A Scalable and Accurate Rectilinear Steiner Minimal Tree Algorithm

Yiu-Chung Wong
Rio Design Automation
Santa Clara, CA 95054
Email: ycwong@rio-da.com

Chris Chu
Iowa State University
Ames, IA 50011
Email: cnchu@iastate.edu

Abstract

FLUTE [1, 2] is a very fast and accurate rectilinear Steiner minimal tree (RSMT)[1] algorithm particularly suitable for VLSI applications. It is optimal for nets up to degree 9 and is still very accurate for nets up to degree 30. However, for higher degree nets, the original FLUTE algorithm is not effective.

In this paper, we present an improvement of FLUTE which is more effective in handling nets with degree tens or more. The main idea is to partition a net according to a spanning tree into small subnets that can be handled effectively by the original FLUTE algorithm. Several novel techniques are proposed to partition a net into small subnets and to merge the Steiner trees for the subnets together. Some improvements of the original FLUTE algorithm, and a scheme to allow users to control the tradeoff between accuracy and runtime are also presented.

We show experimentally that the resulting algorithm FLUTE-3.0 achieves a much better accuracy-runtime tradeoff than the original FLUTE algorithm for degree 30 or more. It produces better quality of result than the well-known near-optimal BI1S algorithm [3] in a runtime shorter than the highly scalable BGA algorithm [4]. FLUTE-3.0 is also highly scalable. It can route a 3-million-pin net in about 25 minutes.

1 Introduction

Rectilinear Steiner minimal tree (RSMT) construction is a fundamental problem that has many applications in VLSI design. In early design stages like physical synthesis, floorplanning, interconnect planning and placement, it can be used to estimate wireload, routing congestion and interconnect delay. In global and detailed routing stages, it is used to generate the routing topology of each net. Many previous works have addressed this problem. For example, the GeoSteiner package is currently the fastest optimal RSMT implementation [5,6]. Griffith et al. [3] (Batched Iterated 1-Steiner (BI1S) heuristic) and Mandoiu et al. [7] are two well-known near-optimal algorithms. These optimal and near-optimal algorithms have high runtime complexities and are impractical for nets with more than a few hundred pins.

Recently, RSMT algorithms capable of routing nets with more than a few hundreds pins are increasingly important. Huge-degree nets like scan enable, which may have up to tens of thousands of pins in a large module, are becoming more common in modern designs due to the increased emphasis on design for test [4]. In addition, to model non-zero pin dimensions (e.g., nets with pre-routes) by representing each pin with a set of electrically equivalent points, very large RSMT instances with as many as 100,000 points have been created [8]. The contribution of this paper is a highly scalable RSMT algorithm that produces near-optimal solutions to both low-degree and high-degree nets.

To handle high-degree nets, many attempts have been made to design RSMT algorithms with low runtime complexity. Borah et al. [9] presented an $O(n^2)$ time algorithm in which a spanning tree is iteratively improved by connecting a point to a nearby edge and deleting the longest edge on the created cycle. An $O(n \log n)$ time but very complicated alternative implementation was also proposed. Zhou [10] used spanning graph [8] to help both generating the initial spanning tree and finding good candidates for the edge substitution idea in [9]. The resulting algorithm runs in $O(n \log n)$ time, and produces better solution in slightly less runtime than the one in [9]. Kahng et al. [4] gave a practical $O(n \log^2 n)$ heuristic called BGA based on a batched version of the greedy triple contraction algorithm. This algorithm matches the solution quality of [9] and [10] but requires a much shorter runtime in practice.

Recently, a very fast and accurate RSMT algorithm called FLUTE is presented [1,2]. In FLUTE, low-degree nets (up to degree 9 in current implementation) are handled optimally and efficiently by a lookup table approach. All other nets are recursively broken down until lookup table can be used. FLUTE is extremely fast and accurate for low-degree nets. So it is particularly suitable for traditional VLSI applications in which most nets have a degree 30 or less. FLUTE is also very efficient for high-degree nets as it has a runtime complexity of $O(n \log n)$ with a small hidden constant. However, the accuracy of FLUTE worsens rather quickly for higher net degrees. The reason is that the simple net break techniques described in [1, 2], while computationally very efficient, introduce significant error to the solution every time the net is broken.

In this paper, we investigate the problems of how to partition a net into small subnets and how to merge the Steiner trees for the subnets together so that the error introduced will be minimized. Several novel heuristics for partitioning and merging will be introduced. In addition, some improvements of the original FLUTE algorithm, and a scheme to allow users to control the tradeoff between accuracy and runtime will also be presented. The resulting algorithm FLUTE-3.0 has a runtime complexity of $O(n \log^2 n)$, and is extremely fast and accurate in practice. Comparing with the original FLUTE algorithm [2], the accuracy-runtime tradeoff achieved by FLUTE-3.0 is better for degree 10 or more, and is significantly better for degree 30 or more. It produces better quality of result than the well-known near-optimal BI1S algorithm [3] in a runtime shorter than the highly scalable BGA algorithm [4]. FLUTE-3.0 is also highly scalable. For instance, it can route a 100000-pin net in less than 1 minute and a 1-million-pin net in less than 10 minutes.

