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. The typical use of force-directed layout is 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 nodes 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], for example:

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

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} is the ideal distance between nodes i and j based on the graph path length between them.

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 the edge crossings 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

M. Kaufmann and D. Wagner (Eds.): GD 2006, LNCS 4372, pp. 8–19, 2007.

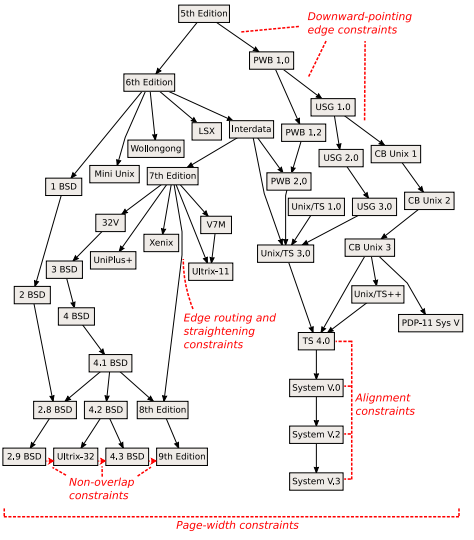


Fig. 1. A directed graph drawn using constrained force-directed layout. Separation constraints are used to ensure: (1) directed edges point downwards; (2) selected nodes are horizontally or vertically aligned; (3) the drawing fits within the page boundaries; and (4) nodes do not overlap edges or other nodes. Allowing constraints between nodes and edges means that edge routing criteria can also be considered, e.g., keeping edges as straight as possible while avoiding unnecessary crossings. The "history of unix" graph data is from `http://www.graphviz.org`.

Tamassia [5] allowed the edges to bend by treating dummy nodes as control points for splines. The problem with these methods is 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 $u + g \leq v$, enforcing a minimum gap g between the positions u and v of pairs of objects in either the x or y dimensions in the drawing [6]. The basic idea is to modify the iterative step in *functional majorisation* [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 placement of nodes below other nodes in directed graphs or containment of nodes in clusters—to be integrated into force-directed layout [6]. In this paper 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 so 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, and which 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.

Our method—explored in detail in Section 2—has three 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 nodes or between nodes and edges. Second, 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]. Third, as illustrated in Figure 1, we can use additional separation constraints to enforce other aesthetic criteria in the layout.

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 post-processing steps to improve the layout. The procedure described here can also be used for this purpose.

In Section 3 we introduce an alternative approach for obtaining the initial layout. Our technique first positions the nodes by performing a force-directed layout on the graph, ignoring edge routing. Next, we use the incremental connector routing library described in [12] to find the connector routes that minimize edge length and amount of bend. We then iteratively improve this using a simple greedy heuristic, re-routing the edges with the largest number of crossings to reduce crossings as long as this does not lead to very long edges or a large number of bends, as seen in Figure 7. 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 have 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.

[1] Of course there is no guarantee that such reinsertion strategies produce a drawing that is optimal with respect to crossings since the general crossing minimisation problem is NP-complete [10].

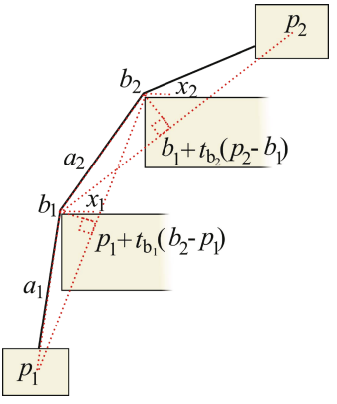


Fig. 2. The bend b_1 will be straightened to its projection point on the line $\boldsymbol{p_1b_2}$, e.g., in the x-dimension, to x_1. Bend b_2 will be similarly straightened towards the line $\boldsymbol{b_1p_2}$. The potential bend points a_1 and a_2 will be straightened to the line segment between actual bend points or the ends of the line, i.e., a_1 will be straightened to the line segment $\boldsymbol{p_1b_1}$.

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 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]).

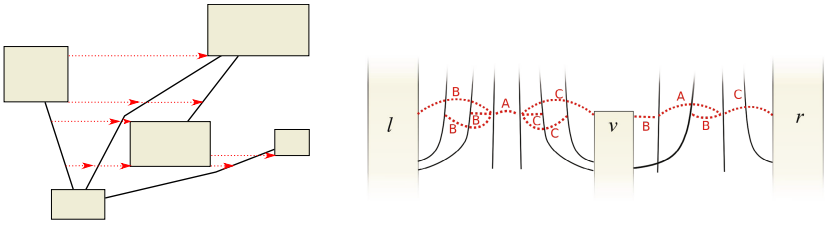
The input to our new layout problem includes bend points for some edges. Ideally we would like such edges to be straightened while still satisfying the original node separation objectives of the goal function and without creating any node/edge, node/node intersections.

