Prim-Dijkstra Revisited: Achieving Superior Timing-driven Routing Trees

Charles J. Alpert[1], Wing-Kai Chow[1], Kwangsoo Han[1,2], Andrew B. Kahng[2,3], Zhuo Li[1], Derong Liu[1] and Sriram Venkatesh[3]
[1]Cadence Design Systems, Inc., Austin, TX 78759
[3]CSE and [2]ECE Departments, UC San Diego, La Jolla, CA 92093
{kwhan, abk, srvenkat}@ucsd.edu, {alpert, wkchow, kwangsoo, zhuoli, derong}@cadence.com

ABSTRACT

The Prim-Dijkstra (*PD*) construction[1] was first presented over 20 years ago as a way to efficiently trade off between shortest-path and minimum-wirelength routing trees. This approach has stood the test of time, having been integrated into leading semiconductor design methodologies and electronic design automation tools. *PD* optimizes the conflicting objectives of wirelength (WL) and source-sink pathlength (PL) by blending the classic Prim and Dijkstra spanning tree algorithms. However, as this work shows, *PD* can sometimes demonstrate significant suboptimality for both WL and PL. This quality degradation can be especially costly for advanced nodes because (i) wire delays form a much larger component of total stage delay, i.e., timing-driven routing is critical, and (ii) modern designs are severely power-constrained (e.g., mobile, IoT), which makes low-capacitance wiring important. Consequently, achieving a good timing and power tradeoff for routing is required to build a market-leading product[2]. This work introduces a new problem formulation that incorporates the total detour cost in the objective function to optimize the detour to every sink in the tree, not just the worst detour. We then propose a new *PD-II* construction which directly improves upon the original *PD* construction by *repairing* the tree to simultaneously reduce both WL and PL. The *PD-II* approach achieves improvement for both objectives, making it a clear win over *PD*, for virtually zero additional runtime cost. *PD-II* is a spanning tree algorithm (which is useful for seeding global routing); however, since Steiner trees are needed for timing estimation, this work also includes a post-processing algorithm called *DAS* to convert *PD-II* trees into balanced Steiner trees. Experimental results demonstrate that this construction outperforms the recent state-of-the-art academic tool, SALT [36], for high-fanout nets, achieving up to 36.46% PL improvement with similar WL on average for 20K nets of size $\geq$ 32 terminals from DAC 2012 contest benchmark designs[37].

ACM Reference Format:
Charles J. Alpert[1], Wing-Kai Chow[1], Kwangsoo Han[1,2], Andrew B. Kahng[2,3], Zhuo Li[1], Derong Liu[1] and Sriram Venkatesh[3]. 2018. Prim-Dijkstra Revisited: Achieving Superior Timing-driven Routing Trees. In *ISPD '18: 2018 International Symposium on Physical Design, March 25–28, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3177540.3178239

1 INTRODUCTION

In recent technology nodes, wire capacitance has become a key challenge to design closure, and this problem only worsens with each successive technology node[2]. Today, a digital implementation flow cannot simply use minimum wirelength (WL) trees for routing estimates in placement and optimization, nor can they be used for timing-driven routing of critical nets. Routing an advanced-node design with minimum WL trees leads to untenable source-to-sink distances, yielding high delays for many nets. On the other hand, one cannot afford to use a shortest path tree which achieves optimal source-to-sink pathlength (PL) for each sink, due to the increased WL which degrades dynamic power and worsens routing congestion. For these reasons, timing-driven tree construction that trades off WL and PL becomes a critical technology for modern designs.

The Prim-Dijkstra (*PD*)[1] construction is generally regarded as the best available spanning tree algorithm for achieving this tradeoff and has the additional advantage of simplicity[4].[1] This algorithm has been used for over 20 years to construct high-performance routing trees in leading semiconductor design methodologies and electronic design automation (EDA) tools, as can be seen by related patents assigned to IBM, Synopsys, Cadence and other entities ([25] [26] [27] [28] [29] [30] [31] [32]). Further, the authors of [9] performed an evaluation that compared *PD* to other spanning tree constructions such as BRBC[10], KRY[11], etc. in 2006; they concluded that *PD* obtained the best tradeoff between WL and PL. That paper [9] argues that the *PD* wirelength cost is minimal enough to be practically free. However, this claim is now suspect because today's designs are significantly more power-sensitive than a decade ago: now, a 1% reduction in power is viewed as a big win for today's design teams performing physical implementation. Consequently, even a small WL savings with similar timing can have a high impact on value. A deeper discussion of prior art is given in Section 2.

