Implementing a General-Purpose Edge Router

David P. Dobkin,[1]*
Emden R. Gansner,[2] Eleftherios Koutsofios,[2] Stephen C. North[2]

[1] Princeton University, Princeton NJ 08544, USA
[2] AT&T Laboratories, Florham Park NJ 07974, USA

Abstract. Although routing is a well-studied problem in various contexts, there remain unsolved problems in routing edges for graph layouts. In contrast with techniques from other domains such as VLSI CAD and robotics, where physical constraints play a major role, aesthetics play the more important role in graph layout. For graphs, we seek paths that are easy to follow and add meaning to the layout. We describe a collection of aesthetic attributes applicable to drawing edges in graphs, and present a general approach for routing individual edges subject to these principles. We also give implementation details and survey difficulties that arise in an implementation.

1 Introduction

Edge placement is an important problem in graph drawing. Once nodes are positioned, edges must be added in a way that clearly exhibits the relation between nodes, without adding clutter or deceptive artifacts. For example, it is usually desirable that an edge between two nodes does not pass through a third node. When nodes are not drawn as points but are represented by symbols or shapes that have area, the difficulties of edge placement increase. Layouts need to employ edges that bend significantly to avoid touching non-incident nodes. Bent edges can be drawn as polylines, or as curves such as Bezier splines. Though polylines are often easier to compute, their sharp corners create unwanted visual discontinuities that smooth curves do not have.

Curves have been employed previously in drawing edges in layered graphs. In these layouts, nodes are assigned to discrete layers and edges between nodes that are more than one layer apart are replaced by chains of virtual nodes. Virtual node chains provide a useful framework for spline fitting. In VCG [18] and dag [7], splines are computed by straightening virtual node chains, then connecting the virtual nodes with line segments and replacing the sharp corners with bends that fit within individual virtual node boxes. Though the results are usually acceptable, it is not unusual for edges to curve abruptly because turns are constrained to fit inside virtual node boxes.

The method used in dot [6] incorporates a more general heuristic with two phases. For a given edge to be drawn, the first phase computes the "white space" or constraint polygon where the edge may be drawn so as not to touch any other nodes or create

* Portions of the work of this author done while visiting AT&T Laboratories. This work supported in part by NSF Grant CCR-9643913 and by the US Army Research Office under Grant DAAH04-96-1-0181

unwanted edge crossings. The second phase fits a smooth spline connecting the edge's endpoints and staying within the constraint polygon. The endpoints can have optional slope constraints. The spline fitting heuristic computes a trial spline between the endpoints. If the spline goes outside the constraint polygon, and it cannot easily be repaired by slight re-aiming, then the spline fitter subdivides the polygon near the broken constraint and draws the top and bottom halves recursively.

We simplified dot's spline fitter by assuming that the input constraint polygon is represented by a list of connected isothetic rectangles, and that the output spline passes through these sequentially. The list of rectangles is easy to compute from virtual node coordinates and spacing between layers. Its structure also makes it straightforward to test if a Bezier curve[1] stays inside the polygon, and to subdivide it for recursion.

We would naturally like good edge placement not only in layered graphs but in other types of layouts as well, *e.g.*, layouts based on virtual physical models, but the rectangular technique is limited to edges in layered graphs. Even in layered graphs, this technique is not always satisfactory. Problems arise in representing the region for an edge that is constrained by another edge sharing the same endpoint if the angle between the 2 edges is small. In such cases, it takes many "steps" in the boundary rectangles to keep them apart. Other complications arise when drawing self-edge loops, or flat edges between nodes in the same layer, particularly if endpoint "ports" or other path constraints are allowed. These problems suggest re-examining the edge drawing problem in a more general setting.

As an initial simplification, we consider drawing edges independently, *i.e.*, we draw an edge without regard to how any other edges may be drawn. We recognize that this ignores any consideration of the global properties of edge layouts, such as reducing edge crossings or emphasizing edges running in parallel.

