Construction of All Rectilinear Steiner Minimum Trees on the Hanan Grid*

Sheng-En David Lin
Washington State University
Pullman, Washington
slin3@eecs.wsu.edu

Dae Hyun Kim
Washington State University
Pullman, Washington
daehyun@eecs.wsu.edu

ABSTRACT

Given a set of pins, a Rectilinear Steiner Minimum Tree (RSMT) connects the pins using only rectilinear edges with the minimum wirelength. RSMT construction is heavily used at various design steps such as floorplanning, placement, routing, and interconnect estimation and optimization, so fast algorithms to construct RSMTs have been developed for many years. However, RSMT construction is an NP-hard problem, so even a fast RSMT construction algorithm such as GeoSteiner [7] is too slow to use in electronic design automation (EDA) tools. FLUTE, a lookup-table-based RSMT construction algorithm, builds and uses a routing topology database to quickly construct RSMTs [5]. However, FLUTE outputs only one RSMT for a given set of pin locations. In this paper, we develop an algorithm to build a database of all RSMTs on the Hanan grid for up to nine pins. The database will be able to help minimize routing congestion and maximize the routability in the design of modern very-large-scale integration layouts.

CCS CONCEPTS

• **Hardware → Wire routing**;

KEYWORDS

Rectilinear Steiner Minimum Tree, RSMT, Routing, Wirelength, Congestion

ACM Reference Format:
Sheng-En David Lin and Dae Hyun Kim. 2018. Construction of All Rectilinear Steiner Minimum Trees on the Hanan Grid. In *ISPD '18: 2018 International Symposium on Physical Design, March 25–28, 2018, Monterey, CA, USA.*. ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3177540.3178240

1 INTRODUCTION

The Rectilinear Steiner Minimum Tree (RSMT) construction problem is finding an Rectilinear Steiner Tree (RST) having the minimum length. Since there could be infinitely many RSMTs for a given set of pin locations, the RSMT construction problem is generally limited to finding an RSMT on the Hanan grid [9]. RSMT construction is heavily used in many very-large-scale integration (VLSI) computer-aided design (CAD) tools at various steps such as floorplanning, placement, routing, and interconnect estimation and optimization. Thus, several fast algorithms have been proposed in the literature to construct an RSMT for a given set of pin locations [5, 7]. However, the RSMT construction problem is NP-hard [6], so several papers proposed Rectilinear Minimum Spanning Tree (RMST) or RST construction algorithms for practical use [1, 7, 8, 10–12, 17].

FLUTE builds a database of potentially optimal wirelength vectors (POWVs) and potentially optimal Steiner trees (POSTs) and constructs an RSMT in no time for a given set of pin locations using the database for up to nine pins. FLUTE achieves the shortest wirelength on average for all the 18 IBM benchmarks among five RSMT and one RMST construction algorithms in [5]. In addition, its runtime is 5.56× to 64.92× shorter than the runtimes of all the other RSMT algorithms compared in [5].

One of the applications heavily using RSMT construction in VLSI CAD tools is global routing, in which RSMTs are used for routing topologies. For example, BoxRouter [4], DpRouter [2], Archer [14], MaizeRouter [13], FastRoute [16], GRIP [15], and NTHU-Route [3] use FLUTE for routing topology generation. If there are multiple POWVs having the same minimum wirelength for given pin locations, FLUTE can also construct multiple RSMTs. However, FLUTE constructs only one POST for each POWV, so there is no guarantee that the multiple RSMTs constructed by FLUTE look quite different.

In this paper, we propose an efficient algorithm constructing all RSMTs on the Hanan grid for given pin locations. The algorithm builds a database of all POSTs on the Hanan grid for each POWV for up to nine pins so that applications can use the database to quickly obtain all RSMTs. For more than nine pins, we use the proposed algorithm with a wirelength vector found by FLUTE to construct all RSTs in a reasonable amount of time. Various applications such as global routing and congestion estimation can use the proposed algorithm to quickly generate meaningfully different routing topologies.

2 THE ALGORITHM OF FLUTE

Our all RSMT construction algorithm is based on FLUTE, so we briefly review the idea of FLUTE in this section.

2.1 Position Sequence (Pin Group)

Let $P = \{p_1, p_2, ..., p_n\}$ be a set of n pins and assume that all the pins have distinct x- and y- coordinates. In other words, if the location of p_i is (x_{p_i}, y_{p_i}), $x_{p_i} \neq x_{p_j}$ and $y_{p_i} \neq y_{p_j}$ for any i and j ($i \neq j$). Then, the Hanan grid constructed for the n pins has n horizontal lines and n vertical lines. Let x_i be the x-coordinate of the i-th vertical line from the left and y_i be the y-coordinate of the

*Produces the permission block, and copyright information

ISPD'18, March 25–28, 2018, Monterey, CA, USA.

ACM ISBN 978-1-4503-5626-8/18/03…$15.00
https://doi.org/10.1145/3177540.3178240

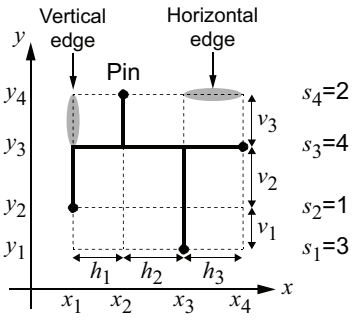

Figure 1: Four pins on the Hanan grid, their position sequence (3142)**, and an RSMT constructed on the Hanan grid.**

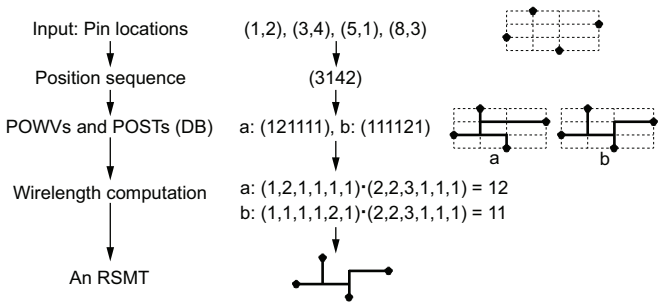

Figure 2: Four pins located on the Hanan grid and its position sequence.