Consider a bend point $b = (x_b, y_b)$ on a line from $p_1 = (x_1, y_1)$ to $p_2 = (x_2, y_2)$. The minimal change to the position of b which straightens the line is to move b to its projection p_b on to the line $\boldsymbol{p_1p_2}$. We let t_b be distance of p_b along $\boldsymbol{p_1p_2}$, that is, $p_b = p_1 + t_b(p_2 - p_1)$ where $t_b = \frac{\boldsymbol{p_1b} \cdot \boldsymbol{p_1p_2}}{||\boldsymbol{p_1p_2}||^2}$ (from the dot product rule for scalar projection) Since all paths have minimal length and bends we have that the projection point must lie between p_1 and p_2, i.e. that $0 \leq t_b \leq 1$.

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 $f(x_i, x_j, x_b) = (x_b - (1 - t_b)x_i - t_b x_j)^2$. If an edge is routed through multiple bend points $b[1], b[2], ..., b[n]$ 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. This is illustrated in Figure 2.



(a) Identifying bends and potential bends for horizontal placement

(b) A complex set of constraints generated by the opening of node v

Fig. 3. Examples showing the separation constraints (dashed arrows) required to prevent the creation of new node/node and node/edge crossings when moving nodes horizontally

In summary, for the three variables u, v, b involved in each bend b, we have $f(u, v, b) = (u, v, b)^T H(u, v, b)$ where

$$H(f(u,v,b)) = \nabla^2 f(u,v,b) = 2 \cdot \begin{pmatrix} (1-t_b)^2 & t_b(1-t_b) & t_b - 1 \\ t_b(1-t_b) & t_b^2 & -t_b \\ t_b - 1 & -t_b & 1 \end{pmatrix} \tag{3}$$

Thus, if B is the 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 matrix A that contains the quadratic terms for each bend and the ideal node separation terms of the graph Laplacian:

$$A = L' + \sum_{b \in B} A_b \tag{4}$$

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 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:

$$x^T A x + l'^T x \tag{5}$$

Where the linear terms in (2) are unaffected by the new quadratic terms so l' is simply an m-vector s.t. $l' = [l|0, \ldots]$. 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 when a sparse matrix data-structure is used.

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 3 shows the constraints generated for when moving nodes horizontally. Notice that additional bend points are introduced in some straight edge segments. A new bend-point is created wherever the top or bottom of a node is visible from that segment where nodes restrict visibility but edges do not. We distinguish between the original *active* bends and these additional *potential* bends. We only want an edge to bend at a potential bend point if the node associated with that bend point (which is currently not touching the edge) is moved as a result of straightening other edges and collides with the edge. We therefore straighten potential bend points to the line segment between active bend points (see Figure 2) and weight them significantly more than active bend points, to avoid introducing new bends where possible.

Figure 4 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]. To generate horizontal constraints, we perform 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 neighbours, l and r, are computed. The function *neighbourhood_x_constraints* searches along the scan line at y between l and r for intersections with open edges. Bend variables are created at these intersection points and constraints are generated between them. Constraints between edges and the nodes to which they are connected should not be generated or else edges may become "wrapped" around their endpoints. We therefore consider several cases for constraint generation. Case A generates constraints between adjacent bend points for edges not connected to l, r or v. Case B considers edges to the right of l or v, skipping those edges connected to l or v, and Case C similarly handles edges to the left of v or r. These three cases are illuminated in Figure 3(b).

