Construction of All Rectilinear Steiner Minimum Trees on the Hanan Grid and Its Applications to VLSI Design

Sheng-En David Lin, *Student Member, IEEE*, and Dae Hyun Kim, *Member, IEEE*

***Abstract*—A rectilinear Steiner minimum tree (RSMT) is a rectilinear Steiner tree connecting a given set of pins with the shortest wirelength. RSMT construction is one of the most frequently used algorithms in the physical design automation, including floorplanning, placement, routing, and interconnect estimation and optimization. Thus, efficient algorithms to construct RSMTs have been developed for many years in academia and industry. Unfortunately, RSMT construction is an NP-hard problem, so even a fast RSMT construction algorithm, such as GeoSteiner is too slow to use in physical design automation tools. FLUTE, a fast lookup-table-based RSMT construction algorithm, builds and uses a routing topology database to quickly construct RSMTs. In this paper, we present an algorithm to build a database (ARSMT DB) to construct all RSMTs on the Hanan grid for a given set of pins. ARSMT DB constructs all RSMTs in almost no time, so numerous applications could use it for various purposes. We apply the ARSMT DB to two applications, timing-driven RSMT construction and congestion-aware global routing, and show that the ARSMT DB can help reduce source-to-critical-sink lengths, source-to-critical-sink delays, and routing congestion significantly. Since the size of the original ARSMT DB is too large, we present techniques to reduce the database size.**

***Index Terms*—Congestion, rectilinear Steiner minimum tree (RSMT), routing, wirelength.**

I. 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 [3]. RSMT or RST construction is heavily used in many computer-aided design (CAD) tools. For example, floorplanning and placement use RSMTs and RSTs to estimate the total wirelength and routing congestion. Routing uses RSMTs and RSTs to find routing topologies minimizing the total wirelength, routing congestion, and the critical path delay. Interconnect optimization, such as buffer insertion also uses RSMTs and RSTs to optimize timing and reduce dynamic power consumption. Thus, several fast algorithms have been proposed in the literature to construct an RSMT for a given set of pin locations [1], [2], [4]–[6]. However, the RSMT construction problem is NP-hard [7], so several papers also proposed rectilinear minimum spanning tree (RMST) or RST construction algorithms for practical use [1], [8]–[12].

FLUTE, a lookup-table-based RSMT construction algorithm, builds a database composed 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. Among five RSMT and one RMST construction algorithms, FLUTE achieves the shortest wirelength on average for all the 18 IBM benchmarks in [2]. In addition, its runtime is 5.56× to 64.92× shorter than the runtime of all the other RSMT algorithms compared in [2].

One of the applications heavily using RSMT construction is global routing in which RSMTs are used for routing topologies. For example, BoxRouter [13], DpRouter [14], Archer [15], MaizeRouter [16], FastRoute [17], GRIP [18], and NTHU-Route [19] use FLUTE for routing topology generation. However, FLUTE constructs only one RSMT for a net. In this paper, we propose an efficient algorithm to construct all RSMTs on the Hanan grid for given pin locations. The algorithm builds a database (called ARSMT DB) of all POSTs on the Hanan grid for each POWV so that applications can quickly obtain all RSMTs from the ARSMT DB. We perform sequential congestion-aware global routing using FLUTE and the ARSMT DB and show that the ARSMT DB can help reduce routing congestion significantly without runtime overhead. We also use the ARSMT DB to minimize the source-to-critical-sink length (SCSL) and source-to-critical-sink delay (SCSD) of a given net. Minimizing the length and delay from the source to a specific sink of a net without degrading the total length of the net is very crucial for timing closure and optimization. The simulation results show that using the ARSMT DB significantly outperforms using the FLUTE for the SCSL and SCSD minimization. We also present techniques to reduce the database size.

The rest of this paper is organized as follows. We review the RSMT construction algorithm used in FLUTE in Section II. In Section III, we propose an algorithm to build the ARSMT DB. Section IV shows simulation results and detailed analysis for the database generation.

Manuscript received June 15, 2018; revised November 22, 2018; accepted March 21, 2019. Date of publication May 20, 2019; date of current version May 22, 2020. This work was supported in part by the Defense Advanced Research Projects Agency Young Faculty Award under Grant D16AP00119, and in part by Washington State University New Faculty Seed under Grant 125679-002. This paper was recommended by Associate Editors I. Bustany and B. Chu. *(Corresponding author: Dae Hyun Kim.)*

The authors are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164 USA (e-mail: dl23ee.lin@wsu.edu; daehyun@eecs.wsu.edu).

Digital Object Identifier 10.1109/TCAD.2019.2917896

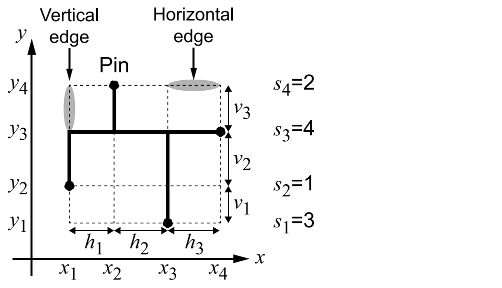

Fig. 1. Four pins on the Hanan grid, their position sequence (3142), and an RSMT constructed on the Hanan grid.

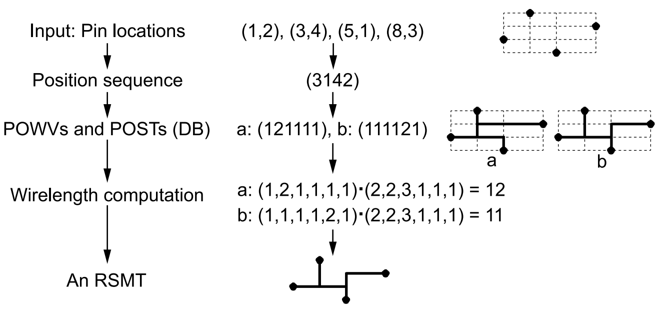

Fig. 2. RSMT construction from two POSTs.

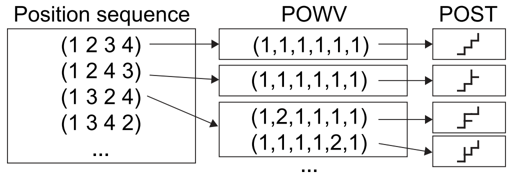

Fig. 3. Overview of the FLUTE database.

In Section V, we present timing-driven RSMT construction and congestion-aware global routing using the ARSMT DB. Section VI explains techniques for the database size reduction and we conclude in Section VII.

II. Algorithm of FLUTE

In this section, we briefly review the idea of FLUTE [2].

A. Position Sequence

Let $P = \{p_1, \ldots, p_n\}$ 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 ith vertical line from the left and y_i be the y-coordinate of the ith horizontal line from the bottom on the Hanan grid as shown in Fig. 1. Then, we can characterize the distribution of the pins using a *position sequence* 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 has a position sequence $(s_1 s_2 \ldots s_n)$. Fig. 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.

B. Potentially Optimal Wirelength Vector

The Hanan grid constructed for a position sequence can be decomposed into horizontal and vertical edges as shown in Fig. 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, we can express the wirelength of any RST constructed on the Hanan grid using the lengths of the horizontal and vertical edges. For example, the wirelength of the RSMT shown in Fig. 1 is

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

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 $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ are given, if $a_i < b_i$ holds for at least one $i = 1, \ldots, n$ and $a_j \leq b_j$ holds for all the other $j = 1, \ldots, n$, the dot product $A \bullet H$ between A and an edge length vector H is always less than $B \bullet H$. However, if $a_i < b_i$ holds for some i and $a_j > b_j$ holds for some j, $A \bullet H$ 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. Each element in C is called a POWV.