i-th horizontal line from the bottom on the Hanan grid as shown in Figure 1. If the n pins are placed on the Hanan grid, we can characterize the distribution of the pins on the Hanan grid using a *position sequence (pin group)* as follows. Suppose the x-coordinate of the pin whose y-coordinate is y_i is x_{s_i}. Then, the distribution of the pins on the Hanan grid has a position sequence $(s_1 s_2 ... s_n)$. Figure 1 shows four pins, the Hanan grid constructed for them, and its position sequence (3142). Notice that the position sequence is based on not the actual x- and y-coordinates, but the relative locations of the pins. Thus, any set of pin locations can be mapped into one of the $n!$ position sequences for n pins.

2.2 Potentially Optimal Wirelength Vector

The Hanan grid constructed for a position sequence can be decomposed into horizontal and vertical edges as shown in Figure 1. A horizontal edge is a horizontal segment $([x_i, x_{i+1}], y_j)$ and a vertical edge is a vertical segment $(x_k, [y_m, y_{m+1}])$. The length of the horizontal edge whose end points are (x_i, y_j) and (x_{i+1}, y_j) is $h_i = x_{i+1} - x_i$. Similarly, the length of the vertical edge whose end points are (x_k, y_m) and (x_k, y_{m+1}) is $v_m = y_{m+1} - y_m$. Then, any RST constructed on the Hanan grid can be decomposed into horizontal and vertical edges. For example, the RSMT shown in Figure 1 uses one h_1, one h_2, one h_3, one v_1, two v_2, and one v_3 edges on the Hanan grid. The wirelength of the RSMT is

$$L = 1 \cdot h_1 + 1 \cdot h_2 + 1 \cdot h_3 + 1 \cdot v_1 + 2 \cdot v_2 + 1 \cdot v_3, \qquad (1)$$

which can also be expressed as a dot product between $(1, 1, 1, 1, 2, 1)$ and $(h_1, h_2, h_3, v_1, v_2, v_3)$. We call $(h_1, h_2, h_3, v_1, v_2, v_3)$ the *edge length vector* of the given set of pin locations. The edge length vector is dependent on the actual pin locations, but the coefficient vector $(1, 1, 1, 1, 2, 1)$ is dependent only on the RSMT topology. When two coefficient vectors $A = (a_1, ..., a_n)$ and $B = (b_1, ..., b_n)$ are given, if $a_i < b_i$ holds for at least one $i = 1, ..., n$ and $a_j \leq b_j$ holds for all the other $j = 1, ..., n$, the dot product $A \bullet H$ between A and an edge length vector H is always less than $B \bullet H$. We denote this relation by $A < B$. However, if $a_i < b_i$ holds for some i and $a_j > b_j$ holds for some j $(j \neq i)$, $A \bullet H$ is greater or less than $B \bullet H$ depending on H. We denote this relation by $A \leftrightarrow B$. FLUTE finds the set of all coefficient vectors C for each position sequence such that any two coefficient vectors c_i and c_j in C are in the $c_i \leftrightarrow c_j$ relation and there is no c_k in C such that $c_k < c_i$ or $c_k < c_j$. Each element in C is called a ***potentially optimal wirelength vector (POWV)*** because it can be a candidate for an RSMT.

FLUTE builds a database of all POWVs for each position sequence. Then, when the locations of n pins are given, FLUTE finds the position sequence of the pins and obtains all the POWVs from the database. Since the actual wirelength is the dot product between a POWV and the edge length vector for the given pins, FLUTE computes the wirelength for each POWV by computing the dot product between the POWV and the edge length vector and finds a POWV having the shortest wirelength.

2.3 Potentially Optimal Steiner Tree

Since FLUTE returns an RSMT for a given set of pin locations, FLUTE has to construct an actual RSMT. Thus, FLUTE also stores a topology corresponding to each POWV in the FLUTE database. A topology stored for each POWV is called a ***potentially optimal Steiner tree (POST)***. Figure 2 shows an example. For given four pins located at $(1, 2)$, $(3, 4)$, $(5, 1)$, and $(8, 3)$, FLUTE extracts the position sequence (3142) and obtains two POWVs $(1, 2, 1, 1, 1, 1)$ and $(1, 1, 1, 1, 2, 1)$ from the database. The POSTs corresponding to the POWVs are also shown in the figure. Then, FLUTE computes the wirelength of each POWV and returns the POST corresponding to a POWV having the minimum wirelength. We refer readers to [5] for the details of the FLUTE database construction.

3 CONSTRUCTION OF ALL RSMTS

A POST becomes an RSMT if the POWV of the POST has the minimum wirelength for given pin locations. Thus, constructing all RSMTs on the Hanan grid means constructing all POSTs for all POWVs so that we can return all POSTs of all POWVs having the minimum wirelength for the given pin locations. In this section, we explain our algorithm to construct all POSTs on the Hanan grid for a given set of pin locations.

3.1 Terminologies and Notations

Figure 3 shows the Hanan grid constructed for n pins. There exist $n(n-1)$ horizontal edges, $n(n-1)$ vertical edges, and n^2 vertices. If a vertex is a pin, we call the vertex a *pin vertex*. A vertex is connected to two, three, or four edges. We call these edges the *neighboring edges* of the vertex and denote the set of all the neighboring edges of vertex d by $NE(d)$. An edge connects two vertices. If edge e_i

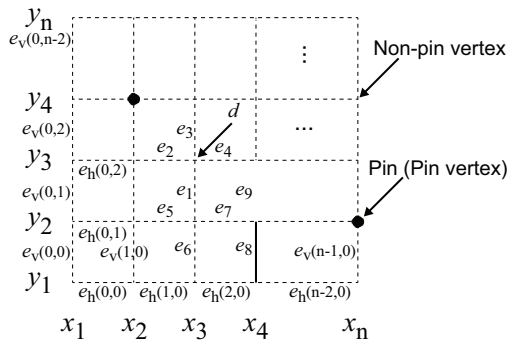

Figure 3: The Hanan Grid for n pins.

