Incremental Connector Routing

Michael Wybrow[1], Kim Marriott[1], and Peter J. Stuckey[2]

[1] Clayton School of Information Technology,
Monash University, Clayton, Victoria 3800, Australia,
{mwybrow,marriott}@csse.monash.edu.au
[2] NICTA Victoria Laboratory,
Dept. of Comp. Science & Soft. Eng.,
University of Melbourne, Victoria 3010, Australia,
pjs@cs.mu.oz.au

Abstract. Most diagram editors and graph construction tools provide some form of automatic connector routing, typically providing orthogonal or poly-line connectors. Usually the editor provides an initial automatic route when the connector is created and then modifies this when the connector end-points are moved. None that we know of ensure that the route is of minimal length while avoiding other objects in the diagram. We study the problem of incrementally computing minimal length object-avoiding poly-line connector routings. Our algorithms are surprisingly fast and allow us to recalculate optimal connector routings fast enough to reroute connectors even during direct manipulation of an object's position, thus giving instant feedback to the diagram author.

1 Introduction

Most diagram editors and graph construction tools provide some form of automatic connector routing. They typically provide orthogonal and some form of poly-line or curved connectors. Usually the editor provides an initial automatic route when the connector is created and again each time the connector end-points (or attached shapes) are moved. The automatic routing is usually chosen by an ad hoc heuristic.

Editors such as OmniGraffle [1] and Dia [2] provide connector routing when attached objects are moved, though these routes may overlap other objects in the diagram. Both Microsoft Visio [3] and ConceptDraw [4] provide object-avoiding connector routing. In both applications the routes are updated only after object movement has been completed, rather than as the action is happening. In the case of ConceptDraw, its orthogonal object-avoiding connectors are updated as attached objects are dragged, though not if an object is moved or dropped onto an existing connector's path. The method used for routing does not use a predictable heuristic and often creates surprising paths. Visio offers orthogonal connectors, as well as curved connectors that follow roughly orthogonal routes. Visio's connectors are updated both when the attached shapes are moved and when objects are placed over their paths, but only in response to either of these

events. Again, these connectors do not follow a predictable heuristic, such as minimizing distance or number of segments. Visio does update these connectors dynamically as objects are resized or rotated, though if there are too many objects for this to be responsive Visio reverts to calculating paths only when the operation finishes. The Graph layout library yFiles [5] and demonstration editor yEd offers both orthogonal and "organic" edge routing—a curved force directed layout where nodes repel edges. Both of these are layout options that can be applied to a diagram, but are not maintained throughout further editing. We know of no editor which ensures that the connectors are optimally routed in any meaningful sense.

Automatic connector routing in diagram editors is, of course, essentially the same problem as edge routing in graph layout, especially when edge routing is a separate phase in graph layout performed after nodes have been positioned. Like connector routing it is the problem of routing a poly-line or orthogonal poly-line or spline between two nodes and finding a route which does not overlap other nodes and which is aesthetically pleasing, i.e., short, with few bends and minimal edge crossings. The main difference is that edge routing is typically performed for a once-off static layout of graphs while automatic connector routing is dynamic and needs to be performed whenever the diagram is modified.

One well-known library for edge routing in graph layout is the Spline-o-matic library developed for GraphViz [6]. This supports poly-line and Bezier curve edge routing and has two stages. The first stage is to compute a *visibility graph* for the diagram. The visibility graph contains a node for each vertex of each object in the diagram. There is an edge between two nodes iff they are mutually visible, i.e., there is no intervening object. In the second stage connectors are routed using Dijkstra's shortest path algorithm to compute the minimal length paths in the visibility graph between two points. A third stage, actually the responsibility of the diagram editor, is to compute the visual representation of the connector This might include adding rounded corners, ensuring connectors don't overlap unnecessarily when going around the same object vertex, etc.

Here we describe how this three stage approach to edge routing can be modified to support incremental shortest path poly-line connector routing in diagram editors. We support the following user interactions:

- *Object addition:* This makes existing connector routes invalid if they overlap the new object and requires the visibility graph to be updated.
- *Connector addition:* This simply requires routing the new connector.
- *Object removal:* This makes existing connector routes sub-optimal if there is a better route through the region previously occupied by the deleted object. It also requires the visibility graph to be updated.
- *Connector removal:* This is simple–just delete the connector.
- *Direct manipulation of object placement:* This is the most difficult since it is essentially object removal followed by addition.