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

C. 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 database. A topology stored for each POWV is called a POST. Fig. 2 shows an example. For the four pins located at $(1, 2)$, $(3, 4)$, $(5, 1)$, and $(8, 3)$, FLUTE obtains the position sequence (3142) and two POWVs $(1, 2, 1, 1, 1, 1)$ and $(1, 1, 1, 1, 2, 1)$ belonging to the position sequence. The POSTs corresponding to the POWVs are also shown in the figure. Then, FLUTE computes the wirelength for each POWV and returns the POST corresponding to the POWV having the minimum wirelength. We refer readers to [2] for the details of the FLUTE database construction.

Fig. 3 shows an overview of the FLUTE database. It stores all position sequences for n pins $(n = 2, 3, \ldots, 9)$. Each position sequence has one or several POWVs. Each POWV has a POST in the database.

III. Construction of All RSMTs

A POST becomes an RSMT if the POWV of the POST has the minimum wirelength for given pin locations. Thus,

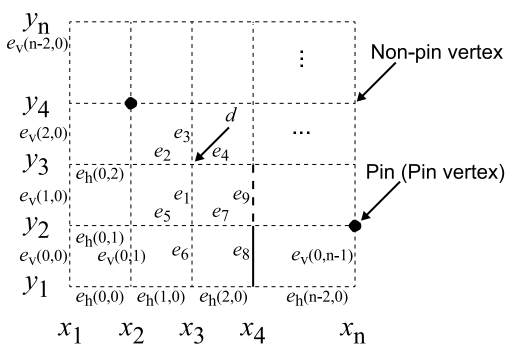
Non-pin vertex
Pin (Pin vertex)
d
y_n y_4 y_3 y_2 y_1
x_1 x_2 x_3 x_4 x_n
$e_v(n-2,0)$ $e_v(2,0)$ $e_v(1,0)$ $e_v(0,0)$ $e_v(0,1)$ $e_v(0,n-1)$
$e_h(0,2)$ $e_h(0,1)$ $e_h(0,0)$ $e_h(1,0)$ $e_h(2,0)$ $e_h(n-2,0)$
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9

Fig. 4. Hanan grid for n pins.

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.

A. Terminologies and Notations

Fig. 4 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*. We call the edges connected to a vertex the *neighboring edges* of the vertex and denote the set of all the neighboring edges of vertex d by NE(d). If edge e_i connects vertices d_j and d_k, we call $\{\mathrm{NE}(d_j) \cup \mathrm{NE}(d_k)\} - \{e_i\}$ the set of the neighboring edges of edge e_i and denote it by NE(e_i). In Fig. 4, NE(d) is $\{e_1, e_2, e_3, e_4\}$ and NE(e_1) is $\{e_2, e_3, e_4, e_5, e_6, e_7\}$. 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 $e_h(i,j)$ and each vertical edge by $e_v(i,j)$, where i and j are the indices to locate the edge. The indices are shown in Fig. 4. If a vertex of an edge is not a pin vertex and is not connected to any other edges, the edge 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 Fig. 4 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 Fig. 4. powv(e) for given edge e is the POWV element corresponding to e. If a POWV is $(q_1, q_2, \ldots, r_1, r_2, \ldots,)$, 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_{i+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 $\{e_h(0,0), \ldots, e_h(0, n-1)\}$.

B. 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

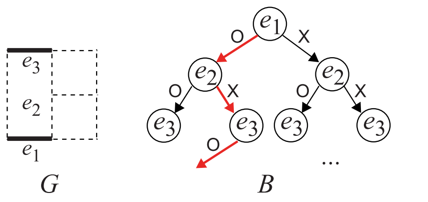

Fig. 5. Rectilinear graph G constructed on the Hanan grid and a binary tree B corresponding to G. The red path shows a decision sequence. e_2 is removed in G because the red path traverses through the right arrow of e_2. O and X mean the edge is used or removed in G, respectively.

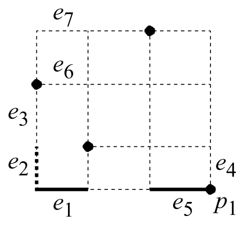

Fig. 6. Must-use and must-remove edges.

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. Fig. 5 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 check the connectivity.

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. 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 the n pins has $2^{2n(n-1)}$ leaf nodes. Since we use the BFS algorithm for the connectivity check of G and there are $2n(n-1)$ edges, the complexity to check the connectivity is $O(n^2)$. Thus, the complexity of the exhaustive POST construction algorithm is $O(n^2 \cdot 2^{2n(n-1)})$. When we find all POSTs, however, we apply several pruning algorithms as follows to reduce the search space.

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 Fig. 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 Fig. 1 should be removed from G.

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, using an edge causes another edge to be a must-use edge. For example, suppose we decide to use edge e_1 in Fig. 6. If

Algorithm 1: Construction of All POSTs for Given Pin Locations and a POWV

```
input: Pin locations and a POWV (powv).
1: Ordered set $E = (e_h(0, 0), ..., e_v(n-2, n-1))$;
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 $\geq$ 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 $\geq$ 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.
```

$N_L(e_1)$ 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. Second, removing an edge causes another edge to be a must-remove edge. For example, suppose we decide to remove e_1 in Fig. 6, which causes e_2 to be dangling. As a result, e_2 becomes a must-remove edge. If we remove e_2, e_3 also becomes a must-remove edge, so we should remove e_3 too. We can remove multiple edges consecutively in this way. Third, using or removing an edge can cause some of its neighboring edges to be must-remove or must-use edges, respectively. For example, if powv(e_1) is 1 and we use e_1 in Fig. 6, e_6 and e_7 become must-remove edges. On the other hand, if we remove e_4, e_5 becomes a must-use edge because e_5 is the only edge connecting pin p_1.

3) 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 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 predetermined threshold value.[1]

Evaluation of G checks whether G 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. We construct B 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 $e_v(\lfloor k/n \rfloor - (n-1), k \bmod n)$ 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 as shown in the next section.

C. 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 $(e_h(0, 0), e_h(0, 1), \ldots, e_h(1, 0), \ldots, e_h(n-2, n-1), e_v(0, 0), e_v(0, 1), \ldots, e_v(n-2, n-1))$. Array R will contain all the POSTs for the given pin locations and POWV (line 2). Then, we call 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). If e is an available edge, we check whether powv(e) is greater than zero (line 16). If it is greater than zero, we try using e and prune additional edges (line 17). Notice that we also try removing e from G and prune additional edges later (line 27). Once the pruning is done, we perform an intermediate connectivity check (lines 18 and 19) if 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. Then, we roll back all the changes by restoring G to its previous state (line 25). Lines 27–35 try removing edge e from G.