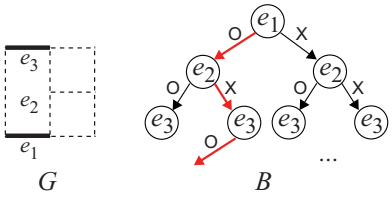

Figure 4: A rectilinear graph G constructed on the Hanan grid and a binary tree B corresponding to G. The red path shows a decision sequence. e_2 is removed in G because the red path traverses through the right arrow of e_2. O and X mean the edge is used or removed in G, respectively.

connects vertices d_j and d_k, we call $NE(d_j) \cup NE(d_k) - \{e_i\}$ the set of the neighboring edges of edge e_i and denote it by $NE(e_i)$. In Figure 3, $NE(d)$ is $\{e_1, e_2, e_3, e_4\}$ and $NE(e_1)$ is $\{e_2, e_3, e_4, e_5, e_6, e_7\}$.

A horizontal edge connects two vertices, one on the left and the other on the right. We denote the left and right vertices of horizontal edge e_i by $V_L(e_i)$ and $V_R(e_i)$, respectively. Similarly, we denote the top and the bottom vertices of vertical edge e_i by $V_T(e_i)$ and $V_B(e_i)$, respectively. Thus, for example, $NE(V_L(e_i))$ is the set of all the neighboring edges connected to the left vertex of horizontal edge e_i. We denote each horizontal edge by $e_h(i, j)$ where i and j are the indices to locate the edge and each vertical edge by $e_v(i, j)$. The indices are shown in Figure 3.

If a vertex of an edge is not a pin vertex and is not connected to any other edges, the edge is *dangling*. For example, if $NE(V_L(e)) - \{e\}$ is the empty set and $V_L(e)$ is not a pin vertex for edge e, e is dangling. If an edge is dangling, it cannot be a part of a POST.

An edge on the Hanan grid can be *available*, *used*, or *removed*. An available edge is an edge that is not used nor removed, but we will decide to use or remove it to construct a POST. e_1 and e_2 in Figure 3 are available edges. A used (or removed) edge is an edge that we have decided to use (or remove) to construct a POST. e_8 is a used edge and e_9 is a removed edge in Figure 3. $powv(e)$ for given edge e is the POWV element corresponding to e. If a POWV is $(q_1, q_2, ..., r_1, r_2, ...)$ where q_k is for the horizontal edges and r_k is for the vertical edges, $powv(e_h(i, j))$ is q_{i+1} and $powv(e_v(i, j))$ is r_{j+1}. We also denote the set of all edges whose POWV element is k by $PE(k)$. For example, $PE(powv(e_h(0, 0)))$ is $\{e_h(0, 0), ..., e_h(0, n-1)\}$.

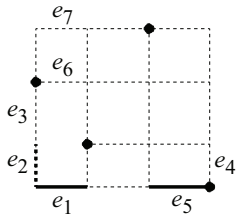

Figure 5: Must-use and must-remove edges.

3.2 Binary Tree-Based POST Construction

We construct a rectilinear graph G on the Hanan grid using a binary tree B to find all POSTs for a given position sequence and a POWV as follows. An internal node in B corresponds to an edge in the Hanan grid. The left and right arrows of an internal node means that we decide to use or remove the edge in G, respectively. Figure 4 shows an example. When we traverse B starting from the root node e_1, we decide to use or remove e_1 in G. When we reach a leaf node, we evaluate the graph G, i.e., we check whether all the pins in G are connected through the used edges. We use the breadth-first search (BFS) algorithm to evaluate a graph.

An exhaustive POST construction algorithm using B uses the in-order traversal to traverse B and evaluates each graph G constructed by B whenever it reaches a leaf node because each leaf node corresponds to a decision sequence $(e_h(0, 0) = O, e_h(0, 1) = X, ...)$ where O and X denote that the edge is used and removed, respectively. However, the exhaustive POST construction algorithm is too slow. The Hanan grid constructed for n pins has $2n(n-1)$ edges, so the total number of leaf nodes in the complete binary tree constructed for n pins has $2^{2n(n-1)}$ leaf nodes. Since we use the BFS algorithm for evaluation of G and there are $2n(n-1)$ edges, the complexity to check whether all pins are connected is $O(n^2)$. Thus, the complexity of the exhaustive POST construction algorithm is $O(n^2 2^{2n(n-1)})$.

When we construct all POSTs, however, we use the binary tree with various pruning criteria to reduce the search space. Although the runtime of the algorithm still seems to increase exponentially, it can find all POSTs for up to nine pins in a reasonable amount of time.

3.2.1 Pruning by Zero POWV Elements. When element q in a POWV becomes zero, we can remove all the available edges in $PE(q)$ from graph G. For example, if the position sequence for four pins is (3142) as shown in Figure 1 and a given POWV is $(1, 2, 1, 1, 1, 1)$, taking the left arrow of node $e_h(0, 0)$ in B uses the edge in G and decreases the first element of the POWV by 1, so the POWV becomes $(0, 2, 1, 1, 1, 1)$. Since the first element of the POWV is zero, $e_h(0, 1)$, $e_h(0, 2)$, and $e_h(0, 3)$ in Figure 1 should be removed from G, which also means we remove the left arrows of all the nodes corresponding to these three edges in B.

3.2.2 Pruning by Must-Use and Must-Remove Edges. When an edge on the Hanan grid is used or removed, there might be edges that should also be used or removed. We call the edges that should be used *must-use* edges and the edges that should be removed *must-remove* edges. The reason that there exist must-use and must-remove edges are as follows.

