Orthogonal Hyperedge Routing

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

[1] National ICT Australia, Victoria Laboratory,
Clayton School of Information Technology,
Monash University, Clayton, Victoria 3800, Australia,
{Michael.Wybrow,Kim.Marriott}@monash.edu
[2] National ICT Australia, Victoria Laboratory,
Department of Computing and Information Systems,
University of Melbourne, Victoria 3010, Australia,
pstuckey@unimelb.edu.au

Abstract. Orthogonal connectors are used in drawings of many network diagrams, especially those representing electrical circuits. Such diagrams frequently include hyperedges—single edges that connect more than two endpoints. While many interactive diagram editors provide some form of automatic connector routing we are unaware of any that provide automatic routing for orthogonal hyperedge connectors. We give three algorithms for hyperedge routing in an interactive diagramming editor. The first supports *semi-automatic routing* in which a route given by the user is improved by local transformations while the other two support *fully-automatic routing* and are heuristics based on an algorithm used for connector routing in circuit layout.

Keywords: orthogonal routing, hyperedges, circuit diagrams

1 Introduction

Orthogonal connectors are used in drawings of many network diagrams, especially those representing electrical circuits. Such diagrams frequently include hyperedges—single edges that connect more than two endpoints. While many interactive diagram editors provide some form of automatic connector routing we are unaware of any that support automatic routing for orthogonal hyperedge connectors. This is the problem we address.

In this paper we describe how we have extended the connector routing library `libavoid`[3] to support orthogonal object-avoiding hyperedge routing in a commercial diagramming tool for circuit diagrams and the Dunnart diagram editor.[4]

We give three algorithms for hyperedge routing that support interactive construction and routing of hyperedges in interactive diagramming tools. The first

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

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

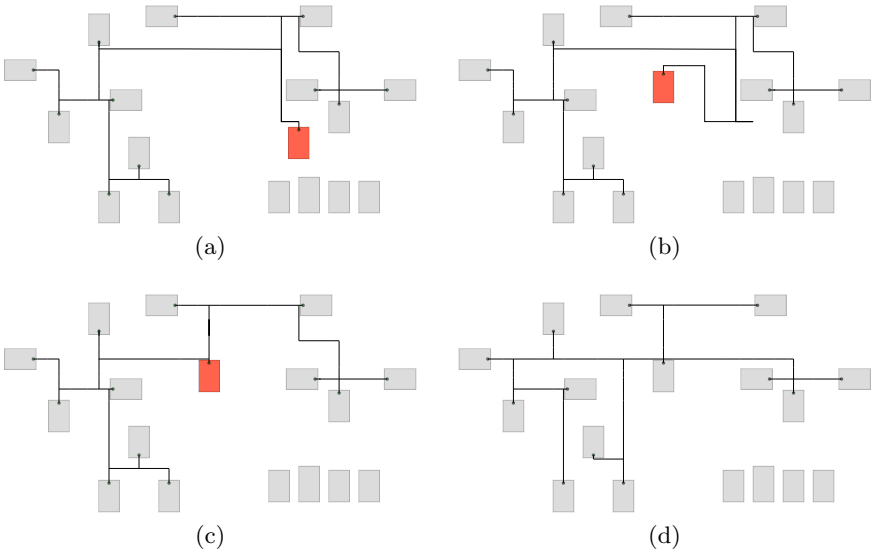


Fig. 1. Demonstration of the interaction model. (a) The diagram initially contains a single hyperedge made up of shapes, junction points and the connectors linking them. (b) The user drags a shape to a new location causing the connector between it and the junction point to be rerouted automatically. (c) Semi-automatic routing is performed to improve the hyperedge route. (d) If the user desires, they can perform fully-automatic routing to find a better topology for the hyperedge.

performs *semi-automatic routing* in which a route given by the user is improved by local transformations. The other two algorithms perform *fully-automatic routing* and are heuristic approaches extending an algorithm used for connector routing in circuit layout.

Previous research on connector routing in interactive diagramming tools has focused on poly-line and orthogonal routing for edges, i.e. arcs that connect two nodes [9, 10]. Hyperedge routing generalises this by allowing the connector to connect multiple nodes. Here we focus on computing orthogonal routes, i.e. routes composed of horizontal and vertical segments, reflecting the drawing conventions used in circuit design.

