Orthogonal 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,
{Michael.Wybrow,Kim.Marriott}@infotech.monash.edu.au
[2] National ICT Australia, Victoria Laboratory,
Department of Computer Science & Software Engineering,
University of Melbourne, Victoria 3010, Australia,
pjs@csse.unimelb.edu.au

Abstract. Orthogonal connectors are used in a variety of common network diagrams. Most interactive diagram editors provide orthogonal connectors with some form of automatic connector routing. However, these tools use *ad-hoc* heuristics that can lead to strange routes and even routes that pass through other objects. We present an algorithm for computing optimal object-avoiding orthogonal connector routings where the route minimizes a monotonic function of the connector length and number of bends. The algorithm is efficient and can calculate connector routings fast enough to reroute connectors during interaction.

1 Introduction

Most interactive diagram editors provide some form of automatic connector routing between shapes whose position is fixed by the user. Usually the editor computes an initial automatic route when the connector is created and updates this each time the connector end-points (or attached shapes) are moved. Orthogonal connectors, which consist of a sequence of horizontal and vertical line segments, are a particularly common kind of connector, used in ER and UML diagrams among others. However, in all current tools that we are aware of, automatic routing of orthogonal connectors uses *ad-hoc* heuristics that lead to aesthetically unpleasing routes and unpredictable behaviour.

For example, the graphic editors OmniGraffle Pro 5.1.1, and Dia 0.97, provide automatic orthogonal connector routing but these routes may overlap other objects in the diagram. Both Microsoft Visio 2007, and ConceptDraw Pro 5 provide object-avoiding orthogonal connector routing but in both applications connector routing does not use a predictable heuristic, such as minimizing distance or number of segments. Furthermore, the routes are mostly updated only after object movement has been completed, rather than as the action is happening. The Graph layout library yFiles[3] and demonstration editor yEd offers orthogonal edge routing but routing is not maintained throughout further editing.

Thus, we know of no interactive diagram authoring tool which ensures that the orthogonal connectors are optimally routed in any meaningful sense. On the

[3] http://www.yworks.com/products/yfiles/

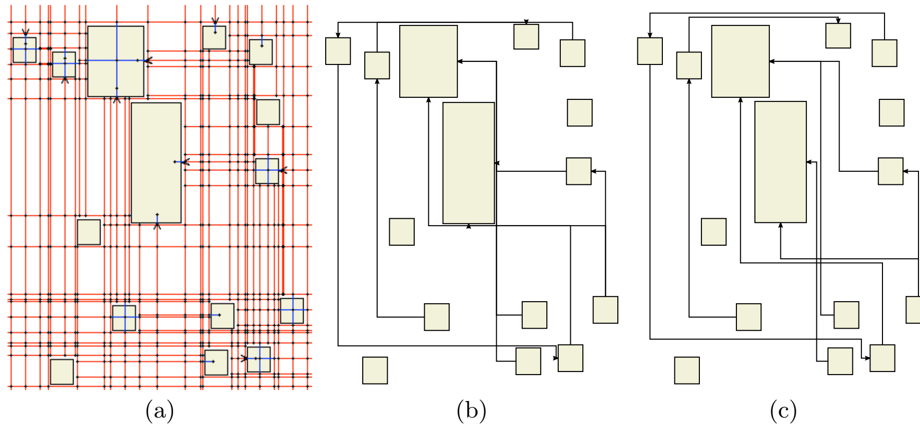


Fig. 1. Three stage routing: (a) the orthogonal visibility graph, (b) the optimal connector routes, (c) the final routes after centering and nudging. Arrows indicate routing direction for connectors.

other hand, automatic routing of poly-line connectors is better supported: two tools, Dunnart and Inkscape, provide real-time poly-line connector routing which is optimal in the sense that it minimizes edge bends and connector length. Both use the connector routing library `libavoid`[4] which has three steps in connector routing [1]. The first stage is to compute a visibility graph for the diagram which contains a node for each vertex of each object in the diagram and an edge between two nodes iff they are mutually visible. The second stage uses $A^\star$ search to find the optimal route through the visibility graph for each connector. The third stage computes the visual representation of the connector. This three step approach is also used in the Spline-o-matic library[5] developed for GraphViz which supports poly-line and Bezier curve edge routing [2].