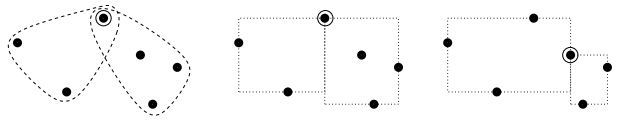
The remainder of the paper is organized as follows. In Section 2, a brief review and several improvements of the original FLUTE algorithm are provided. In Section 3, a tree-based net breaking technique for high degree nets is discussed. In Section 4, experimental results are shown. The paper is concluded in Section 5.

[1] A rectilinear Steiner minimal tree is a tree with minimum total edge length in Manhattan distance to connect a given set of nodes possibly through some extra (i.e., Steiner) nodes.

2 Original FLUTE and Improvements

FLUTE is a lookup table based technique originally designed for wirelength estimation [1]. It is extended to RSMT construction and improved in [2]. For a given degree n, a table of very compact representations called potentially optimal wirelength vectors (POWV) is used to represent all the possible routing topologies of the nets of the same degree. To find the RSMT of a net, it is only necessary to search through a subset of these POWVs and evaluate the corresponding topologies, according to the relative positions of the pins. For low degree, the size of the table is very small and it is extremely economical to evaluate a net. For example, it takes only 10.81MB to store the table for all nets up to degree 9 and there is an average of only 30 POWVs to search for with degree 9.

For high-degree nets, however, both the table size and the number of operations to evaluate a net will be impractically large. So in [1, 2], a lookup table is constructed for nets with degree up to a user-defined parameter D ($D = 9$ in the latest implementation). Nets with higher degree are recursively divided into sub-nets by a net breaking technique until the lookup table can be used. The net breaking technique first checks if the net can be broken optimally according to a condition derived in [2]. If the condition is not satisfied, it tries to divide the net at all pin positions and in both directions (i.e., horizontal and vertical). Then the best solution is picked. For a given pin and breaking direction, pins with a smaller coordinate than the given pin in the breaking direction form one sub-net and other pins form another sub-net. The given pin is also included into both sub-nets as shown in Figure 1(a). However, if recursive calls are really made to evaluate each of the possible pins and directions, the runtime will be very significant. So a score is computed for each possible way of breaking. Recursive calls are made only for the few ways with the highest scores.



(a) (b) (c)

Figure 1: Illustration of the original net breaking technique.

Without loss of generality, consider breaking the net according to y-coordinate at pin r. In [1], the score $S(r)$ is defined as:

$$S(r) \quad = \quad -S_0(r)$$

where $S_0(r)$ is the total half-perimeter wirelength (HPWL) of the two subnets. Note that negative of $S_0(r)$ is used as smaller total HPWL is more desirable. For example, according to the HPWL score, Figure 1(c) is a better selection than Figure 1(b).

In [2], the score is computed as follows. For an n-pin net, let x_i be the x-coordinate of some vertical Hanan grid line such that $x_1 \leq x_2 \leq \cdots \leq x_n$, and let y_j be the y-coordinate of some horizontal Hanan grid line such that $y_1 \leq y_2 \leq \cdots \leq y_n$. Assume the pins are indexed in ascending order of y-coordinate. Let s_i be the rank of pin i if all pins are sorted in ascending order of x-coordinate. The score for breaking at pin r is a weighted sum of three components:

$$S(r) \quad = \quad S_1(r) - 0.3 \times S_2(r) - 0.3 \times S_3(r)$$

where

$$S_1(r) \quad = \quad y_{r+1} - y_{r-1}$$

$$S_2(r) \quad = \quad \begin{cases} 2 \times (x_3 - x_2) & \text{if } s_r = 1 \text{ or } 2 \\ x_{s_r+1} - x_{s_r-1} & \text{if } 3 \leq s_r \leq n-2 \\ 2 \times (x_{n-1} - x_{n-2}) & \text{if } s_r = n-1 \text{ or } n \end{cases}$$

$$S_3(r) \quad = \quad \left| s_r - \frac{n+1}{2} \right| \times d_x + \left| r - \frac{n+1}{2} \right| \times d_y$$

$$d_x = \frac{x_{n-1} - x_2}{n-3}, \; d_y = \frac{y_{n-1} - y_2}{n-3}.$$

The accuracy of FLUTE can be controlled by changing the number of ways of breaking each net. It is observed in [2] that a better tradeoff between accuracy and runtime can be obtained if sub-nets in lower recursive level are handled with less accuracy. Based on this observation, an accuracy control scheme is presented in [2] to allow users to select the tradeoff between accuracy and runtime. A user-defined accuracy parameter A is introduced. The original net is handled with accuracy A. That means A different ways of breaking are tried. Then for each level of recursive call, the accuracy is reduced by 1 unless it is already 1. The default value of A is set to 3.

In this paper, we propose the following two improvements of the original FLUTE algorithm. First, we modify the score to:

$$S(r) \quad = \quad S_1(r) - 0.3 \times S_2(r) - \frac{7.5}{n+10} \times \left(S_3(r) + 12 \times \frac{S_0(r)}{n-3} \right).$$

The weights for the components $S_3(r)$ and $S_0(r)$ are experimentally determined. This modification increases the accuracy of net breaking significantly but makes the algorithm marginally slower (as the computation of HPWL takes time). Overall, the accuracy-runtime tradeoff is much improved. Second, in the accuracy control scheme, for each level of recursive call, instead of reducing the accuracy parameter by 1, the parameter is divided by 2. This modification decreases the accuracy but dramatically reduces the runtime by making fewer recursive calls at the lower levels. Overall, the accuracy-runtime tradeoff is also much improved. These two modifications together produce significantly improved accuracy-runtime tradeoff over previous versions of FLUTE in [1] and [2].

