# Efficient Octilinear Steiner Tree Construction Based on Spanning Graphs*

Qi Zhu[1, 3], Hai Zhou[2], Tong Jing[1], Xianlong Hong[1], Yang Yang[1]

[1] Dept. of CST, Tsinghua Univ., Beijing 100084, P. R. China
[2] Dept. of ECE, Northwestern Univ., Evanston, IL 60208-3118, USA
[3] Dept. of EECS, UC Berkeley, Berkeley, CA 94720, USA
e-mail: zhuqi@eecs.berkeley.edu; haizhou@ece.northwestern.edu
e-mail: {jingtong, hxl-dcs, yycs99}@{mail, mail, mails}.tsinghua.edu.cn

Abstract --Octilinear interconnect is a promising technique to shorten wire lengths. We present two practical heuristic octilinear Steiner tree (OSMT) algorithms in the paper. They are both based on octilinear spanning graphs. The one by edge substitution (OST-E) has a worst case running time of *O(nlogn)* and similar performance as the batched greedy algorithm[9]. The other one by triangle contraction (OST-T) has a small increase in running time and better performance. Experiments on both industry and random test cases are conducted.

## I. Introduction

In an interconnect architecture, the Steiner minimal tree (SMT) construction is one of the key problems, and it will be computed hundreds of thousand times during floorplan and placement. Many of them have very large input sizes for increased emphasis on design for test and nets with pre-routes. Therefore, the Steiner minimal tree problem definitely deserves good performances and highly efficient solutions. As the foundation of octilinear interconnect, an octilinear Steiner minimal tree (OSMT) interconnects given points in octilinear plane, which allows 45 degree diagonal interconnections in addition to traditional horizontal and vertical orientations.

Some studies about octilinear Steiner tree have been made since 1990's [1]-[12]. Among the previous approaches, those that have relatively better performances may not achieve good efficiency.

The main contributions of this paper are two efficient octilinear Steiner minimal tree heuristic algorithms, which are faster than previous algorithms and have similar performances as the best heuristics such as the batched greedy algorithm [9]. One approach (OST-E) derives its efficiency from combining the edge substitution approach of Borah et al. [13] and spanning graphs similar to Zhou et al. [14]. The other one (OST-T) combines the triple contraction [10] and spanning graphs. Both approaches run in *O(nlogn)* time and take *O(n)* storage. The former one has a smaller constant factor and the later one has a better performance. Another advantage is that they are easy to be implemented since all the stages of these two algorithms can be integrated together, rather than using several separate complicated algorithms. Furthermore, because they are graph-based, they can be extended to any λ-geometry easily.

* This work was supported partly by the National Natural Science Foundation of China under Grant No.60373012, the NSF of USA under Grant No.CCR-0238484, the SRFDP of China under Grant No.20020003008, and Key Faculty Support Program of Tsinghua Univ. under Grant No.[2002]4.

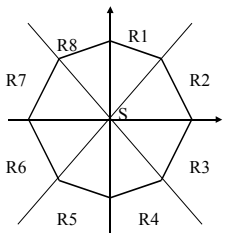


Fig.1. Equi-distant Points in octilinear plane.

The rest of this paper is organized as follows. In Section 2, we show how to define and generate spanning graphs on the octilinear plane. Then in Section 3, the two approaches of octilinear Steiner tree construction are presented. Section 4 gives experimental results and the last section is the concluding remarks.

## II. Octilinear Spanning Graphs

We construct an octilinear Steiner tree from a minimal spanning tree, so the first step is constructing an octilinear minimal spanning tree efficiently. Using Prim's or Kruskal's algorithm on a complete graph will take at lease $\Omega(n^2)$ time. Although Delaunay triangulation can be an efficient way to reduce the number of candidate edges, it may not be easily defined in rectilinear or octilinear metric. Zhou et al. [14] introduced the spanning graph as a base for the minimal spanning tree construction. Given a set of points on the plane, a spanning graph is a graph that contains at least one minimal spanning tree. They presented an *O(nlogn)* algorithm to construct a spanning graph of cardinality (that is, the number of edges) *O(n)*.

In the octilinear case, we first consider the contour of equidistant points from point *s*, as shown in Fig.1. We can divide the plane into eight regions based on the octagonal contour.

