

# Local Improvement in Steiner Trees

*F. D. Lewis<sup>†</sup>, Wang Chia-Chi Pong<sup>†</sup>, and N. Van Cleave<sup>\*</sup>*

Department of Computer Science

<sup>†</sup>University of Kentucky  
Lexington, Kentucky 40506

<sup>\*</sup>Texas Tech University  
Lubbock, Texas

## 1. Introduction.

Next to the traveling salesman problem, the Steiner problem is possibly the most mentioned NP-complete problem in existence. This is due to the ease with which it can be stated (join a set of points with the smallest collection of connections) and its many, varied applications. For example, with a rectilinear metric, Steiner spanning trees are used in several phases of CAD for VLSI systems.

Since this problem is NP-complete [GJ77], most attempts at solving it have been heuristics based on paradigms such as minimal spanning tree modification, computational geometry, stochastic evolution, and iterated improvement. An extensive recent survey of methods for solving Steiner tree problems is contained in [HRW92].

Of special interest in this paper are local improvement algorithms. This kind of approach involves exploring chains of closely related feasible solutions to a problem in hope of discovering a better answer. Two of recent vintage [CH90, HaVW90] involve redesigning small subtrees and adding Steiner points to small subtrees.

Our approach differs significantly from these. Using a new representation for Steiner trees which allows us to separate them into subtrees and reconnect the subtrees with comparative ease, we are able to easily define traversals between neighboring configurations. This results in an algorithm which is easily describable and easily implementable, as well as providing substantial improvement over existing heuristic algorithms for the rectilinear Steiner problem.

## 2. The Steiner Problem.

A *rectilinear Steiner spanning tree* over a set of points is a connected collection of vertical and horizontal lines which spans the points. We shall refer to these sets of lines as *Steiner trees*. Presenting the very special member of this collection of spanning trees known as the *shortest tree* is the goal of our construction efforts.

We are able to restrict our feasible solution search space to a finite collection of trees by applying Hanan's classic result which states that a shortest Steiner tree exists on the *grid induced by the points* [Han66]. This induced grid is formed by drawing vertical and horizontal lines through the points. An example of a grid induced by four points is found in figure 1a.

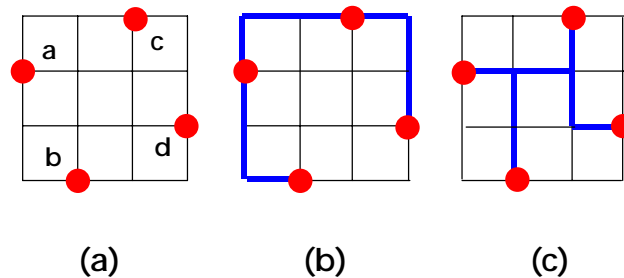


Figure 1 - Points, a Grid, and Spanning Trees

The shortest Steiner tree over these points appears in figure 1c. Another type of spanning tree, the *rectilinear minimum spanning tree* over the same set of points occupies figure 1b. These differ from Steiner trees in that they are sets of edges between points while Steiner trees are collections of grid edges spanning the points. This means that any overlaps between edges in minimum spanning trees are counted twice when computing the size of the tree whereas Steiner trees contain no overlaps. The relationship between the sizes of these two kinds of spanning trees has been formulated by Hwang [Hw76].

**Theorem 1** (Hwang). *The shortest rectilinear Steiner spanning tree over a set of points is no less than two-thirds the size of the rectilinear minimal spanning tree over the points.*

Minimal spanning trees are easily defined as sets of edges. For example, the tree in figure 1b is exactly the three edges:  $\langle a, c \rangle$ ,  $\langle a, b \rangle$ , and  $\langle c, d \rangle$ . The Steiner tree of figure 1c contains these three edges too, but they are drawn differently, and if the overlaps between edges are not included twice when determining the length of the tree, the shortest possible Steiner spanning tree over the points is formed.