Orthogonal hyperedge routing generalizes the problem of finding a minimal length rectilinear Steiner tree (MRST) connecting a set of points in the plane [5]. Computing the MRST is NP-Complete [2] and several heuristics and exact methods are given in [5]. A number of heuristic methods have also been developed for finding obstacle-avoiding rectilinear Steiner minimal trees (OARSMTs) [1, 6] for automatic connector routing in VLSI design. Our problem differs from the standard problem studied in the VLSI setting because we are interested in supporting circuit construction in an interactive diagramming tool. This means that the algorithms need to be fast enough to support interaction and that the visual appeal and readability of the routes is important. Thus when comput-

ing the routes we penalize the number of bends as well as the total length and the semi-automatic routing step ensures that routes are visually distinct and pleasing in the sense that their paths are not obviously "bad."

Hyperedge routing is loosely related to edge bundling in which edge segments originating at the same node are collapsed together [3, 4, 8].

The remainder of this paper is organized as follows. In the next section we define our interaction model and formalize the automatic and semi-automatic routing problem. In Section 3 we discuss the semi-automatic routing algorithm which improves a route without changing topology. In Section 4 we give two algorithms for fully-automatic routing. We give experiments showing the effectiveness of the methods in Section 5. Finally in Section 6 we conclude.

2 Interaction Model and Problem Statement

Our algorithms are designed to support the following interaction model designed for interactive diagramming tools (Figure 1). It is designed to provide predictable automatic layout but allow the user to guide and override this. The four kinds of interaction are:

Creation: The user can create a new hyperedge by defining an initial route through specification of the bends and junctions that comprise the route. This specifies the topology of the route. The route is automatically improved so as to reduce segment lengths or bends but without changing the topology. We call this *semi-automatic* routing.

Editing: Whenever the user edits the diagram components, such as moving or deleting a node or diagram object, semi-automatic routing is used to improve the hyperedge routes while preserving the topology. See Figure 1(b) and (c).

Automatic routing: If the user is unhappy with a particular hyperedge route they can explicitly request that the tool performs *fully-automatic routing* in which case the system uses heuristic approaches to MRST to find an initial route which is then improved using semi-automatic routing. See Figure 1(d).

Manual adjustment: If the user is still unhappy they can manually modify the hyperedge route.

Since explicit support for hyperedges is not common in interactive diagramming tools, users will often work around this by using multiple individual connectors converging at junction points (or small dummy shapes if the software doesn't support junctions). This is actually a reasonably natural representation for hyperedges since it allows for easy incremental construction and alteration by the user, e.g., drawing a connector from a shape to an existing point on the hyperedge path. The difference with our approach is that the junction positions do not need to be tediously managed by the user but can be automatically positioned in response to diagram changes. Without semi-automatic routing its the users responsibility to manually modify Figure 1(b) to become Figure 1(c).

Notice that the interaction model requires that semi-automatic routing is performed very quickly since it must be applied to all hyperedges after most

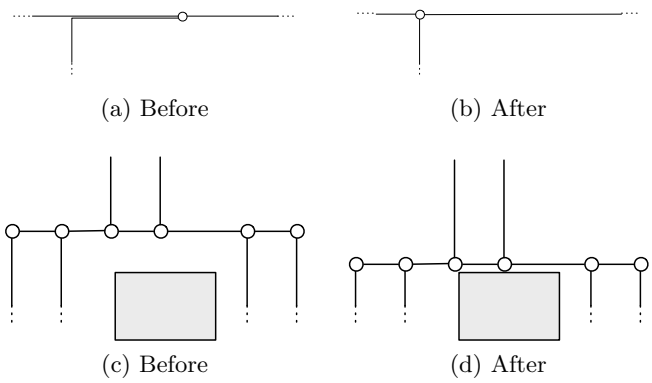


Fig. 2. The two local transformations used to improve the initial route. At the top is moving a junction to merge parallel routes, at the bottom moving a segment to reduce overall hyperedge length.

editing actions. In contrast, automatic routing can be slower since it is only performed for a subset of hyperedges at a time, and only when explicitly requested by the user.

We formalize automatic and semi-automatic hyperedge routing as follows. We have a set of nodes N and a set of hyperedges H. Each $i \in N$ has a fixed position, width and height as well as a set of connector ports P on its perimeter. Each $p \in P$ is a connector port with a direction of visibility. Each hyperedge $h \in H$ is a non-empty subset of connector ports. We wish to find a route R for each h. This is a set of horizontal and vertical segments R that form a tree whose leaf vertices are the connector ports h. The route should not pass through any of the nodes and should minimize a penalty function $\mathsf{p}(R)$ that is a monotonic function f of the length of R, $||R||$, and the number of bends (or equivalently segments) in R, $bends(R)$, i.e. $\mathsf{p}(R) = \mathsf{f}(||R||, bends(R))$. We sometimes refer to the leaf vertices of the route as the *terminals* and to the internal nodes as *junctions* and *bendpoints*. In the case of semi-automatic routing we are given an initial route R' for h which we must improve.