*Theorem 1*: Each region $R_i$ in Fig.1 has the uniqueness property. That is, for any points $p, q \in R_i$, $||pq||_4 \le \max(||sp||_4, ||sq||_4)$.

Since each region $R_0$ has the uniqueness property, we can design a sweep line algorithm with a worst-case running time of *O(nlogn)* to construct the spanning graph. It works as follows. First, the points will be sorted in non-decreasing order of their distance to an imaging point, in order to find the closest points in each region. For example, in region $R_1$, we set the imaging point at the position $(-\infty, -\infty)$ and sort the points of $\sqrt{2} * x + y - x$. In this order, after each point $p$ is swept, the first point seen in its $R_i$ region will be its closest point in that region, which will be connected to $p$. We use an active set $A_i$ to keep the swept points waiting for closest points. When a

new point is swept, we search $A_i$ to find points with the new point in their $R_i$ region. These founded points will be deleted from $A_i$ after adding edges from the new point to them. And that new point will be added to $A_i$ to wait for its closest point. We just need to consider regions $R_1$ through $R_4$ in our algorithm because other regions have been implied by connections in these regions.

After the spanning graph is constructed by adding edges in the above way, we can get octilinear minimal spanning tree easily by using some greedy approaches, such as Prim's algorithm or Kruskal's algorithm.

## III. Octilinear Steiner Tree Construction

We choose Kruskal's algorithm to construct a minimal spanning tree on the spanning graph because it can be used with margining binary tree to find the longest edges in formed cycles [18], which is a crucial part both in OST-E and in OST-T algorithm.

Kruskal's algorithm first sorts all the edges of a spanning graph by non-decreasing length and then considers them in the order. If the ends of the current edge have not been connected, the edge will be included in the minimal spanning tree; otherwise, it will be excluded. We can represent these connection operations by a binary tree, where the leaves represent the points and the internal nodes represent the edges. For each internal node, its two children represent the two components connected by the corresponding edge in the minimal spanning tree. To illustrate this, a spanning tree and its merging binary tree are shown in Fig.2. As we can see, the longest edge between any two points is the lease common ancestor of the two points in the binary tree. For example, the longest edge between *p* and *b* in Fig.2 is (*b, c*).

In our OST-E (octilinear Steiner tree by edge substitution) algorithm, to find the longest edge in the cycle formed by connecting a point to an edge, we should find which end of the edge is in the same component with the point. For example, while connecting *p* with edge (*a*, *b*), because *p* and *b* are in the same component, the longest edge to be deleted in the cycle is (*b*, *c*).

In the OST-T (octilinear Steiner tree by triple contraction) algorithm, while we contract a triple, that is, connect 3 points by a Steiner point, two cycles will be formed in the graph. Thus we should find two longest edges, corresponding to two least common ancestors in the binary tree. Then we need to choose two two-point pair. One way is to find the least common ancestors of the lowest point with the other two. For example, in Fig.2, if the triple of points *p, a, b* is contracted, we can consider pairs *p, a* and *p, b* to get two longest edges (*a*, *b*) and (*b*, *c*). The other way is to find the lease common ancestor of each two-point pair. Among the three edges we got, there are two same edges. For above example, the three edges corresponding to three two-point pair are (*a*, *b*), (*a*, *b*) and (*b*, *c*). We can omit the repeat edge and get two longest edges to delete.

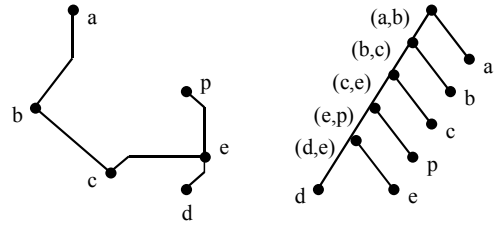


Fig.2. A minimal spanning tree and its merging binary tree.

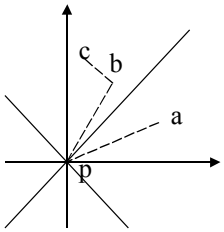


Fig.3. An empty triple usually has at least two connected edges in the spanning graph.

It is crucial to find point-edge pairs in OST-E and to find triples in OST-T algorithm. We can utilize spanning graph to greatly reduce the number of candidates.