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

Algorithm 2: Use or Remove a Given Edge and Process Must-Use and Must-Remove Edges

```
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 $e \in U$ do
 6:             if $e$ is a removed edge or powv($e$) == 0 then
 7:                 return invalid_topology;
 8:             end if
 9:             Use $e$ in $G$; Remove $e$ from $U$;
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 $e \in M$ do
19:             if $e$ is a used edge then
20:                 return invalid_topology;
21:             end if
22:             Remove $e$ from $G$; Remove $e$ from $M$;
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 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 lines 4–16) and must-remove edges (from lines 17–26). 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 in G and remove e from U (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). If any of the edges in NE(e) are must-use edges, 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 and U (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.

D. Example

Fig. 7 shows an example. In Fig. 7(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 Fig. 7(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 Fig. 7(c), we use e_{13} in G, decrease powv(e_{13}) by 1, and remove e_2, e_3, and e_4. Since powv(e_{13}) becomes zero, e_{16}, e_{19}, and e_{22} become must-remove edges. At the same time, e_{15} is a dangling edge. If we remove e_{15}, e_{14} becomes a dangling edge. Thus, we remove e_{14}, e_{15}, e_{16}, e_{19}, and e_{22} in Fig. 7(d). e_{13} is not dangling because the top vertex of e_{13} is a pin vertex.

When we remove e_{16} in Fig. 7(d), e_5 becomes a must-use edge because e_1 will be dangling if e_5 is not used. For the same reason, e_9 becomes a must-use edge. Thus, we use these two edges and decrease powv(e_5) and powv(e_9) by 1 in Fig. 7(e). Since the third element of the POWV is zero, e_{10}, e_{11}, and e_{12} become must-remove edges. If we remove them, e_{23} and e_{24} become dangling, so we remove them too in Fig. 7(f), which shows the final result of using e_1. 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 in Fig. 7(g). e_{13} becomes dangling in this case, so we remove e_{13} in Fig. 7(h). Then, we move on to e_2.

IV. Statistics of the Database Generation

In this section, we present several results obtained from the ARSMT DB construction. We implemented the algorithm using C/C++ and ran all simulations in a 3.3-GHz Intel Core i5-3550 system with 32-GB memory. We used only one core to build the database.

A. # POWVs, # POSTs, and DB Construction Time

Table I shows some statistics of the ARSMT DB construction. First, the total number of POSTs and the average number of POSTs per POWV increase exponentially. The construction time is almost negligible for up to five pins, but it increases exponentially as the pin count goes up. We also show the construction efficiency measured by the total number POSTs divided by the construction time in seconds. As the table shows, the construction efficiency goes down exponentially as the pin count increases. Thus, constructing the ARSMT DB for more than nine pins might be practically impossible.

B. 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 select a position sequence. The 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 ARSMT DB 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

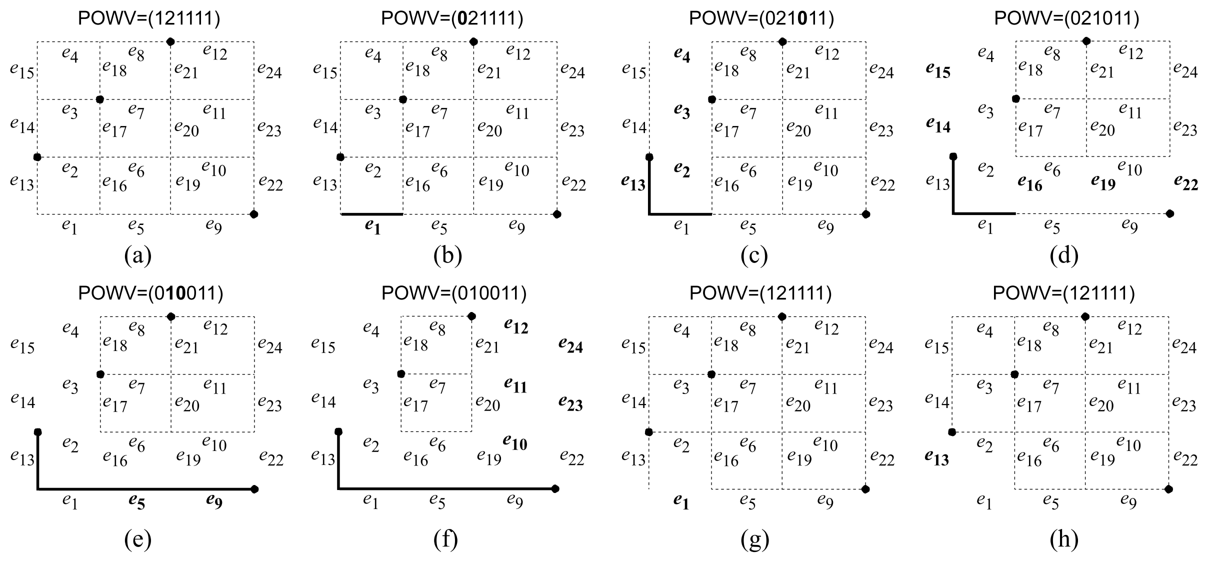

Fig. 7. Example of edge pruning.

TABLE I
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)	# position sequences ($n!$)	# POWVs in a group			# POSTs for a POWV			# POSTs	Con. time	Con. eff.	Database size	
		Min.	Avg.	Max.	Min.	Avg.	Max.				Raw data	Reduced
2	2	1	1	1	2	2	2	4	-	-	-	-
3	6	1	1	1	2	2.7	4	16	0.0002 s	80,000	0.3 KB	0.1 KB
4	24	1	1.7	2	2	7.1	12	284	0.0035 s	81,142	5.0 KB	1.2 KB
5	120	1	2.5	3	4	14.4	38	4,260	0.079 s	53,924	98 KB	11 KB
6	720	1	4.4	8	4	37.7	216	120,212	3.72 s	32,315	3.5 MB	0.24 MB
7	5,040	1	7.9	15	4	98.1	852	3,920,832	254 s	15,436	141 MB	6.7 MB
8	40,320	1	15.3	33	6	290.0	6,558	178,313,916	9.06 hr	5,465	7.7 GB	0.3 GB
9	362,880	1	30.0	79	8	929.6	52,010	10,133,050,012	1,700 hr	1,656	525 GB	19 GB

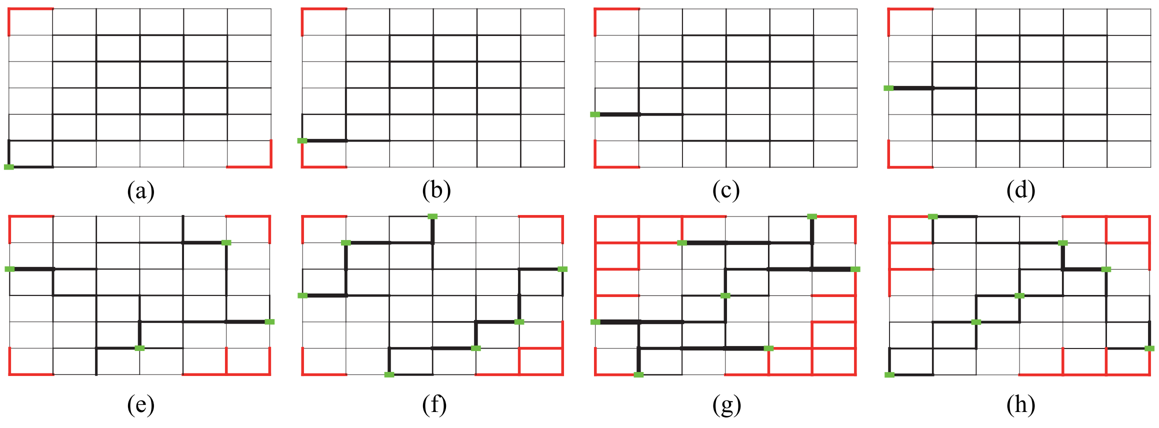

Fig. 8. Statistics of POSTs for seven pins. The red edges are not used at all in any POSTs. Thicker edges are used in more POSTs than thinner edges. Green rectangles are pins. Position sequences are as follows. (a) (1XXXXXX), (b) (X1XXXXX), (c) (XX1XXXX), (d) (XXX1XXX), (e) (X47X16X), (f) (3561724), (g) (2514736), which is a position sequence having the fewest POSTs, and (h) (1734652), which is a position sequence having the most POSTs. X is a don't-care.

noncongested 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.

Fig. 8 shows eight examples for seven pins. In the figures, 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, Fig. 8(a) shows the usage of the edges for (1XXXXXX), i.e., one of the pins is located at (0, 0). The two edges adjacent to vertex (0, 0) are heavily used in the POSTs and the edges in the center area are also used in many POSTs. Thus, it would not be possible to route a net through the center area if the position sequence of the net is (1XXXXXX). Fig. 8(b) shows the edge usage for (X1XXXXX). In this case, none of the POSTs uses edges $e_h(0,0)$, $e_v(0,0)$, $e_h(0,6)$, and $e_v(5,0)$ no matter where the other six pins are located. Similarly, position sequences (XX1XXXX) and (XXX1XXX)

TABLE II
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×

do not use the same four edges and heavily use the right edge of the pin vertex and the edges in the center of the grid as shown in Fig. 8(c) and (d). Fig. 8(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 $e_v(4, 5)$ around the middle of the grid are used in many POSTs. Fig. 8(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 center area are used in many POSTs. Fig. 8(g) shows the usage for (2514736), which is a position sequence having the fewest POSTs and Fig. 8(h) shows the usage for (1734652), which is a position sequence having the most POSTs.