The *PD* construction balances between WL and source-to-sink PL by blending the Prim and Dijkstra spanning tree constructions[5][6] via a weighting factor, α. When $\alpha = 0$, *PD* is identical to Prim's algorithm[5] and constructs a minimum spanning tree (MST). As α increases, *PD* constructs a tree with higher WL but better PL; when $\alpha = 1$, *PD* is identical to Dijkstra's algorithm[6] and constructs a shortest-path tree (SPT). *PD* begins with a tree consisting just the source node, then iteratively adds the edge e_{ij} that minimizes $d_{ij} + \alpha \cdot l_i$, where node v_i is in the current tree and v_j is not in the current tree, d_{ij} is the distance between nodes v_i and v_j, and l_i is the PL from the source to v_i in the current tree.

One problem with the *PD* algorithm is that it greedily adds edges, which becomes problematic with higher fanout trees. Once an edge is added, it is never removed from the final solution, making it

[1]For global routing, spanning trees are often preferred to Steiner trees since global routing commonly decomposes multi-fanout nets into two-pin nets. A spanning tree provides the router with an obvious decomposition. However, Steiner trees are not well-suited for this because the Steiner points become unnecessary constraints that restrict the freedom of the router to resolve congestion.

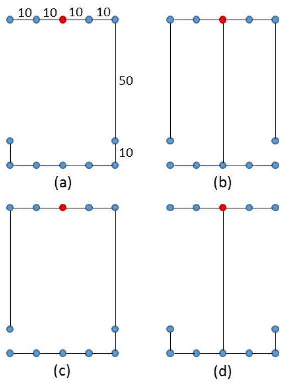

Figure 1: An example instance showing suboptimality of *PD*. The red node is the source. (a) shows the MST obtained when $\alpha = 0.2$, (b) shows the SPT obtained when $\alpha = 0.8$, and (c) shows the solution when $\alpha = 0.4$. The tradeoff in (c) is clearly suboptimal in both WL and PL, as compared to (d).

impossible for *PD* to recover from a potentially poor choice. This can lead to trees that are suboptimal in both WL and maximum PL. Figure 1 shows such an example. When α is small (0.2), *PD* obtains the MST solution (a) with $WL = 150$ and $PL = 130$. When α is large (0.8), *PD* obtains the SPT solution (b) with $WL = 240$ and $PL = 80$. However, when $\alpha = 0.4$, *PD* obtains the solution (c) with suboptimal values of both WL and PL ($WL = 190$ and $PL = 120$). This solution (c) is inferior for both objectives than the solution (d) with $WL = 160$ and $PL = 90$. Thus, $\alpha = 0.4$ generates a poor solution for both WL and PL.

This paper makes the following contributions:

- To fix the shortcomings in *PD*, one needs to directly optimize PL in the tree construction, which requires a new problem formulation. We propose incorporating total *detour cost*, the amount of suboptimal PL for each node, into the tradeoff. The correct formulation of the objective is paramount since it drives any optimization which follows. This work seeks to optimize the detour cost to all sinks instead of just the worst one, as proposed in prior works [36].
- Next, a new algorithm, which we call *PD-II*, is proposed. The idea is to recover the tree, that has any edges poorly chosen by *PD*, using an iterative improvement method according to the proposed objective function.
- Since Steiner trees are most commonly useful for timing prediction and physical synthesis, an algorithm for converting balanced spanning trees into balanced Steiner trees is proposed. The resulting *Detour-Aware Steinerization* (DAS) algorithm optimizes both WL and detour cost to achieve a tree with similar properties to those obtained by the *PD-II* spanning tree algorithm.
- Finally, three sets of experiments are presented. The first shows that *PD-II* is able to meaningfully shift the Pareto curve obtained by the *PD* algorithm, obtaining up to 18% improvement in PL for the same WL. The second experiment demonstrates the value of the *DAS* algorithm versus more standard Steinerization methods. The third experiment shows that the proposed Steiner construction outperforms those of SALT [36] for medium- and high-fanout nets, a recent state-of-the-art academic tool, achieving up to 36.48% PL improvement for similar WL.

The remainder of this paper is organized as follows. Section 2 briefly reviews related works in the areas of spanning and Steiner tree constructions. Section 3 presents the proposed problem formulation that incorporates both WL and detour cost. Section 4 presents the *PD-II* heuristic for spanning tree optimization, and Section 5 presents the *DAS* heuristic for Steiner tree optimization. Section 6 reports our experimental results, and Section 7 concludes the paper.

2 PREVIOUS WORK

There is a rich history on spanning tree and Steiner tree constructions. Many focus on minimizing WL or minimizing longest PL. (Our present work studies constructions that consider both metrics.)