In this reduced formulation, edge placement is essentially a routing problem. Routing paths around obstacles has been studied extensively in VLSI layout, computational geometry, and robotics path planning. Dubins' classic result [4] shows how to find shortest paths of bounded curvature between two points in the plane with defined tangents at the endpoints. This result has been extended in various ways [1, 10, 17].

Many path planning problems address limitations of robot carts or manipulators, where physical constraints must be accommodated. Schwartz and Sharir [20] studied planning collision-free paths for obstacles. Fortune and Wilfong [5], Kanayama and Hartman [11], Nelson [14], Laumond [13] and others have studied reachability and path planning problems for a point or object subject to a curvature constraint. Latombe [12] (who also presents a thorough survey of the field) studied potential field methods for path planning. This is an interesting approach, though simulating electrostatic fields created by polygonal antennae seems complicated. Suri [21] showed how to find minimum link paths in simple polygons. These paths are interesting because they have a minimal number of corners. Consequently, they can take extreme routes that deviate significantly from shortest paths.

[1] dot uses piecewise cubic Bezier curves because this family of spline curves is implemented in the PostScript graphics language and in public domain graphics code, and so is convenient to work with.

Physical constraints such as robot size, mass, acceleration, or turning radius do not have a clear relationship to natural-looking curves in graph drawings. In general, the bulk of work on route planning does not seem to capture the properties we feel are desirable in drawing edges in graphs. We turn next to a discussion of these properties, and our approach to achieving them.

2 Problem Definition

Though we cannot formally define what it means for an edge connecting two vertices to appear natural, we believe good solutions avoid other vertices in the graph, stay close to a shortest path between the endpoints, do not turn too sharply, and avoid unnecessary inflections.

Taking one approach to satisfying these criteria, we restate our edge routing problem. As a model for the problem, we consider a graph layout to consist of a polygon $\mathcal{P}$ consisting of a simple polygon containing a collection of disjoint simple polygonal holes, corresponding to the node obstacles. Given two points p and q on or in the interior of $\mathcal{P}$, possibly with tangents v_p and v_q, respectively, we want to find a smooth piecewise cubic Bezier curve from p to q, satisfying v_p and v_q, that stays on or within $\mathcal{P}$.

As a first attempt at solving this problem, we considered constructing a simple path L between p and q (typically a shortest path) within the interior of $\mathcal{P}$. We could then compute some simple polygon $\mathcal{Q}$ that surrounded L and did not contain any holes. Finally, we could apply some procedure for fitting Bezier curves within the simple polygon $\mathcal{Q}$.

One question was how to define "good" choices of $\mathcal{Q}$. We speculated that good choices might be ones that are as wide as possible at the bisectors of bends of L, so that the output curve has as much room as possible to turn. We experimented with a heuristic that constructs a route constraint polygon on each side of L. On each side, the polygon is calculated by taking each segment of L and building a polygon with the segment as a base. For a segment $s = (p_i, p_{i+1})$ of L, let b_i and b_{i+1} be the bisectors of the corners at p_i and p_{i+1}. Find the set of obstacles between b_i and b_{i+1} on the given side of s. If there are no obstacles, then the polygon formed by s, b_i and b_{i+1} is added. Otherwise, the convex hull of the obstacles is removed from the polygon first.

We found this heuristic complicated to implement because of degeneracies, and because if L is allowed to be any simple path (*e.g.* one entered interactively), ensuring that $\mathcal{Q}$ does not intersect itself involved many cases. Because of these difficulties, we abandoned this approach and evolved the following heuristic.

3 Spline Fitting Heuristic

Let $\mathcal{P}$ be a polygon and S a set of forbidden segments. (Typically, these segments will be the edges of the polygonal holes and the bounding polygon.) Let p and q be points in $\mathcal{P}$. The spline fitter is divided into two algorithms. The first finds a shortest path L within $\mathcal{P}$, connecting p to q, that intersects no edge in S except possibly at an endpoint. There is a choice of algorithms here depending on whether $\mathcal{P}$ can contain holes. The

second algorithm fits a Bezier curve along L that intersects no edge in S except at an endpoint.