C. 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 II shows that pruning by must-use edges is the most effective technique. However, the other three pruning techniques also help reduce the runtime considerably.

D. POSTs Using/Not Using Specific Edges

A representative application of the proposed algorithm is generating multiple routing topologies 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 ARSMT DB to avoid nonpreferred (such as congested) area and/or use preferred (such as noncongested) area. Suppose a set of pin locations and nonpreferred region are given. Then, we search the ARSMT DB to find all POWVs belonging to the position sequence of the pin locations and having the shortest wirelength. For each POST belonging to the POWVs, we check whether the POST uses any edges in the nonpreferred region. Finally, we return all the POSTs not using any edges in the nonpreferred region. Fig. 9 shows an example for position sequence (3561724) shown in Fig. 8(f). We searched for POSTs not containing the removed edges in Fig. 9. 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 wirelength; and 3) checking whether each POST in P contains specific edges.

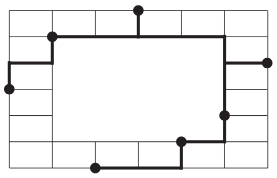
Fig. 9. POST not using specific edges for position sequence (3561724).

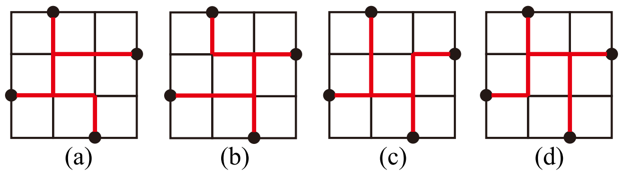

Fig. 10. Four RSMTs for position sequence (3142). (a) and (b) for POWV 121111 and (c) and (d) for POWV 111121.

The runtime of the first step is negligible and the complexity of the second step is approximately $\mathcal{O}(n \cdot 2^n)$, 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 I 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 $\mathcal{O}(k \cdot 3^n)$, 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 nonpreferred 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.

E. Multiple RSTs for More Than Nine Pins

For high-degree nets having more than nine pins, it might be inefficient or impossible (due to the large database size) to build and use a POST database. However, the proposed algorithm is not limited to constructing RSMTs. Rather, if pin locations and a wirelength vector (WV) are given, the algorithm can construct all RSTs belonging to the given WV. For high-degree nets, therefore, we can run FLUTE to construct an RST, obtain its WV, and run the proposed algorithm to obtain multiple RSTs. In this experiment, we tried constructing multiple RSTs using FLUTE for a few cases. Constructing all RSTs for a 10-pin, an 11-pin, and a 12-pin cases (each with one WV) found 324, 6390, and 870 RSTs in 10.9 s, 73.0 s, and 7.9 s, respectively. The 12-pin case had a smaller search space than the 10- and 11-pin cases, so it took only 7.9 s.

V. APPLICATIONS

In this section, we present two applications that can use the ARSMT DB: 1) timing-driven RSMT construction and 2) congestion-aware global routing.

A. Timing-Driven RSMT Construction

The sinks of a net generally have different timing constraints. Thus, generating routing topologies that can minimize

TABLE III
COMPARISON OF THE FLUTE- AND ARSMT DB-BASED ROUTING FOR SCSL AND SCSD OPTIMIZATION. "AVG. DIFF." AND "MAX. DIFF." ARE THE AVERAGE AND THE MAXIMUM OF THE SCSL AND SCSD DIFFERENCES, RESPECTIVELY. THE NUMBERS IN THE PARENTHESES ARE THE MAXIMUM SCSL DIFFERENCES DIVIDED BY THE STANDARD CELL HEIGHT. "# D. NETS (%)" IS THE RATIO BETWEEN THE NUMBER OF NETS WHOSE SCSL OR SCSD DIFFERENCES ARE NONZERO AND THE TOTAL NUMBER OF NETS. "AVG. DIFF. (D.)" IS THE AVERAGE OF THE SCSL (OR SCSD) DIFFERENCES ONLY FOR THE NETS WHOSE SCSL (OR SCSD) DIFFERENCES ARE NONZERO

Benchmark	Source-to-critical-sink length (SCSL)				Source-to-critical-sink delay (SCSD)			
	Avg. Diff.	Max. Diff.	# d. nets (%)	Avg. Diff. (d.)	Avg. Diff. (ps)	Max. Diff. (ps)	# d. nets (%)	Avg. Diff. (d.) (ps)
ibm01	4.6	224 (14)	11.7	39.0	2.3	179	39.8	5.7
ibm02	5.8	480 (30)	11.8	48.6	3.4	725	38.8	8.7
ibm03	4.4	320 (20)	10.1	44.1	4.2	1,104	36.1	11.7
ibm04	5.5	800 (50)	10.5	52.3	3.7	1,532	36.9	10.0
ibm05	5.0	872 (54)	9.9	50.1	5.2	1,068	36.7	14.3
ibm06	5.7	400 (25)	11.2	50.7	2.8	190	40.1	6.9
ibm07	5.6	560 (35)	11.4	49.1	3.8	419	39.5	9.5
ibm08	4.8	240 (15)	10.9	44.1	3.4	476	37.2	9.2
ibm09	4.5	480 (30)	11.2	40.5	3.1	1,000	37.6	8.1
ibm10	4.8	640 (40)	11.1	43.6	3.4	1,068	38.3	8.9
ibm11	4.9	872 (55)	11.1	43.9	3.7	1,204	38.8	9.5
ibm12	5.0	560 (35)	11.7	42.3	3.8	988	40.1	9.6
ibm13	4.9	880 (55)	11.4	43.4	3.6	2,056	38.7	9.3
ibm14	4.8	1,120 (70)	10.3	46.5	4.2	2,659	36.0	11.7
ibm15	5.1	2,640 (165)	11.0	46.2	4.9	8,274	38.4	12.8
ibm16	4.6	1,120 (70)	11.7	39.7	3.9	2,608	40.4	9.8
ibm17	5.1	2,800 (175)	11.5	44.9	5.4	8,837	39.3	13.8
ibm18	4.9	1,040 (65)	11.2	43.7	4.0	1,374	38.2	10.5
Geo. mean	5.0		11.1	45.0	3.7		38.4	9.8

not only the total wirelength of the net but also the SCSL[2] and SCSD is very important in the routing. However, applying only one RSMT to a net will not be able to optimize the SCSL and SCSD of the net effectively. Fig. 10 shows an example for the position sequence (3142). Fig. 10(a) and (b) correspond to POWV (121111) and Fig. 10(c) and (d) correspond to POWV (111121). Assuming all the edges have the same length l, the four topologies have the same wirelength, $7l$. Suppose the source pin is the one located at $(2, 0)$ and the critical sink pin is the one located at $(3, 2)$. In this case, the SCSLs of the topologies in Fig. 10(a)–(d) are $5l$, $3l$, $3l$, and $3l$, respectively. Thus, the first topology gives the longest SCSL, whereas the other three topologies give the shortest SCSL. In addition, suppose each edge, each pin, and each internal node are replaced by a resistor having resistance R, a capacitor having capacitance C_L, and a capacitor having capacitance C_i, respectively. Then, the SCSDs (using the Elmore delay model) of the four topologies in Fig. 10(a)–(d) are $R(10C_L + 10C_i)$, $R(6C_L + 6C_i)$, $R(5C_L + 5C_i)$, and $R(6_L + 8C_i)$, respectively. Thus, the topology in Fig. 10(c) has the shortest SCSL and smallest SCSD.