First, suppose we decide to use edge e_1 in Figure 5. If $N_L(e_1)$ is not a pin vertex, we should use e_2 too, otherwise e_1 becomes a dangling edge. Thus, e_2 becomes a must-use edge. Notice that this does not guarantee that e_1 and e_2 will be included in a POST. Rather, we reduce the search space in the binary tree by removing the right arrows of all the nodes corresponding to e_2 in B.

Second, suppose we remove an edge from G. In this case, some of the neighboring edges of the removed edge might become dangling, so we also have to remove them. For example, suppose we remove e_1 in Figure 5, which causes e_2 to be dangling, so e_2 becomes a must-remove edge and should be removed. If we remove e_2, e_3 also becomes a must-remove edge, so we remove e_3 too. We can remove multiple edges consecutively in this way.

Using or removing an edge can cause some of its neighboring edges to be must-use or must-remove edges. For example, if $powv(e_1)$ is 1 and we use e_1 in Figure 5, e_2 becomes a must-use edge and e_6 and e_7 become must-remove edges. If we remove e_1, e_2 becomes a must-remove edge. If we remove e_4, however, e_5 becomes a must-use edge because e_5 is the only edge connecting pin p_1.

3.2.3 Conditions for POST Evaluation. Evaluation of G checks whether G on the Hanan grid connects all the pins. However, evaluating graphs too often increases the runtime meaninglessly. Thus, we evaluate G only when 1) the current POWV becomes a zero vector or 2) we reach a leaf node in B.

3.2.4 Intermediate Connectivity Check. In many cases, using or removing edges occurs consecutively as explained above. Using edges decreases the POWV elements corresponding to them, so some of the POWV elements might become zero if many edges become must-use edges during pruning. If some POWV elements become zero, all the available edges corresponding to the POWV elements become must-remove edges, so we remove them. If many edges are removed, G is highly likely to be disconnected. Thus, we also check whether all the pins are still connected through the used and available edges in G during the pruning if the number of used and removed edges at a pruning step is greater than a pre-determined threshold value[1].

3.2.5 Binary Tree Construction and Traversal. We construct a binary tree for given pin locations and POWV as follows. The root node (at level 0) is $e_h(0,0)$ and the two child nodes (at level 1) of the root node are $e_h(0,1)$. In general, the nodes at level k are $e_h(\lfloor k/n \rfloor, k \bmod n)$ if $k < n(n-1)$ and $e_v(k \bmod n, \lfloor k/n \rfloor - (n-1))$ if $k >= n(n-1)$. Although we used a binary tree above to explain the proposed algorithm, we implemented the algorithm using a recursive function call without explicitly constructing a binary tree to reduce the memory usage.

3.3 Overall Algorithm

Algorithm 1 shows the overall algorithm for constructing all POSTs for given pin locations and a POWV. We first prepare an ordered set (array) E of all the edges (Line 1). The edges are sorted in the traversal order, so E is $(e_h(0,0), e_h(0,1), \ldots, e_h(1,0), \ldots, e_h(n-2,n-1), e_v(0,0), e_v(1,0), \ldots, e_v(n-1,n-2))$. Array R will contain all the POSTs for the given pin locations and POWV (Line 2). Then, we call

```
Input: Pin locations and a POWV (powv).
 1: Ordered set E = (e_h(0, 0), ..., e_v(n − 1, n − 2));
 2: R = {};
 3: Call recursive_construction (powv, E, R, 0);
 4: Return R;
Function: recursive_construction (powv, E, R, index)
 5: if powv == 0 or index == E.size then
 6:     if Current graph G connects all the pins then
 7:         Insert G into R;
 8:     end if
 9:     return;
10: end if
11: e = E[index];
12: if e is a used or removed edge then
13:     Call recursive_construction (powv, E, R, index+1);
14:     return;
15: end if
16: if powv(e) > 0 then
17:     Call use_or_remove_and_prune (e, NULL, powv);
18:     if # must-use and must-remove edges ≥ threshold then
19:         if Current graph G connects all the pins then
20:             recursive_construction (powv, E, R, index+1);
21:         end if
22:     else
23:         recursive_construction (powv, E, R, index+1);
24:     end if
25:     Roll back the must-use and must-remove edges.
26: end if
27: Call use_or_remove_and_prune (NULL, e, powv);
28: if # must-use and must-remove edges ≥ threshold then
29:     if Current graph G connects all the pins then
30:         recursive_construction (powv, E, R, index+1);
31:     end if
32: else
33:     recursive_construction (powv, E, R, index+1);
34: end if
35: Roll back the must-use and must-remove edges.
```

Algorithm 1: Construction of all POSTs for given pin locations and POWV.

function *recursive_construction* with the current POWV, E, R, and the edge index 0 (Line 3). Once the recursive function call finishes, we return R (Line 4).

At the beginning of function *recursive_construction*, we check whether the current POWV is equal to the zero vector or the edge index has reached the end of E (Line 5). If the condition is true, we check whether the current graph G connects all the pins by performing a BFS starting from a pin only through the used edges (Line 6). If G is connected, it is a POST, so we insert G into R (Line 7) and finish the current function call because there is no reason to explore using/removing edges further (if the POWV is zero) or there is no more edge to process (if the current node is a leaf node).

If the POWV is not equal to the zero vector and there are remaining edges to process (Line 11), we keep constructing POSTs as follows. If the current edge e is a used or removed edge (Line 12), we move on to the next edge (Line 13), which is the same as immediately taking the left or the right arrow at the node corresponding to e in B if it is a used or removed edge, respectively. If e is an available edge, we check whether $powv(e)$ is greater than zero (Line 16). If it

[1]We use the number of pins for the threshold.

is greater than zero, we try using e (which is traversing through the left arrow of the node corresponding to e in B) and prune additional edges (Line 17). Notice that we also try removing e from G (which is traversing through the right arrow of the node) and prune additional edges (Line 27). Once the pruning is done, we perform intermediate connectivity check (Line 18 and 19) when the number of must-use and must-remove edges is greater or equal to a threshold number. In this case, if we can reach all the pins in G through the used and available edges, we call function *recursive_construction* to continue to construct POSTs. If the number of must-use and must-remove edges is less than the threshold number, we just call function *recursive_construction* to move on to the next edge. If G is not connected, we immediately roll back all the changes by restoring G to its previous state (Line 25). Line 27 to Line 35 tries removing edge e from G.