3 Tree-based Net Breaking

Since FLUTE in [1, 2] recursively cuts a net into subnets using either a horizontal line or a vertical line only, it may miss some better way of cutting. This is usually not a problem for small nets, when the solution space is not large, as FLUTE compensates this drawback by trying many different break points under the control of the parameter A. The impact on the quality, however, begins to show up as the net size grows, say, beyond a few tens. To overcome this problem, we use a divide-and-conquer approach based on a spanning tree breaking idea.

Let P be the input pin set. If $|P|$ is small enough, we simply apply FLUTE directly on it. Otherwise the following recursive heuristics is used. Suppose we start with a base spanning tree T_{base} of P that we know is a good approximation to the RSMT we are searching for. Obviously a minimum spanning tree of P will be a reasonable choice, but we shall see in Section 3.3 that there are other choices as well. T_{base} is broken into a number of subtrees. Each of these subtrees is optimized recursively. The optimized subtrees are then merged together to form the resulting Steiner tree T.

It is clear that the merging step plays an important role in achieving a good quality of the resulting Steiner tree. We have considered two variants on implementing the above divide-and-conquer scheme with different merging strategies. The first one puts the recursive call inside a loop. The idea consists of simply blending the subtrees at their touching points to form T. We then make use of the information that T provides to try to improve T_{base}. The new T_{base} is used to start another round of the divide-and-conquer procedure. The intuition is that this new T_{base} is likely to be a better RSMT than the original one, and it may in turn lead to a better new T. The loop finishes either when a certain number, R, of rounds have been tried, or when there is no further improvement on T. We call this multi-round strategy FLUTE-MR.

The second strategy, which we call FLUTE-AM, uses a more aggressive merging step to obtain the final Steiner tree T, but it eliminates the use of the loop. The idea is to find potential improvement

during the merge. Exactly two subtrees are created and individually optimized at each recursive step. At merging, a heuristics similar to the edge-based heuristics [9] is applied to look for local improvement arising from connecting a node from one subtree to an edge on the other subtree.

FLUTE-AM is usually much faster than FLUTE-MR. On the other hand, FLUTE-MR in general gives better result when enough number of rounds is allowed. We therefore leverage the advantages of these two heuristics by applying them in different situations: FLUTE-MR is used to handle medium-degree nets and FLUTE-AM is applied to high-degree nets. Note that low-degree nets are handled by the original FLUTE with the improvements described in Section 2. We rename it as FLUTE-LD here.

Figures 2, 3 and 4 show the pseudo code of the above description. We call the whole algorithm FLUTE-3.0. D_1 and D_2 are pre-defined parameters that determine when each heuristics should be used. Details of the major steps in the algorithm are explained below.

```
Algorithm FLUTE-3.0(P)
Input: Set of pins P.
Output: A rectilinear Steiner tree of P.
Begin
  if |P| ≤ D1
    return FLUTE-LD(P)
  else if |P| ≤ D2
    return FLUTE-MR(P)
  else
    return FLUTE-AM(P)
End
```

Figure 2: Algorithm of FLUTE-3.0.

```
Algorithm FLUTE-MR(P)
Input: Set of pins P.
Output: A rectilinear Steiner tree of P.
Begin
  T_base ← MST(P)
  T_best ← T_base
  for each round ∈ (1, ..., R) {
    Let S = PARTITION-TREE(T_base, τ) be the set of subtrees
    for each T_i ∈ S
      T_i ← FLUTE-LD(pins of T_i)
    pop the first element T from S
    while S is non-empty {
      pop an element T' from S that is touching T
      T ← SIMPLE-MERGE(T, T')
    }
    if T is no better than T_best then
      return T_best
    else
      T_best ← T
    T_base ← IMPROVE-SPANNING-TREE(T)
  }
  return T_best
End
```

Figure 3: A multi-round tree-based heuristics for FLUTE.

3.1 Partitioning the Spanning Tree

In FLUTE-MR, we limit the size of each subtree by a threshold τ. The PARTITION-TREE function takes a spanning tree T and the threshold τ as input and proceeds as follows. First it identifies the least-expensive edge of T (according to the effective distance defined in Section 3.3), and chooses one of its endpoints as the root.

```
Algorithm FLUTE-AM(P)
Input: Set of pins P.
Output: A rectilinear Steiner tree of P.
Begin
  T_base ← MST(P)
  (T1, T2) ← BIPARTITION-TREE(T_base)
  T1' ← FLUTE-3.0(pins of T1)
  T2' ← FLUTE-3.0(pins of T2)
  T ← AGGRESSIVE-MERGE(T1', T2')
  return T
End
```

Figure 4: A tree-based approach with aggressive merging.

Then it imposes a child-parent relationship along each edge in the obvious way (i.e., the node being closer to the root is the parent). Now denote the subtree rooted at a node u by T_u. The partitioning is done by repeatedly finding a node u such that $|T_u| \leq \tau$ but $|T_{parent(u)}| > \tau$, and then cutting the tree at u. Node u is replicated before the subtree T_u is cut out. By doing so we make sure that adjacent subtrees that are cut out share a common node.

