways. We ignore three.
        continue
      else if  $a1 = b1$  then
        # The segments share an endpoint
         $a0 \leftarrow$  point on previous segment ( $a0, a1$ ) of  $A$  if it exists
         $b0 \leftarrow$  point on previous segment ( $b0, b1$ ) of  $B$  if it exists
        if  $a2 = b0$  then
          # In terms of  $A$ , the shared path is ahead. Skip it.
          continue
        else if ( $a0 = b0$ ) or ( $a0 = b2$ ) then
          # In terms of  $A$ , we are leaving a shared path.
          if  $A$  and  $B$  are connected to the same node then
            # Don't want to the count case where edges route from or to a common end point.
            continue
          endif
          # Note: the shared portion of the path can go in the forwards or backwards direction along  $B$ .
           $bL \leftarrow$  last shared point on  $B$  (will be  $b0$  or  $b2$ )
           $endSide \leftarrow$  cornerSide( $a0, a1, a2, bL$ )
          Scan backwards (or forwards) through  $A$  and  $B$  until reaching the beginning of the shared path.
           $aS \leftarrow$  point on  $A$  leading into the shared path.
           $bS \leftarrow$  point on  $B$  leading into the shared path.
           $aS2 \leftarrow$  the first point along the shared path.
           $aS3 \leftarrow$  the second point along the shared path.
           $startSide \leftarrow$  result of cornerSide( $aS, aS2, aS3, bS$ )
          if  $startSide \neq endSide$  then
            # The shared path contains a crossing.
            increment  $crossings$ 
          endif
        else
          # The edges  $A$  and  $B$  cross or touch at this point.
           $preSide \leftarrow$  cornerSide( $a0, a1, a2, b0$ )
           $postSide \leftarrow$  cornerSide( $a0, a1, a2, b2$ )
          if  $preSide \neq postSide$  then
            # The edges cross at this point.
            increment  $crossings$ 
          endif
        endif
      endif
    else
      if segments ( $a1, a2$ ) and ( $b1, b2$ ) intersect then
        # Standard line segment intersection.
        increment  $crossings$ 
      endif
    endif
  endfor
endfor
return  $crossings$ 

procedure cornerSide( $c1, c2, c3, p$ )
  #  $c1, c2, c3$  is a line  $C$  with two segments.
  if point  $p$  is to the left of line  $C$  then
    return 1
  else if point  $p$  is to the right of line  $C$  then
    return -1
  endif
  return 0

```

Fig. 4. Poly-line edge crossing detection algorithm.

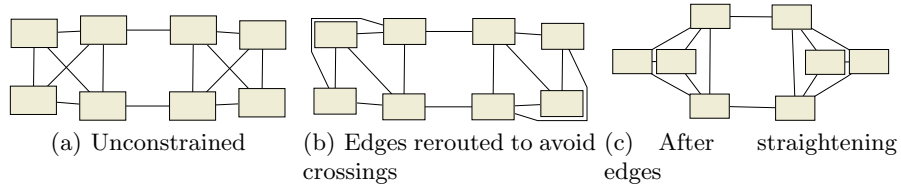


Fig. 5. A small example showing progress from unconstrained layout to layout with crossings removed and edge bends straightened.

Finally, in step four, line segments are adjusted to slightly separate edges routed around the same corner of a node. This involves nudging the bend points of edges along shared paths, or at the points they cross or touch, as shown in Figure 3. To do this a sorted order for each of these points is kept when determining crossings. Such nudging prevents the creation of additional edge crossings during the layout step.

4 Discussion

Figure 5 demonstrates how the edge routing and straightening procedures are applied in practice. We begin with the output of an unconstrained stress-majorisation layout. Edges are all close to their ideal length thus minimising (1). We apply edge routing using Algorithm 4 to penalise routes with crossings. A planar layout is obtained but with longer edges and a number of bends. Constraints are then generated as per Algorithm 2 and the stress majorisation layout is rerun subject to these constraints. The result retains the planar layout but with edges straightened.