Spanning Tree Constructions. As discussed previously, the Prim and Dijkstra constructions achieve optimal WL and PL, respectively. Spanning tree algorithms that optimize both are called *shallow-light* constructions [10] [12]; they seek to optimize WL and PL simultaneously to within constant factors of optimal. Shallow-light constructions have in many ways been a "holy grail" in VLSI CAD literature for over 25 years. The *PD* algorithm is "shallow-light in practice", but no such formal property has ever been established[1]. Cong et al.[13] give the Bounded Prim (BPRIM) extension of Prim's MST algorithm[5], which produces trees with low average WL and bounded PLs, but possibly unbounded WL. The BRBC algorithm of Cong et al.[10] produces a tree that has WL no greater than $1 + 2/\epsilon$ times that of an MST, and radius no greater than $1 + \epsilon$ times that of an SPT. Khuller et al.[12] contemporaneously develop a method similar to BRBC.

Minimum WL Heuristic Steiner Tree Constructions. Several works describe heuristic algorithms for Steiner tree constructions with minimized WL. Kahng and Robins[15] give the iterated 1-Steiner (I1-S) heuristic which greedily constructs a Steiner tree through iterative Steiner point insertion, resulting in trees with close to optimal WL. Ho et al.[7] propose an algorithm (HVW) to optimally edge-overlap *separable* MSTs to obtain Steiner trees, while Borah et al.[16] present a greedy heuristic (BOI) to convert spanning trees to RSMTs with performance similar to the I1-S heuristic. Chu and Wong[33] propose FLUTE which uses pre-computed look-up tables for Steiner construction to find solutions more efficiently than the prior art.

Rectilinear Steiner Arborescence (RSA) Constructions. The NP-complete[17] *rectilinear Steiner arborescence* (RSA) problem seeks to find a minimum-WL tree in the Manhattan plane that achieves optimal PL for every sink. Rao et al.[3] present the first heuristic for the RSA problem. Cong et al.[18] address the construction of RSAs with the A-tree algorithm, while Kahng and Robins[19] give a simple adaptation of their Iterated 1-Steiner algorithm to the RSA problem.

Steiner Constructions that Optimize WL and PL. Recently, Scheifele[35] has proposed a method to construct Steiner trees for which Elmore delays are bounded. Given an RMST solution (i.e., FLUTE), [35] iteratively finds the vertex that breaks its ϵ-based metric, and reroutes the vertex to the source via a shortest path, which indirectly balances between RMST and RSA. On the other hand, Elkin and Solomon[34] propose a more direct shallow-light Steiner tree construction method (ES). The main idea is to identify breakpoints and reconnect those breakpoints to the root directly by a Steiner SPT so that there is no detour from the root to the breakpoints. The authors of [34] build a Hamiltonian path and check the accumulated distance along the Hamiltonian path to find proper breakpoints, such that the final Steiner tree meets the given shallowness and lightness criteria. Recently, Chen et al.[36] present SALT, which further improves the ES method[34]. The key contributions are (i) tighter criteria to identify breakpoints, and (ii) using an MST instead of a Hamiltonian path. With some post-processing such as L-shape flipping, the method shows superior tradeoffs between pathlength and wirelength compared to any state-of-the-art spanning/Steiner tree construction methods. Comparisons to the method of [36] are included in Section 6 below.

Table 1: Notation

Notation	Meaning
V	signal net, $V = \{v_0, v_1, ... v_{n-1}\}$ having $n-1$ sinks
G	routing graph in the spanning tree context
T	routing tree, which is a spanning subgraph of G
v_0	source node of the signal net V, which is the root of T
e_{ij}	edge from node v_i to v_j
$par(v_i)$	parent node of v_i
l_j	cost of the unique v_0 to v_j path in a tree, $v_0, v_j \in T$
d_{ij}	cost of the edge e_{ij}
m_{ij}	Manhattan distance from node v_i to v_j
W_T	total wirelength of a tree
Q_i	detour cost of node v_i, $Q_i = l_i - m_{i0}$
Q_T	detour cost of a tree, $= \sum_{n-1}(Q_i)$
C	weighted cost of a tree, $= \alpha \cdot Q_T + (1-\alpha) \cdot W_T$
$\Delta C_{e,e'}$	the change in the weighted cost that results from removing edge e and adding e', used in *PD-II*
α	weighting factor used in *PD* and *PD-II*
D	flipping distance used in *PD-II*
P_T	sum of pathlengths of a tree, $= \sum_{n-1}(l_j)$

3 PROBLEM FORMULATION

A *signal net* $V = \{v_0, v_1, ..., v_{n-1}\}$ is a set of n terminals, with v_0 as the *source* and the remaining terminals as *sinks*. We define the underlying routing graph to be a connected weighted graph $G = (V, E)$, where each edge $e_{ij} \in E$ has a cost d_{ij}. We are concerned with the case where G is a complete graph with each e_{ij} having cost equal to the Manhattan distance d_{ij}. A *routing tree* $T = (V, E')$ is a spanning subgraph of G with $|E'| = n-1$.[2] Given a routing tree T, the cost of the unique $v_0 - v_i$ path in T is l_i, the *radius* of T is $r(T) = max_{1 \le i \le n-1} l_i$, and the *wirelength* (WL) of T is $W_T = \sum_{e_{ij} \in T} d_{ij}$. All notations used in our work are listed in Table 1.