To be useful in a diagram or graph editor we need these operations to be fast enough for reasonable size diagrams or graphs with up to say 100 nodes. The performance requirement for direct manipulation is especially stringent if we

wish to update the connector routing during direct manipulation, i.e., to reroute the connectors during the movement of the object, rather than re-routing only after the final placement. This requires visibility graph updating and connector re-routing to be performed in milliseconds.

Somewhat surprisingly our incremental algorithms are fast enough to support this. Two key innovations allowing this are: an "invisibility graph" which associates each object with a set of visibility edges that it obscures (this speeds up visibility graph update for object removal and direct manipulation); and a simple pruning function which significantly reduces the number of connectors that must be considered for re-routing after object removal. In addition we investigate the use of an $A^\star$ algorithm rather than Dijkstra's shortest path algorithm to compute optimal paths.

There has been considerable work on finding shortest poly-line paths and shortest orthogonal poly-line paths between two objects in part because of the importance of these problems in path-planning for robots and VLSI circuit design. Most of this has focused on finding paths given a fixed object layout and has not considered the problem of dynamically changing objects and the need to incrementally update an underlying visibility structure. The most closely related work is that of Miriyala *et. al.* [7]. They have given an efficient $A^\star$ algorithm for computing orthogonal connector paths. Like us they are interested in doing this incrementally and rely on a *rectangulation* of the graph and previously drawn edges which is essentially a visibility graph. The main difference to this paper is that they only consider orthogonal paths. Other differences are that their algorithm is heuristic and routes are not guaranteed to be optimal even if minimizing edge crossings is ignored (see e.g. Figure 9 of [7]). They do not discuss object removal and how to maintain optimality of connectors.

There are several well known algorithms for constructing visibility graphs that run in less than the naive $O(n^3)$ approach. In [8] Lee provided an $O(n^2 \log n)$ solution with a rotational sweep. Later, Welzl presented an $O(n^2)$ duality-based arrangements approach [9]. Asano, et. al., presented two more arrangement based solutions in [10] both running in $O(n^2)$ time. Another $O(n^2)$ approach was given by Overmars and Welzl using rotational trees in [11]. Ghosh and Mount showed an output sensitive solution to the problem in [12] which uses plane sweep triangulation and funnel splits. It runs in $O(m + n \log n)$ time, where m is the number of visibility edges. Only Lee's algorithm and Asano's algorithm support incremental update of a visibility graph.

Given a visibility graph with m edges and n nodes the standard implementation of Dijkstra's shortest path algorithm is $O(n^2)$. Fredman and Tarjan [13] give an $O(m + n \log n)$ implementation using Fibonacci heaps. The $A^\star$ algorithm has similar worst case complexity but in practice we have found it to be faster and believe it has $O(n \log n)$ average case complexity. There are techniques based on the continuous Dijkstra method to compute a single shortest path in $O(n \log n)$ time which do not require computation of a visibility graph [14]. These methods are more complex and so we chose to use a visibility graph-based approach. In

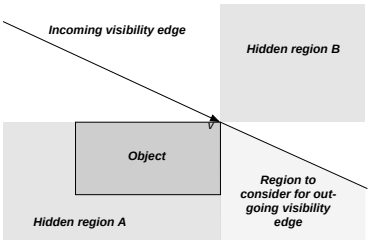


Fig. 1. Hidden regions which can be ignored when constructing the visibility graph and when finding the shortest path

practice we conjecture that they will lead to similar complexity assuming $O(n^2)$ connectors.

2 Algorithms

2.1 Basic data structures

We assume that we have objects which have an associated list of vertices and connector points. For simplicity we restrict ourselves to convex objects and for the purposes of complexity analysis we assume the number of vertices is a fixed constant, i.e. say four, since the bounding box can just be used for many objects. The algorithms themselves work for arbitrary convex polygons, so circles can be approximated by a dodecagon for example.

We also have connectors, these are connected to a particular connection point on an object and have a current routing which consists of a list of edges in the visibility graph. Of course, connectors are not always connected to objects, and may have end-points which are neither object vertices or connection points. In this case an extra node is added to the visibility and invisibility graphs for this end-point.

