Short Papers

An Edge-Based Heuristic for Steiner Routing

Manjit Borah, Robert Michael Owens,
and Mary Jane Irwin, *Fellow, IEEE*

***Abstract*— A new approximation heuristic for finding a rectilinear Steiner tree of a set of nodes is presented. It starts with a rectilinear minimum spanning tree of the nodes and repeatedly connects a node to the nearest point on the rectangular layout of an edge, removing the longest edge of the loop thus formed. A simple implementation of the heuristic using conventional data structures is compared with previously existing algorithms. The performance (i.e., quality of the route produced) of our algorithm is as good as the best reported algorithm, while the running time is an order of magnitude better than that of this best algorithm. It is also shown that the asymptotic time complexity for the algorithm can be improved to $O(n \log n)$, where n is the number of points in the set.**

I. Introduction

Routing is one of the most time-consuming phases in layout of VLSI circuits and printed circuit boards. Routing involves connecting disjoint sets of points together using metal wires, usually along rectangular gridlines. Minimizing the length of the wires reduces the interconnect capacitance, cost of metal, and wiring area. A minimum rectilinear Steiner tree of a set of points connects the points together using the minimum total length of wire. Finding the minimum rectilinear Steiner tree is an NP-hard problem [8], however several heuristics for finding a good approximation exist [10], [14], [15], [5], [4], [3], [17].

The cost (i.e., the total rectilinear length of all the edges) of the minimum rectilinear spanning tree (MST) is at most 1.5 times more than that of the minimum rectilinear Steiner tree [11]. Therefore, any reasonable approach starting with a minimum rectilinear spanning tree will produce a Steiner tree no costlier than this bound. Many heuristics for Steiner tree approximations start with an MST. Ho, Vijayan, and Wong [10] proposed algorithms for finding optimal *L-shaped* and *Z-shaped* embeddings of rectangular layouts of a *separable* MST. Chao and Hsu [5] start with the MST and introduce Steiner points to the tree based on local and global refinements. Lee, Bose, and Hwang [15] modified Prim's algorithm to expand the current subtree by adding a point nearest to either an existing vertex on the tree or any point on the rectangular layout of some edge on the existing tree. Hwang [13] improved the above heuristic to make use of the MST to guide the search, which resulted in a faster algorithm. Bern and Carvalho [4] developed algorithms based on Kruskal's MST algorithm. Recently, Lim *et al.* [17] formulated another Steiner tree problem based on critical path delay reduction in the circuit. They also gave a heuristic that is similar in form to the Prim's MST heuristic with constraints on the edges to include the shortest path for the critical nets. Sarrafzadeh and Wong [20] took a hierarchical approach to the problem and developed a recursive algorithm that divides the tree into two fairly equal subtrees at each step until subtrees of sufficiently smaller size are reached, which are then solved explicitly. They also formulated another version of Steiner tree with the maximum weight edge minimized that they used on their global router [6].

Manuscript received August 4, 1993; revised August 12, 1994. This paper was recommended by Associate Editor M. Sarrafzadeh.

The authors are with the Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802 USA.

IEEE Log Number 9405723.

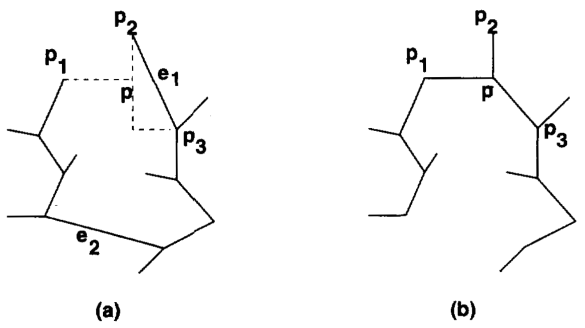

Fig. 1. Example of the edge-based update.

All of the above methods are based on the MST. However, there are other heuristics that approach the problem differently. Kahng and Robins' [14] 1-Steiner heuristic starts with the set of nodes and iteratively adds a new Steiner point to the set such that the MST for the new set is minimized among all such sets with one extra node. It has been shown with detailed comparisons in [14], [19] that the 1-Steiner heuristic has the best empirical performance in terms of reduction in cost of the Steiner tree with respect to the MST for the original set of points.