Initially, the tree consists only of v_0. The *PD* algorithm iteratively adds edge e_{ij} and sink v_i to T, where v_i and v_j are chosen to minimize

$$(\alpha \cdot l_j) + d_{ij} \; s.t. \; v_j \in T, \; v_i \in V - T \tag{1}$$

The *PD* algorithm can result in trees with either large WL or PL, as shown in Figure 1. To alleviate this issue, conventional *shallow-light* tree constructions[10] [13][36] focus on bounding the *shallowness* and *lightness* to optimize the tree cost. Lightness η means that the WL of a tree is at most η times of the MST WL. A tree has shallowness ζ if PL to each sink in the tree is at most ζ times the source-to-sink Manhattan distance (MD). However, shallowness alone does not adequately represent the quality of a routing tree. Figure 2 shows two examples that have the same shallowness and lightness. It is clear that Figure 2(b) is preferable to Figure 2(a) since the left sinks have shorter PLs, but shallowness does not capture the difference.

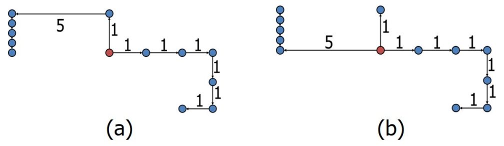

Figure 2: Two routing trees that have the same lightness and shallowness.

With the above in mind, we define a new *detour cost* metric as follows. Detour cost Q_i of a sink v_i is the difference between PL from v_0 to v_i in T and the Manhattan distance from v_0 to v_i. The detour cost of the tree T, denoted by Q_T, is the sum of the detour cost values of all the sinks in the tree, i.e., $Q_T = \sum_{1 \le i \le n-1} Q_i$.

[2]Our use of G and T pertains to the spanning tree context. In the rectilinear Steiner tree context, the underlying routing graph would be the Hanan grid [20], and a Steiner routing tree would be a spanning tree over $\{V \cup S\}$, where S is a set of Steiner points taken from the Hanan grid. For simplicity, as long as meanings are obvious, we will use terms from the spanning tree context in the Steiner tree context as well.

Since *PD* iteratively adds edges and nodes to the growing tree, if a sink v_j close to the source incurs high detour, then all downstream sinks (descendants of v_j) will also have high detour and hence long PL. We therefore propose the following formulation to capture the problem of simultaneously reducing WL and detour cost of a spanning tree:

Simultaneous WL and Detour Cost Reduction (SWDCR) Problem. Given a spanning tree $T = (V, E)$, minimize the weighted sum of WL and detour cost of the tree.

$$\text{Minimize } \alpha \cdot \sum Q_i + (1-\alpha) \cdot W_T \tag{2}$$

where $0 \le \alpha \le 1$. We present a heuristic algorithm *PD-II* in Section 4 for tackling the SWDCR problem.

Once the spanning tree construction is converted into a Steiner tree, there is a change in the tree topology. We propose and address the following formulation to further optimize the detour cost of a Steiner tree:

Detour Cost Reduction in Steiner Trees (DCRST) Problem. Given a Steiner tree, minimize the tree detour cost.

$$\text{Minimize } Q_T \tag{3}$$

$$\text{s.t. } W_{T,new} \le W_{T,init} \tag{4}$$

$$Q_{T,init} \ge Q_{T,new} \tag{5}$$

To address the DCRST problem, we present our algorithm *DAS* below in Section 5.

4 THE *PD-II* SPANNING TREE CONSTRUCTION

This section presents the *PD-II* algorithm that performs iterative edge-swapping which simultaneously improves the detour cost and WL. The key idea of the *PD-II* algorithm is to start with a spanning tree and swap edges to improve the tradeoff between detour cost and WL. The algorithm can take any spanning tree as input, but it makes sense to start with the *PD* solution since it should already be relatively strong for both objectives. We note that while *PD* can be quite slow for higher-fanout nets, it can be sped up significantly by using a sparsified nearest-neighbor graph instead of the complete graph.