3 Semi-Automatic Routing

Semi-automatic routing has two steps. The first step is to perform *local improvement* on the initial route to improve it by rectifying bad routing that is obvious to the human observer. The local improvement step is novel and is a result of examining many routes and identifying how to improve these manually.

Local improvement is designed to make local changes to the hyperedge which reduce edge length and bends. We first build R, a tree representing the routing for the hyperedge, with the root node of the tree being one of the junctions and the leaf nodes being the terminal points. Other nodes within the tree are made

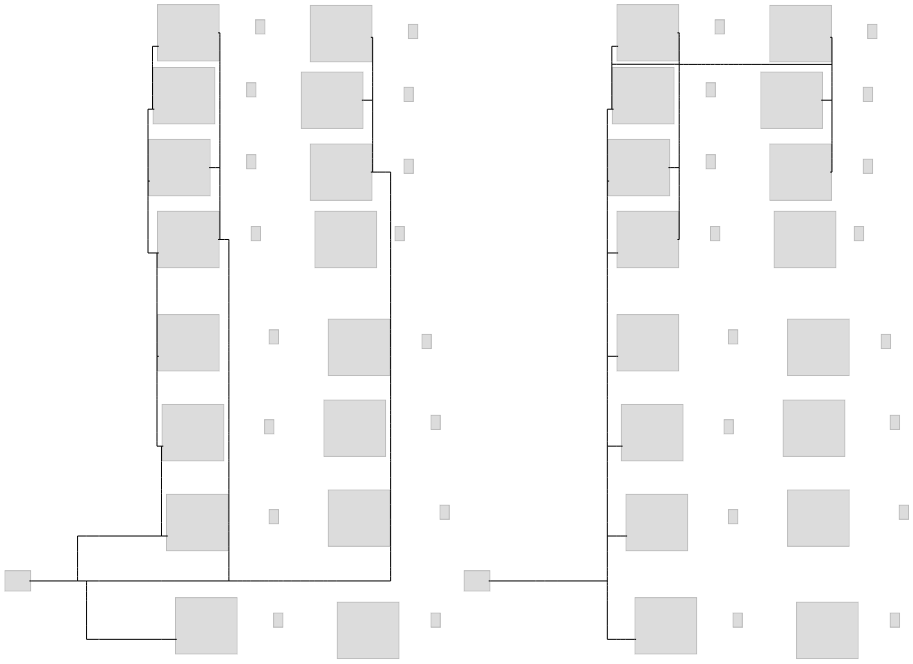
up of bendpoints and the remaining junctions from the hyperedge. We use two local transformations on this tree. They are illustrated in Figure 2.

The first transformation is to *remove redundant edges*. This looks at each junction node and if any of the edges in the tree both have another common endpoint (as well as the junction node), then the junction is moved to that node and the redundant edge is removed. This reduces the overall connector length and removes a bend point. The transformation is often not initially necessary but is still important in cases where the user has manually placed junctions at positions that may cause the shortest paths from multiple terminals to converge together before reaching the junction.

The second transformation is first applied in the horizontal dimension, then the vertical. In the case of the horizontal dimension we move a vertical line segment (an edge from our path tree) horizontally within the available space bordered by obstacles, and in the direction with the most divergent paths. This is shown in Figure 2(c–d) for a horizontal line segment in the vertical dimension. A line segment will consist of one or more collinear edges from the path tree, and thus multiple nodes. Using these nodes we maintain for each segment a count of the edges that diverge to each side. We also perform a sweep of the diagram, similar to that described in [10], to give us available space to shift each segment in either direction. We then repeatedly shift unbalanced segments, either to an obstacle boundary, or to the endpoint of the shortest diverging segment in that direction, updating the balance counts and merging segments as we do this. We will also shift balanced segments up to a diverging segment when doing so reduces the overall "length" of the hyperedge taking into consideration the penalty for each bend. Once there are no more segments to shift in that dimension, we remove redundant edges and perform the symmetric vertical process.