3.1 Path Finding

If $\mathcal{P}$ does not contain holes (*e.g.*, when emulating the spline router for layered graphs used in dot), we can apply a standard "funnel" algorithm [2, 9] for finding Euclidean shortest paths in a simple polygon. To find a shortest path from point p to point q in a simple polygon, we first triangulate the polygon, using only vertices of the polygon. We then find the triangles that contain points p and q, say t_p and t_q. We then find the sequence of triangles that connect t_p and t_q by doing a depth-first search starting from t_p and searching for t_q. This induces a list of triangle sides (a_i, b_i) interior to $\mathcal{P}$ and crossed by the shortest path from p to q. We then iteratively build a funnel composed, at each step, of the two shortest paths from p to a_i and b_i. Once q is added to the funnel, we have the shortest path from p to q. Given the requisite triangulation, the funnel construction phase runs in linear time.

If $\mathcal{P}$ may contain holes, then we compute the visibility graph of its points plus the two endpoints, and apply Dijkstra's algorithm to find a shortest path between the endpoints. The details of this algorithm are omitted.

3.2 Spline Fitting

Using the shortest path L as a guide, we recursively attempt to fit a curve to it that avoids the obstacles. We use an approach based on the curve-fitting method introduced by Schneider [19]. As input, we start with a collection of points $p_0, p_1, ..., p_n$ that we wish to fit, initially using the vertices of L, plus tangents t_0 and t_n specified at p_0 and p_n. Applying Schneider's method once, we compute a single cubic Bezier segment corresponding to four control points w_0, w_1, w_2, w_3, where $p_0 = w_0$, $p_n = w_3$, and the segments $[w_0, w_1]$ and $[w_2, w_3]$ are parallel to the given tangents at p_0 and p_n, respectively.[2] If this curve does not intersect any of the obstacles, we are done.

If there is an intersection, we attempt a series of local adjustments, in which we move w_1 and w_2 closer to w_0 and w_3, respectively, along the appropriate tangents. If, at any stage, we obtain a viable curve, we are done. Otherwise, we pick the point p_i that is furthest from the Bezier curve, compute a tangent t_i that bisects the angle turned at p_i, and recursively apply the algorithm to the points $p_0, p_1, ..., p_i$ with tangents t_0 and t_i, and to the points $p_i, ..., p_n$ with tangents t_i and t_n.

Note that, using this construction, there is no guarantee that the resulting curve is topologically equivalent to L. At some point, a Bezier segment may "leap over" an intervening obstacle. The approximation is good enough, however, that the resulting curve would still be adequate for our purposes and, in practice, this situation does not seem to arise.

[2] If there are only two input points, we put w_1 and w_2 along the appropriate tangents at a distance of $d/3$ away from p_0 and p_n, respectively, where d is the distance between p_0 and p_n.

4 Implementation

With general edge routing as a goal, we have implemented the path planner as a C library. Its main primitives are:

- *shortestpath(P, p, q)*: finds a shortest path between p and q in a simple polygon P.
- *obspath(obstacles, p, p_{poly}, q, q_{poly})*: finds a shortest path between p and q not intersecting the interior of any polygons in the list *obstacles*. When an endpoint is inside an obstacle (as in Figure 2), the obstacle must be ignored for the route. It is possible that the caller knows when this happens. For example, a graph layout algorithm may know the endpoint nodes, or an interactive diagram editor may perform hit detection on canvas objects. When an endpoint is known to be inside an obstacle, the obstacle may be passed as an argument, viz. p_{poly} or q_{poly}.
- *splinefit(barriers, L, v_p, v_q)*: returns a piecewise cubic Bezier that fits around the input path L and avoids the list of segments in $barriers$. The arguments v_p and v_q provide tangent vectors for the first and last points of L. Generally, L was obtained from one of the two previous primitives. Note that $barriers$ can be any collection of segments, not necessarily forming closed polygons.