In order to be able to define Steiner trees as sets of edges between points (rather than collections of grid edges or lines on the grid), we use a result from [JLV96] which guarantees that there is a shortest Steiner tree with the following two properties.

- a) Every line in the tree passes through a point.
- b) Every intersecting pair of lines contains at least two points.

More ease of definition and construction is provided if we require the points to induce an  $n$  by  $n$  grid and only consider trees with no *dangling lines* (lines which do not end at another line or a point). We claim that this is not overly restrictive since dangling lines are never found in shortest spanning trees and there is little difference in spanning trees for a set of points containing two which share a coordinate and a set where the points nearly share a coordinate.

Thus given a set of  $n$  points which do not share horizontal or vertical coordinates, a Steiner spanning tree of the kind we have defined consists of exactly  $n$  lines, one for each point. We call these *1-1 rectilinear Steiner spanning trees* (or *1-1 Steiner trees*) because the lines and points form a one to one correspondence. As the following theorem states, they may be defined as sets of edges.

**Theorem 2** [Van92]. *Every 1-1 rectilinear Steiner spanning tree over  $n$  points may be represented as a collection of  $n-1$  edges between points.*

**Proof.** Assuming that none of the points share horizontal or vertical coordinates, there are exactly  $n$  lines in the 1-1 tree. Since it is a spanning tree, there are exactly  $n-1$  intersections between the lines. Let each of these intersections represent an edge between the points on the lines which meet there.

Either this collection of edges (when drawn through the intersections which represent them) is the original tree or there is a portion of a line in the tree which does not fall on any of the edges.

If there is such a line segment in the tree, it cannot intersect with a line in the tree because the edges contain the intersections and adjacent parts of the lines which meet there. For the same reason, this segment also cannot lie between a point and an intersection. Thus it must be on a line which goes through a point and ends somewhere out on the grid rather than at an intersection. These dangling line segments are not allowed in the trees we consider.

As we have seen, figure 1c contains such a tree. The result concerning shortest Steiner trees and 1-1 trees mentioned above is stated as follows.

**Theorem 3** [JLV96]. *For any set of points, there is a shortest rectilinear Steiner spanning tree which is a 1-1 rectilinear Steiner spanning tree.*

These results also provide limits which we may use to restrict our feasible solution search space. Thus we may now confine our investigations only to 1-1 Steiner trees represented as sets of edges or point pairs and be confident that a shortest Steiner spanning tree is contained within this abridged solution space. Keeping in mind that an edge is a pair of points, we now redefine Steiner spanning trees.

**Definition.** *A one-one Steiner spanning tree over a set of  $n$  points (which do not have common vertical or horizontal coordinates) is a connected collection of  $n-1$  edges which spans the set of points.*

Since the Steiner trees which interest us are now merely sets of edges, we shall employ the notation  $T = \{ \langle u_1, v_1 \rangle, \dots, \langle u_{n-1}, v_{n-1} \rangle \}$  when referring to the  $n-1$  edges which make up one of these trees.

As we found when examining figure 1, there is no unique way to draw a tree represented by a set of edges. If we are drawing 1-1 Steiner trees though, there are *exactly two ways* since each line in the tree can be vertical or horizontal. To draw a tree from a set of edges we begin with an edge (or pair of points) and connect them with a vertical line through one and a horizontal line through the other. Then we take another edge containing one of these points and connect in the new point (from this new edge) with a line perpendicular to an existing line and possibly an extension of that existing line. We continue on in the same manner until all of the edges have been used and the entire tree is drawn. The process produces a spanning tree over the points since the set of edges spans the points.

For example, the tree in figure 1c can be constructed by drawing the edge  $\langle a, c \rangle$  with a horizontal line through  $a$ , connecting this to the point  $b$  with the edge  $\langle a, b \rangle$ , and then filling in the edge  $\langle c, d \rangle$ . This construction sequence is presented in figure 2.