To show the effectiveness of the ARSMT DB for timing-driven RSMT construction, we compared two global routing approaches for the ISPD 2004 benchmarks [20]. The experiment is as follows. We place the instances using NTUPlace3 [21]. Then, for each global net whose degree is greater than or equal to 4 and less than or equal to 8, we randomly choose two pins, one for the source and the other for the critical sink of the net. The first approach uses only FLUTE to generate one RSMT for the net. The second approach searches the ARSMT DB to find the best RSMT minimizing the SCSL (or SCSD) of the net. Then, we compute the difference between the SCSL (or SCSD) values obtained from the two approaches. Notice that the SCSLs and SCSDs of the RSMTs obtained from the ARSMT DB-based approach are always smaller than or equal to those obtained from the FLUTE-based approach.

[2]The critical sink of a net is the sink having the smallest slack in the net.

Table III compares the FLUTE- and the ARSMT DB-based approaches. The average SCSL difference is computed by

$$\text{Avg. Diff.} = \frac{\sum_{n \in N}(\text{SCSL}_F(n) - \text{SCSL}_A(n))}{|N|} \qquad (2)$$

where N is the set of all the nets whose degree is in the range of $[4, 8]$, $|N|$ the size of N, $\text{SCSL}_F(n)$ the SCSL obtained from FLUTE for net n, and $\text{SCSL}_A(n)$ the minimum SCSL obtained from the ARSMT DB for net n. The average SCSL differences are relatively small (the height of a cell is 16), but it is because the FLUTE- and the ARSMT DB-based approaches produce the same SCSL for most of the nets in N. In fact, the ratio ("# d. nets" in the table) between the number of nets having different SCSLs for the two approaches and the total number of nets in N is approximately 11.1%. Thus, 88.9% of the nets in N obtain the same SCSL from the FLUTE- and ARSMT DB-based approaches. If we calculate the average SCSL differences only for the nets having different SCSLs, the average value increases to 45.0 on average as shown in the table.

However, the maximum SCSL differences between the two approaches are very large as shown in the third column of Table III. The numbers in the parentheses are the maximum SCSL differences divided by the standard cell height (16). As the table shows, the maximum SCSL differences in the relatively small benchmarks, such as ibm01 and ibm02 are between 14 and 50 rows, but those in the large benchmarks, such as ibm15 and ibm17 are greater than 160 rows. Thus, this comparison study shows that we can effectively minimize the SCSL by an exhaustive search of the ARSMT DB. The runtime overhead for the exhaustive search is negligible.

Fig. 11 shows several representative cases that have the maximum SCSL differences between two pins. In Fig. 11(a),

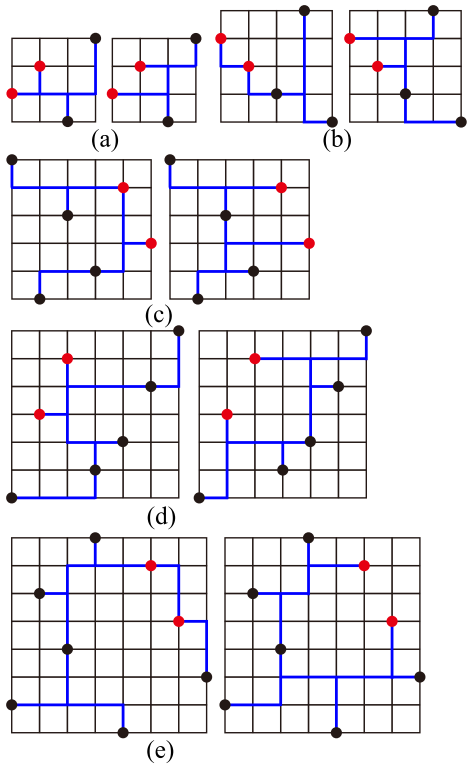

Fig. 11. Two RSMTs having the maximum SCSL difference between two red pins for each position sequence. (a) Four-pin net. (b) Five-pin net. (c) Six-pin net. (d) Seven-pin net. (e) Eight-pin net.

for example, the lengths between the two red pins are 2 and 4 on the left- and right-hand sides, respectively, so the difference is 2. Similarly, the differences in Fig. 11(b)–(e) are 2, 4, 6, and 10, respectively. Thus, the ARSMT DB could help timing-driven global routers find one or multiple RSMTs minimizing the SCSL for a given net.

Table III also compares the FLUTE- and the ARSMT DB-based approaches for SCSD optimization. We used the Elmore delay model to calculate the delay values. The output resistance of a driver is 100 Ω, the unit wire resistance and capacitance are 2 Ω per length and 0.4 fF per length, respectively, and the capacitance of each pin is 5 fF. The average SCSD difference between the two approaches is 3.7 ps, which is almost negligible. In addition, the average SCSD difference for the nets having difference SCSDs is only 9.8%. However, the maximum SCSD differences are very large. For example, the maximum SCSD differences in the ibm15 and ibm17 benchmarks are greater than 8 ns. Overall, the ARSMT DB can help find RSMTs having the minimum SCSD efficiently. Notice that buffer insertion along the RSMTs will reduce the SCSD differences between the two approaches, but the ARSMT DB can still help find minimum-SCSL and minimum-SCSD RSMTs.

B. Congestion-Aware Global Routing

Another representative application of the ARSMT DB is the congestion-aware (routability-driven) global routing. Many global routing algorithms use RSMTs to find high-quality routing topologies [13]–[19]. However, all the RSMT generators

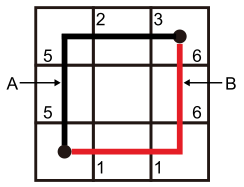

Fig. 12. Example for congestion cost computation.

generate only one RSMT for a net, which could result in serious routing congestion. In this experiment, we perform global routing using the ARSMT DB to reduce the congestion without wirelength overhead.

The experiment is as follows. First, we place instances using NTUPlace3 [21] and build a global routing data structure (bins, edges, etc.). Then, for each global net, we generate a routing topology using FLUTE and proceed to the next net in the FLUTE-based global routing. In the ARSMT DB-based global routing, we obtain all RSMTs for each global net, compute the congestion cost for each RSMT, and obtain the best RSMT that has the lowest cost. We use the amount of overflows for the congestion cost, which is defined as follows:

$$g_i = \sum_{e \in E_i} (d_e - c_e) u[d_e - c_e] \tag{3}$$

where g_i is the congestion cost of net i, E_i the set of all the global routing edges that net i goes through, d_e the number of nets going through e, c_e the maximum capacity of e, and $u[x]$ is the unit-step function. Fig. 12 shows an example for a two-pin net. The maximum capacity of each edge in the grid is 5. The numbers in the figure show the numbers of nets going through the global routing edges. The congestion cost for topology A is $(5+1-5)+(5+1-5)+0+0 = 2$, whereas that for topology B is $(6+1-5)+(6+1-5)+0+0 = 4$, so we choose topology A to minimize the total overflow. For high-degree nets, we use FLUTE to get an RST for both FLUTE- and ARSMT DB-based global routing.