The worst-case time complexity of procedure *generate_x_constraints*(V, E) is $O(|V|(|V| + |E| \log |E|))$ and 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 is to use 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 and so our algorithm can be used for this purpose. An example is shown in Figure 8. We have also explored an alternative approach for obtaining the initial layout and edge routing. This is novel, so we describe it in more detail. It has four main steps.

```
procedure generate_x_constraints(V, E)
    C ← ∅
    Events ← {(top(v), Open, v), (bottom(v), Close, v)|v ∈ V}
        ∪{(top(e), Open, e), (bottom(e), Close, e)|e ∈ E}
    sort Events by decreasing y position
    # OpenNodes is maintained in order of each node's x position
    OpenNodes← ∅
    OpenEdges← ∅ % OpenEdges is unordered
    for each (y, type, obj) ∈ Events do
        if obj is a node then
            v ← obj
            l ← first node to left of v in OpenNodes
            r ← first node to right of v in OpenNodes
            Z ← Z∪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 ← ∅, minx = −∞
    if l ≠ 0 then
        L ← {l}, minx = xpos(l)
    endif
    L ← L ∪ {dummyNode(e, x, y)|e ∈ OpenEdges ∧ x ∈ xpos(e, y)
        ∧ minx < x < xpos(v) ∧ e is not connected to l or v} ∪ {v} ∪ ...
    ... similarly add dummy nodes for edges between v and r including r (if r ≠ 0)
    u_A ← u_B ← u_C ← 0
    for each w ∈ L in ascending xpos order do
        A: if edge(w) ∧ ¬end(w) ∈ {v, l, r} then
                if u_A ≠ 0 then
                    Z ← Z ∪ {xpos(u_A) + (width(u_A) + width(w))/2 ≤ xpos(w)}
                u_A ← w
            else if node(w) then
                u_A ← 0
            endif
        B: if edge(w) then
                if isend(u_B, edge(w)) then
                    skipList ← skipList ∪ {w}
                else
                    for each s ∈ skipList do
                        Z ← Z ∪ {xpos(s) + (width(s) + width(w))/2 ≤ xpos(w)}
                    skipList ← ∅
                endif
            else if node(w) then
                skipList ← {w}
                u_B ← w
            endif
    for each w ∈ 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. 4. 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 3(b).

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 $A^\star$ 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 $A^\star$ based-search to find the best route for each edge, though this time the cost includes a penalty for each crossing. 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.

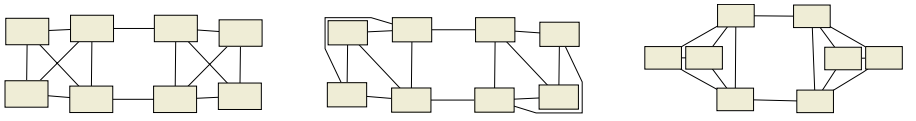
The penalty for an edge route is simply the sum of the penalties for its segments where the penalty for an edge segment is given by:

$$cost = l + \alpha s + \frac{\beta a \log(a+1)}{10} + \gamma scc \tag{6}$$

where: α, β, γ are user specifiable penalties for, 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 segment.

The penalty function incorporates the three main features of poly-line edge routing that have been shown to affect user comprehension: edge length, number of bends and degree of bendiness [14]. The ability to use a flexible penalty function allows us to adjust the initial routing to match the desired combination of aesthetic criteria and their tradeoffs. Setting a very high penalty for crossings will produce routings with few crossings but this is not always ideal. By reducing the penalty we produce more pleasing routings such as the one shown in Figure 7(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.

Finding poly-line edge crossings for a graph is not as simple as just determining the intersections between the segments of all edge paths. It is common for edges to bend around the same node corner or to share paths for part of their route, i.e., running along the same paths in the visibility graph. In these cases we want to distinguish between situations where they cross and where they only



(a) Initial force-directed layout