For an individual connector the transformations either remove a segment from the route or move the segment against the side of an obstacle (i.e., a shape expanded slightly with some buffer space). Thus, they can be applied no more than $O(n + s)$ times where s is the number of segments in the hyperedge. The sweep takes $O(n \log n)$ time for each dimension. However, in practice local improvement is very fast.

The second step in semi-automatic routing is nudging and centering. This is performed on all hyperedges (and edges) together and is based on that for orthogonal edge routing [10]. We first determine the relative ordering of connectors in shared edges. In order to make the connector route clearer we want to nudge these paths apart to make the paths visually distinct. It is important to do so in a manner which does not introduce unnecessary crossings or bends in segments. Based on this we 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. This is described more fully in [10]. If n is the number of diagram objects and s the total number of connector segments then this step has $O((s + n)^2)$ worst-case complexity.



(a) Heuristic 1 (b) Heuristic 2

Fig. 3. Real-world circuit diagram example showing difference between the two fully-automatic routing heuristics. (a) Sequential construction of MTST does not appropriately discount any shared segments on the paths in the MCST. (b) Interleaved construction of SPTF and MTST creates better routes, closer to what a human would draw. These diagrams show the raw output of each heuristic, before the hyperedge improvement step is performed. Note that Heuristic 2 also tends to result in less work needing to be performed in the subsequent improvement stage.

4 Fully Automatic Routing

As we have seen, computing an optimal hyperedge route is NP-Hard. In this section we describe two polynomial time heuristics for computing an initial hyperedge route. When combined with the preceding approach for semi-automatic layout they give a method for fully-automatic routing. Computing the initial route is quite an expensive operation. However in our model for user interaction its use is explicitly controlled by the user who must select a set of hyperedges to be rerouted by the tool.

The basis for our heuristics is the observation that when finding routes minimizing the penalty function we need only consider routes in the *orthogonal visibility graph*. This was introduced in [10] and 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

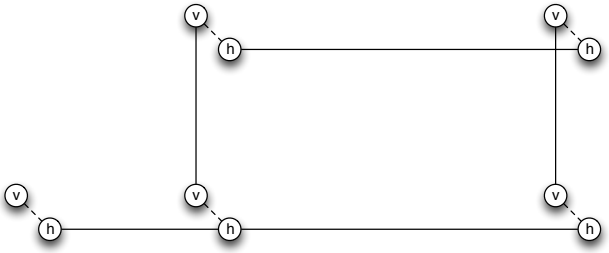


Fig. 4. A separated orthogonal visibility graph: v nodes and vertical edges are on a plane above the h nodes and horizontal edges. The dashed edges connect the two planes and correspond to a bend.

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.

We slightly modify the orthogonal visibility graph to produce a *separated orthogonal visibility graph* in which each node is split into two nodes (conceptually on two different planes) corresponding to whether it is connected to horizontally or vertically neighbouring nodes. Additionally, we add a link between these two nodes with a weight representing the bend penalty. This simplifies the algorithms because length and bend penalties are treated uniformly. Figure 4 illustrates the separated graph.

The orthogonal visibility graph can be constructed in $O(n^2)$ time for a diagram with n objects and contains $O(n^2)$ vertices and $O(n^2)$ edges. An example orthogonal visibility graph is shown in Figure 5(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.

4.1 Heuristic 1: Sequential construction of MTST

The starting point for our first heuristic is the VLSI routing algorithm of Long et al. [7]. This algorithm uses the standard (non-orthogonal) visibility graph (using the Manhattan distance on visibility edges) and is designed to find the route of minimal length. We have altered it to work with the separated orthogonal visibility graph in order to take the number of bends into account when computing the cost of a route. Figure 5 gives an overview of the main steps in fully-automatic routing with this heuristic for an example layout.

Our heuristic works by first constructing the *shortest path terminal forest (SPTF)* for the orthogonal visibility graph where the terminals are the hyperedge