In FLUTE-AM, we always partition the base spanning tree into two subtrees that are roughly equal in size. BIPARTITION-TREE is similar to PARTITION-TREE but it implicitly assumes that the size threshold is $|P|/2$ and returns exactly two subtrees. Hence it is possible that one of them could be slightly larger than $|P|/2$.

3.2 A Simple Merging Step

First let us define some notations. For any pin node u of a Steiner tree T, let $\mathcal{N}(u, T)$ denote the set of pin nodes of T that are reachable from u without going through any other pin nodes. And let $\mathcal{FSC}(u, T)$ be the subtree of T induced by $\mathcal{N}(u, T)$ (i.e., it also includes all the associated Steiner nodes connecting $\mathcal{N}(u, T)$). Note that when u is also a leaf node of T then this is a *full Steiner component* of T [4].

Given two adjacent Steiner trees T_1 and T_2 that share a common node c, the SIMPLE-MERGE step in FLUTE-MR joins them together at c in a straight-forward manner to form Steiner tree T. It then performs a minimal-effort improvement on T in the neighborhood of c by the following operation: the subset $\mathcal{FSC}(c, T_1) \cup \mathcal{FSC}(c, T_2)$ is replaced by FLUTE-LD($\mathcal{N}(c, T_1) \cup \mathcal{N}(c, T_2)$). Since this neighborhood is usually very small in size, FLUTE-LD is sufficient to handle it and the cost of the call is very low. At the same time, we observe empirically that there is usually slight improvement on the quality of T.

3.3 Improving the Base Spanning Tree

The partitioning of the base spanning tree T_{base} in FLUTE-MR can be viewed as providing a way of breaking the net into groups of subnets. The Steiner tree T obtained after the simple merging step provides some insight on how the pin nodes could be grouped in a better way. The rationale is that there are probably some groupings of pin nodes that are obvious in T but not in T_{base}. The basic idea is that the closeness of two edges in T may help to shorten the *effective distance* between their end-points, as a potential Steiner point may be added to bridge one of the end-points to the other edge. We would therefore like to include such consideration when we create the new T_{base} in the next round.

The algorithm in Figure 5 shows how it is done. Let $dist_{eff}$ be the effective distance matrix. In the first round of FLUTE-MR, we do not yet have any information of T. So $dist_{eff}$ is the original Manhattan distance matrix. Subsequently $dist_{eff}$ is updated in the function IMPROVE-SPANNING-TREE as follows. It loops through each edge of the input Steiner tree T in a scan line fashion. For each edge $e = (u, v)$ it checks for any other edge (p, q) that is disjoint from e and falls within a neighborhood of e. For each such edge pair, it finds the closest node pair $(x, y) \in \{u, v\} \times \{p, q\}$ such that

both x and y are pin nodes. If such pair exists, then their effective distance is updated as

$$dist_{eff}(x, y) = \text{shortest-distance}(bbox(u, v), bbox(p, q)) \quad (1)$$

where $bbox()$ is the bounding box function. After all scanning is done, a new T_{base} is calculated by computing the minimum spanning tree of the pin set with respect to the updated matrix $dist_{eff}$.

Algorithm IMPROVE-SPANNING-TREE(T)
Input: A Steiner tree.
Output: A spanning tree T_{base} of the pin nodes of T.
Begin
  $S \leftarrow$ edges of T in increasing x order
  for each $e = (u, v)$ in S {
    if both u and v are Steiner nodes, then continue;
    for each $e' = (p, q)$ in $neighborhood(e)$ {
      if e and e' are touching, then continue;
      if both p and q are Steiner nodes, then continue;
      find shortest-dist pair $(x, y) \in \{u, v\} \times \{p, q\}$
        s.t. both x and y are pin nodes
      update $dist_{eff}(x, y)$ according to eq. (1)
    }
  }
  $T_{base} \leftarrow$ MST of pin nodes of T w.r.t. $dist_{eff}$
  return T_{base}
End

Figure 5: The algorithm to improve the base spanning tree.

Algorithm AGGRESSIVE-MERGE(T_1, T_2)
Input: Two adjacent Steiner trees.
Output: A combined Steiner tree.
Begin
  Let c be the common node of T_1, T_2.
  $T \leftarrow T_1 \cup T_2$
  Make T rooted at c
  repeat {
    $S \leftarrow$ edges of T sorted in horizontal order
    for each $(p, q) \in S$ {
      for each pin node u in $neighborhood(p, q)$ {
        $d \leftarrow distance(bbox(p, q), u)$
        $e \leftarrow$ longest-edge(p, u)
        if $(d < length(e))$ then {
          remove e from T
          create Steiner edge from u to (p, q)
          if new Steiner node x is created in last step {
            insert (p, x) into S
            $p \leftarrow x$
          }
        }
      }
    }
    repeat the above loop with
      $S \leftarrow$ edges of T sorted in vertical order
  } until no more improvement
  return T
End

Figure 6: The aggressive-merge algorithm.

3.4 An Aggressive Merging Step

The heuristics in Section 3.3 allows FLUTE-MR to repeatedly improve the quality of T in a global sense, as all the information about the potential additional Steiner nodes are collected first before T is re-generated. However, the advantage of such global improvement is obtained at the expense of runtime.