Algorithm 2 shows the proposed algorithm for pruning must-use and must-remove edges after using or removing a given edge. First, insert given edge u into set U (Line 1) and insert given edge m into set M (Line 2). Then, we keep repeating processing must-use edges (from Line 4 to Line 16) and must-remove edges (from Line 17 to Line 26). For each edge e in U, we check whether e is a removed edge or $powv(e)$ is zero (Line 6). If e is a removed edge or $powv(e)$ is zero, we cannot use e in G because it is contradictory, so the current graph G cannot be a POST. Thus, if any of the two conditions is true, we stop processing the must-use edge and return *invalid_topology* (Line 7). Otherwise, we use e (Line 9) and decrease $powv(e)$ by 1 (Line 10). If $powv(e)$ becomes zero, we insert all the available edges in $PE(powv(e))$ into M so that we can remove the edges later (Line 12). Then, we check whether any of the edges in $NE(e)$ are must-use edges. If any, we insert them into U (Line 14) so that we can process them later.

Once we process all the must-use edges in U, we move on to the must-remove edges in M (Line 17). If e in M is a used edge (Line 19), removing e from G leads to a contradiction. Thus, we stop processing the must-remove edge and return *invalid_topology* (Line 20). Otherwise, we remove e from G (Line 22). Then, we insert all dangling edges and must-use edges in $NE(e)$ into M (Line 23) and U (Line 24), respectively, to process them later.

```
Function: Use_or_remove_and_prune (u, m, powv)
Input: (Edge u to use, Edge m to remove, a POWV (powv)).
 1: U = {u};
 2: M = {m};
 3: while U.size + M.size > 0 do
 4:    while U.size > 0 do
 5:       for each e ∈ U do
 6:          if e is a removed edge or powv(e) == 0 then
 7:             return invalid_topology;
 8:          end if
 9:          Use e in G;
10:          powv(e) = powv(e) - 1;
11:          if powv(e) == 0 then
12:             Insert all available edges in PE(powv(e)) into M;
13:          end if
14:          Insert all must-use edges in NE(e) into U.
15:       end for
16:    end while
17:    while M.size > 0 do
18:       for each e ∈ M do
19:          if e is a used edge then
20:             return invalid_topology;
21:          end if
22:          Remove e from G;
23:          Insert all dangling edges in NE(e) into M;
24:          Insert all must-use edges in NE(e) into U;
25:       end for
26:    end while
27: end while
```

Algorithm 2: Use or remove a given edge and process must-use and must-remove edges.

3.4 Example

Figure 6 shows an example. In Figure 6(a), four pins, their position sequence (4123), and a POWV (121111) are given. Starting with edge e_1, $powv(e_1)$ is 1, so we try using it first by marking it used and reducing $powv(e_1)$ by 1 in Figure 6(b). In this case, e_1 will be a dangling edge if e_{13} is not used, so e_{13} becomes a must-use edge. In addition, e_2, e_3, and e_4 become must-remove edges because the POWV element corresponding to the edges is zero. In Figure 6(c), we use e_{13} in G and decrease $powv(e_{13})$ by 1, so the POWV becomes (021011). Since $powv(e_{13})$ becomes zero, e_{16}, e_{19}, and e_{22} become must-remove edges. In Figure 6(d), we remove e_2, e_3, and e_4 in this order. If we remove e_4, e_{15} becomes a must-remove edge. In Figure 6(e), $powv(e_{13})$ is zero, so we remove e_{16}, e_{19}, and e_{22}. Then, we remove e_{15} in Figure 6(f), which causes e_{14} to be dangling, so we remove e_{14} too. However, e_{13} is not dangling when we remove e_{14} because the top vertex of e_{13} is a pin vertex.

When we remove e_{16} in Figure 6(e), e_5 becomes a must-use edge because e_1 will be dangling if e_5 is not used. Similarly, when we remove e_{19} and use e_5 after removing e_{16}, e_9 becomes a must-use edge because e_5 will be dangling if e_9 is not used. Thus, e_5 and e_9 become must-use edges. We use these two edges in Figure 6(g) and decrease $powv(e_5)$ and $powv(e_9)$ by 1, so the POWV becomes (010011). Since the third element of the POWV is zero, e_{10}, e_{11}, and e_{12} become must-remove edges, so we remove them in Figure 6(h). Removing the three edges causes e_{23} and e_{24} to be dangling as shown in Figure 6(h), so we remove them in Figure 6(i). Figure 6(j) shows the result of using e_1.

Overall, using e_1 leads to using three additional edges (e_5, e_9, e_{13}) and removing 13 edges (e_2, e_3, e_4, e_{10}, e_{11}, e_{12}, e_{14}, e_{15}, e_{16}, e_{19}, e_{22}, e_{23}, e_{24}). Since the total number of must-use and must-remove edges at this step is 16, which is greater than the total number of pins (four), we perform the intermediate connectivity check. Since the pins are disconnected, using e_1 will not generate POSTs. Thus, we roll back all the used and removed edges and try removing e_1 from the graph in Figure 6(k). e_{13} becomes dangling in this case, so we remove e_{13} too in Figure 6(l). Then, we move on to e_2.