We initially populate the *neighbors* of each node using the following method. We say that v_i is a *neighbor* of v_j if the smallest bounding box containing v_i and v_j contains no other nodes. The worst-case number of neighbor nodes for each node is $\Theta(n)$. For example, every red point in Figure 3 is a neighbor of every green point, and vice versa. However, Naamad et al.[23] show that the expected number of maximal empty boxes amidst n random points in a plane is bounded above by $O(n \log n)$, so it is reasonable to expect the average number of neighbors per node to be $O(\log n)$.

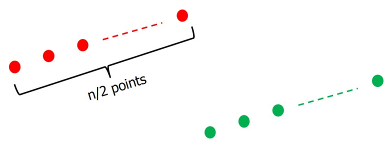

Figure 3: Example showing $\Theta(n^2)$ asymptotic worst-case complexity of the number of *neighbor* relationships. Each green node is a neighbor to each red node.

Analysis of random placements of net sinks show this to be true. The number of neighbors for 100K random point sets of size 16, 32, and 64 yields an average number of neighbors per node of 6.3, 8.7 and 11.3, respectively. Real placements should generally have even fewer neighbors, since cells tend to align horizontally or vertically. For the testcases described in Section 6.1, the average number of neighbors is 2.58, 4.27, 6.15 and 8.24 for small, medium, large and huge nets, respectively. Hence, in practice, runtime complexity of iterating through the neighbors of a node has logarithmic complexity.

An $O(n \log n)$ runtime complexity can be obtained for *PD* using a binary heap implementation and an adaptation of Scheffer's MST code[21][22]. Since *PD* solutions are generally good, though sometimes suboptimal, it makes sense to post-process the *PD* solution to obtain a better one. The key technique for *PD-II* is *edge flipping*, whereby one edge is removed from the original tree and replaced with a new edge. Figure 4(a) shows an example tree, represented as a DAG, representing a topological ordering starting at the source. Figure 4(b) shows an example transform in which one edge is removed and replaced with a new red edge, thereby obtaining a different tree. Note that one of the directed edges in the new (rooted) tree is reversed from its previous orientation in order to maintain a well-formed rooted tree. This approach recalls the iterative improvement operation used in BOI[16], but the application of *flipping* is more restricted to focus on WL vs. PL improvements.

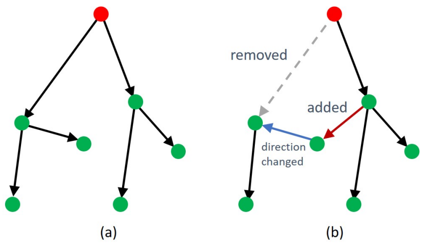

Figure 4: Illustration of *PD-II* edge *flipping*.

For each edge pair, we define the *flip cost* as the cost associated with edge flipping, i.e., the cost of removing edge e_{ij} and adding edge $e_{i'j'}$. *Flip cost* $\Delta C_{e,e'} = \alpha \cdot (Q_{T_{i'j'}} - Q_{T_{ij}}) + (1-\alpha) \cdot (d_{i'j'} - d_{ij})$, where α is a weighting factor;[3] $Q_{T_{ij}}$ and $Q_{T_{i'j'}}$ are the detour costs of the trees before and after edge flipping, respectively; and d_{ij} and $d_{i'j'}$ are the lengths of edges being removed and inserted, respectively.

Pseudocode for *PD-II*, Algorithm 1 is given below. Essentially, *PD-II* takes an input tree and searches for edge flips that improve flip cost.[4] If the flip cost improves, the swap is taken. Considering all pairs of possible swaps could be expensive, so we define the *flipping distance D* to be equal to the number of edges in the DAG that require a change in direction to preserve topological ordering, i.e., rooted orientation. For the swap in Figure 4, $D = 1$. In practice, using $D > 1$ has little benefit (but more runtime) compared to $D = 1$, so we use $D = 1$ for all experiments.

Line 3 of Algorithm 1 initializes the best flip cost to zero. Line 5 computes the set of candidate edges E_e that can be flipped with edge e, as restricted by the flipping distance D. For each candidate edge $e' \in E_e$, we calculate the flip cost for the edge pair (e, e') and find the edge pair (e_{best}, e'_{best}) with lowest flip cost in Lines 6-12. These edges are swapped if the lowest flip cost is less than zero (Lines 14-16). The algorithm continues until no more flip-cost improvement is obtained (Line 17).