Our edge-based heuristic also starts with an MST and incrementally improves the cost by connecting a node to the rectangular layout of a neighboring edge and removing the longest edge in the loop thus formed. A simple $O(n^2)$ implementation (n-the number of points in the set) of the algorithm using conventional data structures was tested and compared with the batched 1-Steiner algorithm of [14]. The results show that the average percent reduction produced over the MST by our algorithm is the same as that of the batched 1-Steiner algorithm, while our algorithm is an order of magnitude faster than the batched 1-Steiner algorithm. Moreover, we also show that our edge-based algorithm has $O(n \log n)$ asymptotic time complexity using sophisticated data structures, which matches the lower bound on computing the MST of a set of points and thus matches the lower bound on any MST-based heuristic [12].

Lewis *et al.* [16] also independently developed a somewhat similar heuristic for computing the Steiner tree based on local improvements. They iteratively migrate to the best 'neighbor' of the existing tree that is obtained by removing an edge from the tree and connecting the two subtrees using another edge. Though the two heuristics are similar, their algorithm has a complexity of $O(n^4)$ and therefore is not suitable for real applications.

The rest of this paper is organized as follows. We describe our heuristic in Section II. A simple implementation of the algorithm is also described in Section II. Results of the test and comparison are given in Section III. Section IV describes an improvement on the asymptotic time complexity to $O(n \log n)$. Conclusions are drawn with remarks in Section V.

```
Algorithm Edge-based-Steiner()

Begin
  1.Compute the rectilinear minimum spanning tree of the set of nodes
  2.Compute all possible <node, edge> pairs that give positive gain
  3.Sort all the pairs in descending order of gain
  4.While (there are pairs with positive gain) do
        If (the two edges to be replaced exist in the tree) then
           Replace the pair of edges with three new edges and a new node.
        End-if
  End-while
End
```

Fig. 2. The basic edge-based algorithm.

II. The Edge-Based Heuristic

Our edge-based algorithm is based on connecting a node to the nearest point on the rectangular layout of an edge in the tree and removing the longest edge in the loop thus formed. Consider the tree fragment in Fig. 1(a). Here and in the rest of the paper, the edges are shown in euclidian only for clarity sake. All the distance measures are in rectilinear metric. If we connect the node p_1 to the nearest point p on the rectangular layout of the edge e_1, then it forms a loop in the tree. Suppose e_2 is the longest edge in the path between p_1 and p_2 in the tree. We can make the following modification to the tree (Fig. 1(b)):

1) Add node p
2) Remove edge e_1
3) Remove edge e_2
4) Add edge connecting p to p_1
5) Add edge connecting p to p_2
6) Add edge connecting p to p_3.

The above procedure adds a new node to the tree (*Steiner point*) and replaces a pair of existing edges with three new edges. After this modification, the resulting graph becomes a spanning tree for a new set with one extra node. Observe that the total cost of the two edges (p, p_2) and (p, p_3) together is equal to the cost of the edge e_1. Therefore, the reduction in the cost of the tree (*gain*) due to this operation is given by

$$gain = length(e_2) - length(p, p_1) \tag{1}$$

Our algorithm computes all such possible $\langle$node, edge$\rangle$ pairs that produce a positive gain and applies as many such edge-pair replacements as possible to the existing tree. The main block of the algorithm is given in Fig. 2. The total number of $\langle$node, edge$\rangle$ pairs possible for the whole tree is $O(n^2)$. However, each edge can be replaced only once in a single pass; i.e., the edge no longer exists after it has participated in an update. For example, in Fig. 1(a), e_1 may have more than one node (apart from p_1) that would result in a positive gain when connected to e_1. We can apply only one of these cases, since after the update is done (Fig. 1(b)) e_1 no longer exists. Therefore, we need to compute only the pair with maximum gain for any given edge, reporting only the 'best' pair for each edge. Observe that since our algorithm works in a 'batched mode' (i.e., in one pass it first computes all the possible tuples with positive gain before applying any of the updates), the tree is fixed for this phase of computing the tuples for the entire tree. Hence, the total number of pairs considered for steps 3 and 4 are linear. However, while applying the updates to the tree in step 4, we check to see that both the edges participating in the update exist in the tree.

The degree of any vertex in a rectilinear minimum spanning tree is bounded by six [10]. Therefore, the edge-pair replacement of Fig. 1 can be done in $O(1)$ time. A very simple $O(n^2)$ implementation of the algorithm is possible with conventional data structures, such as adjacency lists, as illustrated below.