An alternative is to update T locally whenever a good Steiner node is found during the search, but to avoid the recursive call in the multiple rounds. This forms the basis of the AGGRESSIVE-MERGE heuristics for FLUTE-AM. Figure 6 describes the algorithm. The inputs are two Steiner subtrees T_1 and T_2 that share a common node c. It first joins the trees at c, and makes the combined tree T rooted at c (essentially imposing a parent-child relationship on the nodes of each edge). The edges are then sorted and examined one by one in a scan line manner similar to the way as in Section 3.3. However, only pin nodes (instead of tree edges) in the neighborhood of an edge are collected this time. For an edge (p, q) (where p is the parent of q in T) and a pin node u in its neighborhood, the Manhattan distance between $bbox(p, q)$ and u is calculated. It is compared to the length of the current longest edge e between (p, q) and u in T (if e happens to be (p, q) then the second longest one, if it exists, replaces e in the subsequent discussion). If the former is smaller, T is updated by creating a Steiner edge from u to (p, q) and breaking the cycle at e.

Obviously the order of the edges being scanned has a direct impact on the quality of the final T. In order to optimize the chances of improvement, AGGRESSIVE-MERGE does it by first scanning the edges horizontally and then scanning the updated edge set again vertically. The whole process is then repeated until no further improvement can be found. Since the inputs T_1 and T_2 come from a recursive call in FLUTE-AM, their qualities are already very good and so we expect that the number of repetition for the process to converge is very small. In our empirical finding, it usually takes no more than three iterations to exit.

The most expensive step in AGGRESSIVE-MERGE is identifying the longest edge between (p, q) and u. We have implemented a simple approach as follows. First the least common ancestor lca of p and u is computed. This can be done by following the respective paths from p and u to the root c of T until a common ancestor is found. There are three cases: (i) lca is neither p nor u, (ii) lca is p, or (iii) lca is u. In any case, the rest of the computation reduces to one of finding the longest edge from a node to an ancestor, and it is done by tracing the corresponding path.

With two major changes, there is also a batched version that helps speeding up the heuristics. First, only those edge-pin pair (p, q) and u that come from opposite input trees are considered. That is, if (p, q) belongs to T_1, then u is searched from T_2, and vice versa. Second, instead of updating T immediately during the scan loop, the potential edge-pin pair candidates are collected but not processed yet. After the loop, this set of candidates is re-examined and T is updated accordingly. In this way, the longest-edge calculation within the scan loop can be done very efficiently as follows. Before the scan loop, we calculate for each node the longest edge along the path from it to the node c in the corresponding tree. This is achieved with linear complexity through a depth-first traversal of T_1 and T_2 respectively, starting from the common node c. This information allows the longest edge between edge (p, q) and node u to be computed in constant time during the scanning.

The set of candidates collected during the scan loop is relatively small in general. This strategy effectively helps cutting the run time significantly by avoiding a lot of dynamic longest-edge computation. The quality is compromised slightly. Therefore, we only use it for extremely high degree nets. Another threshold, D_3, is added to control it: the heuristics is switched from the non-batched version to the batched one when $|T_1| + |T_2| \geq D_3$.

3.5 Accuracy Control Scheme

As in the original FLUTE, we introduce a user-defined accuracy parameter A. A can take any integral value ≥ 1 and is practical up to about 20. Based on A, other parameters of the algorithm are experimentally determined to control the tradeoff between accuracy and runtime.

The threshold τ on the partition size in FLUTE-MR is:

$$\tau \quad = \quad 8 + 1.3A$$

To control the accuracy of FLUTE-MR, the number of rounds R and the accuracy parameter $\widehat{A}$ used when calling FLUTE-LD are defined below:

$$R \quad = \quad \begin{cases} 1 & \text{if } A \leq 6 \\ A - 5 & \text{if } A > 6 \end{cases}$$

$$\widehat{A} \quad = \quad \begin{cases} A & \text{if } A \leq 6 \\ 6 + 2 \times \lfloor \frac{A-5}{4} \rfloor & \text{if } A > 6 \end{cases}$$

Note that $\widehat{A}$ is always set to an even number when $A > 6$. Because the accuracy parameter in the accuracy control scheme of FLUTE-LD is always divided by 2, we notice that an even value for $\widehat{A}$ produces marginally better results. In fact, powers of 2 are even better but the difference is insignificant.

The accuracy of FLUTE-AM is controlled by setting the accuracy parameter $\widehat{A}$ of the recursive calls to FLUTE-3.0 as A. We could control the

accuracy of AGGRESSIVE-MERGE by adjusting the stopping criteria of its main loop, but we have not incorporated this idea in the current implementation. As a result, we currently are not very effective in controlling the accuracy of FLUTE-AM.

Based on the tradeoff characteristics of FLUTE-LD, FLUTE-MR, and FLUTE-AM over nets of various degree, the thresholds determining which heuristics to use are set as below:

$$D_1 = 25 + 120/A^2$$

$$D_2 = \begin{cases} 500 & \text{if } A \leq 6 \\ 75 + 5A & \text{if } A > 6 \end{cases}$$

$$D_3 = \begin{cases} 1000 & \text{if } degree \leq 10000 \\ 10000 & \text{otherwise} \end{cases}$$

Note that D_2 is set in such a way that the use of FLUTE-AM is prohibited when $A \leq 6$ unless the degree is high.