The number of candidates for edge flipping can be very large when D is unbounded. The worst-case number of edges is $(n/2)^2$, giving Algorithm *PD-II* a worst-case time complexity of $O(n^3)$,

Algorithm 1 Algorithm *PD-II*

```
Input: Spanning tree $T_{in} = (V, E_{in})$, with $E_{in} \subseteq E$
Output: Spanning tree $T_{out} = (V, E_{out})$, with $E_{out} \subseteq E$
 1: Initialize $T_{out} \leftarrow T_{in}$
 2: repeat
 3:     Initialize largest detour cost reduction, $\Delta C_{best} \leftarrow 0$
 4:     for all $e \in E_{out}$ do
 5:         $E_e \leftarrow candidateEdges(e, D)$
 6:         for all $e' \in E_e$ do
 7:             $\Delta C_{e,e'} \leftarrow flipCost(e, e')$
 8:             if $\Delta C_{e,e'} < \Delta C_{best}$ then
 9:                 $\Delta C_{best} \leftarrow C_{e,e'}$
10:                 $e_{best} \leftarrow e$ ; $e'_{best} \leftarrow e'$
11:             end if
12:         end for
13:     end for
14:     if $\Delta C_{best} < 0$ then
15:         Remove $e_{best}$, insert $e'_{best}$ and change direction of associate edges
16:     end if
17: until $\Delta C_{best} == 0$
```

where n is the number of sinks. However, with the distance restriction, the complexity reduces to $O(D \cdot n^2)$, and in practice it converges rapidly. To show this, we take two large blocks from an industrial design and run a production Steiner package on an Intel Xeon 2.7GHz machine (CPU E5-2680), using RHEL5. The first design has 1.9 million datapath nets, and the total runtime for the Steiner package which uses *PD* for its spanning tree construction requires 59.3 seconds. Adding *PD-II* to the Steiner package increases the runtime to 62.7 seconds, for a net penalty of 3.4 seconds. The second design with 4.0M datapath nets requires 124.0 seconds for running the default Steiner package. Adding *PD-II* to the Steiner package increases the runtime from 124.0 seconds to 125.8 seconds, for a net penalty of 1.8 seconds. Consequently, the runtime cost of using *PD-II* is negligible, averaging less than one additional second of runtime per million nets.

5 THE DETOUR-AWARE STEINERIZATION ALGORITHM (*DAS*)

For global routing, spanning tree constructions such as *PD-II* are sometimes preferred to Steiner trees since global routing commonly decomposes multi-pin nets into two-pin nets. However, for timing estimation, congestion prediction, or general physical synthesis optimization, a Steiner tree is required since spanning trees will have too much WL. The previous spanning tree formulation can easily be extended to Steiner trees; the definitions of WL and PL do not change. However, since finding the minimum wirelength Steiner is NP-complete, FLUTE WL is used as a proxy for minimum Steiner tree cost.

To transform a spanning tree into a Steiner tree, the linear-time algorithm of [7] is invoked. It maximizes edge-overlaps in the spanning tree by creating a Steiner node. We call the algorithm *HVW* after the algorithm's creators: Ho, Vijayan, and Wong. *HVW* traverses the tree from the leafs and iteratively maximize overlaps with the currently visited edge and its immediate children edges. However, this basic construction can be inefficient both in terms of WL and PL. Hence a new Steinerization algorithm, called *DAS* for Detour-Aware Steinerization is proposed below.

DAS has two phases of optimization. The first phase seeks to reduce WL while minimizing the detour cost penalty (Lines 1-14). This phase does a bottom-up tree traversal and makes edge swaps which reduce WL. For each edge e_{ji} in the Steiner tree, the edge e_{ji} is removed from the tree and replaced with e_{ki} where v_k is a nearest neighbor of v_i if the WL improves and the PL is not overly degraded. (i.e., $p_i \leq 0.5 \cdot p_T^{max}$).

After the first phase, since PL (or detour cost) is not targeted, there still may be room to improve for that dimension. Hence,

[3]The parameter α can be determined by the timing constraints. If a net is critical, a higher value of α can be used to achieve lower delays, but if arcs through the net have positive slacks, α can be small to save wirelength. Hence, α allows topology optimization and can be chosen to best satisfy the design specifications on a per-net basis.

[4]Flipping cannot be added into the original PD cost function since the flip cost objective cannot be correctly computed until an entire tree is constructed. Hence, we propose PD-II as a post-processing algorithm which improves a given spanning tree.