The most important data structure is the visibility graph. Edges in this graph are stored in a distributed sparse fashion. Each object has a list of vertices, and each of these vertices has a list of vertices that are visible from it. We treat the connection points on objects as if they are vertices. They behave like standard vertices in the visibility graph, but, of course, must occur at the start or end of a path, not in the middle. They have connectors associated with them.

Actually, not all visibility edges are placed in the visibility graph. As noted in [15] in any shortest path the path will bend around vertices, making an angle of less than 180^o around each object. This means that we do not need to include edges to vertices which are visible in the sector opposite to the object. Consider the vertex v with incoming visibility edge shown in Figure 1. Clearly in any shortest path the outgoing visibility edge must be to a vertex in the region indicated. And so, in general, we need not include edges in the visibility graph

that are between v and vertices in either of the two "hidden" regions. Note that this generalizes straightforwardly for any convex object.

The other important data structure is the "*invisibility graph*." This is a new data structure we have introduced to support incremental recomputation of the visibility graph when an object is deleted. It associates each non-visible edge with a blocking object. This should not be confused with the invisibility graph of [**?**] which simply represents the visibility graph negatively.

The invisibility graph consists of all edges between object vertices which could be in the visibility graph except that there is an object intersecting the edge and so obscuring visibility. Edges in the invisibility graph are associated with the obscuring object and with the objects they connect. Thus each object has a list of visibility edges that it obscures. If the edge intersects more than one object the edge is associated with only one of the intersecting objects.

2.2 Connector Addition and Deletion

Connector deletion is simple, all we need to do is to remove the connector and references to the connector from its component edges in the visibility graph.

Connector addition requires us to determine the optimal route for the connector. The simplest approach is use a shortest path algorithm such as Dijkstra's [16]. Dijkstra's method has $O(n^2)$ worst case complexity while a priority queue based approach has worst case complexity $O(m \log n)$ where m is the number of edges in the visibility graph and n the number of objects in the diagram.

A (hopefully) better approach is to use an A^* algorithm which uses the Euclidean distance between the current vertex on the path as a lower bound on the total remaining cost [14]. The idea is to modify the priority queue based approach so that the priority for each frontier node x is the cost of reaching x plus $||(x, v)||$ where v is the endpoint of the connector. In practice we would hope that this is faster than Dijkstra's shortest path algorithm since the search is directed towards the goal vertex v rather than exploring all directions from the start vertex in a breadth-first fashion.

2.3 Object Addition

When we add an object we must first incrementally update the visibility and invisibility graphs, then recompute the route for any connectors whose current route has been invalidated by the addition. The precise steps when an object o is added are

1. Find the set of edges E_o in the visibility graph that intersect o.
2. Find the set of connectors C_o that use an edge from E_o.
3. Remove the edges in E_o from the visibility graph and place them in the invisibility graph, associating them with o.
4. For each vertex (and connection point) v of o and for each vertex (and connection point) u of each other object in the diagram o' determine if there is another object o'' which intersects the segment (v, u). If there is add (v, u)

to the invisibility graph and associate it with o''. If not add (v, u) to the visibility graph.

5. For each connector $c \in C_o$ find its new route.

The two steps with greatest expected complexity are Step 1, computing the visibility edges E_o obscured by o, and Step 4, computing the visible and obscured vertices for each vertex v of o.

The simplest implementation of Step 1 has $O(m)$ complexity since we must examine all edges in the visibility graph to see if they intersect o. We could reduce this to an average case $O(\log m)$ using a spatial data structure such as a PMR quad-tree [17].

Naive computation of the visible and obscured vertices from a single vertex has $O(n^2)$ complexity. However more sophisticated algorithms for computation of the visibility graph have been developed.

One reasonably simple approach which appears to work well in practice Lee's rotational sweep algorithm [8] in which the vertices of all objects are sorted w.r.t. the angle they make with the original vertex v of o and then these are processed in sorted order. At each stage the algorithm keeps a heap of the object edges that are currently open ordered by their distance from v. Only the closest edge is visible since the others are obscured by it. This has $O(n \log n)$ complexity.

One complication with Lee's Sweep Algorithm is that it does not handle intersecting objects since these give rise to "implicit" vertices in which object edges cross. Thus the algorithm requires that when an object is added that we identify any objects that it intersects with and insert the implicit vertices.

2.4 Object Deletion

Perhaps surprisingly, object deletion is potentially considerably more expensive than object creation. The first stage is to incrementally update the visibility graph.