The current implementations provide room for increased efficiency. The *obspath* routine uses a naive $O(n^3)$ visibility graph algorithm. We intend to replace this with a more efficient algorithm [8, 16]. For interactive or incremental layout, incremental visibility computation would obviously be desirable. The current version of *splinefit* tests each spline against all barrier edges separately for intersections. This is a quadratic algorithm. When this becomes too slow, a bucketing technique can be implemented to get near-linear behavior. Finally, we use an $O(n\log n)$ triangulation algorithm in *shortestpath*.

5 Observations and Conclusions

Figures 1 - 4 are screen dumps from sample runs of the edge router. Although the obstacles in these examples are different from what would be expected in graph drawing, they exhibit the wide applicability of our technique and the quality of the resulting paths. With each figure we give the time in milliseconds needed to compute the route, measured on a Silicon Graphics Indigo2 computer with a 250MHz MIPS R4400 processor and R4010 floating point unit. The measurement is divided into the time to compute the visibility graph (which may be amortized) plus the time to find a shortest path and fit a spline.

Figures 5 and 6 show the router in a graph editor, its intended domain. Figure 5 was made with manual node placement, and figure 6 with the *dynadag* hierarchical layout manager [15].

We are not aware of any formal studies about what kind of edge routes are most effective for information visualization. Without a formal definition of what it means for a spline to be "good," an evaluation of our results must be subjective. We speculate that when drawing edges manually, humans try to interpolate between a shortest path and a path of minimum curvature while avoiding obstacles.

We have ignored the problem of routing multiple edges. Instead, we route individual edges independently. Further work is needed to understand how edge routes interact. One simple approach, foreshadowed by the edge layout in dot, is to route edges consecutively, determining the "territory" available to an edge based on neighboring edges already routed. Clearly, the order of edge drawing becomes significant, as an edge route can affect others drawn afterward. How should the order be planned? Should previously drawn edges ever be moved? Would a more global approach to routing multiple edges offer better results?

Polygons only approximate regions bounded by curved edges and nodes with curved boundaries (such as ellipses or boxes with rounded corners). Re-implementation of the router with splinegons [3] is worth consideration.

6 Acknowledgments

We thank John Ellson from Lucent Corp. for the TCL/tk interface.

References

1. J.-D. Boissonnat, A. Cerezo, and J. Leblond. Shortest paths of bounded curvature in the plane. *J. Intell. and Robotics Systems*, 11:5–20, 1994.
2. B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd IEEE Symp. Foundations of Computer Science*, pages 339–349, 1982.
3. D. P. Dobkin, D. L. Souvaine, and C. J. Van Wyk. Decomposition and intersection of simple splinegons. *Algorithmica*, 3:473–486, 1988.
4. L. E. Dubins. On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents. *Amer. J. Math.*, 79:497–516, 1957.
5. S. Fortune and G. Wilfong. Planning constrained motion. *Annals of Mathematics and Artificial Intelligence*, 3:21–82, 1991.
6. E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, March 1993.
7. E.R. Gansner, S.C. North, and K.P. Vo. Dag – a program that draws directed graphs. *Software – Practice and Experience*, 18(11):1047–1062, 1988.
8. S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Computing*, 20(5):888–910, 1991.
9. J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, pages 331–342. Springer-Verlag, 1991.
10. P. Jacobs. Minimal length curvature constrained paths in the presence of obstacles. Technical Report 90042, Laboratoire d'Automatique et d'Analyse des Systemes, 7 Avenue du Colonel Roche - 31077 Toulouse, France, February 1990.
11. Y. Kanayama and B. I. Hartman. Smooth local path planning for autonomous vehicles. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, volume 3, pages 1265–1270, 1989.
12. J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.
13. J. P. Laumond. Finding collision-free smooth trajectories for a non-holonomic mobile robot. In *Proc. Intl. Joint Conf. on Artificial Intelligence*, pages 1120–1123, 1987.
14. W. Nelson. Continuous-curvature paths for autonomous vehicles. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, volume 3, pages 1260–1264, 1989.