Our two approaches are both based on spanning graphs, which can be used not only to construct octilinear minimal spanning trees, but also to get point-edge pairs in OST-E or triples in OST-T. Here, point-edge pairs and triples are units to be optimized, similar as in Borah et al.'s edge substation algorithm [13] and in Zelikovsky's triple contraction algorithm [10], respectively.

For edge substitution, in order to reduce the number of point-edge candidates from $O(n^2)$ to $O(n)$, Borah et al. [13] suggested using the visibility of a point from an edge, that is, only a point visible from an edge can be connected to that edge This requires a sweep line algorithm to find visibility relations between points and edges. A crucial observation is that if a point is visible to an edge then the point is usually connected to at lease one end of that edge in the spanning graph. Thus, for each edge in the minimal spanning tree, we may just consider the points that are neighbors of either end of the edge to form point-edge pairs. Since the cardinality of the spanning graph is $O(n)$, the number of possible point-edge pairs generated by this way is also $O(n)$.

For triple contraction, Kahng et al. [9] proposed a divide-and-conquer algorithm, which can compute in $O(n\log n$ time a set of $O(n\log n)$ triples containing all empty tree triples [15]. As shown in [15], there are at most 36n empty tree triples which do not contain any other terminals in the minimum rectangle bounding each triple. These triples are sufficient to get a nearly optimal solution by triple contraction. We observed that an empty triple usually has at least two connected edges in the spanning graph. As shown in Fig.3, triple *(p, a, b)* and triple *(p, b, c)* are two examples. Therefore, in our algorithm, we use the edges of a spanning graph to get triples. For each edge in the spanning graph, all points that are neighbors of either end point will be considered to form triples with this edge. Because the cardinality of a spanning graph is $O(n)$, we will have a set of $O(n)$ triples, which nearly contains all the empty triples. Furthermore, if point-edge pairs are considered as triples, they are a subset of triples considered in the OST-T. Thus, the OST-T theoretically has a better performance than OST-E, but it will be slower since it considers more triples.

Based on the above discussion, the pseudo-code of these two algorithms is described in Fig.4. At the beginning of each algorithm, the octilinear spanning graph is generated. Then Kruskal's algorithm is used on the graph to generate a minimal spanning tree and the corresponding merging binary tree. In OST-E, during this process, point-edge pairs will also be added to query list by lcd_add_query. And by using Tarjan's off-line least common ancestor algorithm [16], we can get the longest edge for each pair in lcd_answer_queries. Thus the gain of each point-edge pair can be calculated and sorted. If neither the connection edge nor the deletion edge has been

deleted, a Steiner point can be added and a connection will be realized.

The running time of these two algorithms are dominated by spanning graph generation and edge sorting, which takes *O(nlogn)*. And since the number of edges in the spanning graph is *O(n)*, both Kruskal's algorithm and Tarjan's offline least common ancestor algorithm take $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse of Ackermann's function [16], which grows extremely slow.

## IV. Experimental Results

We implement the Octilinear Spanning Graph (OSG) algorithm and the two Octilinear Steiner Tree algorithms (OST-E and OST-T) in the C language, following the pseudo-code in Fig.4, with the exception that the program starting from the first "for" loop will be repeated if there are improvements in the previous iteration.

In TABLE I, we compare our octilinear Steiner tree results with rectilinear minimal spanning tree. The results in TABLE I show that our two octilinear approaches can greatly reduce the wire length for test cases with different size.

In TABLE II, we report the comparisons between our two programs with Kahng's octilinear Steiner tree program—the batched greedy aglorithm, which theoretically has a $O(nlog^2n)$ running time and a good performance. For fair comparison, we compiled and run all the programs on the same machine--a Sun V880 running UNIX operating system. The test bed for the experiments consists of two categories: random test cases ranging from 10 to 100000 terminals; and test cases extracted from recent industrial designs, ranging in size from 330 to 23000. For each input size, we report the average improvement ratio of the Steiner tree over minimal spanning tree (in percentage) and the average running time (in seconds) of 10 repeated experiments.

TABLE II shows that the OST-E is faster than Kahng's batched greedy algorithm and has a similar performance. The main reason why the OST-T is slower than the batched greedy algorithm is that the implementation of the OST-T is highly structural and has much overhead. Furthermore, its running time can be greatly reduced by pruning the generated redundant triples. We plan to make these improvements in future work.