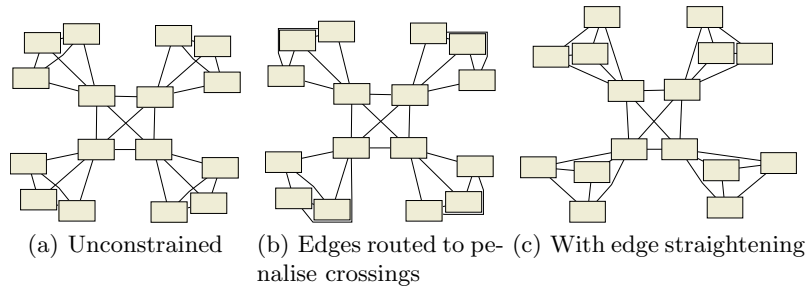


Fig. 6. An example graph from Kamada-Kawai [2].

Figure 6 shows an example graph from Kamada and Kawai [2]. Note that the graph is planar, yet our routing allows one crossing to remain because removing the final crossing would require a very long edge. The result is a layout with more

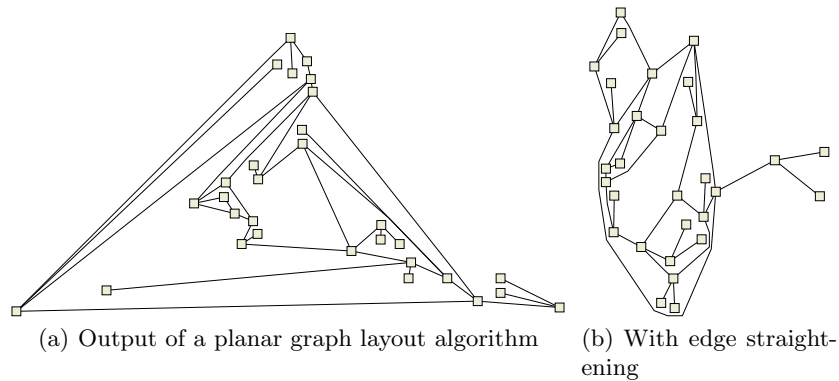


Fig. 7. Applying edge straightening to the output of a planar layout algorithm.

consistent edge length and greater symmetry than a completely planar drawing would permit.

Figure 7 shows the result of applying the edge straightening method directly to the output of a typical planar graph layout algorithm. The layout is considerably compacted by allowing edges to bend while the edge routing topology is preserved. Note that the layout is somewhat compressed by the requirement to preserve the ideal distance between two nodes on the outer face that have finished up on opposite sides of the drawing. This could perhaps be alleviated by a further refinement step that relaxed ideal distances between nodes connected by an edge following a long route.

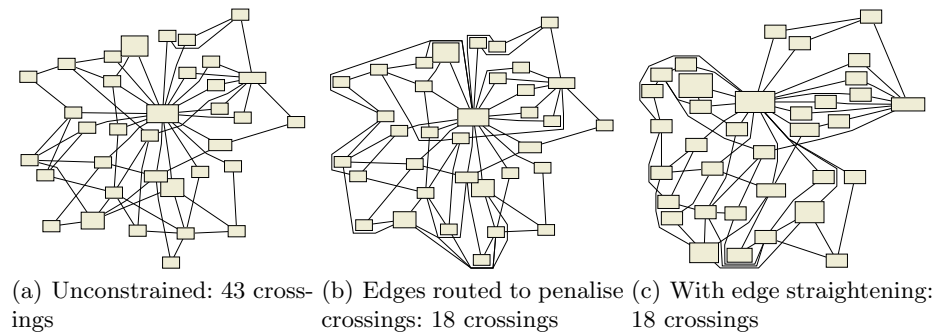


Fig. 8. A more realistic graph visualisation application (a bayesian net).

Our final example in Figure 8 is more typical of the type of graphs encountered in actual applications, namely: nodes have variable size bounding boxes depending on the labels required; there are many more edges than our previous examples; and there are a small number of nodes of high degree, e.g. the two highest degree nodes have 19 and 8 connections. In the final layout edges that share a common path have been separated using the method described in

Section 3. Note that we have not applied such a separation where edges share a common end point—the main goal of nudging being to disambiguate routes that come together and then diverge.

The entire process including computation of an initial unconstrained layout, routing of all edges and then the straightening phase takes only a few seconds for each of the examples shown, but our implementation is currently far from optimal. We expect that using sparse matrix techniques to compute the stress majorisation, as discussed in Section 2 will greatly decrease run times.