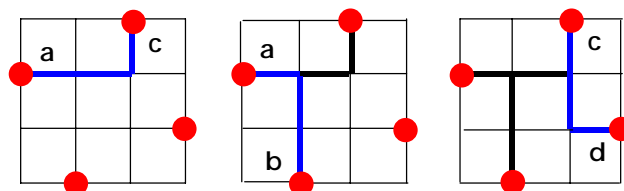


Figure 2 - Constructing a 1-1 Spanning Tree

Note that figure 1b is the other way to draw the 1-1 Steiner tree represented by the same set of edges. It was constructed by connecting edges in the same sequence, but beginning with a vertical line through a and a horizontal one through c.

Since we now have two physically different tree drawings for each set of edges, we often have two different sizes also. Thus we must define what is meant by the size of a tree for a set of edges. The following definition sets this to our best advantage.

**Definition.** *The **size** of a 1-1 tree  $T = \{ \langle u_1, v_1 \rangle, \dots, \langle u_{n-1}, v_{n-1} \rangle \}$  is the sum of the lengths of the grid segments forming the smallest of the two 1-1 trees which can be constructed from the edges.*

### 3. Neighborhoods.

Local improvement algorithms involve sequences of small changes to some initial feasible solution which lead (one hopes) to an optimum solution. Each small change should not produce a new tree which is very different from the previous one. Small changes such as these are said to lead to *neighbors* of the original tree. If we are to use this technique to develop a shortest Steiner tree construction algorithm, we must formulate exactly what the *neighborhood* for a solution consists of, which in this case is the neighborhood of a 1-1 Steiner tree.

Intuitively, this process shall be very simple. We break a tree into two subtrees by removing an edge and then patch it back together with a different edge to produce a neighbor. The collection of trees constructed in this manner forms the neighborhood of our original tree.

For example, removing the edge  $\langle a, c \rangle$  from the tree of figure 1c breaks the tree  $T$  into a subtree containing a (which we call  $T_a$ ) and a subtree containing c (called  $T_c$ ). The two ways of drawing each of these subtrees are shown in figure 3.

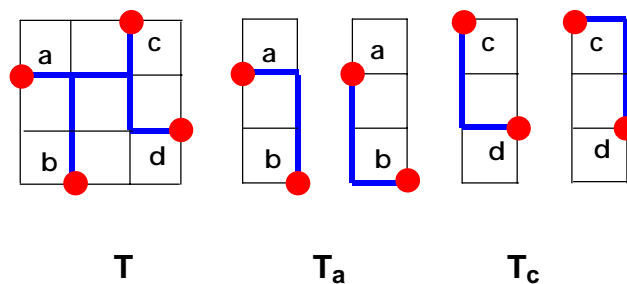


Figure 3 - Breaking Apart a Tree

We define this tree separation formally as follows.

**Definition.** If  $T$  is a 1-1 Steiner tree and the edge  $\langle u, v \rangle \in T$  then:

$T_u =$  subtree of  $T - \{\langle u, v \rangle\}$  rooted at  $u$ , and

$T_v =$  subtree of  $T - \{\langle u, v \rangle\}$  rooted at  $v$ .

Our next step is to patch the tree back together to produce a neighboring spanning tree over the set of points. This means adding an edge which joins the two subtrees. The edges possible to add to the subtrees of our example are  $\langle a, d \rangle$ ,  $\langle b, c \rangle$ , and  $\langle b, d \rangle$ . Adding these edges and drawing each new tree in both possible ways is demonstrated in figure 4.