Assume initially that we do not have an invisibility graph. We first need to remove all edges in the visibility graph that are connected to the object being deleted, o. Then when need to add edges to the visibility graph that were previously obscured by o. For each vertex (and connection point) v of each object and for each vertex (and connection point) u of some other object in the diagram we must check that (u, v) is not in the visibility graph and that it intersects o. If so we need to check whether there is any other object which intersects the segment (u, v). If there is not then it must be added to the visibility graph.

Identifying these previously obscured edges is potentially very expensive: $O(n^2)$ to compute the candidate new edges and then an $O(n)$ test for non-overlap for each edge of which there may be $O(n^2)$. Thus the worst case complexity of this method is $O(n^3)$.[3]

In order to reduce the expected (but not the worst case) cost we have introduced the invisibility graph. By recording the reason for not including an edge

[3] Based on this one might consider recomputing the entire visibility graph using the Sweep Algorithm since this has $O(n^2 \log n)$ complexity.

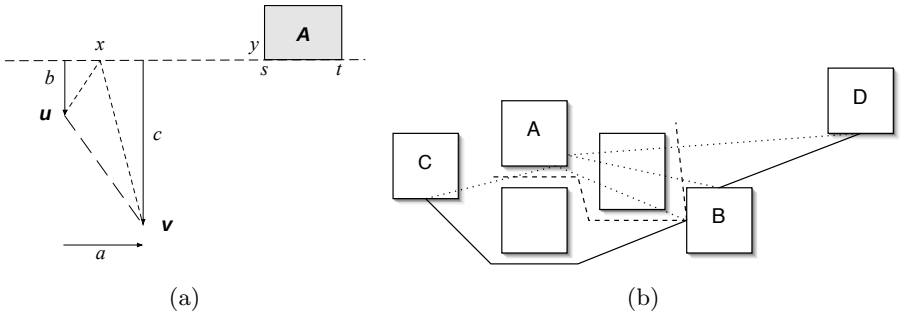


Fig. 2. (a) Computing the closest point $y \in A$ to the segment (u, v). (b) Example recomputation of connectors after deleting object A. Connectors are shown as solid lines and lower-bound connector paths through A are shown as dotted lines. The re-exploration to try and find a better path from B to C is shown as dashed lines.

between two vertices in the visibility graph we know almost exactly which edges we need to retest for visibility. More exactly when we remove o we take the set of edges I_o associated with o in the invisibility graph and then test for each of these whether they intersect any other objects. Note that I_o can be expected to be considerably smaller than the candidate edges identified above since an edge (u, v) is only in I_o if it intersects o *and* o was the object first discovered to intersect (u, v).

Thus, although the invisibility graph does not reduce the worst case cost, we can expect it to substantially reduce the average cost of updating the visibility graph. Furthermore, construction of the invisibility graph does not introduce substantial overhead in any of the other operations, the only overhead is the space required to store the edges. Note that when we remove an object we also need to remove edges to it that are in the invisibility graph.

The second stage in object deletion is to recompute the connector routes. This is potentially very expensive since removing an object means that we could have to recompute the best route for all connectors since there may be a better route that was previously blocked by the object just removed.

However, we can use a simple strategy to limit the number of connectors reconsidered. Let A be the region of the object removed and let u and v be the two ends of the connector C. We need only reconsider the current route for C if $\exists y \in A$ s.t. $||(u, y)|| + ||(y, v)||$ is less than the cost of the current route since otherwise any route going through A will be at least as expensive as the current one.

Thus we need to compute $\min_{y \in A} ||(x, y)|| + ||(y, v)||$. Assuming A is convex we can compute this relatively easily. If the line segment (u, v) intersects A then the lower bound is $||(u, v)||$ and we must reroute C. Otherwise we find for each line segment (s, t) on the boundary of A the closest point y to A on that segment. The closest point in A is simply the closest of these. Now consider the

line segment (s,t). We first compute the closest point x on the line $\overline{st}$. If x is in the segment (s,t) it is the closest point, otherwise we set y to s or t whichever is closest to x. W.lo.g. we can assume that (s,t) is horizontal. Let b and c be the vertical distance from $\overline{st}$ to u and v respectively and a the horizontal distance between u and v, as shown in Figure 2(a). We are finding the value for x which minimizes $\sqrt{x^2+b^2}+\sqrt{(x-a)^2+c^2}$. There are two solutions: $x=\frac{ab}{b+c}$ when $b\cdot c\geq 0$ and $x=\frac{ab}{b-c}$ when $b\cdot c\leq 0$. In the case $b=c=0$, x is any value in $[0,a]$.