The time complexity of FLUTE-3.0 is $O(n \log^2 n)$. The analysis is omitted due to space limitation.

4 Experimental Results

We have implemented FLUTE-3.0 in C. We perform two sets of experiments. The first set focuses on nets with low and medium degree, and is run on a 1.8GHz Pentium 4 laptop with 256MB memory. We compare FLUTE-3.0 with four other algorithms: an efficient $O(n^2)$ implementation of Prim's algorithm (RMST) [11], the near-optimal Batched Iterated 1-Steiner (BI1S) heuristic [3], the $O(n \log^2 n)$ near-optimal batched greedy heuristics (BGA) reported in [4], and FLUTE-2.0 with accuracy $A = 3$ and $A = 6$ [2]. We also run the exact RSMT software GeoSteiner 3.1 [6] to generate the optimal solutions as the reference. 1000 randomly generated nets is used for each degree. BI1S crashes for degrees 400 and 500.

Tables 1 and 2 show respectively the average wirelength error and runtime comparisons. It is clear from the tables that FLUTE-3.0 has comparable quality with the well-known near-optimal BI1S heuristics. In particular, with accuracy $A \geq 10$, it gives the best quality of result among all the algorithms being compared.[2] At the same time, it is much faster than BI1S (by at least one to two orders of magnitude). It is also faster than the highly scalable BGA algorithm whenever $A \leq 12$. We would also like to remark that whenever $A \geq 12$, FLUTE-3.0 consistently produces a result that is within 0.5% of the optimial solution. The tables also show that although FLUTE-2.0 can be extremely fast (with a small A), it fails to generate accurate solutions for medium-degree nets even with a large A value (i.e., a high runtime). FLUTE-3.0 can generate far more accurate solutions in less runtime.

Figures 7 and 8 show the tradeoff between accuracy and runtime for nets with degree 30 and 300, respectively. As expected, these figures confirm that the quality of FLUTE-3.0 improves consistently when A increases. On the other hand, they show that the runtime increases only moderately, and is within reasonable range even when $A = 18$. The figures also show that FLUTE-3.0 provides a much better accuracy-runtime tradeoff than FLUTE-2.0 and BGA. Note that in Figure 8, FLUTE-2.0 is not included because its wirelength errors are far too high to be plotted in the same graph (wirelength error $> 4.4\%$ for the corresponding runtime range). Figure 8 shows a large jump in accuracy and runtime between $A = 6$ and $A = 7$ for degree 300. The jump is caused by the application of FLUTE-AM when $A \geq 7$ and degree $> D2 = 75 + 5A$. Such a jump does not exist when degree $\leq D2$ as illustrated in Figure 7.

The second set of experiment illustrates the scalability of FLUTE-3.0. It is run on a 3GHz Xeon desktop with 4GB memory. Figures 9 and 10 compare respectively the runtime and quality of FLUTE-3.0 with those of BGA for very high degree nets. Note that BGA runs out of memory for nets with over 1 million pins. Since GeoSteiner cannot handle such large nets, we measure the quality of result by the percentage improvement over the rectilinear minimum spanning tree of the same set of pins. For each degree, 10 randomly generated nets are routed and the average runtime and quality is reported. It can be seen that both FLUTE-3.0 and BGA are highly scalable. However, the runtime of FLUTE-3.0 is not only smaller but also increases in a much slower rate than BGA. In particular, when $A = 12$, FLUTE-3.0 is 3.7 times faster than BGA and achieves a better quality in routing one million pins. Moreover, BGA refuses to handle more than one million pins due to insufficient memory, while FLUTE-3.0 can handle 3-million-pin nets in our experiment.

[2] In fact, FLUTE-3.0 gives the best quality of result among all the algorithms being compared when $A \geq 9$.

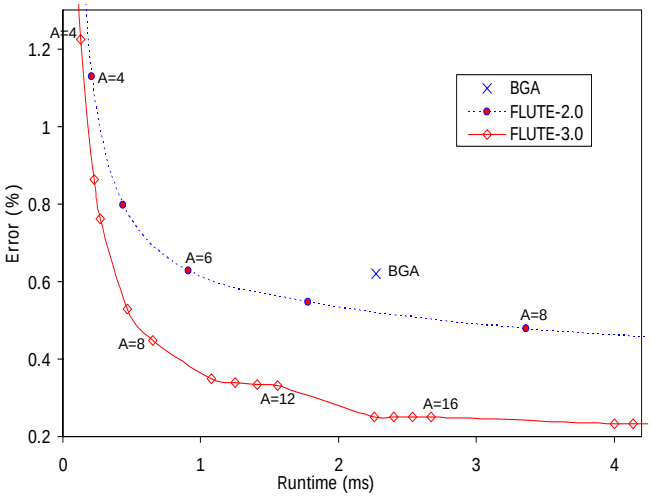


Figure 7: Tradeoff between wirelength error versus runtime for nets with degree 30.