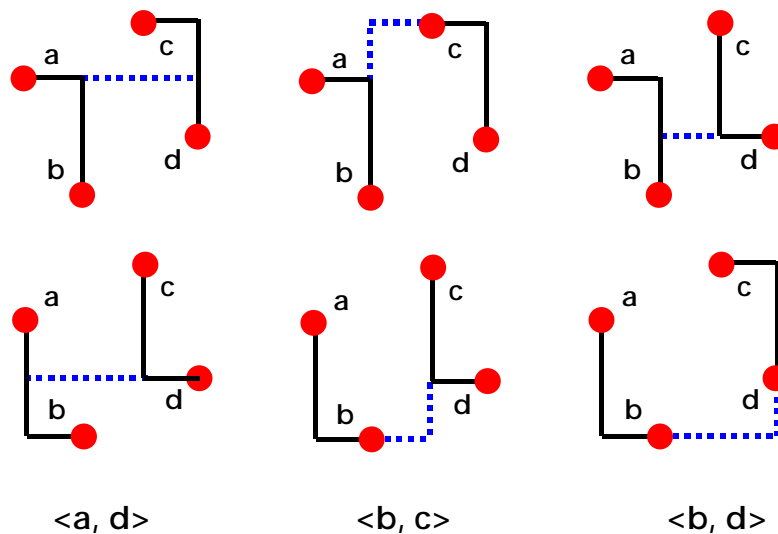


Figure 4 - Reconnecting  $T_a$  and  $T_c$

In reconnecting the subtrees we did not use the edge  $\langle a, c \rangle$  because we had no further interest in that spanning tree. We had already considered it and we wished to construct new and different spanning trees. Note that the definitions provided below do include a tree in its own neighborhood - but we shall ignore this when developing tree construction algorithms.

Definitions of this transformation process and its transitive closure are as follows.

**Definition.** The 1-1 Steiner tree  $T$  can be **transformed** into the 1-1 Steiner tree  $T'$  (written  $T \rightarrow T'$ ) if and only if for an edge  $\langle u, v \rangle \in T$ , there is a point  $p \in T_u$  and a point  $q \in T_v$  such that:

$$T' = T_u \cup T_v \cup \{\langle p, q \rangle\}.$$

**Definition.** *The 1-1 Steiner tree  $T$  can be **eventually transformed** into the 1-1 Steiner tree  $T'$  (written  $T \Rightarrow T'$ ) if and only if there is a sequence of 1-1 Steiner trees  $T_1, T_2, \dots, T_k$  such that:*

$$T = T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_k = T'.$$

Now we can define neighbors and neighborhoods. A neighbor of a 1-1 Steiner tree is just a transformation of the tree via the above definition. And the *neighborhood* of a 1-1 tree  $T$  is  $N(T) = \{ T' / T \rightarrow T' \}$ .

We now have the tools with which to implement a local improvement strategy based upon traversing sequences of best neighbors. But, it is not clear that it is possible to get to the optimum solution, the smallest 1-1 Steiner tree from just any starting place. We would like to be sure that the restrictions so carefully formulated above place no additional barriers in our way when attempting to find the optimum spanning tree. The following lemma provides a way to select specific neighbors which might be found on the path to an optimum tree.

**Lemma (Splitting).** *For any two points  $p$  and  $q$  in a 1-1 Steiner tree  $T$ , there is an edge  $\langle u, v \rangle \in T$  such that  $p \in T_u$  and  $q \in T_v$ .*

**Proof.** Since  $p$  and  $q$  are points in a 1-1 Steiner tree, there is a path between them which is made up of portions of edges. If the path contains just one edge, then this edge is  $\langle p, q \rangle$  and may also function as  $\langle u, v \rangle$ . If the path contains portions of more than one edge (such as in figure 5 below),

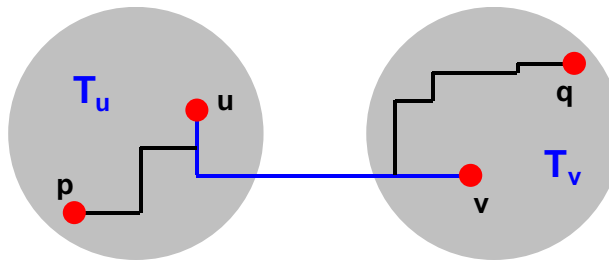


Figure 5 - Path from  $p$  to  $q$

With this lemma we now have the mechanism necessary to transform one tree into any other by selecting a sequence of transformations leading from that 1-1 tree to the one we wish to construct.