4 SIMULATION RESULTS

In this section, we present various simulation results obtained from the construction of all POSTs on the Hanan grid. We implemented the proposed algorithm using C/C++ and ran all simulations in a

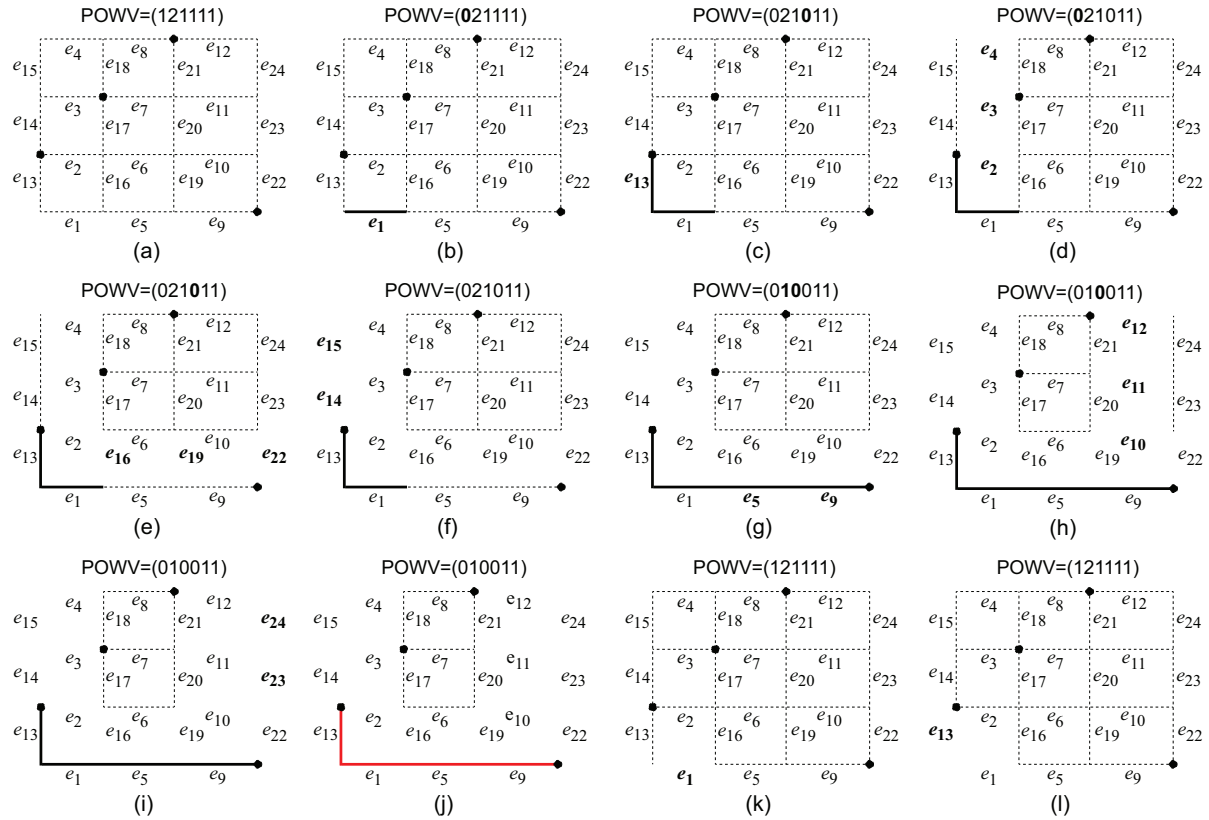

Figure 6: An example of edge pruning.

Table 1: Statistics of the construction of all POSTs. "Con. time" is the construction time for all the POSTs for each pin count and "Con. eff." is the construction efficiency measured by the number of total POSTs over the construction time (in seconds).

# pins (n)	# pin groups ($n!$)	# POWVs in a group			# POSTs for a POWV			# POSTs	Con. time	Con. eff.	Table size
		Min.	Avg.	Max.	Min.	Avg.	Max.				
2	2	1	1	1	2	2	2	4	0.0 s	-	0 MB
3	6	1	1	1	2	2.667	4	16	0.0002 s	80,000	0 MB
4	24	1	1.667	2	2	7.100	12	284	0.0035 s	81,142	0 MB
5	120	1	2.467	3	4	14.392	38	4,260	0.079 s	53,924	0.1 MB
6	720	1	4.433	8	4	37.661	216	120,212	3.72 s	32,315	3.5 MB
7	5,040	1	7.932	15	4	98.080	852	3,920,832	254 s	15,436	141 MB
8	40,320	1	15.251	33	6	289.972	6,558	178,313,916	9.06 hr	5,465	7.7 GB
9	362,880	1	30.039	79	8	929.600	52,010	10,133,050,012	1,700 hr	1,656	525 GB

3.3GHz Intel Core i5-3550 system with 32GB memory. We used only one core to build the database of all POSTs.

4.1 POWVs and POSTs

Table 1 shows various statistics about the construction of all POSTs for up to nine pins. The number of pin groups is the number of position sequences for a given pin count (n). The total number of POSTs for each pin count and the average number of POSTs per POWV increase exponentially. The construction time is almost negligible for up to five pins, but then it increases exponentially, 3.72 seconds for six pins, 254 seconds for seven pins, 9.06 hours for eight pins, and approximately 1,700 hours for nine pins. We also show the construction efficiency measured by the total number POSTs divided by the construction time in seconds. As shown in the table, the construction efficiency goes down exponentially as the pin count goes up. However, the proposed algorithm can still construct 5,465 POSTs per second for eight pins and 1,656 POSTs per second for nine pins on average.