1) The minimum spanning tree of the set of points (step 1) is computed in $O(n^2)$ time using *Prim's algorithm* [7].
2) We use a recursive routine similar to *depth first search* for each edge to compute the $\langle$node, edge$\rangle$ pair giving maximum gain involving that edge. (Start with the given edge as root and pass the maximum edge seen until now as parameter to the recursive calls.) It takes $O(n)$ time for each edge, thus this step takes $O(n^2)$ time.
3) Sorting the $O(n)$ pairs (one pair for each edge) in step 3 requires $O(n^2)$ worst case time using *Quicksort.*
4) Finally, applying the $O(n)$ updates to the tree (step 4) requires only $O(n)$ time.

Repeating the algorithm more than once, each time on the updated tree of the previous pass, usually produces further improvements. However, it is found from experimental results based on about 5000 examples of each size (Table I) that three iterations are sufficient in most cases and the need for more than four iterations is rare. Moreover, on the average the improvement obtained in the fourth pass is at most 0.01% (for netsize 50) and the maximum improvement obtained on the fourth pass, considering all netsizes is 0.18%, which is negligible. Thus, we can use only three iterations without much loss in performance.

III. Comparison with Other Existing Algorithms

The algorithm has been implemented and tested on a large number of random samples ranging in size (i.e., number of points) from 4 to 1000. Since there is no absolute measure of the quality of the route produced by a Steiner routing heuristic or algorithm, the average percent improvement over the cost of the minimum rectilinear spanning tree (also called performance) is commonly used to compare different algorithms [19]. Using this metric, the 1-Steiner heuristic has been shown to produce the best average percent improvement among the previously reported heuristics with proven time bounds. Recently, it is also claimed, based on empirical results, that on the average the 1-Steiner algorithm is only 0.25% away from optimal for up to 8 node sets and less than 0.50% away from optimal for 20 nodes [1]. We compared our algorithm directly with the batched 1-Steiner algorithm on the same set of examples. The points were drawn at random from an uniform grid of size 10000 × 10000. The percent improvement obtained by our edge-based algorithm and the batched 1-Steiner algorithm for problems varying in size between 4 and 200 are shown in Table II. This result is based on the average of about 5000 random examples of each size. The comparison shows that our edge-based

TABLE I
IMPROVEMENTS OBTAINED ON DIFFERENT PASSES OF THE ALGORITHM

size	Passes Required			Avg. improv./pass (%)				Max. improv./pass (%)			
	Avg	Max	Min	1st	2nd	3rd	4th	1st	2nd	3rd	4th
6	1.41	3	1	8.88	.99	.02	0	19.69	11.21	5.19	0
10	1.7	3	1	9.15	.97	.01	0	15.04	4.65	.47	0
50	2.3	4	1	9.08	1.31	.06	.01	12.69	2.87	.61	.07
100	2.5	5	2	9.28	1.40	.07	.002	11.26	3.12	.49	.18
200	2.7	4	2	9.48	1.36	.07	.001	11.05	2.24	.37	.04
500	3.3	4	3	9.46	1.41	.05	.002	10.39	1.79	.15	.02

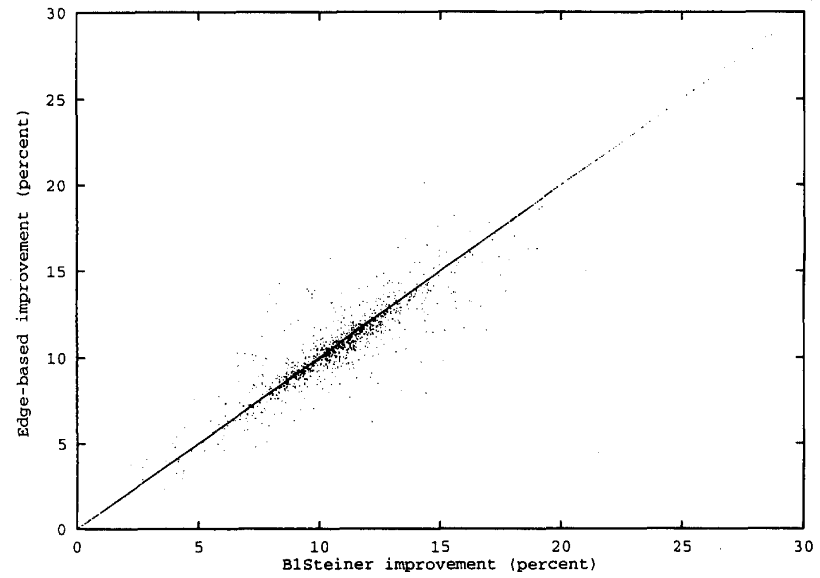

Fig. 3. Batched 1-Steiner algorithm versus edge-based algorithm on 6000 random samples.