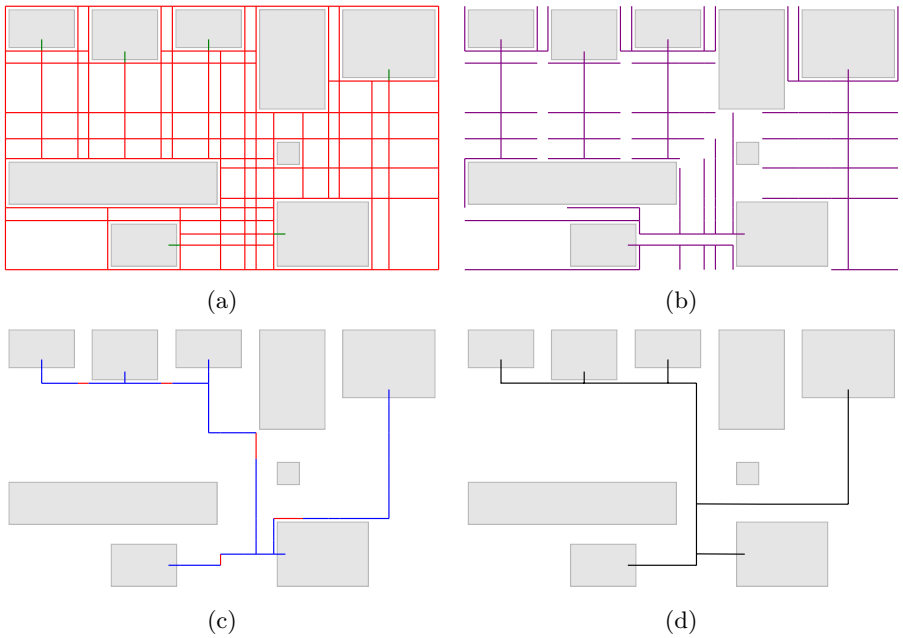


Fig. 5. Heuristic 1: In the sequential fully-automatic hyperedge routing the initial route is found by: (a) computing the separated orthogonal visibility graph, (b) constructing the shortest path terminal forest (SPTF) and using it to compute an (c) initial hyperedge routing from the minimum terminal spanning tree (MTST). Semi-automatic hyperedge routing takes an initial routing and improves it by (d) performing local optimization followed by centering and nudging.

nodes and the edges are weighted by their length. This is computed using an extended Dijkstra shortest path algorithm. This processes edges in order of least distance from a terminal creating a shortest cost tree around each terminal. When processing an edge, if its endpoint is not already in a tree it is added to the tree, otherwise it is marked as a *bridge edge* if its endpoint belongs to a different terminal trees or else ignored if its endpoint belongs to the same tree (*self edge*). An example SPTF can be seen in Figure 5(b).

Importantly, the use of a high bend penalty along with our separated orthogonal visibility graph modification results in shortest cost trees that grow a long way in a straight line before branching. This helps our approach produce ideal two-segment connections between far apart terminals that could otherwise be blocked by the "bushy" growth of traditional SPTFs.

At the end of this step all nodes in the orthogonal visibility graph belong to exactly one terminal's shortest cost tree, and edges between these trees are marked as bridges. Next an extended Kruskal minimum spanning tree algorithm is used to find the *minimum cost spanning tree* (MCST) using bridge edges that connects the terminal trees where the cost of a bridge edge is its weight plus

the cost to reach the terminal node from each of its endpoints. The minimum terminal spanning tree $MTST$ is then the bridge nodes in the minimum cost spanning tree and the associated path to each terminal. For more details see [7].

As the number of nodes and edges in the visibility graph is $O(n^2)$ the time complexity of computing the SPTF, MCST and MTST is $O(n^2 \log n)$.

4.2 Heuristic 2: Interleaved construction of SPTF and MTST

The main limitation of Heuristic 1 is that for efficiency the SPTF is computed before the MCST. This means that the cost when computing the MCST does not appropriately discount any shared segments on the paths in the MCST, see for example Figure 3. In essence, we can improve the quality of the solution found by interleaving computation of the SPTF with that of the MCST and building the MTST as we go. Our algorithm for this interleaved computation is given in in Figure 6[5] and an example of the algorithm's operation is shown in Figure 7.

The algorithm works by constructing sub-routes connecting disjoint subsets of the terminals in the original hyperedge, repeatedly combining these using a bridge edge from MCST until all of the terminals are connected in which case a MTST has been found.

The algorithm uses two priority queues *MCSTpq* and *SPTFpq* for computing the MCST from the sub-routes and the SPTF. Nodes n in the separated orthogonal visibility graph (VG) are annotated to indicate whether they have been reached in the SPTF ($reached[n]$), and if so the cost of the path to them ($cost[n]$), and the sub-route ($route[n]$) from which the path originates (a set of edges). Elements in *SPTFpq* are tuples (c, n, n', R) indicating that node n can be reached from node n' with a path of cost c from sub-route R. Elements in *MCSTpq* are tuples (c, n, w, n') indicating that nodes n and n' have been reached in the SPTF from different sub routes, say R and R', and that R and R' can be connected by a path with a total cost of c passing through bridge edge (n, w, n'). We assume a function $\mathsf{terms}(R)$ which returns the set of terminal nodes appearing in route (set of edges) R.