15. S.C. North. Incremental layout in dynadag. In F.J. Brandenburg, editor, *Symp. on Graph Drawing GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418, 1996.
16. M. H. Overmars and E. Welzl. New methods for computing visibility graphs. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 164–171, 1988.
17. J. A. Reeds and L. A. Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific J. of Mathematics*, 145(2), 1990.
18. G. Sander, M. Alt, C. Ferdinand, and R. Wilhelm. Clax, a visualized compiler. In F.J. Brandenburg, editor, *Symp. on Graph Drawing GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 459–462, 1996.
19. Philip J. Schneider. An algorithm for automatically fitting digitized curves. In Andrew S. Glassner, editor, *Graphics Gems*, pages 612–626. Academic Press, Boston, Mass., 1990.
20. J. T. Schwartz and M. Sharir. Algorithmic motion planning in robotics. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 391–430. Elsevier, Amsterdam, 1990.
21. S. Suri. A linear time algorithm for minimum link paths inside a simple polygon. *Computer Vision and Graphical Image Processing*, 35:99–110, 1986.

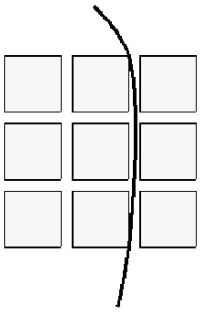

Fig. 1. (28.1+7.8 ms.)

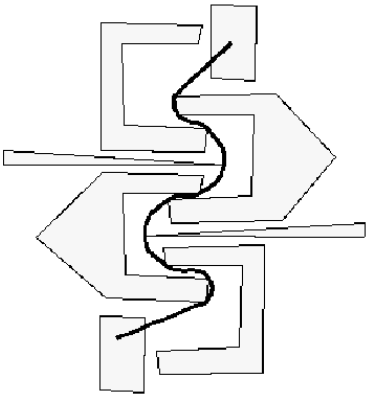

Fig. 2. (31+38 ms.)

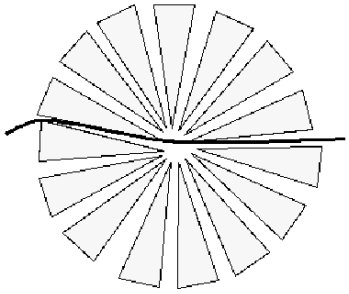

Fig. 3. (46.8+15.0 ms.)

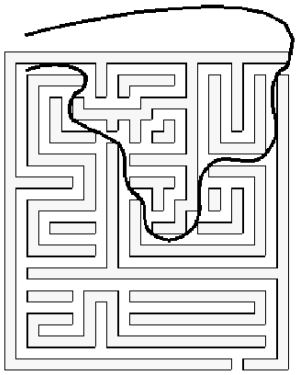

Fig. 4. (742.0+123.0 ms.)

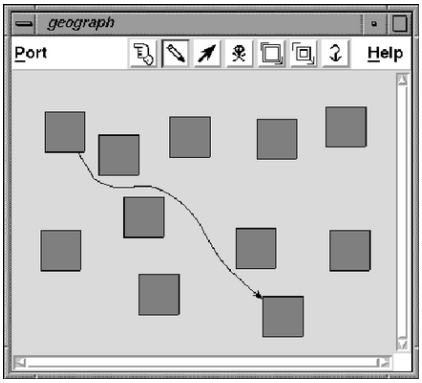

Fig. 5.

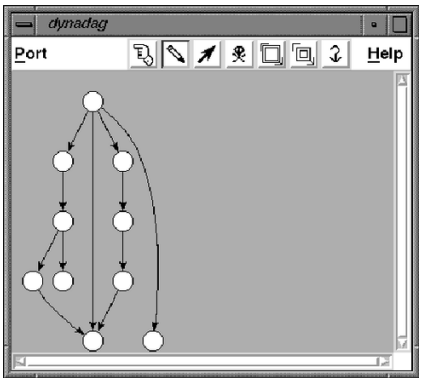

Fig. 6.