Table IV shows the global routing result for the ISPD 2004, 2005, and 2006 benchmark suites [22], [23]. "# g. nets" is the number of global nets. Since we use the ARSMT DB only for low-degree nets, the number of nets routed using the ARSMT DB ("# r. nets" in the table) is slightly less than the total number of global nets. However, most of the global nets are low-degree nets, so they can show the effectiveness of the ARSMT DB. Notice that they have the same total wirelength because both of them use RSMTs for low-degree nets and the same RSTs for high-degree nets.

The table shows that the FLUTE-based routing has much more overflows than the ARSMT DB-based routing. On average, the ARSMT DB-based routing has 85.39% less overflows than the FLUTE-based routing. In addition, the maximum overflow of the ARSMT DB-based routing is less than that of the FLUTE-based routing for all the benchmarks except adaptec5. On average, the ARSMT DB-based routing has 83.71% lower maximum overflow than the FLUTE-based routing. The average overflow (the total overflow over the number of edges) of the ARSMT DB-based routing is 83.89% less than that of the FLUTE-based routing. Overall, the ARSMT DB-based sequential global routing optimizes the total overflow, the number of overflow edges, the maximum overflow,

TABLE IV
SEQUENTIAL GLOBAL ROUTING RESULT. "# G. NETS" IS THE TOTAL NUMBER OF GLOBAL NETS. "# R. NETS" IS THE NUMBER OF GLOBAL NETS ROUTED USING THE ARSMT DB. "# OV" IS THE TOTAL NUMBER OF OVERFLOWS. "(F)" AND "(O)" ARE THE FLUTE- AND THE ARSMT DB-BASED ROUTING, RESPECTIVELY. "% OV IMP." IS (# OV(O) - # OV(F)) * 100/# OV(F). "M. OV" IS THE MAXIMUM OVERFLOW."A. OV" IS THE AVERAGE OVERFLOW. "RT" IS THE RUNTIME (IN SECONDS) FOR GLOBAL ROUTING. GM: GEOMETRIC MEAN. AM: ARITHMETIC MEAN

Benchmark	# bins	# g. nets	# r. nets	# OV(F)	# OV(O)	% OV imp.	M. OV(F)	M. OV(O)	A. OV(F)	A. OV(O)	RT(F)	RT(O)
ibm01	39×39	9,368	9,010	0	0	-	0	0	0.000	0.000	0.01	0.23
ibm02	44×44	14,255	13,161	27	0	100.00	10	0	0.007	0.000	0.01	0.44
ibm03	49×49	19,211	18,340	88	0	100.00	20	0	0.019	0.000	0.01	0.46
ibm04	55×55	24,307	23,829	1,099	637	42.04	158	89	0.185	0.107	0.01	0.31
ibm05	56×56	21,216	18,511	5,334	552	89.65	72	19	0.866	0.090	0.02	0.57
ibm06	51×51	23,975	22,047	0	0	-	0	0	0.000	0.000	0.02	0.42
ibm07	68×67	35,063	32,988	216	16	92.59	22	11	0.024	0.002	0.02	1.22
ibm08	69×68	36,224	33,340	5,430	2,878	47.00	275	246	0.587	0.311	0.03	0.60
ibm09	74×74	45,130	42,917	529	12	97.73	39	7	0.049	0.001	0.03	1.21
ibm10	94×94	60,836	57,139	2,429	85	96.50	74	16	0.139	0.005	0.04	2.22
ibm11	84×84	63,860	61,695	1,104	25	97.74	58	22	0.079	0.002	0.03	1.43
ibm12	98×97	64,730	60,277	10,423	2,183	79.06	224	147	0.554	0.116	0.05	2.41
ibm13	90×90	76,452	72,241	602	15	97.51	23	7	0.038	0.001	0.05	1.80
ibm14	123×122	122,686	116,594	4,130	815	80.27	146	119	0.139	0.027	0.07	2.74
ibm15	122×122	149,164	140,166	22,214	7,170	67.72	110	73	0.752	0.243	0.10	3.99
ibm16	140×139	156,095	144,118	9,585	2,927	69.46	234	101	0.248	0.076	0.12	5.88
ibm17	153×152	163,017	146,188	15,252	1,611	89.44	79	19	0.330	0.035	0.14	6.52
ibm18	146×x145	164,764	150,822	4,030	1,173	70.89	55	24	0.096	0.028	0.13	4.03
adaptec1	324×324	172,210	156,127	52,103	7,790	85.05	478	215	0.249	0.037	0.22	3.14
adaptec2	424×424	204,971	189,307	53,609	2,956	94.49	179	136	0.149	0.008	0.29	3.40
adaptec3	774×779	355,705	332,621	25,066	1,814	92.76	164	59	0.021	0.002	0.51	5.76
adaptec4	774×779	379,772	360,214	2,516	795	68.40	349	172	0.002	0.001	0.52	4.86
adaptec5	465×468	524,009	486,835	8,939	809	90.95	401	423	0.021	0.002	0.67	9.65
bigblue1	227×227	193,722	177,849	12,456	791	93.65	333	95	0.121	0.008	0.25	4.05
bigblue2	468×471	407,306	387,912	57,528	7,001	87.83	534	519	0.131	0.016	0.57	4.61
bigblue3	555×557	595,622	569,160	82,341	9,701	88.22	540	177	0.133	0.016	0.52	8.19
bigblue4	403×405	1,091,465	1,020,759	111,213	2,572	97.69	890	244	0.342	0.008	1.22	24.12
newblue1	399×399	270,692	255,709	57	0	100.00	28	0	0.000	0.000	0.42	5.00
newblue2	557×463	367,425	347,363	42,735	2,316	94.58	166	89	0.083	0.004	0.45	4.85
newblue3	973×1,256	441,733	425,918	80,431	8,454	89.49	661	321	0.033	0.003	1.28	4.78
newblue4	455×458	511,278	481,906	94,177	9,552	89.86	1,693	583	0.226	0.023	0.52	8.68
newblue5	637×640	855,614	786,465	119,445	5,523	95.38	948	294	0.147	0.007	2.00	13.80
newblue6	463×464	802,737	749,235	196,466	4,913	97.50	634	57	0.458	0.011	1.01	16.72
newblue7	488×490	1,571,798	1,461,358	616,303	10,233	98.34	2,390	1,041	1.291	0.021	1.94	28.54
						GM (85.39)	AM (0.221)	AM (0.036)		Ratio	1.00	21.76

and the average overflow much more effectively than the FLUTE-based sequential global routing.

The runtime of the FLUTE-based routing is less than two seconds for all the cases, whereas the runtime of the ARSMT DB-based routing is less than 30 s. The runtime ratio between the ARSMT DB- and FLUTE-based routing is 21.76 on average. However, the absolute runtime overhead is negligible as the table shows.

VI. DATABASE SIZE REDUCTION

As shown in Section IV, the database size is prohibitively large, especially when the pin count is nine. In this section, therefore, we present several techniques to reduce the size of the ARSMT DB, database loading time, and memory usage.

A. Raw Database

The raw ARSMT DB uses the database structure shown in Fig. 13 to store the POST data. The database consists of position sequence lines, POWV lines, and POST lines. A position sequence line is "P *seq*," where "P" denotes a new position sequence and *seq* is its actual position sequence. All the POWVs belonging to the position sequence appear between the position sequence line and the next position sequence line or the end of file. A POWV line is "V *vec*," where "V" denotes a new POWV and *vec* is its actual POWV. All the POSTs belonging to the POWV appear between the POWV line and the next POWV line or the next position sequence line or the end of file. A POST line is "H*h_edges*V*v_edges*," where *h_edges* and *v_edges* are the coordinates of the horizontal and vertical edges, respectively.