**Theorem 4.** *Any two 1-1 Steiner trees over the same set of points, can be eventually transformed into each other.*

**Proof.** Suppose that we wish to transform an arbitrary 1-1 Steiner tree  $T_0$  into  $T = \{ \langle u_1, v_1 \rangle, \dots, \langle u_{n-1}, v_{n-1} \rangle \}$ . We begin by splitting  $T_0$  into two subtrees, one containing  $u_1$  and one containing  $v_1$ . Then we patch the tree back together with  $\langle u_1, v_1 \rangle$  and call it  $T_1$ . We continue this until all the edges of  $T$  have been placed in the tree we are constructing. The following algorithm outlines the transformation of  $T_0$  into  $T$ . Breaking trees into subtrees is accomplished with the aid of the Splitting Lemma.

for  $i = 1$  to  $n-1$   
     Split  $T_{i-1}$  into  $T_u$  and  $T_v$  containing  $u_i$  and  $v_i$   
      $T_i = T_u \cup T_v \cup \{ \langle u_i, v_i \rangle \}$

If we make sure that the pairs  $\langle u_1, v_1 \rangle, \dots, \langle u_{n-1}, v_{n-1} \rangle$  are never deleted from the tree when we are applying the Splitting Lemma, we end up with  $T_{n-1} = T$  as we wish. To see that this is possible, consider the following argument.

Suppose that at some stage of the algorithm we are trying to break  $T_i$  into two parts and the entire path from  $u_i$  to  $v_i$  is made up of edges in  $\{ \langle u_1, v_1 \rangle, \dots, \langle u_{i-1}, v_{i-1} \rangle \}$ . Adding  $\langle u_i, v_i \rangle$  to this set will create a cycle, so either  $T$  was not a tree or the path must have an edge not in  $T$  upon it. Thus the algorithm can transform any 1-1 Steiner tree to  $T$  with proper care in selecting edges to delete from intermediate trees when applying the Splitting Lemma.

This means that by restricting our field of inquiry to 1-1 Steiner trees it is possible to construct an optimum solution to the rectilinear Steiner spanning problem by traversing a sequence of transformations between neighboring 1-1 trees, starting from any tree. Thus theorem 4 assures us that a local search algorithm based upon the neighborhoods defined above always has a chance of finding the best solution.

#### 4. Local Improvement Strategy.

A classic local search solution to an NP-complete problem is the graph partitioning heuristic of Kernighan and Lin [KL70]. It, like most local improvement strategies involves traversing sequences of neighboring solutions on the way to a locally optimal solution.

This method involves starting with some feasible solution and examining all of its neighbors. The best of these is selected and then its neighbors are examined. This process usually continues for  $O(n)$  steps (with problems of size  $n$ ) producing a sequence feasible solutions found in a chain of neighborhoods. The best solution in the sequence is selected and the entire algorithm is repeated until no improvement is achieved. A template for this local search procedure follows.

```
S = initial feasible solution
repeat
  S0 = S
  for i = 1 to n do
    Si = best neighbor of Si-1
  S = best of S0, ... , Sn
until S is no better than S0
```

One reason that these algorithms work so well is that a local optimum within a certain radius of neighborhoods is found in each of the repeat loops. If the sequence of neighbors traversed is large enough then there is a chance of encountering a global optimum.

We begin by taking as our initial 1-1 Steiner spanning tree the set of edges in the minimal rectilinear spanning tree. Then we construct the smallest neighboring 1-1 tree by applying a transformation of the kind discussed in the last section. (Breaking the tree apart by removing an edge and then adding another edge to reconnect it.) Then we examine this tree in turn and find its smallest neighbor. And so on.

To insure that this process halts, we do not allow returning to previous configurations. This is accomplished by *locking in place* the replacement edges used to form neighboring trees. Thus we do not undo what we have accomplished and we prevent cycling.