Algorithm 2 The Detour-Aware Steinerization Algorithm (*DAS*)

```
Input: Steiner tree $T_{St,in}$
Output: Improved Steiner tree $T_{St,out}$
1:  //First phase: wire recovery at the cost of small additional PL
2:  $p_T^{max}$ ← maximum PL of the Steiner tree
3:  Do Breadth-First Search (BFS) from the leaf node
4:  for all $v_i$ do
5:      $v_j \leftarrow par(v_i)$ ; $d_{ji}$ ← edge length to $v_i$;
6:      $o_{ji}$ ← overlap length with other edges to $v_i$
7:      $\Delta d_{ji} \leftarrow d_{ji} - o_{ji}$
8:      for all $v_k$ in {all neighbors of $v_i$} do
9:          $\Delta d_{ki} \leftarrow d_{ki} - o_{ji}$; $p_i$ ← PL to node $v_i$
10:         if ($\Delta d_{ki} < \Delta d_{ji}$ && ($p_i \leq 0.5 \cdot p_T^{max}$) then
11:             Disconnect $v_i$ to $v_j$ and reconnect $v_i$ to $v_k$
12:         end if
13:     end for
14: end for
15: //Second phase: detour cost reduction with bounded WL
16: $W_{T,init}$ ← Init. Steiner tree WL; $Q_{T,init}$ ← Init. Steiner tree detour cost
17: Do Breadth-First Search (BFS) from the source node
18: for all $v_i$ do
19:     $v_j \leftarrow par(v_i)$; $d_{ji}$ ← Initial edge length to $v_i$
20:     for all $v_k$ in {all neighbors of $v_i$} do
21:         $e_{ki}$ ← Edge from $v_k$ to $v_i$; $d_{ki}$ ← Edge length from $v_k$ to $v_i$
22:         $W_{T,new} \leftarrow W_{T,init} + d_{ki} - d_{ji}$
23:         $Q_{T,new}$ ← detour cost tree with edge $e_{ki}$
24:         if ($W_{T,new} \leq W_{T,init}$) && ($Q_{T,new} < Q_{T,init}$) then
25:             Disconnect $v_i$ to $v_j$ and reconnect $v_i$ to $v_k$
26:             $W_{T,init} \leftarrow W_{T,new}$; $Q_{T,init} \leftarrow Q_{T,new}$
27:         end if
28:     end for
29: end for
```

a second phase (Lines 15-29) seeks to optimize detour cost Q_T without degrading WL. This second phase performs a *top-down* tree traversal to minimize Q_T. This is because the detour cost Q_i to a node v_i affects not only the PL to the node, but also the PL to the downstream nodes of v_i. Thus, more opportunity for large Q_T reductions exists in the edges near the source v_0. For each edge e_{ji} in the Steiner tree, the edge e_{ji} is removed and replaced with e_{ki}, where v_k is the possible parent among the nearest neighbors of v_i, to reduce Q_T without degrading WL. This process is repeated for all the nodes in the tree with non-zero detour cost.

Algorithm *DAS* has a worst-case time complexity of $O(n^2)$. However, with the sparsified nearest neighbor graph implementation described in Section 4, *DAS* runs much faster than $O(n^2)$ and is closer to $O(n\ log\ n)$ in practice. For 100K nets, *DAS* runs in 0.86 seconds for 16-terminal nets, 1.71 seconds for 32-terminal nets and 4.83 seconds for 64-terminal nets.

6 EXPERIMENTAL SETUP AND RESULTS

6.1 Experimental Setup

The algorithms described above are implemented in C++. The following experiments are performed on a 2.7 GHz Intel Xeon server with 8 threads. Testcases are generated from the DAC 2012 contest benchmarks[37], with pin locations for each net are extracted from ePlace placement solutions[38]. Since finding a solution with optimal WL and PL is trivial for two- and three-pin nets, our experiments focus on nets with fanout larger than two. The roughly 749K total nets are divided into four groups (small, medium, large, huge) by their terminal count, as shown in Table 2.

Table 2: Net Statistics for Superblue Benchmark Designs

	small	medium	large	huge
$\|V\|$	4 – 7	8 – 15	16 – 31	32+
#nets	533029	128463	46486	20853

While our algorithms optimize Q_T, Q_T itself does not adequately capture the quality of the tree. Instead, results are reported based on two normalized metrics, W_{Tnorm} (normalized WL) and P_{Tnorm} (normalized PL). W_{Tnorm} is defined as the ratio of the tree WL to the MST WL for spanning trees. P_{Tnorm} is defined as the ratio of sum of PLs of each node in the tree to the sum of Manhattan distances from source to each node. The optimal value any tree could have for either metric is one, which makes the corresponding Pareto curve more intuitive.

6.2 Experiment I - Spanning Tree Results

In the following results, *PD* and *PD-II* refer respectively to the *spanning trees* constructed using the *PD* and *PD-II* algorithms. Figure 5 shows normalized WL and PL tradeoff curves for *PD* and *PD-II*, for the 46486 large nets. Each point in the curves represents the average (W_{Tnorm}, P_{Tnorm}) over all the nets for a particular value of α. We sweep α from 0.05 to 0.95, in steps of 0.05, to obtain both the *PD* and *PD-II* curves. We observe that the blue *PD-II* Pareto curve is clearly better than the red *PD* curve.