## V. Conclusions

In summary, we developed two efficient octilinear Steiner tree algorithms which are based on Zhou et al.'s spanning graph, and combined with Borah et al.'s edge substitution and Zelikovsky's triple contraction, respectively. The implementations have a runtime of *O(nlogn)* and a storage requirement of *O(n)*, without large hidden constant.

```
Algorithm Octilinear Steiner Tree with Edge Substitution (OST-E)
    T = NULL;
    Generate the spanning graph G by OSG algorithm;
    For (each edge (u,v)of G in non-decreasing length){
        s1 = find_set(u); s2 = find_set(v);
        if (s1 != s2){
            add (u,v) in tree T;
            for (each neighbor w of u,v in G)
                lca_add_query(w, (u,v));
            lca_add_edge((u,v), s1.edge); lca_add_edge((u,v), s2.edge);
            s = union_set(s1, s2); s.edge = (u,v);
        }
    }
    Generate point-edge pairs by lca_answer_queries;
    For (each pair (p, (a,b), (c,d)) in non-increasing positive gains)
        if ((a,b), (c,d) have not been deleted from T){
            connect p to a, b by adding three edges and one Steiner point to T;
            delete (a,b), (c,d) from T;
        }
    Output T;

Algorithm Octilinear Steiner Tree with Triple Contraction (OST-T)
    T = NULL;
    Generate the spanning graph G by OSG algorithm;
    Generate the minimal spanning tree T and merging binary tree by Kruskal's algorithm,
    For (each edge (u,v)of G in non-decreasing length)
        for (each neighbor w of u,v in G)
            lca_add_query(w, u, v);
    Generate triple info by lca_answer_queries;
    For (each triple info (w, u, v, (a,b), (c,d))) in non-increasing positive gains)
        if ((a,b), (c,d) have not been deleted from T){
            connect w, u, v by adding three edges and one Steiner point to T;
            delete (a,b), (c,d) from T;
        }
    Output T;
```

Fig.4. OST-E and OST-T algorithms.

## REFERENCES


[1] M. Sarrafzadeh and C. K. Wong, "Hierarchical Steiner tree construction in uniform orientations", *IEEE Trans on CAD,* 1992, 11(9): pp.1095.

[2] C. K. Koh, "Steiner Problem in octilinear routing model", *Master's thesis,* National University of Singapore, 1995

[3] D. T. Lee, C. F. Shen, and C. L. Ding, "On Steiner tree problem with 45˚ routing", *In: Proc. of IEEE ISCAS,* Seattle, WA, USA, 1995: pp.1680.

[4] G. Robins, "On Optimal Interconnections", *Ph.D. thesis,* University of California, Los Angels, 1992

[5] C. K. Koh and P. H. Madden, "Manhattan or non-Manhattan? a study of alternative VLSI routing architectures", *In: Proc. of GLSVLSI,* San Diego, CA, USA, 2000: pp.47.

[6] S. L. Teig, "The X architecture: not your father's diagonal writing", *In: International Workshop on System Level Interconnect Prediction (SLIP),* San Diego, CA, USA, 2002: pp.33.

[7] C, Chiang and C. S. Chiang, "Octilinear Steiner tree construction", *In: Proc. of IEEE MWSCAS,* Tulsa, USA, 2002: pp.TAM1l-216.

[8] http://www.xinitiative.org/

[9] A. B. Kahng, I. I. Mandoiu, and A. Z. Zelikovsky, "Highly Scalable Algorithms for Rectilinear and Octilinear Steiner Trees", *In: Proc. Of ASP-DAC,* 2003: pp.-.

[10] A. Zelikovsky, "An 11/6-approximation algorithm for the network Steiner problem", *Algorithmica,* 1993, 9: pp.463

[11] A. B. Kahng and G. Robins, "A new class of Steiner tree heuristics with good performance", *IEEE Trans on CAD,* 1992, 11(7): pp.893

[12] Chris S. Coulston, "Constructing Exact Octagonal Steiner Minimal Tree", *In: Proc. of GLSVLSI,* Washington DC, USA. 2003: pp.-.

[13] M. Borah, R. M Owens, and M. J. Irwin, "An edge-based heuristic for Steiner routing", *IEEE Transactions on CAD,* 1994,13: pp.1563.

[14] H. Zhou, N. Shenoy, and W. Nicholls, "Efficient spanning tree construction without Delaney triangulation", *Information Processing Letter,* 2002, 81(5): pp.-.