Since the database has all POWVs and POSTs, we can construct all RSMTs for given pin locations as follows. First, we obtain all

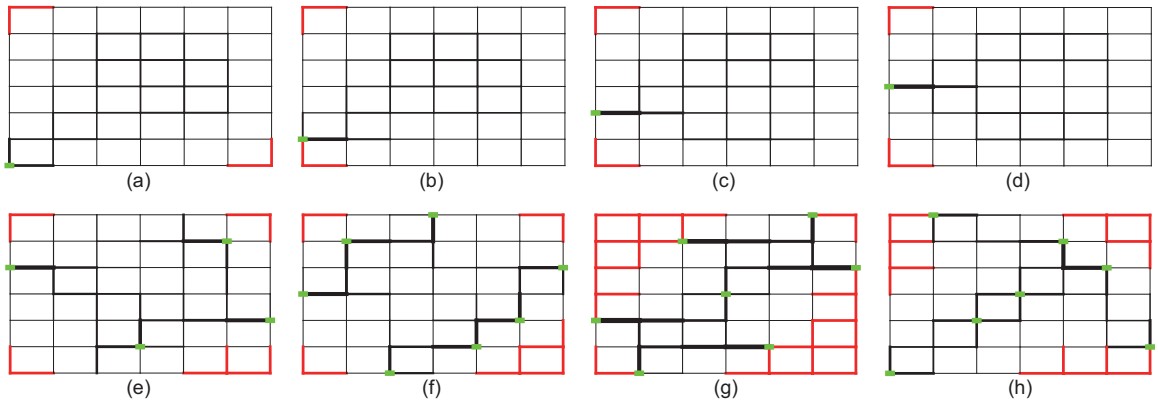

Figure 7: Statistics of POSTs for seven pins. The red edges are not used at all in any POSTs. Thicker edges are used in more POSTs than thinner edges. Green rectangles are pins. Position sequences are as follows. (a) (1XXXXXX), (b) (X1XXXXX), (c) (XX1XXXX), (d) (XXX1XXX), (e) (X47X16X), (f) (3561724), (g) (2514736), which is one of the position sequences having the fewest POSTs, (h) (1734652), which is one of the position sequences having the most POSTs. X is a don't-care.

the POWVs of the position sequence for the pin locations from the database. Then, we compute a dot product between each POWV and the edge length vector $(h_1, ..., v_1, ...)$ and obtain all POWVs having the minimum wirelength. Then, we return all POSTs belonging to the POWVs from the database.

4.2 Statistics of POSTs

In this simulation, we investigate how many times each edge is used in all POSTs for given pin locations. The simulation methodology is as follows. We first come up with a position sequence for seven pins. Each position sequence can be an exact sequence such as (1234567) or include some don't-cares (X). For example, position sequence (12345XX) includes two position sequences (1234567) and (1234576). Then, we search the database to find all POSTs matching the position sequence and count how many times each edge is used in the POSTs. This statistics help estimate whether we can route a given net through non-congested area. If an edge is used in most of the POSTs for given pin locations, for example, it would be hard to route the net without using the edge.

Figure 7 shows eight examples for seven pins. In the figure, the thickness of a black edge is proportional to the number of times it is used. Red edges are not used at all. Green rectangles are pins. First, Figure 7(a) shows the usage of the edges for (1XXXXXX), i.e., when a pin is located at (0,0). As the figure shows, the two edges adjacent to vertex (0,0) are used in almost all POSTs. The edges in the middle area are also used in many POSTs, which means that it would not be possible to route through the middle area if the position sequence of a seven-pin net is (1XXXXXX). Figure 7(b) shows the edge usage for (X1XXXXX), i.e., when a pin is located at (0,1). In this case, none of the POSTs uses edges $e_h(0,0)$, $e_v(0,0)$, $e_h(0,6)$, and $e_v(0,5)$ no matter where the other six pins are located. Similarly, position sequences (XX1XXXX) and (XXX1XXX) do not use the same four edges and heavily use the right edge of the pin vertex and the edges in the middle of the grid as shown in Figure 7(c) and (d). Figure 7(e) shows the usage for position sequence (X47X16X), which we picked randomly. For this position sequence, some edges such as $e_h(1,3)$, $e_h(4,2)$, and $e_v(5,4)$ around the middle of the grid are used in many POSTs. Figure 7(f) shows the usage of an exact position sequence (3561724). Since the pins are distributed around the boundaries of the grid, the edges in the middle of the grid are used many times. Figure 7(g) shows the usage for (2514736), which is one of the position sequences having the fewest POSTs and Figure 7(h) shows the usage for (1734652), which is one of the position sequences having the most POSTs.

Table 2: Effectiveness (runtime in seconds) of the pruning algorithms. All: Enabling all pruning algorithms. The other four columns are disabling (1) zero POWV elements, (2) must-use edges, (3) must-remove edges, and (4) intermediate connectivity check.

	All	(1)	(2)	(3)	(4)
6 pins	3.72	13.58	3,850	22.39	9.15
	Ratio (1.00)	3.65×	1,035×	6.02×	2.46×
7 pins	254	2,837	∞	6,973	964
	Ratio (1.00)	11.17×	-	27.45×	3.80×

4.3 Effectiveness of the Pruning Algorithms

We use four pruning algorithms, 1) pruning by zero POWV elements, 2) pruning by must-use edges, 3) pruning by must-remove edges, and 4) intermediate connectivity check, to reduce the POST construction time. Thus, we measured the effectiveness of each algorithm by disabling each of them while enabling all the other techniques. Table 2 shows that pruning by must-use edges is the most effective technique and pruning by must-remove edges is also effective when the pin count goes up. However, the other two pruning techniques also help reduce the runtime considerably, especially when the pin count goes up. For example, if the intermediate connectivity check is disabled, finding all POSTs for the nine-pin case would take more than 8,500 hours instead of 1,700 hours.

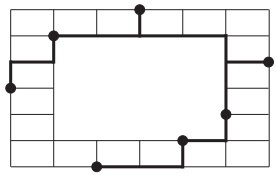

Figure 8: A POST not using specific edges for position sequence (3561724).