5 Conclusion

Extending stress majorisation techniques to handle separation constraints allows us to naturally handle a wide number of aesthetic criteria and drawing conventions. Here we have shown that separation constraints allow edge routing to be integrated into force-directed layout. The precise encoding is not obvious, and relies on using separation constraints in combination with a modification to the one-dimensional quadratic objective function to essentially model an arbitrary linear inequality.

Another contribution of the paper is to present a simple heuristic for finding poly-line edge routes that tries to minimise the number of edge crossing while still taking into account the edge length and degree of bendiness. There has been surprisingly little research into such heuristics and we believe that there is considerable scope for more research.

We think it is significant that our algorithm can be used to improve layouts obtained with planarization based methods. We plan to further explore how force-directed layout can be combined with such combinatorial techniques.

References

1. Fisk, C.J., Isett, D.D.: ACCEL: automated circuit card etching layout. In: DAC'65: Proceedings of the SHARE design automation project, ACM Press (1965) 9.1–9.31
2. Kamada, T., Kawai, S.: An algorithm for drawing general undirected graphs. *Information Processing Letters* **31** (1989) 7–15
3. Bertault, F.: A force-directed algorithm that preserves edge crossing properties. In: Proc. 7th Int. Symp. on Graph Drawing (GD'99). Volume 1731 of LNCS., Springer (1999) 351–358
4. Brandes, U., Wagner, D.: Using graph layout to visualize train interconnection data. In: Proc. 6th Int. Symp. on Graph Drawing (GD'98). Volume 1547 of LNCS., Springer (1998) 44–56
5. Finkel, B., Tamassia, R.: Curvilinear graph drawing using the force-directed method. In: Proc. 12th Int. Symp. on Graph Drawing (GD'04). Volume 3383 of LNCS., Springer (2004) 448–453
6. Dwyer, T., Koren, Y., Marriott, K.: IPSEP-COLA: An incremental procedure for separation constraint layout of graphs. In: Proc. IEEE Symp. on Information visualisation (Infovis'06). (To appear 2006)

7. Gansner, E., Koren, Y., North, S.: Graph drawing by stress majorization. In: Proc. 12th Int. Symp. Graph Drawing (GD'04). Volume 3383 of LNCS., Springer (2004) 239–250
8. Fruchterman, T., Reingold, E.M.: Graph drawing by force-directed placement. *Software - Practice and Experience* **21** (1991) 1129–1164
9. Gutwenger, C., Mutzel, P., Weiskircher, R.: Inserting an edge into a planar graph. In: SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2001) 246–255
10. Garey, M.R., Johnson, D.S.: Crossing number is NP-complete. *Journal of Algebraic Discrete Methods* **4** (1983) 312–316
11. Harel, D., Sardas, M.: Randomized graph drawing with heavy-duty preprocessing. In: AVI '94: Proceedings of the workshop on Advanced visual interfaces, New York, NY, USA, ACM Press (1994) 19–33
12. Wybrow, M., Marriott, K., Stuckey, P.J.: Incremental connector routing. In: Proc. 13th Int. Symp. on Graph Drawing (GD'05). Volume 3843 of LNCS., Springer (2006) 446–457
13. Purchase, H.C., Cohen, R.F., James, M.: Validating graph drawing aesthetics. In: Proc. 4th Int. Symp. on Graph Drawing (GD'96), Springer (1996) 435–446
14. Ware, C., Purchase, H., Colpoys, L., McGill, M.: Cognitive measurements of graph aesthetics. *Information Visualization* **1** (2002) 103–110
15. Dobkin, D.P., Gansner, E.R., Koutsofios, E., North, S.C.: Implementing a general-purpose edge router. In: Proc. 5th Int. Symp. on Graph Drawing (GD'97). Volume 1353 of LNCS., Springer (1997) 262–271
16. Freivalds, K.: Curved edge routing. In: FCT '01: Proceedings of the 13th International Symposium on Fundamentals of Computation Theory. Number 2138 in LNCS, London, UK, Springer (2001) 126–137
17. Preparata, F.P., Shamos, M.I. In: *Computational Geometry*. Springer (1985) 359–365
18. Dwyer, T., Marriott, K., Stuckey, P.: Fast node overlap removal. In: Proc. 13th Int. Symp. on Graph Drawing (GD'05). Volume 3843 of LNCS. (2006) 153–164