In this paper we describe how we have extended the connector routing library `libavoid` to support orthogonal connector routing. The main contribution is to show that a similar three step process to that used for poly-line connector routing can also be used for optimal orthogonal connector routing. We introduce the *orthogonal visibility graph* in which edges in the graph represent horizontal or vertical lines of visibility from the vertices and connector ports of each object (Section 4). Connector routes are found using an $A^\star$ search through the orthogonal visibility graph (Section 5). The algorithm is guaranteed to find a route for each connector that is optimal in the sense that it minimizes bends and overall connector length. Finally, the actual visual route is computed (Section 6). This step orders and *nudges* apart the connectors in shared segments so as to ensure that unnecessary crossings are not introduced and that crossings occur at the start or end of the shared segment. It also tries to ensure that connectors pass down the middle of "alleys" in the diagram when this does not lead to additional cost. Figure 1 shows an example layout using the three step process. Our

[4] `http://adaptagrams.sourceforge.net/libavoid/`

[5] `http://www.graphviz.org/Misc/spline-o-matic/`

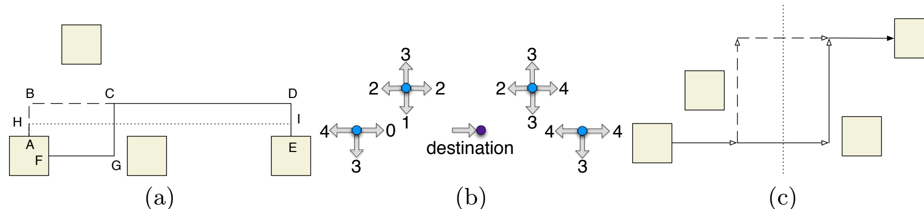


Fig. 2. (a) Comparison of our approach with that of Miriyala *et al.* [3], which chooses the solid path (FGCDE) while our approach computes the dotted path (AHIE). (b) Minimal required additional bends for reaching the destination with correct direction from each point and direction. (c) The solid path is preferred to the dashed path since it is the "initially straighter" path. The dotted line shows the middle of the "alley" of possible paths for the middle segment of the connector.

algorithms are surprisingly efficient and fast enough to reroute connectors even during direct manipulation of reasonably sized diagrams, thus giving instant feedback to the diagram author (Section 7).

2 Related Work

Our work is a significant extension to the previously mentioned research into three-step optimal routing of poly-line connectors to handle orthogonal connector routing. There has been some previous work on finding good orthogonal connector routings between fixed position shapes. The most closely related work in the graph drawing literature is that of Miriyala *et al.* [3] who also use an $A^\star$ algorithm for computing orthogonal connector paths. The main difference is that they search through the *rectangulation* of the diagram rather than the orthogonal visibility graph. The rectangulation is obtained by drawing vertical lines from the vertices of each shape. The search is then through these rectangles. While superficially similar, the rectangulation is actually quite different to the orthogonal visibility graph. The key difference is that the rectangulation does not directly model horizontal visibility. This means that their algorithm is heuristic and routes are not guaranteed to be optimal in any meaningful sense even if minimizing edge crossings is ignored. Figure 2(a) shows the (solid) route FGCDE computed by the approach of [3] to connect the left and right objects even though the (dashed) route ABCDE is clearly better. Our approach will generate the (dotted) route AHIE which is the shortest route with fewest bends (where AH is the minimal distance from shapes allowed). The disadvantage of our approach is that the rectangulation is $O(n)$ in size while the orthogonal visibility graph is $O(n^2)$ in size for n shapes.

Other related work includes algorithms for orthogonal graph layout [4]. The standard technique is to solve a network flow problem in order to compute an orthogonal representation for the graph which minimizes the total number of connector bends. A compaction step is then applied to the orthogonal representation to assign positions to the nodes which minimize the area of the drawing but do not introduce additional crossings or overlap. The key difference to the

problem we address is that in orthogonal graph layout the layout algorithm is responsible for positioning nodes so as to minimize bends, while in our context nodes, i.e. shapes, have a fixed position. It is also worth mentioning that the two stage approach of orthogonal graph layout means that minimizing bends always takes precedence over minimizing connector length so the layouts can contain connectors with very long routes. We also mention incremental approaches to orthogonal graph layout which incrementally construct the layout as vertices (and all of their associated edges) are added one at a time [4]. Again nodes are allowed to move and the focus is on bend minimization.