When we have replaced  $n-1$  edges (for an  $n$  point problem) we examine the sequence of 1-1 trees we have built and select the smallest. Then we begin again (with this as our starting point) and keep iterating until no improvement is found. This process is described in the algorithm of figure 6.

**LocalSearch**( $P, T$ )  
*PRE:  $P = \text{set of } n \text{ points}$*   
 *$T = \text{rectilinear minimum spanning tree over } P$*   
*POST:  $T = \text{rectilinear Steiner spanning tree over } P$*

```

repeat
   $T_0 = T$ 
  Unlock all pairs of points  $\langle u, v \rangle$  in  $T_0$ 
  for  $i := 1$  to  $n-1$  do
     $T_i = \text{smallest neighbor of } T_{i-1}$ 
     $\langle p, q \rangle = \text{replacement edge used in building } T_i \text{ from } T_{i-1}$ 
    Lock  $\langle p, q \rangle$ 
   $T = \text{smallest of } T_0 \dots T_{n-1}$ 
until  $T$  is the same size as  $T_0$ 

```

Figure 6 - Local Improvement Algorithm

At the core of the local improvement algorithm is the transformation of a tree into its most attractive neighbor. We accomplish this by examining all of the possible ways to break a tree into subtrees by removing one edge as well as every manner in which we can reconnect it. To prevent cycling in the main algorithm though, we do not consider neighbors formed by removing edges which were added earlier. (These are referred to as *locked edges*.) Figure 7 contains this neighbor selection procedure.

**BestNeighbor**( $P, T, T', p, q$ )  
*PRE:  $P = \text{set of points}$*   
 *$T = \text{Steiner tree spanning } P$*   
*POST:  $T' = \text{smallest unforbidden neighbor of } T \text{ spanning } P$*   
 *$\langle p, q \rangle = \text{new pair connected to build } T' \text{ from } T$*

```

for every unlocked pair  $\langle u, v \rangle$  in  $T$  do
   $T_u = \text{subtree of } T - \{\langle u, v \rangle\} \text{ rooted at } u$ 
   $T_v = \text{subtree of } T - \{\langle u, v \rangle\} \text{ rooted at } v$ 
  for every point  $p$  in  $T_u$ 
    for every point  $q$  in  $T_v$ 
      if  $\langle p, q \rangle \neq \langle u, v \rangle$  then  $T_{p,q} = T_u \cup T_v \cup \{\langle p, q \rangle\}$ 
   $T_{u,v} = \text{smallest } T_{p,q}$ 
 $T' = \text{smallest } T_{u,v}$ 

```

Figure 7 - Selecting the Smallest Neighbor

Two properties of the algorithm need to be demonstrated, *termination* and *correctness*. Termination is assured since the main algorithm does halt when it encounters no better 1-1 Steiner tree at the end of the *repeat* loop. Since we began with a particular tree (the minimal spanning tree) and required a smaller tree to be found before iterating once more, this is a finite process.

Correctness (producing a tree spanning the points) is certain primarily due to our definitions. We begin with a spanning tree and always select neighbors of trees in the sequence  $T_0, \dots, T_{n-1}$  to examine. Since neighbors span the same points, every tree selected is a correct one.

An *efficiency limit* for our local search algorithm comes directly from Hwang's theorem [Hw76] and the observation that a tree larger than the minimal spanning tree cannot be selected.

**Theorem 5.** *The 1-1 rectilinear Steiner spanning tree built by the local improvement algorithm can be no larger than one and a half the size of the optimum solution.*

*Complexity* depends upon the time spent in examining neighbors. Examination of the neighbor selection algorithm of figure 7 reveals that inside the loops we need to find the size of a tree formed by removing an edge and adding one to the original tree. The proper data structure allows us to keep this time to a minimum. We use a list of the intersections on each line. Consider table 1 below and the spanning trees of figure 1.