h_edges consists of uppercase and lowercase alphabets, which are their *x*- and *y*-coordinates, respectively. The mapping between an alphabet and its coordinate is that "A" or "a" corresponds to 0, "B" or "b" corresponds to 1, and the other alphabets are mapped into integers in a similar way. For example, "Ab" for a horizontal edge means $e_h(0, 1)$. To reduce the number of characters, horizontal edges having the same *x*-coordinate share the *x*-coordinate. Thus, "Abde" for horizontal edges denote three edges $e_h(0, 1)$, $e_h(0, 3)$, and $e_h(0, 4)$. We process *v_edges* in the same way. The database size shown in the 12th column (raw data table size) in Table I is the size of the database of the raw data.

B. Congruence and Difference Encoding

The input to the database generator is the raw database explained in Fig. 13 and the output is an encoded database. Users can open the database, decode the data, and load it into their applications and use the raw data.

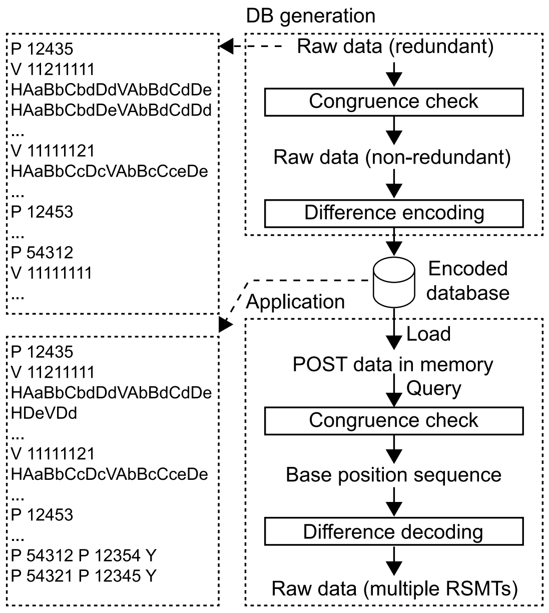

Fig. 13. Raw database structure and the proposed encoding and decoding flow for database size reduction.

Two techniques are used in [2] to reduce the database size of FLUTE. The first technique is to use congruence. For example, position sequences (12345) and (54321) have the same set of POSTs, but the POSTs are symmetric about the *y*-axis. Thus, we need to store POSTs only for one of them (called a base position sequence). For the other position sequence, we can obtain all POSTs using a transformation rule between the position sequences. In Fig. 13, therefore, only one line "P 54321 P 12345 Y" suffices for storing all POSTs for position sequence (54321), where "Y" denotes the transformation rule (reflection over the *y*-axis). A congruence check between two position sequences includes rotation of one of the position sequences and the image of its reflection about the *y*-axis by 90°, 180°, and 270°.

The second technique used in [2] for database size reduction is to store only differences between two POSTs. As explained in [2], many POSTs are similar to each other. Thus, FLUTE stores full data for a few POSTs and difference data for all the other POSTs. We apply a similar technique to the ARSMT DB as follows. Suppose we encode the $(i+1)$th POST by comparing it with the ith POST. The format of the encoded POST is still "H*h_edges*V*v_edges*." If the $(i+1)$th POST has exactly the same horizontal or vertical edges as the ith POST in column k or row k, we skip them. Otherwise, we store the column or row coordinate and the y- or x-coordinates of the edges used in the column or row, respectively. For example, the raw data of the POST in Fig. 14(a) is "HAaBbCbdDdVAbBdCdDe" and that of the POST in Fig. 14(b) is "HAaBbCbdDeVAbBdCdDd." If we store only the differences, however, the latter becomes "HDeVDd."

C. Simulation Results

Table V shows statistics of database encoding and database loading time. We used a 7200-RPM hard disk drive for the simulation. The sizes of the raw and encoded database are the same as the table sizes shown in Table I. As the table shows,

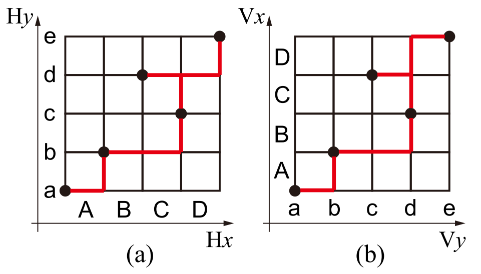

Fig. 14. Difference encoding. (a) HAaBbCbdDdVAbBdCdDe. (b) HDeVDd relative to (a). The coordinate systems in (a) and (b) are for the horizontal and vertical edges, respectively.

TABLE V

STATISTICS OF DATABASE ENCODING AND DATABASE LOADING TIME (FROM A 7200-RPM HDD). "R (C)" AND "R (D)" ARE THE DATABASE SIZE REDUCTION RATIOS BY THE CONGRUENCE CHECK AND THE DIFFERENCE ENCODING, RESPECTIVELY. THE LOADING TIME INCLUDES THE RUNTIME FOR DIFFERENCE DECODING. MEMORY USAGE IS THE SIZE OF THE MEMORY ALLOCATED TO STORE THE POST DATABASE IN MEMORY

# pins	File size		r (C)	r (D)	Loading time	Memory usage
	Raw DB	Encoded DB				
3	0.3 KB	0.1 KB	0.375	-	0.1 ms	0.1 KB
4	5.0 KB	1.2 KB	0.271	0.88	0.2 ms	1.2 KB
5	98 KB	11 KB	0.176	0.63	1.1 ms	11 KB
6	3.5 MB	0.24 MB	0.162	0.42	14 ms	0.25 MB
7	141 MB	6.7 MB	0.139	0.34	0.26 s	7.0 MB
8	7.7 GB	0.3 GB	0.132	0.30	11.2 s	0.3 GB
9	525 GB	19 GB	0.128	0.29	612 s	20 GB

the size of the encoded database is much less than that of the raw database. The database size reduction ratio (# generated POSTs divided by the total # POSTs) by the congruence check is approximately 0.375 and 0.128 for the three- and nine-pin cases, respectively, and varies between the two values for the four- to eight-pin cases as shown in Table V. In the best case, eight position sequences are congruent for which the database size reduction ratio is 0.125. As the pin count increases, more position sequences are congruent to each other, so the database size reduction ratio approaches 0.125. The database size reduction ratio by the difference encoding varies between 0.29 and 0.88. Thus, both the congruence-based and difference-encoding-based database size reduction techniques are very effective.

Table V also shows database loading time and memory usage, which is the "Load" process in Fig. 13. The loading time is almost negligible for all the cases. Although the loading time for the nine-pin case is approximately ten minutes, global routing of large designs generally takes much longer time than that [24], [25]. The maximum memory usage is 20 GB, which is also acceptable for the design of large, complex layouts. Notice that the loaded database is still difference-encoded. Thus, processing a database query requires real-time congruence check and difference decoding. However, the actual runtime of the congruence check and difference decoding is negligible .

VII. CONCLUSION