Orthogonal connector routing has been extensively studied in computational geometry, in part because of its applications to VLSI circuit design. Lee *et al.* [5] provides an extensive survey. One of most common earliest approaches was so-called *maze running* in which objects are assumed to be laid out on a uniform grid and a shortest path algorithm was employed to find the shortest path in the grid [?]. The complexity is proportional to the size of the grid. In our context, the grid needs to be very fine because the user is free to place elements where they like and so the time complexity is prohibitively high. Our approach can be considered a modification to maze running in which we use a non-uniform grid whose mesh size is tailored to the geometry of the diagram. The problem we are addressing is finding a *minimum-cost* path (MCP) where the cost is a function of the number of bend points and path length. Algorithms with $O(n \log^{3/2} n)$ complexity for routing a single connector where n is the number of objects (assuming they are rectangles) are known for this problem. However, it is fair to say that these algorithms are complex, dependent on the kind of penalty function and difficult to implement. Our approach has the advantage of simplicity and, since $A^\star$ search is a generic technique, it can be extended to more complex penalty functions such as one, for instance, penalizing connector crossings (see Section 8). Furthermore, our approach is analogous to the poly-line connector routing approach already used in `libavoid` and so implementation effort is reduced. While the worst-case complexity of our approach is $O(n^2 \log n)$, in practice because of the good heuristic used in the $A^\star$ search, we have found performance is perfectly acceptable.

Another significant contribution of our paper is our algorithm for "nudging" orthogonal connectors apart so as to improve the legibility of the layout. While Miriyala *et. al.* do consider nudging, the issues and approach are quite different. They do not consider the problem of how to avoid introducing unnecessary crossings when separating connectors with a shared path. Our algorithm for ordering connectors in shared paths so as to avoid introducing unnecessary crossings is related to algorithms for metro-line crossing [6, 7]. The main difference is that we have the additional requirement that the ordering should not introduce unnecessary bends in the layout and so crossings are only allowed to occur when a connector enters or leaves the shared path, but not in the shared path itself.

3 Problem Statement

For simplicity we model objects by their bounding rectangle and assume for the purposes of complexity analysis that the number of connector points on

each object is a fixed constant. Also for simplicity, we assume that connectors must start and end at distinct connection points. In practice, connectors are not always connected to objects and may have end-points which are not connection points. This can be handled by adding an extra node to the visibility graph for this endpoint.

We are interested in finding a poly-line route of horizontal and vertical segments for each connector. We wish to find routes that are short and which have few bends. While we also wish to reduce connector crossings we will delay consideration of this until Section 8. We assume our penalty function $\mathsf{p}(R)$ for measuring the quality of a particular route R is a monotonic function f of the length of the path, $||R||$, and the number of bends (or equivalently segments) in R, $bends(R)$, i.e. $\mathsf{p}(R) = \mathsf{f}(||R||, bends(R))$. We require that the routes are *valid*: they do not pass through objects and only contain right-angle bends.

We use the *Manhattan distance* $||(v_1, v_2)||_1 = |x_1 - x_2| + |y_1 - y_2|$ to measure the shortest orthogonal route between points $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$. We make use of 4 cardinal directions: $\mathbb{N}, \mathbb{S}, \mathbb{E}, \mathbb{W}$. We assume the functions *right*, *left*, and *reverse* defined by the mappings:

$$\begin{aligned} right &= \{\mathbb{N} \mapsto \mathbb{E},\ \mathbb{E} \mapsto \mathbb{S},\ \mathbb{S} \mapsto \mathbb{W}, \mathbb{W} \mapsto \mathbb{N}\} \\ left &= \{\mathbb{N} \mapsto \mathbb{W}, \mathbb{E} \mapsto \mathbb{N},\ \mathbb{S} \mapsto \mathbb{E},\ \mathbb{W} \mapsto \mathbb{S}\} \\ reverse &= \{\mathbb{N} \mapsto \mathbb{S},\ \mathbb{E} \mapsto \mathbb{W}, \mathbb{S} \mapsto \mathbb{N},\ \mathbb{W} \mapsto \mathbb{E}\} \end{aligned}$$

We define the directions of point $v_2 = (x_2, y_2)$ from $v_1 = (x_1, y_1)$ as:

$$dirns(v_1, v_2) = \{\mathbb{N} \mid y_2 > y_1\} \cup \{\mathbb{E} \mid x_2 > x_1\} \cup \{\mathbb{S} \mid y_2 < y_1\} \cup \{\mathbb{W} \mid x_2 < x_1\}$$

Note $dirns(v_1, v_2) = \{D\}$ means v_2 is on the line in direction D drawn from v_1.