Point	Direction	Line	Direction	Line
a	vertical	c-a-b	horizontal	a-b-c
b	horizontal	a-b	vertical	a-b
c	horizontal	a-c-d	vertical	c-a-d
d	vertical	c-d	horizontal	c-d

Table 1 - Lines in Spanning Trees

The column of lines on the left represents the spanning tree in figure 1b. Through the point a we have a vertical line which goes from c's row to b's row. In the column of lines on the right (depicting the spanning tree in figure 1c) we note that the line through point a is horizontal, begins at point a, ends at point c, and contains an intersection at b's column.

Deleting an edge from a tree involves changing the lines through the two points defining the edge so that the edge's intersection no longer appears. If the intersection is at the end of the line then the new line is shorter than before. For example, deleting  $\langle a, c \rangle$  from the tree of figure 1b changes the line through a to a-b and the line through c to c-d. Adding the edge  $\langle b, d \rangle$  to this changes

the line through  $b$  to  $a$ - $b$ - $d$  and that through  $d$  to  $c$ - $d$ - $b$ . Thus to add or delete an edge we need to insert or delete an intersection on the list of rows or columns which make up the lines through the points which define the edge. If the line lists are stored as search trees then  $O(\log n)$  steps are required

Computing the length of a new tree is even less complex if pointers to the ends of each line (the leftmost and rightmost leaves in the search tree) are maintained. Since trees change size when intersections are added to or removed from the ends of lines, calculating changes in size requires only a constant amount of time.

The two inner *for* loops of the BestNeighbor procedure contribute  $O(n^2)$  steps for each unlocked pair processed since only  $O(1)$  steps are executed inside the loops. So, the first time the neighbor selection routine is called,  $O(n^3)$  steps are executed, and the last time,  $O(n^2)$  steps are completed since there is only one unlocked pair. This adds up to  $O(n^4)$  steps each time the outer loop is executed (which in practice is usually about twice).

## 5. Empirical Results.

The algorithm was implemented in C and executed on a Sequent Symmetry S81 Dynix 3.0 machine with randomly generated input data sets. There were ten data sets for each number of points (which ranged from 5 to 100). All of the data sets were generated randomly from a uniform distribution in a grid, which has been found to be very similar to pin locations of actual VLSI layouts [KR90] and thus appears to be an appropriate testbed for these problems [Ric89].

Since  $O(n^4)$  steps, while polynomial, is a rather large complexity for an algorithm, the actual execution times for the algorithm are of interest. In table 2, the average time (in seconds) is provided for each group of data sets. As expected, very large data sets do require long times, but for problems of the size expected in most applications, the times are not excessive. And, since the neighborhood search lends itself nicely to parallel computation methods, even large problems would take only minor amounts of time if implemented in this manner.

Points	5	10	15	20	25	30	50	100
Time	0.07	0.61	2.93	8.40	13.8	202	1,078	72,000

Table 2 - Local Algorithm Average Times

Table 3 provides a comparison of the local improvement algorithm to other algorithms for the groups of various sized point sets. Included are a Kruskal style MST-based algorithm from [LV91], two classic Prim style MST-based algorithms from [LBH76, Hw76], and two linear algorithms [HoVW90, LPV91].

Points	LV	LBH	Hw	HoVW	LPV	Loc
5	9.52	8.37	9.52	9.50	8.44	10.10
10	9.42	8.82	9.45	8.95	8.66	10.46
15	9.90	8.99	9.80	8.20	9.51	11.26
20	11.31	9.49	10.18	8.67	8.61	11.70
25	10.09	8.50	9.57	7.85	8.06	10.86
30	11.26	8.93	10.63	8.71	9.09	12.68
35	10.95	8.95	10.29	8.18	9.31	11.91
50	11.88	9.49	10.97	9.56	9.64	12.96
100	11.01	9.17	10.38	8.48	9.02	11.91
Ave	10.59	8.97	10.09	8.68	8.93	11.54

**Table 3 - Percent Improvement over MST**

For the data sets containing five points, the local improvement solutions were the same size as those found by an optimal branch and bound method [LPV92]. It was the only heuristic to accomplish this. And, on ten point data, the local improvement algorithm found an optimum solution in five of the ten cases. In addition, it came close to the optimum branch and bound algorithm's score of 10.84% average improvement over the minimal spanning trees for this group.