4.4 Application – POSTs Using/Not Using Specific Edges

A representative application of the proposed algorithm is multiple routing topology generation for global routing. Generating multiple RSMTs for each net can effectively reduce routing overflows, minimize routing congestion, and reduce the total coupling capacitance. In this section, we show how to use the database of all POSTs to avoid non-preferred (such as congested) area and/or preferred (such as non-congested) area. Suppose a set of pin locations and non-preferred region are given. Then, we search the POST database to find all POWVs belonging to the position sequence of the given pin locations. For each POWV, we compute the wire length by the dot product between the POWV and the edge length vector. Then, we find all POWVs having the shortest wire length. For each POST belonging to the POWVs, we check whether the POST uses any edges in the non-preferred region. Finally, we return all the POSTs not using any edges in the non-preferred region. Figure 8 shows an example for position sequence (3561724) shown in Figure 7(f). We searched for POSTs not containing the removed edges in Figure 8. The POST in the figure shows one of the POSTs satisfying the condition.

The search time consists of 1) finding the position sequence, 2) finding the set P all the POWVs belonging to the position sequence and having the shortest wire length, 3) going through all the POSTs in P and checking whether each POST contains specific edges. The runtime of the first step is negligible and the complexity of the second step is approximately $O(n \cdot 2^n)$ where n is the number of pins. The exponential term comes from the total number of POWVs belonging to a position sequence as shown in Table 1 and the multiplication factor n comes from the total number of multiplications for the dot product computation. The complexity of the third step is approximately $O(k \cdot 3^n)$ where n is the number of pins. The exponential term comes from the total number of POSTs for a POWV and k is the number of edges in the non-preferred and/or preferred regions.

Notice that this does not solve the obstacle-avoiding RSMT construction problem that finds RSTs having the minimum wirelength for given pin locations and obstacles. Rather, we return all POSTs (or RSMTs if their POWVs have the minimum wirelength) that use and/or do not use specific edges.

4.5 Multiple RSTs For More Than Nine Pins

Although it might be inefficient or impossible (due to the large database size) to build and use a database for nets having more than nine pins, if a set of pin locations is given, we can run FLUTE to construct an RST, obtain its wirelength vector (WV), and run the proposed algorithm to obtain multiple RSTs having the same WV. Notice that the proposed algorithm is not limited to constructing RSMTs. Rather, if pin locations and a wirelength vector are given, the proposed algorithm can construct all RSTs satisfying the given WV. Thus, we tried constructing multiple RSTs using FLUTE for a few cases. Constructing all Steiner trees (STs) for a 10-pin, a 11-pin, and a 12-pin cases (each with one WV) found 324, 6,390, and 870 STs in 10.38 seconds, 72.99 seconds, and 7.91 seconds, respectively. The 12-pin case had a smaller search space than the 10- and 11-pin cases, so it took only 7.91 seconds.

5 CONCLUSION

In this paper, we proposed an efficient algorithm to construct all RSMTs for up to nine pins using a lookup table and FLUTE. The generation time and table size are reasonable for up to nine pins, but the number of POSTs, the database generation time, and the database size increase exponentially as the number of pins goes up. Using the database of all POSTs, we investigated several properties of the POSTs. The proposed algorithm and the database of all POSTs will help various VLSI CAD software optimize layouts more efficiently.

ACKNOWLEDGMENTS

This work was supported by the Defense Advanced Research Projects Agency Young Faculty Award under Grant D16AP00119 and the New Faculty Seed Grant (125679-002) funded by the Washington State University.

REFERENCES

[1] Manjit Borah, Robert Michael Owens, and Mary Jane Irwin. 1994. An Edge-Based Heuristic for Steiner Routing, Vol. 13. 1563–1568.
[2] Zhen Cao, Tong Jing, Jinjun Xiong, Yu Hu, Lei He, and Xianlong Hong. 2007. DpRouter: A Fast and Accurate Dynamic-Pattern-Based Global Routing Algorithm. 256–261.
[3] Yen-Jung Chang, Yu-Ting Lee, Jhih-Rong Giao, Pei-Ci Wu, and Ting-Chi Wang. 2010. NTHU-Route 2.0: A Robust Global Router for Modern Designs, Vol. 29. 1931–1944.
[4] Minsik Cho and David Z. Pan. 2007. BoxRouter: A New Global Router Based on Box Expansion and Progressive ILP, Vol. 26. 2130–2143.
[5] Chris Chu and Yiu-Chung Wong. 2008. FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design, Vol. 27. 70–83.
[6] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: Freeman.
[7] GeoSteiner. [n. d.]. Software for Computing Steiner Trees. http://www.geosteiner.com. ([n. d.]).
[8] J. Griffith, G. Robins, J. S. Salowe, and Tongtong Zhang. 1994. Closing the Gap: Near-Optimal Steiner Trees in Polynomial Time, Vol. 13. 1351–1365.
[9] M. Hanan. 1966. On Steiner's Problem with Rectilinear Distance. In *SIAM Journal on Applied Mathematics*, Vol. 14. 255–265.
[10] F. K. Hwang, D. S. Richards, and P. Winter. 1992. *The Steiner Tree Problem.* Elsevier.
[11] Andrew B. Kahng, I. I. Mandoiu, and A. Z. Zelikovsky. 2003. Highly Scalable Algorithms for Rectilinear and Octilinear Steiner Trees. 827–833.
[12] Ion I. Mandoiu, Vijay V. Vazirani, and Joseph L. Ganley. 2000. A New Heuristic for Rectilinear Steiner Trees, Vol. 19. 1129–1139.
[13] Michael D. Moffitt. 2008. MaizeRouter: Engineering an Effective Global Router, Vol. 27. 2017–2026.
[14] Muhammet Mustafa Ozdal and Martin D. F. Wong. 2007. Archer: A History-Driven Global Routing Algorithm. 488–495.
[15] Tai-Hsuan Wu, Azadeh Davoodi, and Jeffrey T. Linderoth. 2009. GRIP: Scalable 3D Global Routing Using Integer Programming. 320–325.
[16] Yue Xu, Yanheng Zhang, and Chris Chu. 2009. FastRoute 4.0: Global Router with Efficient Via Minimization. 576–581.
[17] Hai Zhou. 2004. Efficient Steiner Tree Construction Based on Spanning Graphs, Vol. 23. 704–710.