4 Orthogonal Visibility Graph

The basis for our approach is the observation that when finding routes minimizing the penalty function we need only consider routes in the orthogonal visibility graph. This is defined as follows.

Let I be the set of *interesting points* (x, y) in the diagram, i.e. the connector points and corners of the bounding box of each object. Let X_I be the set of x coordinates in I and Y_I the set of y coordinates in I. The *orthogonal visibility graph* $VG = (V, E)$ is made up of nodes $V \subseteq X_I \times Y_I$ s.t. $(x, y) \in V$ iff there exists y' s.t. $(x, y') \in I$ and there is no intervening object between (x, y) and (x, y') and there exists x' s.t. $(x', y) \in I$ and there is no intervening object between (x, y) and (x', y). There is an edge $e \in E$ between each point in V to its nearest neighbour to the north, south, east and west iff there is no intervening object in the original diagram.

An example orthogonal visibility graph is shown in Figure 1(a). It is quite different to the standard (non-orthogonal) visibility graph used for poly-line routing. In particular, the standard visibility graph has $O(n)$ nodes if there are n objects in the diagram while the orthogonal visibility graph has $O(n^2)$ nodes. Both have $O(n^2)$ edges.

Observation: *Let R be a valid orthogonal route for a connector c. Then there exists a valid orthogonal route R' using edges in the orthogonal visibility graph for c s.t. $p(R') \leq p(R)$.*

Proof. We simply take the route R and "shrink" each segment on the route onto a path in the visibility graph to give R'. By construction R' is no longer than R and has no additional bends. □

The orthogonal visibility graph can be constructed using the following algorithm. It has three steps:

1. Generate the *interesting horizontal segments*

$$H_I = \{\ ((x,y),(x',y)) \mid (x,y),(x',y) \in I \text{ s.t. } x \leq x' \text{ and there is no intervening object between } (x,y) \text{ and } (x',y)\}.$$

2. Generate the *interesting vertical segments*

$$V_I = \{\ ((x,y),(x,y')) \mid (x,y),(x,y') \in I \text{ s.t. } y \leq y' \text{ and there is no intervening object between } (x,y) \text{ and } (x,y')\}.$$

3. Compute the orthogonal visibility graph by intersecting all pairs of segments from H_I and V_I. We note that this could be done lazily, however for simplicity we construct the entire visibility graph in one step.

Theorem 1. *The orthogonal visibility graph can be constructed in $O(n^2)$ time for a diagram with n objects using the above algorithm.*

Proof. The interesting horizontal segments can be generated in $O(n \log n)$ time where n is the number of objects in the diagram by using a variant of the line-sweep algorithm from [8, 9]. This uses a vertical sweep through the objects in the diagram, keeping a horizontal "scan line" list of open objects with each node having references to its closest left and right neighbors. Interesting, horizontal segments are generated, when an object is opened, closed, or a connection point is reached. Dually, the interesting vertical segments can generated in $O(n \log n)$ time by using the line-sweep algorithm with a horizontal sweep. The last step takes $O(n^2)$ time since there are $O(n)$ interesting horizontal and vertical segments. □

5 Routing the Connector

We use an $A^\star$ algorithm which iteratively builds longer and longer partial paths that start from the *source* node s until the *destination* node d is reached. Partial paths are stored in a priority queue and at each step the partial path with lowest *cost* is taken from the queue and expanded. The expanded nodes are placed in the queue. The process stops when the path chosen for expansion is already at d. The cost associated with each partial path is the cost of the partial path so far plus a lower bound on the remaining cost to the destination.