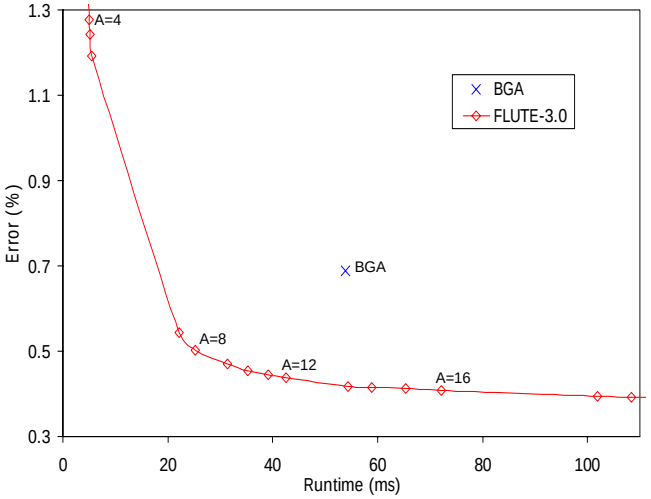


Figure 8: Tradeoff between wirelength error versus runtime for nets with degree 300.

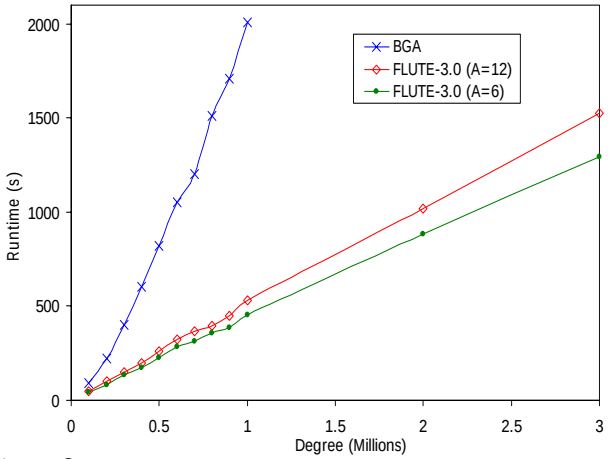


Figure 9: Average runtime comparison for very high degree nets.

Degree	RMST	BI1S	BGA	FLUTE-2.0 $A=3$	FLUTE-2.0 $A=6$	FLUTE-3.0 $A=4$	FLUTE-3.0 $A=6$	FLUTE-3.0 $A=8$	FLUTE-3.0 $A=10$	FLUTE-3.0 $A=12$	FLUTE-3.0 $A=14$	FLUTE-3.0 $A=16$	FLUTE-3.0 $A=18$
10	11.982	0.349	0.443	0.182	0.061	0.083	0.037	0.028	0.028	0.027	0.027	0.027	0.027
20	12.168	0.421	0.518	0.920	0.327	0.583	0.348	0.221	0.187	0.153	0.137	0.124	0.115
30	12.551	0.552	0.619	1.727	0.628	1.223	0.761	0.447	0.338	0.331	0.251	0.250	0.232
40	12.727	0.556	0.624	2.462	0.990	0.999	0.893	0.507	0.349	0.329	0.321	0.320	0.293
50	12.684	0.567	0.628	3.156	1.300	1.032	0.955	0.515	0.384	0.357	0.321	0.316	0.279
60	12.729	0.580	0.647	3.629	1.482	1.034	0.954	0.567	0.443	0.403	0.340	0.336	0.322
70	12.848	0.557	0.630	3.977	1.759	1.083	1.001	0.593	0.447	0.403	0.355	0.349	0.340
80	12.862	0.573	0.639	4.374	1.914	1.106	1.016	0.643	0.473	0.437	0.404	0.395	0.359
90	12.889	0.590	0.669	4.625	2.133	1.128	1.024	0.661	0.493	0.452	0.415	0.404	0.375
100	12.867	0.599	0.678	4.922	2.285	1.181	1.081	0.702	0.547	0.497	0.437	0.427	0.401
200	13.015	0.609	0.689	6.633	3.516	1.250	1.156	0.517	0.444	0.421	0.396	0.390	0.367
300	13.054	0.617	0.689	7.502	4.419	1.277	1.192	0.502	0.453	0.437	0.416	0.408	0.392
400	13.134	N/A	0.704	8.156	5.068	1.302	1.222	0.527	0.460	0.441	0.419	0.416	0.398
500	13.076	N/A	0.681	8.535	5.595	1.306	1.222	0.522	0.463	0.444	0.419	0.412	0.390

Table 1: Average wirelength error (in %) for nets of different degree.