The algorithm repeatedly does one of two things. It can extend the SPTF around the current set of sub-routes by popping a tuple from the *MTSTpq* and adding non-self edges and non-bridge edges to the *SPTFpq*. Whenever a bridge edge is encountered the appropriate path is added to the *MCSTpq*. Or it can pop a tuple (c, n, w, n') from *MCSTpq* and merge the two sub-routes using the path through (n, w, n') and add nodes on the merged route to the *MTSTpq* with a zero cost. The algorithm stops when it has created a sub-route connecting all of the terminals in the hyperedge.

[5] For the sake of pedagogical clarity the algorithm in Figure 6 omits several details important for implementation. For example, an implementation should use a number greater than any potential path in the diagram (including penalties) in place of ∞. Also, the algorithm description assumes that the priority queues will never be empty. This can happen in the case where obstacles prevent all possible paths between sections of the hyperedge. Please see the implementation in `libavoid` for more information.

```
VG := separated orthogonal visibility graph
SPTFpq := {(∞, ⊥, ⊥, ∅)} ; MCSTpq := {(∞, ⊥, 0, ⊥)}
reached[⊥] := true
for each node n in VG do reached[n] := false endfor
for each terminal n in h do
   add (0, n, ⊥, ∅) to SPTFpq
endfor
repeat
   if 2 × cost of top tuple on SPTFpq < cost of top tuple on MCSTpq then
      (c, n, np, R) := pop SPTFpq
      if ¬reached[n] ∧ reached[np] then
         reached[n] := true ; cost[n] := c ; route[n] := R
         for each edge (n, w, n′) in VG do
            if reached[n′] then
               if terms(R) ≠ terms(route[n′]) then /* bridge edge */
                  add (c + cost[n′] + w, n, w, n′) to MCSTpq
               endif
            else /* non self + non bridge edge */
               add (c + w, n′, n, R ∪ {(n, w, n′)}) to SPTFpq
            endif
         endfor
      endif
   else /* found two sub-routes to connect */
      (c, n, w, n′) := pop MCSTpq
      R := route[n] ∪ route[n′] ∪ {(n, w, n′)}
      if terms(R) = h then
         return R /* complete hyperedge route */
      for each node n″ where terms(route[n″]) ⊆ terms(R) do
         reached[n″] := false
      endfor
      for each node n″ in R do
         add (0, n″, ⊥, R) to SPTFpq
      endfor
   endif
forever
```

Fig. 6. Heuristic for computing a route R for hyperedge h using interleaved construction of the SPTF and MCST.

The choice of whether to extend the SPTF or to merge two sub-routes depends on the cost of the top tuples of *SPTFpq* and *MCSTpq*. If the current top tuple on *SPTFpq* has cost c then we can safely commit to joining the two sub-routes R and R' connected by the bridge edge (n, w, n') in the top tuple of *MTSTpq* if its cost c' is no more than $2 \times c$. This is because we have found the minimum cost path between the sub-routes connected by (n, w, n'). To see this consider any other path p' between R and R'. If all nodes on the path have been reached, then the path must be in *MCSTpq* and since it was not the top of the heap, its cost is no less than c. Otherwise not all nodes on the path have been reached. This means there are two nodes on the path n_R and $n_{R'}$ respectively reached from R and R' that are in *SPTFpq*. But this means the cost of getting to n_R from R is at least c and to $n_{R'}$ from R' is also at least c and so the total cost of this path is at least $2 \times c$.

The disadvantage of this algorithm is the increased time complexity. Basically, whenever we process a path from the *MCSTpq* we recompute the SPTF around that path. This means that in the worst case the algorithm has time complexity $O(kn^2 \log n)$ where k is the number of terminals in the hyperedge.

5 Evaluation

We have implemented all algorithms in the open source `libavoid` connector routing library. These features can also be used interactively from within the Dunnart diagram editor.[6] We have used the orthogonal hyperedge routing algorithms to find routes for a variety of diagrams.