If we are only trying to minimize connector length, the only state we need to know about the partial path is the position of its end. However, if the number of bends is also part of the cost we also need to know the direction of the path. Thus, entries in the priority queue have form (v, D, l_v, b_v, p, c_v) where v is the node in the orthogonal visibility graph, D is the "direction of entry" to the node,

l_v is the length of the partial path from s to v and b_v the number of bends in the partial path, p a pointer to the parent entry (so that the final path can be reconstructed), and c_v the cost of the partial path. There is at most one entry popped from the queue for each (v, D) pair. When an entry (v, D, l_v, b_v, p, c_v) is scheduled for addition to the priority queue, it is only added if no entry with the same (v, D) pair has been removed from the queue, i.e. is on the closed list. And only the entry with lowest cost for each (v, D) pair is kept on the priority queue.

When we remove entry (v, D, l_v, b_v, p, c_v) from the priority queue we

1. add the neighbour (v', D) in the same direction with priority $\mathsf{f}(l_v + ||(v, v')||_1 + ||(v', d)||_1, s_v + s_d)$;
2. add the neighbours $(v', \mathit{right}(D))$ and $(v', \mathit{left}(D))$ at right angles to the entry with priority $\mathsf{f}(l_v + ||(v, v')||_1 + ||(v', d)||_1, s_v + 1 + s_d)$;

where s_d is the estimation of the remaining segments required for the route from (v', D') to (d, D_d). The estimation of the remaining segments required is: $s_d =$

0. if $D' = D_d$ and $\mathit{dirns}(v', d) = \{D'\}$;
1. if $\mathit{left}(D_d) = D' \vee \mathit{right}(D_d) = D'$ and $D' \in \mathit{dirns}(v', d)$;
2. if $D' = D_d$ and $\mathit{dirns}(v', d) \neq \{D'\}$ but $D' \in \mathit{dirns}(v', d)$, or $D' = \mathit{reverse}(D_d)$ and $\mathit{dirns}(v', d) \neq \{D_d\}$;
3. if $\mathit{left}(D_d) = D' \vee \mathit{right}(D_d) = D'$ and $D' \notin \mathit{dirns}(v', d)$; and
4. if $D' = \mathit{reverse}(D_d)$ and $\mathit{dirns}(v', d) = \{D_d\}$, or $D' = D_d$ and $D' \notin \mathit{dirns}(v', d)$.

Figure 2(b) shows all the possible scenarios for determining the remaining minimal number of bends. We note that Miriyala *et al.* [3] use a similar cost.

Even taking into account number of bends, there are usually many alternate routes of the same cost from source to destination. To make the routing behaviour more predictable and faster we add a tie break for equal cost routes based on a time stamp of when the entry was added to the priority queue. This means that because the order in which neighbours is added is deterministic—straight, right, left—there is a slight preference for right turns and also that the latest path is extended in preference to earlier paths. See Figure 2(c).

The worst-case complexity of the A* algorithm is that of a priority queue based implementation of the shortest path algorithm over the orthogonal visibility graph. Thus:

Theorem 2. *The above algorithm will find an optimal valid route for a single connector through the orthogonal visibility graph in $O(n^2 \log n)$ time where the diagram has n objects.* □

6 Computing the visual representation

The third and last step in orthogonal connector routing is "nudging" of the connectors to compute their actual position in the drawing. The importance of this step is often overlooked, but feedback from users of Dunnart and Inkscape suggests that it has a significant impact on the perception of layout quality. It has two steps.

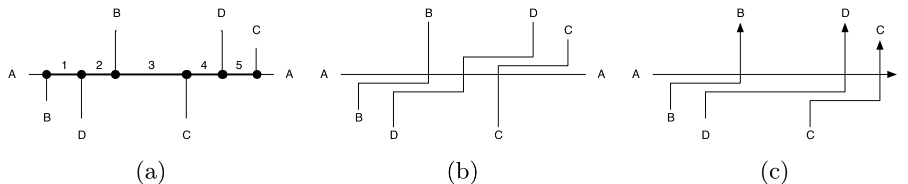


Fig. 3. (a) A set of orthogonal connectors which share edges, and (b) an ordering of shared edges to minimize crossings, (c) an order of shared edges that minimizes crossings and does not introduce additional segments.

6.1 Ordering shared edges

The first aspect is determining the relative ordering of connectors in shared edges. A consequence of routing connectors along the orthogonal visibility graph is that multiple connectors will share edges of their paths. In order to make the connector route clearer we want to nudge these paths apart to make the distinct paths clear. It is important to do so in a manner which does not introduce unnecessary crossings or bends in segments.