## 6. Conclusion.

We have presented an easily described, new approach to solving the rectilinear Steiner problem. The algorithm is very practical for problem sizes corresponding to actual VLSI layouts and provides significant improvement over algorithms employing other popular methods.

## References

- CH90** Chao, T.-H., and Hsu, Y.-C. "Rectilinear Steiner Tree Construction by Local and Global Refinement." *Proc. of IEEE Int. Conf. on CAD*, 1990, 432 - 435.
- GJ77** Garey, M. R. and Johnson, D. S. "The Rectilinear Steiner Problem is NP-Complete." *SIAM J. Appl. Math.*, vol. 32, 1977, 826-834.
- Han66** Hanan, M. "On Steiner's Problem with Rectilinear Distance." *SIAM J. Appl. Math.*, vol. 14, no. 2, 1966, 255-265.
- HaVW90** Hasan, N., Vijayan, G., and Wong, C. K. "A Neighborhood Improvement Algorithm for Rectilinear Steiner Trees," *Proc. ISCAS*, 1990, 2869 - 2872.
- HoVW90** Ho, J.-M., Vijayan, G., and Wong, C. K. "New Algorithms for the Rectilinear Steiner Tree Problem." *IEEE Trans. on CAD*, vol.9, 1990, 185-193.
- Hw76** Hwang, F. K. "On Steiner Minimal Trees with Rectilinear Distance." *SIAM J. Appl. Math.*, vol. 30, 1976, 104-114.
- HRW92** Hwang, F. K., Richards, D. S., and Winter, P. *The Steiner Tree Problem*, Annals of Discrete Mathematics, no. 53, North Holland, 1992.
- JLV96** Joyce, P. M., Lewis, F. D., and VanCleave, N. "The Feasible Solution Space for Steiner Trees." *Proceedings of the Eighth Society for Industrial and Applied Mathematics Conference on Discrete Mathematics*, 1996.
- KR90** Kahng, A. and Robins, G. "On a New Class of Iterative Steiner Tree Heuristics with Good Performance." *Proc. of IEEE Int. Conf. on CAD*, 1990, 428 - 431.
- KL70** Kernighan, B. W. and Lin, S. "An Efficient Heuristic Procedure for Partitioning Graphs." *Bell System Technical J.*, vol. 49, 1970, 291-307.
- LBH76** Lee, J. H., Bose, N. K., and Hwang, F. K. "Use of Steiner's Problem in Suboptimal Routing in Rectilinear Metric." *IEEE Trans. on Circuits Syst.*, vol. CAS-23, 1976, 470-476.
- LPV91** Lewis, F. D., Pong, W. C., and Van Cleave, N. "A Linear-time Heuristic for Rectilinear Steiner Trees." *Proc. of First Great Lakes Sym. on VLSI*, 1991, 152-156.
- LPV92** Lewis, F. D., Pong, W. C., and Van Cleave, N. "Optimum Steiner Tree Generation." *Proc. of Second Great Lakes Sym. on VLSI*, 1992, 207 - 212.
- LV91** Lewis, F. D. and Van Cleave, N. "Correct and Provably Efficient Methods for Rectilinear Steiner Spanning Tree Generation," *Proceedings of the First Great Lakes Computer Science Conference*, and *Springer-Verlag Lecture Notes*, vol. 507, 1991.
- Ric89** Richards, D. "Fast Heuristic Algorithms for Rectilinear Steiner Trees." *Algorithmica*, vol. 4, 1989, 191-207.
- Van92** Van Cleave, N. "The Rectilinear Steiner Problem." *PhD Dissertation*, CS Department, University of Kentucky, Lexington, KY, 1992.