To investigate performance of the algorithms we ran the following experiment on a 2008 MacBook Pro with a 2.53 GHz Intel Core 2 Duo processor and 4GB of memory. Our C++ `libavoid` implementation was compiled using `gcc` 4.2.1 with `-O3`. The experiment used a small representative example from our commercial partner as well as some larger, randomly generated examples. We measured the time for each stage of the routing. The results are shown in Table 1. Note the

[6] `http://www.dunnart.org/`

Table 1. Average times taken to compute fully-automatic routing for hyperedges in a representative circuit diagram and for several larger randomly generated instances. (Please note, P is the total number of connection pins among all hyperedges H, and the VisGraph and Improve times are global rather than per hyperedge.)

	Diagram size			VisGraph size		Times (in msec.) to compute			
Diagram	$\|N\|$	$\|H\|$	$\|P\|$	$\|V\|$	$\|E\|$	VisGraph	Heuristic1	Heuristic2	Improve
Circuit	52	5	57	6,948	11,083	16	155	167	12
Random-1	200	1	25	20,142	35,674	56	47	142	25
Random-2	200	1	50	22,674	40,035	62	61	187	64
Random-3	400	1	50	28,117	46,952	82	70	304	213
Random-4	400	10	250	33,587	52,805	97	74	197	466

visibility graph construction and the improvement is performed only once for each diagram, whereas the times for the full rerouting are average times per hyperedge.

We found that routing hyperedges of 25–50 terminals in a small diagram of up to 200 hundred nodes can be performed in a fraction of a second. In the largest example of 400 nodes, full rerouting for 10 hyperedges (each with 25 terminals) with heuristic 1 took 1.7 seconds, or just under 3 seconds using heuristic 2. Of course this would be considerably less if the user was requesting rerouting of just a single hyperedge. In general, the interleaved heuristic was approximately 3 times slower than the sequential approach, but resulted in much better hyperedge routes. The interleaved heuristic also lead to less improvement work being necessary, though with insignificant gains. Adding additional hyperedges to any example requires the cost of running the heuristic approach when a user require rerouting be performed, as well as a small time increase for the local improvement stage.

6 Conclusion

We have given a practical approach to support hyperedge routing in a diagramming tool designed for electrical circuit design. It produces high-quality routings and is fast enough to be used for interactive diagramming. Our interaction model supports hyperedge creation and semi-automatic routing to improve routes after changes in the diagram, as well as fully-automatic routing, when changes in the diagram suggest the topology of the hyperedge should change. We give two heuristics for fully-automatic routing, one very similar to standard VLSI approaches, and a novel interleaving approach that better captures the cost of resulting hyperedge. The interleaved heuristic creates markedly better routes at about three times the runtime cost of the sequential approach.

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 and DP110101390.

References

1. Ajwani, G., Chu, C., Mak, W.K.: FOARS: FLUTE based obstacle-avoiding rectilinear steiner tree construction. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 30(2), 194–204 (Feb 2011)
2. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
3. Holten, D.: Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. IEEE Transactions on Visualization and Computer Graphics 12, 741–748 (September 2006)

4. Holten, D., van Wijk, J.J.: Force-directed edge bundling for graph visualization. Comput. Graph. Forum 28(3), 983–990 (2009)
5. Hwang, F.K., Richards, D.S., Winter, P.: The Steiner Tree Problem. Annals of Discrete Mathematics, Elsevier, North-Holland (Oct 1992)
6. Lin, C.W., Chen, S.Y., Li, C.F., Chang, Y.W., Yang, C.L.: Efficient obstacle-avoiding rectilinear steiner tree construction. In: Proc. of the 2007 Int. Symp. on Physical Design. pp. 127–134. ISPD '07, ACM, New York, NY, USA (2007)
7. Long, J., Zhou, H., Memik, S.O.: EBOARST: An efficient edge-based obstacle-avoiding rectilinear steiner tree construction algorithm. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 27(12), 2169–2182 (2008)
8. Pupyrev, S., Nachmanson, L., Kaufmann, M.: Improving layered graph layouts with edge bundling. In: Proc. of 18th Int. Symp. on Graph Drawing (GD'10). LNCS, vol. 6502, pp. 329–340. Springer (2011)
9. Wybrow, M., Marriott, K., Stuckey, P.J.: Incremental connector routing. In: Proc. of 13th Int. Symp. on Graph Drawing (GD'05). LNCS, vol. 3843, pp. 446–457. Springer (2006)
10. Wybrow, M., Marriott, K., Stuckey, P.J.: Orthogonal connector routing. In: Proc. of 17th Int. Symp. on Graph Drawing (GD'09). LNCS, vol. 5849, pp. 219–231. Springer (2010)