We now explain our algorithm to generate a relative ordering of connectors in shared edges. Initially we construct the graph of shared edges, that is the subset of the edges in the visibility graph that have two or more connectors routed along that edge (plus their incident nodes). We process each connected component in the graph separately since each defines an independent subproblem in terms of the parts of connectors whose routes enter and exit this connected component of shared edges. Note that one connector may enter and exit the connected component multiple times in which case each sub-route is treated as a separate connector. Processing of each connected component has two steps.

We first try and assign a uniform *pseudo direction* for each of these connector sub-routes. This pseudo direction is independent of the actual direction of the connector—it is simply used for route adjustment. Choose an arbitrary connector sub-route A and fix its pseudo direction in an arbitrary direction. Now fix the pseudo direction of a connector sub-route that shares an edge with A to have the same direction as one of the shared edges. Follow the sub-route assigning the same pseudo-direction until there is a conflict in which case we mark the sub-route with a split point, reverse the pseudo-direction and continue following the sub-route. Continue this until all sub-routes segments have a pseudo direction. The whole process is $O(e)$ where e is the number of edges in all the sub-routes appearing in this tree.

In practice, we have found that the pseudo-direction assignment for each connector sub-route is almost always *consistent*. We say a set of connector paths is *path consistent* if the pseudo-direction assignment is consistent for each connected component of the shared edge graph.

The next step is to determine for each shared edge a relative order, left to right along the pseudo direction, of each of the connectors that share that edge. We do this in an incremental fashion. Each edge starts with an empty sequence of connectors. We choose an, as yet unconsidered, connector sub-route and process each of its consistent sub-sections, one at a time (i.e. the sub-sections without

split point. We insert this consistent sub-section in the ordering for each shared edge it makes use of. The key is that we will ensure that the necessary crossings of this sub-section with other connectors only occur at the end of the last (in the pseudo direction) shared edge between them, which is either at the end of the connector or at a split point.

Consider adding a connector c to a shared edge order O for edge e. We need to insert c in O in the appropriate place. There are three subsequences of connectors in $O = L{+\!+}S{+\!+}R$. Those that enter e (along the pseudo direction) from the left of c, L, those that enter in the same direction as c, S, and those that enter from the right, R. Now c already shares an edge e' with connectors in S, so we can project the order O' for this edge onto the connectors in $S \cup \{c\}$ to determine an order $S_L{+\!+}[c]{+\!+}S_R$.[6] The new order for edge s is hence $L{+\!+}S_L{+\!+}[c]{+\!+}S_R{+\!+}R$. This step is $O(e^2)$.

Example 1. Consider the tree of shared edges shown if Figure 3(a). The ordering of shared edges shown in Figure 3(b), has minimal connector crossings but adds two extra segments in the route for D. The algorithm proceeds as follows. We assign the pseudo direction left to right to connector A, and this propagates to the other edges as shown by the arrow heads in Figure 3(c). The tree is path consistent. We first add connector A as the unique route in each of edges 1–5. Next we add connector B. Since it enters from below it is ordered after A in edge 1 and 2, implicitly crossing A after edge 2. Similarly we add connector C. The resulting ordering is ([A,B],[A,B],[A],[A,C],[A,C]). Next we add connector D, it is added last on edges 2 and 3 but since it enters above C it is ordered between A and C in edge 4. The final resulting ordering is ([A,B],[A,B,D],[A,D],[A,D,C],[A,C]). The resulting diagram is shown in Figure 3(c). □

Theorem 3. *If the shared edge graph is path consistent the above ordering algorithm produces segment orders with the minimal number of connector crossings, and all connector crossings are produced at the end of the last (in the pseudo direction) shared edge of the two connectors.*

Proof. (Sketch) Consider any pair of (sub-routes of) connectors A and B in a tree of shared edges. The algorithm ensures that the relative order of A and B is fixed in all their shared edges. By definition this order is defined by their left to right order on entry to the shared edge. Hence the two connectors can only cross at the exit of the shared edge, and only do so if that is necessary. □

Theorem 4. *If the shared edge graph is planar then the above ordering algorithm produces a planar layout.*

Proof. (Sketch) The relative order of shared edged is always preserved from one endpoint of the connectors, thus a crossing will only be inserted if the relative order of the two endpoints is different, in which case the graph is not planar. □

---

[6] It may be that when starting from a split point that while c already shares an edge e' with connectors in S the ordering is not yet decided, in which case the ordering is determined by following the sub-route back across the split point and along the shared path to find the input ordering.