(b) Edges rerouted to reduce crossings

(c) After straightening edges

Fig. 6. A small example showing the main steps in our layout process

touch or run parallel. We do this by comparing the order of edges entering and leaving shared paths or common bend points.

Since the length of the shared path has no effect on whether the two edges cross, we can treat bend points equivalently to shared paths. We start by looking for cases where the segments of two edges have a single shared endpoint. This is the beginning of a shared path or a common bend point. Such bend points always pass around the corner of a node, so we determine the order of edges entering the shared path at this point by finding which edge runs closest to the node. From here we follow the common segments along the shared path until they diverge again. We then determine the order of edges leaving the shared path, taking into account features of the bends such as the winding directions. If the two orders are different then we can tell that the edges cross along the shared path, rather than running parallel.

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 5. To do this a sorted order for each of these points and shared paths is kept when determining crossings. This nudging step prevents the creation of additional edge crossings during the layout step.

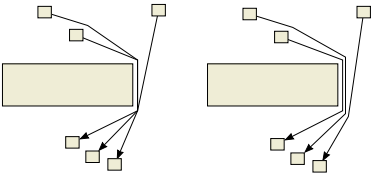
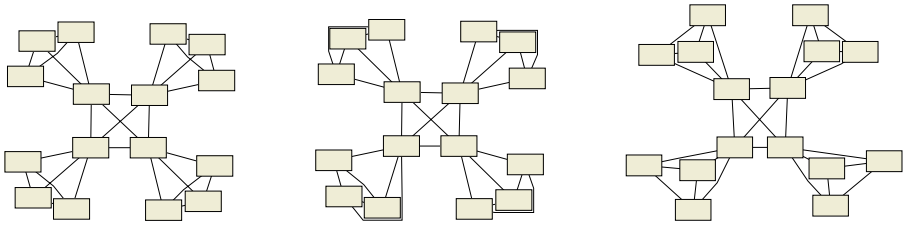


Fig. 5. An exaggerated example of the nudging we perform on shared edge bend points

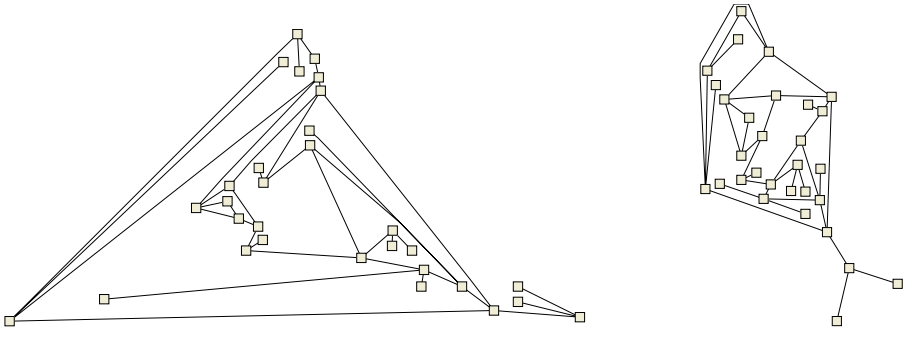
4 Discussion

Figure 6 demonstrates how the edge routing and straightening procedures are applied in practice. We begin with the output of an unconstrained stress-majorisation layout in which edge routing is ignored. Edges are all close to their ideal length thus minimising (1). We apply edge routing as described above, penalising routes with crossings. A planar layout is obtained but with longer edges and a number of bends. Constraints are then generated using the algorithm from Figure 4 and the stress majorisation layout is rerun subject to these constraints. The result retains the planar layout but the edges are straightened.



(a) Initial force-directed layout (b) Edges routed to penalise crossings (c) With edge straightening

Fig. 7. An example graph from Kamada-Kawai [2]



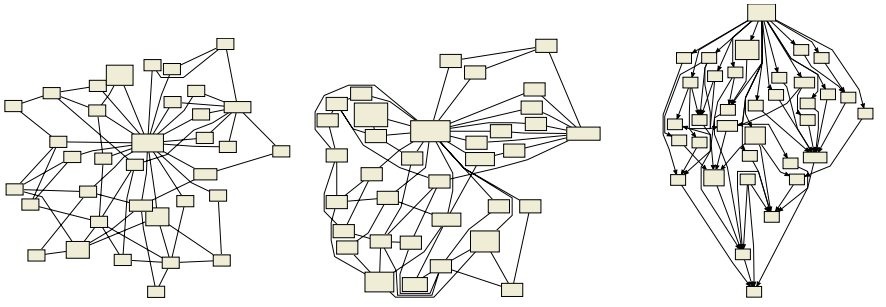
(a) Output of a planar graph layout algorithm (b) With edge straightening

Fig. 8. Applying edge straightening to the output of a planar layout algorithm

Figure 7 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 consistent edge length and greater symmetry than a completely planar drawing would permit.

Figure 8 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.

Our final example in Figure 9 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 Figures 9(b) and 9(c)



(a) Initial force-directed layout: 43 crossings

(b) With edges routed to penalise crossings and straightened: 18 crossings

(c) With additional constraints to require downward pointing edges: 26 crossings

Fig. 9. A more realistic graph visualisation application (a Bayesian net)

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. Figure 9(c) demonstrates how other constraints—in this case downward pointing directed-edges—can be used in addition to edge straightening constraints. This is also demonstrated in the example shown in Figure 1.

The entire process, including computation of the initial layout, routing of all edges and then the straightening phase takes only a few seconds for each of the examples shown.

5 Conclusion and Further Work

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 further work.

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: Proc. of the 12th Annual ACM-SIAM Symp. on Discrete Algorithms, 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: Proc. of the 13th Int. Symp. on Fundamentals of Computation Theory (FCT '01). Number 2138 in LNCS, 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