Now consider the case when we have determined that there may be a better path for the connector because of the removal. Instead of investigating all possible paths for the connector we need only investigate those that pass through the deleted object. Let A be the region of the object removed and let u and v be the two ends of the connector C and assume that the current length of the connector is c. Requiring the path to go through A means that we can use the above idea to provide a better lower-bound when computing a lower bound on the remaining length of the connector. The priority for each frontier node x is the cost of reaching x plus $\min_{y\in A}||(x,y)||+||(y,v)||$ if the path has not yet gone through A. Furthermore we can remove any node whose priority is $\geq c$ since this cannot lead to a better route than the current one.

For example consider deleting object A from the diagram in Figure 2(b).The connector from B to D does not need to be reconsidered since the shortest path from the connection points (dotted) is clearly longer than the current path. But the connector from B to C needs to be reconsidered (even though in this case it will not move). The A^* algorithm will compute the dashed paths whose endpoints fail the lower bound test.

2.5 Direct manipulation of object placement

The standard approach in diagram and graph editors for updating connectors during direct manipulation is to only reroute the connectors once the object has been moved to its final position. The obvious disadvantage is that the user does not have any visual feedback about what the new routes will be and may well be (unpleasantly) surprised by the result. One of the main ideas behind direct manipulation [18] is that the user should be given visual feedback about the consequences of the manipulation as they perform it rather than waiting for the manipulation to be completed. Thus it seems better for diagram and graph editors to reroute connectors during direct manipulation.

We have identified two possible approaches. In the *complete feedback* approach all connectors are rerouted at each mouse move during the direct manipulation. The advantage is that the user knows exactly what would happen if they left the object at its current position. The disadvantage is that this is very expensive. Another possible disadvantage is that it might be distracting to reroute connectors under the object being moved during the direct manipulation—for positions between the first and final position of the object the user knows that the object will not be placed there and so it is distracting to see the effect on

connectors that are only temporarily under the object being manipulated. For these reasons we have also investigated a *partial feedback* approach in which for intermediate positions we only update the routes for connectors attached to the object being manipulated and leave other connectors alone.

The simplest way of implementing complete connector-routing feedback is to regard each move as an object deletion followed by object addition. Assume that we move object o from region R_{old} to R_{new}. Then we

1. Find the set of edges I_o associated with o in the invisibility graph which do not intersect R_{new} and remove them from the invisibility graph.
2. For each edge $(u, v) \in I_o$ determine if there is another object $o' \in O$ which intersects the segment (u, v). If there is add (v, u) to the invisibility graph and associate it with o'. If not add (v, u) to the visibility graph.
3. Find the set of edges E_o in the visibility graph that intersect $R_{new} \setminus R_{old}$ but are not from o.
4. Find the set of connectors C_o that use an edge from E_o.
5. Remove the edges in E_o from the visibility graph and place them in the invisibility graph, associating them with o.
6. For each vertex (and connection point) u of o and edge (u, v) in the invisibility graph check that the object o' associated with the edge still intersects it. If it does not, temporarily add the edge to the visibility graph.
7. For each vertex (and connection point) u of o and edge (u, v) in the visibility graph check if there is another object $o' \in O$ which intersects the segment (u, v). If there is add (u, v) to the invisibility graph and associate it with o'. If not, keep (u, v) in the visibility graph.
8. For each connector $c \in C_o$ find its new route.
9. For every connector not in C_o determine if there is a better route through $R_{old} \setminus R_{new}$.

Note that in the above we can conservatively approximate the regions $R_{new} \setminus R_{old}$ or $R_{old} \setminus R_{new}$ by any enclosing region such as their bounding box.

The simplest way of implementing partial connector-routing feedback is perform object deletion once the object o has moved and then at each step

1. Compute the vertex and connector points which are visible from a vertex of o and temporarily add these to the visibility graph
2. Recompute shortest routes for all connectors to/from o

Once the move has finished we perform object addition for o. Clearly this is substantially less work than required for complete feedback.