6.2 Final placement

The final step in the layout is to determine the exact coordinates of the orthogonal connector segments. This nudges connector routes a minimum distance apart to show the relative order of connectors with shared segments and also ensures that connectors pass down the middle of "alleys" in the diagrams when this does not lead to additional cost or additional edge crossings.

We collapse collinear segments in the connector routes into maximal horizontal and vertical segments. This means that segments in the path alternate horizontal and vertical alignment. We compute the horizontal and vertical position in separate passes. The horizontal pass works as follows and the vertical pass is symmetric.

1. Determine a desired horizontal position for all non-end segments in the connector. For the middle segment in an "S" or "Z" bend, this is the middle of the "alley" that the segment is in. For example, for the solid connector route shown in Figure 2(c), the dotted line shows the desired position for this segment. For the middle segment in an "$\sqsupset$" or "$\sqsubset$" bend, this is that of the vertex of the object that the segment bends around.
2. Generate a set of horizontal separation constraints to ensure that segments maintain their current relative horizontal ordering with each other and with the other objects in the diagram. In the case of shared segments the separation constraints impose the ordering determined previously. The constraints are designed to enforce non-overlap and also to stop segments passing through each other and so introducing additional connector crossings.
3. Project the desired values on to the separation constraints to find the horizontal position of the segments using the approximate projection algorithm *satisfy_VPSC* from [8, 9].

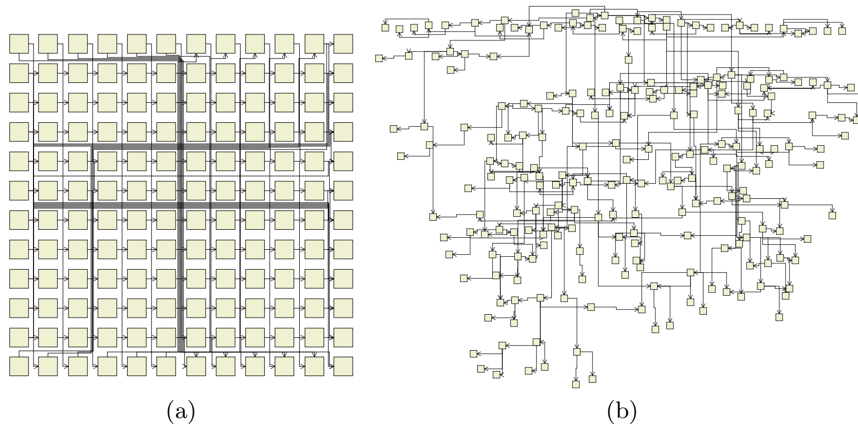
The constraints and desired positions can be generated using a variant of the line-sweep algorithm from [8] in $O((n+s)\log(n+s))$ time where n is the number of diagram objects and s the total number of vertical connector segments. The approximate projection algorithm has $O((s+n)^2)$ worst-case complexity but in practice $O((s+n)\log(s+n))$ complexity [9].

7 Evaluation

We have implemented all algorithms in the open source `libavoid` library and call them from the Dunnart diagram editor.[7] The library is written in C++ and compiled with `gcc` 4.2.1 at `-O3`. We have used the orthogonal routing algorithms to find routes in a variety of diagrams. Some examples are shown in Figure 4.

To investigate performance of the algorithms we ran the following experiment on a MacBook Pro with a 2.53 GHz Intel Core 2 Duo processor and 2GB of memory. The experiment used various sized grid arrangement of nodes, where each outside node is connected to the diagonally opposite node by a connector, and each node except those on the right and bottom edge is connected to the

[7] Dunnart, including the orthogonal routing features, is available for download from `http://www.dunnart.org/`.



(a) (b)

Fig. 4. Some diagrams used in the evaluation: (a) Grid-12x12 and (b) Graph-sparse.

Table 1. Average time taken to construct the orthogonal visibility graph, route all connectors, and compute final positions of all connectors for various sized grids and random graphs.