algorithm performs as well as the batched 1-Steiner algorithm. The last two lines of Table II show the percent improvement obtained by our edge-based algorithm for 500 samples each of sizes 500 and 1000. The batched 1-Steiner algorithm takes too long for sizes 500 and 1000 to collect enough data. Fig. 3 shows a scatter plot of the percent improvements obtained by the two algorithms on 6000 random samples of various sizes. The Y-axis represents the percent improvements obtained by our edge-based algorithm and the X-axis represents the same produced by the batched 1-Steiner algorithm. It can be easily observed that the two algorithms produce about the same improvement in cost in most of the cases. While in a few cases batched 1-Steiner is better than the edge-based algorithm, there are about the same number of other examples where the edge-based algorithm produces better routes than the batched 1-Steiner algorithm.

Next, we looked at the time taken by the two algorithms to compute the above results. We ran our implementation of the edge-based algorithm and the batched 1-Steiner code obtained from the authors of [14] together on the same sets of data on a Sun4 machine and measured the average CPU time taken for about 5000 samples of each size. Table III shows the results of the execution time comparisons for samples varying in size from 5 to 200. The average execution times taken by our edge-based algorithm for net sizes 500 and 1000 based on about 500 runs for each size are also shown in the last two rows of Table III.

TABLE II
AVERAGE PERFORMANCE COMPARISON OF THE TWO ALGORITHMS

Size of net	*Batched 1-Steiner*	*Edge-based*
4	8.63%	8.63%
5	9.54%	9.54%
6	10.03%	10.02%
10	10.36%	10.33%
20	10.44%	10.40%
50	10.71%	10.71%
100	10.89%	10.84%
200	10.88%	10.88%
500	-	10.94%
1000	-	11.04%

Based on experimental results, our edge-based algorithm is 5 times faster than the batched 1-Steiner algorithm for problems of size as small as 5 nodes and about 70 times faster than the batched 1-Steiner algorithm for problems of size 200. Moreover, our algorithm computes the route for a problem with 1000 nodes in less than 3 minutes.

In CAD systems for VLSI layout generation, routing represents a major portion of the running time. Since our algorithm has the

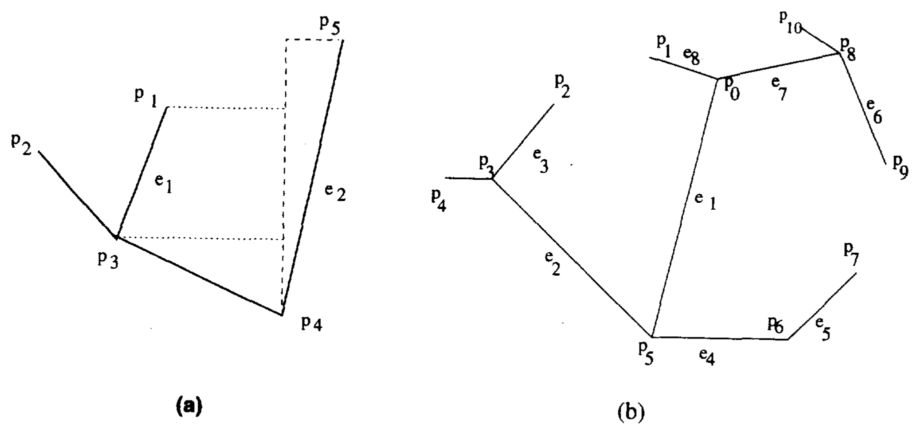

Fig. 4. Visible nodes and blocking.

```
Algorithm Horizontal-sweep()
Input: The Sorted list of vertical segments
Output: The <visible-node, edge> pairs to the left or right
Major data-structures: A balanced tree for maintaining 'active' intervals

Begin
  For (each vertical edge segment) do
    If ('left' vertical segment) then
      Report all the nodes of the 'active' segments overlapped by this
          segment as visible from the incoming edge
      Mark the reported nodes as 'blocked'
      If (any active segment is partially overlapped) then
        Report the appropriate node of the incoming segment as
            visible from the overlapped edge
      If (both nodes of a segment are blocked) then
        Delete that segment from the set of active segments.
    If ('right' vertical segment) then
      Remove the corresponding left segment from 'active' set, if exists
      Insert the segment into the set of active segments
End.
```