In this paper, we proposed an efficient algorithm to create a database for an efficient RSMT construction. The ARSMT DB can return all the RSMTs for a given set of pins in no time, so its application is numerous. We showed two representative applications: 1) SCSL and 2) SCSD minimization and congestion-aware global routing. Use of the ARSMT DB helps minimize the SCSL and SCSD for a given net and also minimize routing congestion during global routing effectively. Since the size of the ARSMT DB is large, we proposed techniques to reduce the size of the ARSMT DB. We believe that the ARSMT DB can help all the electronic design automation software improve the quality of layouts and reduce the runtime for placement, routing, and various optimization.

References

[1] GeoSteiner. *Software for Computing Steiner Trees*. Accessed: Jan. 2017. [Online]. Available: http://www.geosteiner.com

[2] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 1, pp. 70–83, Jan. 2008.

[3] M. Hanan, "On Steiner's problem with rectilinear distance," *SIAM J. Appl. Math.*, vol. 14, no. 2, pp. 255–265, Mar. 1966.

[4] J. L. Ganley and J. P. Cohoon, "Optimal rectilinear Steiner minimal trees in $O(n^2 2.62^n)$ time," in *Proc. Can. Conf. Comput. Geometry*, 1994, pp. 308–313.

[5] J. L. Ganley and J. P. Cohoon, "A faster dynamic programming algorithm for exact rectilinear Steiner minimal trees," in *Proc. Great Lakes Symp. VLSI*, 1994, pp. 238–241.

[6] J. L. Ganley and J. P. Cohoon, "Improved computation of optimal rectilinear Steiner minimal trees," *Int. J. Comput. Geometry Appl.*, vol. 7, no. 5, pp. 457–472, Oct. 1997.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: Freeman, 1979.

[8] J. Griffith, G. Robins, J. S. Salowe, and T. Zhang, "Closing the gap: Near-optimal Steiner trees in polynomial time," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 11, pp. 1351–1365, Nov. 1994.

[9] M. Borah, R. M. Owens, and M. J. Irwin, "An edge-based heuristic for Steiner routing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 12, pp. 1563–1568, Dec. 1994.

[10] I. I. Mandoiu, V. V. Vazirani, and J. L. Ganley, "A new heuristic for rectilinear Steiner trees," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 10, pp. 1129–1139, Oct. 2000.

[11] A. B. Kahng, I. I. Mandoiu, and A. Z. Zelikovsky, "Highly scalable algorithms for rectilinear and octilinear Steiner trees," in *Proc. Asia South Pac. Design Autom. Conf.*, Jan. 2003, pp. 827–833.

[12] H. Zhou, "Efficient Steiner tree construction based on spanning graphs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 5, pp. 704–710, May 2004.

[13] M. Cho and D. Z. Pan, "BoxRouter: A new global router based on box expansion and progressive ILP," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 12, pp. 2130–2143, Dec. 2007.

[14] Z. Cao, T. Jing, J. Xiong, Y. Hu, L. He, and X. Hong, "DpRouter: A fast and accurate dynamic-pattern-based global routing algorithm," in *Proc. Asia South Pac. Design Autom. Conf.*, 2007, pp. 256–261.

[15] M. M. Ozdal and M. D. F. Wong, "Archer: A history-driven global routing algorithm," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, 2007, pp. 488–495.

[16] M. D. Moffitt, "MaizeRouter: Engineering an effective global router," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 11, pp. 2017–2026, Nov. 2008.

[17] Y. Xu, Y. Zhang, and C. Chu, "FastRoute 4.0: Global router with efficient via minimization," in *Proc. Asia South Pac. Design Autom. Conf.*, 2009, pp. 576–581.

[18] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "GRIP: Scalable 3D global routing using integer programming," in *Proc. ACM Design Autom. Conf.*, 2009, pp. 320–325.

[19] Y.-J. Chang, Y.-T. Lee, J.-R. Giao, P.-C. Wu, and T.-C. Wang, "NTHU-Route 2.0: A robust global router for modern designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 12, pp. 1931–1944, Dec. 2010.

[20] N. Viswanathan and C. C.-N. Chu, "FastPlace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model," in *Proc. Int. Symp. Phys. Design*, Mar. 2004, pp. 26–33.

[21] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1228–1240, Jul. 2008.

[22] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter, and M. Yildiz, "The ISPD2005 placement contest and benchmark suite," in *Proc. Int. Symp. Phys. Design*, 2005, pp. 216–220.

[23] G.-J. Nam, "The ISPD 2006 placement contest: Benchmark suite and results," in *Proc. Int. Symp. Phys. Design*, 2006, p. 167.

[24] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "GRIP: Global routing via integer programming," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 1, pp. 72–84, Jan. 2011.

[25] K.-R. Dai, W.-H. Liu, and Y.-L. Li, "NCTU-GR: Efficient simulated evolution-based rerouting and congestion-relaxed layer assignment on 3-D global routing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 3, pp. 459–472, Mar. 2012.

Sheng-En David Lin (S'16) received the B.S. degree in electrical engineering from Washington State University, Pullman, WA, USA, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electrical Engineering and Computer Science.

His current research interests include modeling for very large scale integration (VLSI) circuits and systems and algorithms for VLSI CAD automation with current focus on designing of monolithic 3-D ICs.

Dae Hyun Kim (S'08–M'12) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2002, and the M.S. and Ph.D. degrees in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, GA, USA, in 2007 and 2012, respectively.

He is an Assistant Professor with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA. He researched on physical layout optimization with Cadence Design Systems, Inc., San Jose, CA, USA, from 2012 to 2014. His current research interests include electronic design automation and computer-aided design for very large scale integration (VLSI), high-performance and/or low-power VLSI and computer systems, and 3-D integrated circuits and systems.

Dr. Kim was a recipient of the Defense Advanced Research Projects Agency Young Faculty Award in 2016.