Diagram	Diagram size $\|V\|$	Diagram size $\|E\|$	VisGraph size $\|V\|$	VisGraph size $\|E\|$	Times (in msec.) to compute: VisGraph	RouteConns	FinalPos	Total
Grid-6x6	36	35	594	740	1	9	10	19
Grid-8x8	64	63	941	1,139	3	31	30	64
Grid-10x10	100	99	2,081	3,033	5	188	47	240
Grid-12x12	144	143	2,064	2,559	6	193	107	306
Graph-sparse	231	276	53,342	104,421	161	107	188	456
Graph-dense	305	413	51,255	99,976	118	1,532	357	2,007

node directly down and to the right. We also used two larger random graphs. Figure 4 shows the layout for a 12x12 grid and one of the random graphs.

We measured the time to construct the orthogonal visibility graph, the time to find all connector routes using the A* algorithm, the time to centre routes in channels and the time taken to perform nudging. The results are shown in Table 1. We found for many small examples, the routing process for the entire diagram can be performed in a fraction of a second. The size and construction time for the visibility graph in grid examples is notably smaller, as would be expected since the shapes have visibility just to their neighbours. The routing step is very fast in most cases, especially when close to optimal routes are available. The time required to centre routes is negligible, so should always be performed since it leads to more predictable routes. The `Graph-dense` example is notable for the smaller visibility graph and higher routing time, where both are due to the fact that many of the nodes in this graph are close together and obscure visibility or block the optimal routes of connectors.

8 Conclusion

Most diagram editors and graph construction tools provide some form of automatic orthogonal connector routing. However the routes are typically computed using *ad-hoc* techniques and are not usually updated during direct manipulation. We present algorithms for computing optimal object-avoiding orthogonal connector routings where optimality is w.r.t some monotonic function of the connector length and number of bends. Our approach is based on first computing an orthogonal visibility graph for the diagram, then an optimal route using an $A^\star$ search, followed by computation of the precise connector path. The approach is 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.

We plan to extend our work in three main ways. The first is to incorporate a cost for edge crossings in the penalty function. An advantage of using the orthogonal visibility graph and the shared path ordering step is that it allows easy identification of edge crossings. Thus computing how many times a new connector route crosses the previously routed connectors can be done simply and with little additional overhead. The second extension is to support connector routing which does not pass through cluster boundaries unnecessarily. This is closely related to the first extension since a cluster boundary can be treated as a kind of edge. The third extension is to develop even faster methods allowing incremental changes to the orthogonal visibility graph and connector routes.

Acknowledgments: NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. We acknowledge the support of the ARC through Discovery Project Grant DP0987168.

References

1. Wybrow, M., Marriott, K., Stuckey, P.J.: Incremental connector routing. In: GD 2005. Volume 3843 of LNCS., Springer (2006) 446–457
2. Dobkin, D.P., Gansner, E.R., Koutsofios, E., North, S.C.: Implementing a general-purpose edge router. In: GD 1996. Volume 1353 of LNCS., Springer (1997) 262–271
3. Miriyala, K., Hornick, S.W., Tamassia, R.: An incremental approach to aesthetic graph layout. In: Proceedings of the 6th International Workshop on Computer-Aided Software Engineering, IEEE Computer Society (Jul 1993) 297–308
4. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice-Hall, Inc. (1999)
5. Lee, D., Yang, C., Wong, C.: Rectilinear paths among rectilinear obstacles. Discrete Applied Mathematics **70**(3) (1996) 185–216
6. Argyriou, E., Bekos, M., Kaufmann, M., Symvonis, A.: Two Polynomial Time Algorithms for the Metro-line Crossing Minimization Problem". In: Graph Drawing: 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008, Revised Papers, Springer (2009) 336
7. Bekos, M., Kaufmann, M., Potika, K., Symvonis, A.: Line crossing minimization on metro maps. Lecture Notes in Computer Science **4875** (2008) 231
8. Dwyer, T., Marriott, K., Stuckey, P.: Fast node overlap removal. Lecture Notes in Computer Science **3843** (2006) 153
9. Dwyer, T., Marriott, K., Stuckey, P.: Fast Node Overlap Removal-Correction. Lecture Notes in Computer Science **4372** (2007) 446