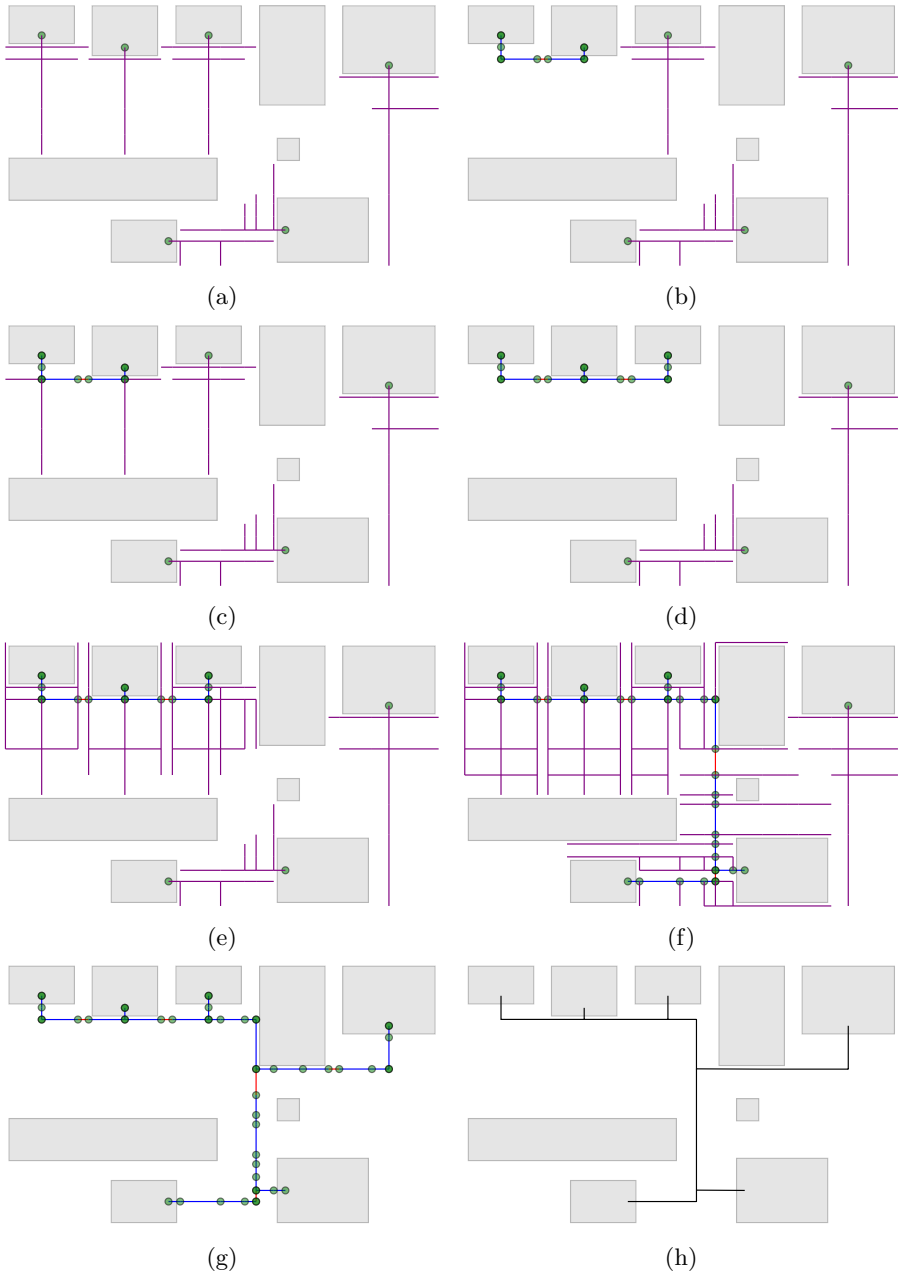


Fig. 7. Heuristic 2: In the interleaved approach we (a) incrementally build the SPTF, storing possible bridging edges. (b) We commit to the cheapest bridge when we reach a vertex with a cost more than twice the cost of the bridge. We then remove the bridged terminal's SPTFs and (c) continue, adding new terminals with zero cost for each vertex along the bridged path. (d–f) We repeat this process, until (g) there is just one terminal group remaining. This route is then improved by (h) performing local optimization followed by centering and nudging.