[15] F. K. Hwang, "An $O(n \log n)$ algorithm for rectilinear minimal spanning trees", *Journal of the ACM,* 1979, 26(2): pp.177.

[16] U. Foßmeier, M. Kaufmann, and A. Zelikovsky, "Faster approximation algorithms for the rectilinear Steiner tree problem", *Discrete & Computational Geometry,* 1997, 18(1): pp.93.

[17] T. H. Cormen, C. E. Leiserson, and R. H. Rivest, "Introduction to Algorithm", *MIT Press,* 1989

[18] H. Zhou, "Efficient Steiner Tree Construction Based on Spanning Graphs", *In: Proc. of ACM ISPD,* Monterey, CA, USA, 2003: pp.-.


TABLE I

PERCENT IMPROVEMENT OVER RECTILINEAR MINIMAL SPANNING TREE (RMST)

| Input Size | RMST | OST-E | OST-T |
|---|---|---|---|
| | Length | Improve (%) | Improve (%) |
| Random Instances (average results over 10 experiments) | | | |
| 10 | 24215 | 20.16932 | 20.16932 |
| 100 | 81679 | 18.10012 | 18.16256 |
| 300 | 143930 | 18.49163 | 18.52081 |
| 1000 | 259277 | 19.42826 | 19.45487 |
| 3000 | 443622 | 19.0739 | 19.10816 |
| 10000 | 81190917 | 19.01136 | 19.03157 |
| 50000 | 181146384 | 19.14284 | * |
| 100000 | 255478245 | 19.13443 | * |
| VLSI Instances (average results over 10 experiments) | | | |
| 337 | 24773 | 14.23727 | 14.23727 |
| 1944 | 45225 | 12.86014 | 12.9508 |
| 2437 | 57879 | 12.68681 | 12.78011 |
| 12052 | 265274 | 13.79404 | 13.84041 |
| 22373 | 1.396E+09 | 16.80561 | 16.82782 |

TABLE II

PERCENT IMPROVEMENT OVER OCTILINEAR MINIMAL SPANNING TREE (OMST) AND CPU TIME OF EACH PROGRAM

| Input | OMST | OST-E | | Batched Greedy | | OST-T | |
|---|---|---|---|---|---|---|---|
| | Length | Imp (%) | CPU (sec.) | Imp (%) | CPU (sec.) | Imp (%) | CPU (sec.) |
| Random Instances (average results over 10 experiments) | | | | | | | |
| 10 | 20040 | 3.537924 | 0 | 3.5379242 | 0 | 3.537924 | 0 |
| 100 | 70167 | 4.663161 | 0.02 | 4.6631607 | 0.03 | 4.735844 | 0.05 |
| 300 | 122122 | 3.936228 | 0.07 | 3.9943663 | 0.18 | 3.97062 | 0.19 |
| 1000 | 218222 | 4.269964 | 0.33 | 4.298375 | 0.75 | 4.301583 | 0.75 |
| 3000 | 375336 | 4.350768 | 1.10 | 4.3680862 | 3.40 | 4.391265 | 4.85 |
| 10000 | 68720982 | 4.315372 | 6.22 | 4.335769 | 14.43 | 4.339244 | 43.89 |
| 50000 | 153045172 | 4.296345 | 47.86 | 4.325813 | 199.39 | * | * |
| 100000 | 215853782 | 4.289864 | 93.12 | * | * | * | * |
| VLSI Instances (average results over 10 experiments) | | | | | | | |
| 337 | 21900 | 2.986301 | 0.06 | 2.9863014 | 0.13 | 2.986301 | 0.11 |
| 1944 | 40723 | 3.226678 | 0.70 | 3.4697837 | 1.58 | 3.327358 | 1.54 |
| 2437 | 52311 | 3.393168 | 0.92 | 3.7774082 | 1.64 | 3.496397 | 3.10 |
| 12052 | 237228 | 3.602442 | 9.03 | 3.7137269 | 13.20 | 3.65429 | 26.06 |
| 22373 | 1.207E+09 | 3.759414 | 13.52 | 3.7414225 | 40.69 | 3.7851 | 270.56 |

(Asterisk * indicates that the results cannot be calculated in the given time, i.e., 1000 seconds)