Degree	RMST	BI1S	BGA	FLUTE-2.0 $A=3$	FLUTE-2.0 $A=6$	FLUTE-3.0 $A=4$	FLUTE-3.0 $A=6$	FLUTE-3.0 $A=8$	FLUTE-3.0 $A=10$	FLUTE-3.0 $A=12$	FLUTE-3.0 $A=14$	FLUTE-3.0 $A=16$	FLUTE-3.0 $A=18$
10	0.01	0.27	0.44	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.03	0.02	0.03
20	0.02	1.57	1.29	0.05	0.24	0.06	0.12	0.23	0.35	0.57	0.78	1.28	1.61
30	0.03	4.81	2.27	0.09	0.90	0.13	0.27	0.65	1.25	1.56	2.40	2.67	4.14
40	0.05	11.08	3.35	0.14	2.04	0.29	0.35	1.05	2.18	2.83	4.10	4.64	6.87
50	0.08	20.91	4.64	0.19	3.66	0.38	0.46	1.37	2.81	3.67	5.72	6.48	10.40
60	0.10	35.48	5.96	0.26	5.69	0.47	0.56	1.77	3.64	4.85	7.47	8.55	13.20
70	0.15	55.08	7.48	0.31	8.06	0.56	0.67	2.19	4.55	6.12	9.57	11.03	17.11
80	0.17	80.73	8.98	0.37	10.79	0.66	0.78	2.64	5.48	7.43	11.58	13.43	20.84
90	0.22	116.19	10.55	0.44	13.84	0.76	0.90	3.07	6.42	8.80	13.78	16.06	24.90
100	0.26	159.37	12.31	0.50	17.19	0.85	1.01	3.56	7.41	10.25	16.06	18.78	29.24
200	0.99	1360.44	31.46	1.22	60.28	2.67	3.04	12.61	20.06	25.57	36.82	42.18	62.24
300	2.20	4711.17	53.88	1.99	113.32	5.06	5.54	25.17	35.31	42.62	58.92	72.15	108.32
400	3.89	N/A	79.82	2.79	172.72	7.88	8.50	39.01	53.17	63.97	86.64	97.54	138.86
500	6.08	N/A	110.22	3.59	237.34	11.23	11.98	57.19	73.65	87.79	117.57	133.62	190.15

Table 2: Runtime per net (in ms) for nets of different degree.

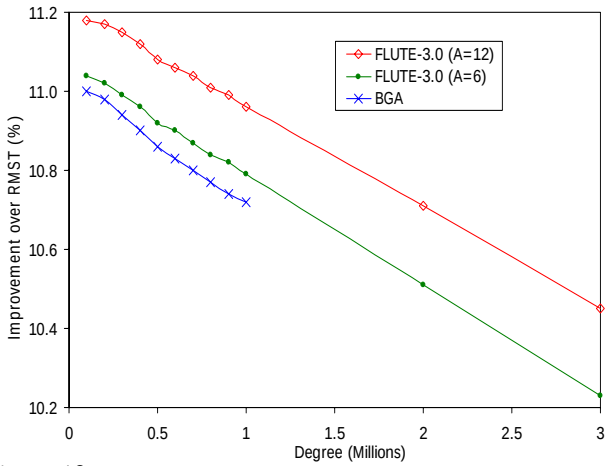


Figure 10: Average quality comparison for very high degree nets.

5 Conclusion

A tree-based net breaking technique and modifications to the original recursive bisection based net breaking technique are proposed to enhance FLUTE [1, 2] for rectilinear Steiner minimal tree construction for all degrees. The performance improvement is significant. It provides a better quality of result, with a smaller runtime, than BI1S [3] and BGA [4], the two state-of-the-art near-optimal RSMT algorithms found in the public domain.

We also provide a scheme to control the runtime and quality tradeoff for the new FLUTE algorithm, through the parameter A. It is shown that the runtime increases only moderately with the accuracy, and is within practical range even when A is set to as large as 20. Therefore the algorithm is very suitable for RSMT computation in various VLSI design applications, ranging from cases where fast computation is important (e.g., congestion estimation during placement), to cases where high accuracy is required (e.g., global / detailed routing).

References

[1] Chris Chu. FLUTE: Fast lookup table based wirelength estimation technique. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pages 696–701, 2004.

[2] Chris Chu and Yiu-Chung Wong. Fast and accurate rectilinear Steiner minimal tree algorithm for VLSI design. In *Proc. Intl. Symp. on Physical Design*, pages 28–35, 2005.

[3] J. Griffith, G. Robins, J. S. Salowe, and T. Zhang. Closing the gap: Near-optimal Steiner trees in polynomial time. *IEEE Trans. Computer-Aided Design*, 13(11):1351–1365, November 1994.

[4] A. Kahng, I. Mandoiu, and A. Zelikovsky. Highly scalable algorithms for rectilinear and octilinear Steiner trees. In *Proc. Asian and South Pacific Design Automation Conf.*, pages 827–833, 2003.

[5] D. M. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane Steiner tree problems: A computational study. In D.Z. Du, J.M. Smith, and J.H. Rubinstein, editors, *Advances in Steiner Trees*, pages 81–116. Kluwer Academic Publishers, 2000.

[6] GeoSteiner – software for computing Steiner trees. `http://www.diku.dk/geosteiner/`.

[7] I. I. Mandoiu, V. V. Vazirani, and J. L. Ganley. A new heuristic for rectilinear Steiner trees. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pages 157–162, 1999.

[8] H. Zhou, N. Shenoy, and W. Nicholls. Efficient minimum spanning tree construction without Delaunay triangulation. In *Proc. Asian and South Pacific Design Automation Conf.*, pages 192–197, 2001.

[9] M. Borah, R. M. Owens, and M. J. Irwin. An edge-based heuristic for Steiner routing. *IEEE Trans. Computer-Aided Design*, 13(12):1563–1568, December 1994.

[10] Hai Zhou. Efficient Steiner tree construction based on spanning graphs. In *Proc. Intl. Symp. on Physical Design*, pages 152–157, 2003.

[11] Andrew B. Kahng and Ion Mandoiu. RMST-Pack: Rectilinear minimum spanning tree algorithms. `http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/RSMT/RMST/`.