Fig. 5. The horizontal sweep of the sweeping algorithm.

advantage of producing very close to optimal Steiner trees and running an order of magnitude faster than the best existing algorithm, it stands as the best candidate for use in such layout generators. Moreover, the implementation of the algorithm requires the use of only simple and conventional data structures, which makes it easy to interface with other tools in the system.

IV. Improving the Asymptotic Time Complexity

In [12] Hwang gave an $O(n \log n)$ algorithm to find the rectilinear minimum spanning tree of a set of nodes by first constructing a voronoi diagram of the points in $O(n \log n)$ time and thus transforming the problem into a planar graph problem that can easily be solved in $O(n \log n)$ time. Computing all possible $\langle$node, edge$\rangle$ pairs with positive gain can be done in $O(n \log n)$ time. This is based on the following simple observations.

A. Visible Nodes and Blocking

Consider the tree in Fig. 4(a). Nodes p_1 and p_3 may be connected to edge e_2 resulting in a modification similar to Fig. 1. But the node p_2 cannot be connected to e_2 in the same way. (In fact, p_2 should probably be connected to e_1.) In a sense, e_1 is 'blocking' p_2 from connecting to e_2 in the tree. We will say node p_1 and p_3 are *visible* to edge e_2 and node p_2 is *blocked* from e_2 by edge e_1. In this example, the nodes p_1 and p_2 are visible from the edge e_2, while p_5 is visible

TABLE III
Comparision of the Average CPU Time Required by the Two Algorithms

Size of net	*Batched 1-Steiner*	*Edge-based*	$\frac{\textit{Batched1-Steiner}}{\textit{Edge-based}}$
5	7.1 *msec*	1.4 *msec*	5
6	13.3 *msec*	2.5 *msec*	5.3
8	32.1 *msec*	4.5 *msec*	7
10	59.6 *msec*	6.4 *msec*	9.3
20	456 *msec*	35 *msec*	13
50	6.53 sec	0.26 sec	25
100	52 sec	1.17 sec	44
200	395 sec	5.6 sec	70
500	>1.5 hrs	40 sec	-
1000	-	173 sec	-

```
Algorithm Edge-based-Steiner()
Begin
  Compute the MST of the set of nodes.
  Sort the vertical segments of the edges in ascending x-coordinates
     Also mark left and right vertical segments.
  Sort the horizontal segments of the edges in ascending y-coordinates
     Also mark top and bottom horizontal segments.
  Use sweepline algorithm to report all the <visible-node, edge> pairs
  Use Tree compression and NCA algorithms to compute the gain for each pair
  Sort all the pairs in descending order of gain
  Apply as many updates as possible from the sorted list.
End.
```

Fig. 6. The main algorithm for the $O(n \log n)$ version.

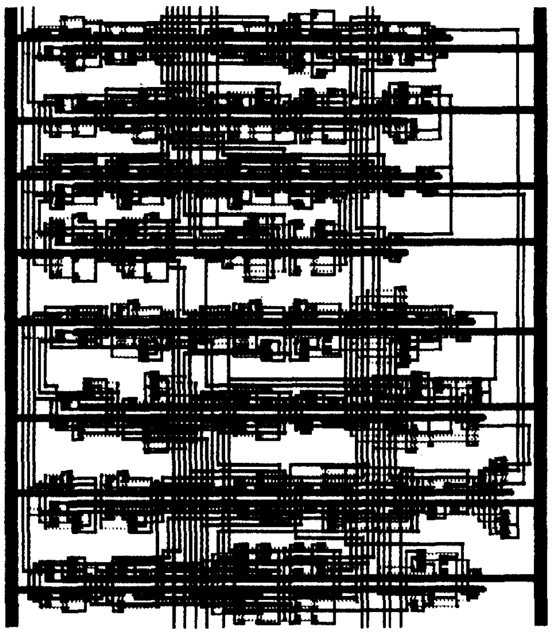

Fig. 7. ArtistII layout for 'alu' using edge-based steiner algorithm.