3 Evaluation

We have implemented our incremental connector algorithms in the Dunnart diagram editor and have conducted an experiment to evaluate our algorithms.[4]

[4] Dunnart including this feature is downloadable from
`http://www.csse.monash.edu.au/~mwybrow/dunnart/`

Dunnart is written in C++ and compiled with `gcc` 3.2.2 at `-O3`. We ran Dunnart on a Linux machine (glibc 2.3.3) with 512MB memory and Pentium 4, 2.4GHz processor.

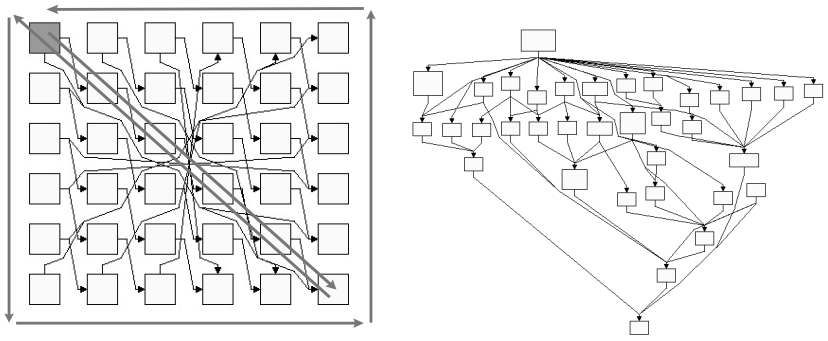
In our experiment we compared the Spline-o-matic (SoM) connector routing library of Graph Viz (which is non-incremental) with a static version (Static) of our algorithm in which the visibility graph and connector routes are recomputed from scratch after each editor action, and the incremental algorithm (Inc) given here with various options.

The experiment used various sized grid arrangement of boxes, where each outside box is connected to the diagonally opposite box by a connector and each box except those on the right and bottom edge is connected to the box directly down and right. Figure 3(a) show an example layout for a 6x6 grid. For an $n \times n$ grid we have n^2 objects and $2(n-1) + (n-1)^2$ connectors. We also used a smallish but more realistic diagram **bayes** from [19] (a Bayesian network with 35 objects and 61 connectors) shown in Figure 3(b). The experiments were: for each object to delete it from the grid and the add it back in. We measured the time for each deletion and each addition giving the average under the **Add** and **Delete** rows. We have separated the time into that for manipulating the visibility graph (*Vis*) and that for performing all connector (re)routing (*Paths*). We also measured the average time taken to compute the new layout for each mouse position when moving the marked corner box through and around the grid as shown in Figure 3(a). The move of **bayes** is similar using the top leftmost box. This results are is given in the **Move** rows.

Both our static (Static) and incremental (Inc) version use the $A^\star$ algorithm to compute connector paths and give complete feedback. The incremental version computes the invisibility graph while the static one does not since this is not needed. We also give times for versions of the incremental algorithm which do not construct the invisibility graph (Inc-noInv). and use Dijkstra's shortest path algorithm rather than the $A^\star$ algorithm (Inc-noA*). For the **Move** sub-experiment we also give times for an incremental version providing partial feedback (Inc-par) rather than complete feedback.

The results are shown in Table 1, for grids of size 6, 8, 10 and 12, and **bayes**. A "—" indicates that the approach failed to complete the total experiment in three hours.

We can see from the table that the static version of our algorithm is considerably faster than Spline-O-Matic. The incremental versions are orders of magnitude faster than static algorithms. While the incremental version is usable for direct manipulation with complete feedback at least until **grid10** (and with difficulty at **grid12**), the static algorithms become unusable at **grid08**. The results show how important incremental recomputation is. The importance of the invisibility graph is clearly illustrated by the difference between Inc-noInv and Inc, improving visibility graph recomputation by orders of magnitude for **Delete** and **Move**. The $A^\star$ algorithm gives around 50% improvement in path re-routing. Partial feedback reduces the overhead of movement considerably particularly as the number of connectors grows.



(a) (b)

Fig. 3. (a) 6x6 Grid layout, showing the path taken through grid for **Move** experiment, and (b) Layout of the **bayes** diagram

4 Conclusion

Most diagram editors and graph construction tools provide some form of automatic connector routing, usually providing orthogonal or poly-line connectors. However the routes are typically computed using ad hoc techniques and updated in an ad hoc manner with no feedback during direct manipulation.