from e_1. From the above example, it is clear that: *A node may be connected to an edge for edge-pair replacement only if the node is visible from the edge.*

In Fig. 4(b), the edge e_1 has p_2 and p_3 visible to its left, p_7 and p_9 to its right, p_1 and p_6 above it and p_8 below. The edges $e_2, e_3, \ldots, e_8$ are the 'neighbor edges' of e_1. Any other point outside the neighbor edges (like p_4 or p_{10}) are blocked by these neighbors and not visible to e_1. This situation is true for any of the other edges in the tree. *The visible nodes for an edge are the nodes of its neighbor edges.* Therefore, we can report all the pairs of ⟨visible-node, longest-edge⟩ using a variation of the sweepline algorithm used to report overlaps of rectangles [22]. A top-level description of this sweepline algorithm is given in Fig. 5.

We represent an edge by the four rectilinear segments (*left, right, top* and *bottom*) of its rectangular layout. Degenerate edges (i.e., edges that are along the gridlines) are considered as special cases. For simplicity, the sweepline algorithm is split into two phases, a horizontal or left to right sweep using the vertical segments and reporting pairs with the visible nodes to the left or right of edges and a vertical or bottom to top sweep using the horizontal segments and reporting visible nodes above or below. We describe only the horizontal sweep here; the vertical sweep is similar. For the horizontal sweep, the vertical segments are sorted in ascending order on their x-coordinates. There are at most $2n$ such segments. An *interval tree* is used to maintain 'active' edge segments at any time of the left to right sweep. The tree is searched to report the visible nodes and blocking edges. When a *left* vertical segment of an edge is encountered, the tree is queried to report all the visible nodes to the left of the edge. These are exactly the active segments that this edge overlaps in the tree. The edges to which the two nodes of the incoming edge are visible to the right are the partially overlapped segments, and are also reported. When a right vertical segment is encountered, it is inserted into the tree. Since the interval tree data structure is balanced, operations like insertion and deletion take $O(\log n)$ time, while queries take $O(\log n + k)$ time where k is the number of nodes reported. Each edge segment in the sweepline algorithm is encountered only once, and only a linear number of nodes are reported. Thus, the sweepline algorithm requires $O(n \log n)$ time.

After all the ⟨node, edge⟩ pairs are found using the sweepline algorithm, the maximum length edges for each of the pairs need to be computed before the gain can be calculated. Since all the pairs are already available and the tree does not change during computation of the gain, we can use an off-line algorithm for this. Tarjan, in [21], gives an $O((n + m)\alpha(m + n, n))$ time algorithm for computing m such queries on a n node tree. Here $\alpha()$ is the inverse of the Ackerman's function and is very slow-growing. Tarjan's algorithm uses path compression along with the *nearest common ancestor* (NCA or LCA) algorithm to compute this. Note that efficient $O(n)$ preprocessing time and $O(m)$ query time algorithms for the nearest common ancestor problem exists [9]. Since m is $O(n)$ in this case, the queries can be computed in $O(n \log n)$ time. Now, computing the gain information for the $O(n)$ pairs takes only $O(n)$ time. Step 4 of sorting the $O(n)$ queries take $O(n \log n)$ time followed by $O(n)$ time for the updates to the tree. Hence, the algorithm has $O(n \log n)$ complexity. Our overall improved algorithm is given as pseudocode in Fig. 6.

Though this algorithm is asymptotically faster than the version of our algorithm described in Section II, an implementation of this algorithm would require the use of complex data structures (e.g., *splay trees*) and sophisticated programming techniques. Thus, we expect the constant involved in the asymptotic time to be large and that this algorithm may be empirically faster than the simpler version only for large problems (say, >100 nodes). We have not tested this version of the algorithm for running time comparisons, although we expect to see a drastic improvement in the running time for large-sized problems.

V. Conclusion

Our edge-based algorithm is used in *artistII* [18], which produces good layouts quickly. Fig. 7 shows a layout produced by *artistII* for the 'alu' benchmark circuit [2] using our edge-based algorithm. This implementation is very simple to understand and easy to use, since it uses conventional data structures like adjacency lists and arrays. The running time of the implemented algorithm is based on a naive $O(n^2)$ implementation, and further improvement on the running time is possible by applying better programming techniques. The approach taken in our algorithm is greedy, i.e., only edge-pair updates that results in a positive gain are considered. Our experiments show that by allowing a certain number of updates with negative gains in each iteration, only marginally better improvements can be obtained at the cost of a significant increase in the running time.