We have investigated the problem of incrementally computing minimal length object-avoiding poly-line connector routings. Our algorithms are surprisingly fast and allow us to recalculate optimal connector routings fast enough to reroute connectors even during direct manipulation of an object's position, thus giving instant feedback to the diagram author.

References

1. The Omni Group: OmniGraffle Product Page. Web Page (2002) `http://www.omnigroup.com/omnigraffle/`.
2. Larsson, A.: Dia Home Page. Web Page (2002) `http://www.gnome.org/projects/dia/`.
3. Microsoft Corporation: Microsoft Visio Home Page. Web Page (2002) `http://office.microsoft.com/visio/`.
4. Computer Systems Odessa: ConceptDraw Home Page. Web Page (2002) `http://www.conceptdraw.com/`.
5. yWorks: yFiles - Java Graph Layout and Visualization Library. Web Page (2005) `www.yworks.com/products/yfiles/`.
6. AT&T Research: Spline-o-matic library. Web Page (1999) `http://www.graphviz.org/Misc/spline-o-matic/`.
7. Miriyala, K., Hornick, S.W., Tamassia, R.: An incremental approach to aesthetic graph layout. In: Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering, IEEE Computer Society (1993) 297–308

		grid06		grid08		grid10		grid12		bayes	
Op	Algorithm	Vis	Paths	Vis	Paths	Vis	Paths	Vis	Paths	Vis	Paths
Add	SoM	152	198	752	881	2449	2831	6669	1166	122	284
	Static	40	67	154	313	475	1024	1209	1064	29	87
	Inc-noInv	0	13	8	90	15	304	14	761	0	1
	Inc-noA*	0	11	9	61	18	195	16	347	0	7
	Inc	0	1	9	20	18	58	16	138	0	0
Delete	SoM	146	185	724	853	2385	2779	6542	1149	110	269
	Static	40	64	153	296	463	1003	1186	1042	29	80
	Inc-noInv	53	16	266	87	1006	298	1146	749	77	3
	Inc-noA*	2	38	17	223	49	734	55	1331	7	9
	Inc	2	8	18	58	48	204	55	504	8	0
Move	SoM	149	188	742	863	—	—	—	—	114	282
	Static	31	69	156	310	461	1004	1214	1026	29	79
	Inc-noInv	43	15	230	80	950	289	1167	708	80	0
	Inc-noA*	0	25	12	102	34	261	37	449	7	8
	Inc	0	10	12	28	34	77	37	213	7	0
	Inc-par	0	3	10	7	26	20	28	11	0	3

Table 1. Average visibility graph and connector routing times (in msec.)

8. Lee, D.T.: Proximity and reachability in the plane. PhD thesis, Department of Electrical Engineering, University of Illinois, Urbana, IL (1978)
9. Welzl, E.: Constructing the visibility graph for n line segments in $O(n^2)$ time. Information Processing Letters **20** (1985) 167–171
10. Asano, T., Asano, T., Guibas, L., Hershberger, J., Imai, H.: Visibility of disjoint polygons. Algorithmica **1** (1986) 49–63
11. Overmars, M.H., Welzl, E.: New methods for computing visibility graphs. In: Proceedings of the fourth annual symposium on Computational geometry, ACM Press (1988) 164–171
12. Ghosh, S.K., Mount, D.M.: An output-sensitive algorithm for computing visibility. SIAM Journal on Computing **20** (1991) 888–910
13. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34** (1987) 596–615
14. Mitchell, J.S.: Geometric shortest paths and network optimization. In Sack, J.R., Urrutia, J., eds.: Handbook of Computational Geometry. Elsevier Science Publishers B.V., Amsterdam (2000) 633–701
15. Rohnert, H.: Shortest paths in the plane with convex polygonal obstacles. Information Processing Letters **23** (1986) 71–76
16. Dijkstra, E.W.: A note on two problems in connection with graphs. Numerische Mathematik (1959) 269–271
17. Nelson, R.C., Samet, H.: A consistent hierarchical representation for vector data. In: SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM Press (1986) 197–206
18. Shneiderman, B.: Direct manipulation: A step beyond programming languages. IEEE Computer **16** (1983) 57–69
19. Woodberry, O.J.: Knowledge engineering a Bayesian network for an ecological risk assessment. Honours thesis, Monash University, CSSE, Australia (2003) http://www.csse.monash.edu.au/hons/projects/2003/Owen.Woodberry/.