Acknowledgment

The authors thank Prof. Piotr Berman of Penn State University for his hints and advice regarding the asymptotic complexity. They also thank A. B. Kahng, G. Robins, and T. Zhang for providing their batched 1-Steiner program.

References

[1] T. Barrera, J. Griffith, G. Robins, and T. Zhang, "Narrowing the GAP: near-optimal steiner trees in polynomial time," Univ. of Virginia, Computer Science Dept., Tech. Rep. CS-93-31, June 1993.

[2] MCNC benchmark. *Physical Design Workshop*, 1989.

[3] Piotr Berman and Viswanathan Ramaiyer, "Improved approximations for the steiner tree problem," in *Proc. Symp. Discrete Algorithms*, Jan. 1992.

[4] M. W. Bern and M. de Carvalho, "A greedy heuristic for rectilinear steiner tree problem," Univ. California-Berkely, Tech. Rep., 1986.

[5] T. H. Chao and Y. C. Hsu, "Rectilinear steiner tree construction by local and global refinement," *IEEE Trans. Computer-Aided Design*, vol 13, no. 3, pp. 303–309, Mar. 1994.

[6] C. Chiang, M. Sarrafzadeh, and C. K. Wong, "Global routing based on steiner min-max trees," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 12, pp. 1318–1325, 1990.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, ch. 24, 1990.

[8] M. Garey and D. S. Johnson, "The rectilinear steiner problem is NP-complete," *SIAM J. Appl. Math.*, vol. 32, no. 4, pp. 826–834, 1977.

[9] Dov Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Computing*, vol. 13, no. 2, pp. 338–355, May 1984.

[10] J. M. Ho, G. Vijayan, and C. K. Wong, "New algorithms for the rectilinear steiner tree problem," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 2, pp. 185–193, 1990.

[11] F. K. Hwang, "On steiner minimal trees with rectilinear distance," *SIAM J. Appl. Math.*, vol. 30, no. 1, pp. 104–114, 1976.

[12] F. K. Hwang, "An $O(n \log n)$ algorithm for rectilinear minimal spanning trees," *J. Assn. for Computing Machinery*, vol. 26, no. 2, pp. 177–182, Apr. 1979.

[13] F. K. Hwang, "An $O(n \log n)$ algorithm for suboptimal rectilinear steiner trees," *IEEE Trans. Circuits Syst.*, vol. 26, pp. 75–77, 1979.

[14] A. B. Kahng and G. Robins, "A new class of iterative steiner tree heuristics with good performance," *IEEE Trans. Computer-Aided Design*, vol. 11, no. 7, pp. 893–902, July 1992.

[15] J. H. Lee, N. K. Bose, and F. K. Hwang, "Use of steiner's problem in suboptimal routing in rectilinear metric," *IEEE Trans. Circuits Syst.*, vol. 23, pp. 470–476, 1976.

[16] F. D. Lewis, W. C. Pong, and N. Van-Cleave, "Local improvements on Steiner trees," in *Proc. 3rd Great Lakes Symp. on VLSI*, 1993, pp. 470–476.

[17] A. Lim, S. W. Cheng, and C. T. Wu, "Performance oriented rectilinear steiner trees," *Proc. DAC*, June 1993, pp. 171–176.

[18] M. J. Irwin and R. M. Owens, "An overview of the penn state design system," in *Proc. DAC*, July 1987, pp. 516–522.

[19] D. Richards, F. K. Hwang, and W. Winter, *Steiner Tree Problems*. New York: North Holland, ch. 2, 1992.

[20] M. Sarrafzadeh and C. K. Wong, "Hierarchical Steiner tree construction in uniform orientations," *IEEE Trans. Computer-Aided Design*, vol. 11, no. 9, pp. 1095–1103, Sept. 1992.

[21] Robert E. Tarjan, "Applications of path compression on balanced trees," *J. Assn. for Computing Machinery*, vol. 26, no. 4, pp. 690–705, Oct. 1979.

[22] Jeffery D. Ullman, *Computational Aspects of VLSI*. Computer Science Press, Inc., ch. 9, 1984.