The Pareto curve makes the improvement trend clear, but makes it difficult to measure the degree of improvement of *PD-II*. To compare the two algorithms more robustly, we analyze the results in the following way; (1) select different percentages of permissible WL degradation with respect to MST WL (i.e., WL thresholds = 1%, 2%, 4%, 7%, 10% and 15%), and (2) for each net, find the minimum P_{Tnorm} solution that meets the WL threshold across all solutions with different α. The results are averaged across all the nets and summarized in Table 3. Each entry in the table corresponds to the normalized PL P_{Tnorm}. To find the percentage improvement, one is subtracted from each value, since 1.0 is a lower bound. For example, a reduction from 1.15 to 1.12 results in an improvement of 20%, i.e., $(1 - (1.12 - 1.0)/(1.15 - 1.0)) \cdot 100\%$.

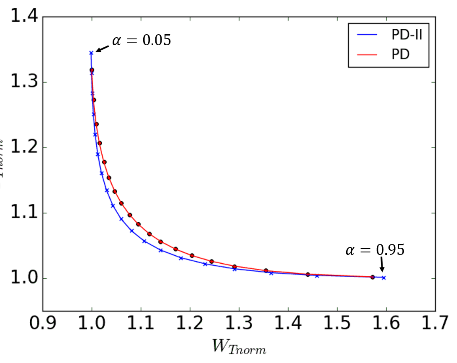

Figure 5: WL and PL tradeoff for various α.

Table 3: Comparisons of the best P_{Tnorm} for *PD* and *PD-II* across different WL thresholds.

$\|V\|$	Method	WL threshold					
		1%	2%	4%	7%	10%	15%
small	*PD*	1.0972	1.0927	1.0819	1.0680	1.0569	1.0427
	PD-II	1.0970	1.0923	1.0812	1.0672	1.0561	1.0420
	Imp. (%)	**0.26**	**0.42**	**0.78**	**1.15**	**1.36**	**1.63**
med.	*PD*	1.1888	1.1746	1.1483	1.1189	1.0974	1.0723
	PD-II	1.1870	1.1706	1.1423	1.1122	1.0909	1.0668
	Imp. (%)	**0.93**	**2.33**	**4.07**	**5.66**	**6.62**	**7.68**
large	*PD*	1.2981	1.2698	1.2216	1.1723	1.1390	1.1006
	PD-II	1.2895	1.2545	1.2025	1.1533	1.1219	1.0870
	Imp. (%)	**2.89**	**5.66**	**8.64**	**11.00**	**12.32**	**13.52**
huge	*PD*	1.3952	1.3550	1.2873	1.2210	1.1777	1.1302
	PD-II	1.3758	1.3238	1.2526	1.1876	1.1488	1.1056
	Imp. (%)	**4.91**	**8.79**	**12.06**	**15.14**	**16.27**	**18.87**

We observe the following:

- *PD-II* gives better results than *PD* for all classes of nets. This makes sense since it strictly improves upon an existing *PD* solution.

- Small nets obtain relatively small improvement, ranging from 0.26% to 1.63%; however, huge nets show significant improvements, ranging from 4.91% to 18.87%. Trends for medium and large nets lie in between. This is because the detour cost is close to optimal for smaller nets, but is much larger for bigger nets. For example, with a 1% WL threshold, the average normalized PL for *PD-II* is 1.097 for small nets but 1.376 for large nets.
- When the WL threshold is tight (such as 1% or 2%), the improvement of *PD-II* is much smaller as compared to looser constraints of 10% or 15%. This makes sense because a looser constraint gives the algorithms more freedom to reduce PL. A threshold of 1% means the topology cannot deviate much from the minimum-length spanning tree.

6.3 Experiment II - Steiner Tree Results

Our next experiments compare (*PD* + *HVW* + *DAS*) and a baseline flow (PD + *HVW*) to show the value of *DAS*. *HVW* refers to the Steiner tree obtained after performing edge-overlapping as described by Ho et al.[7], and *DAS* refers to the Steiner tree after applying *DAS* algorithm to the *HVW* tree. Figure 6 shows the normalized WL and PL tradeoff comparison for the two flows for the set of large nets. Steiner tree W_{Tnorm} is defined as the ratio of total WL of the tree to the FLUTE WL[5][33] and P_{Tnorm} is defined as the ratio of sum of PLs of all sinks in the tree to the sum of source-to-sink Manhattan distances. Each point in the curve represents the average (W_{Tnorm}, P_{Tnorm}) over all nets, for a particular value of α. It is clear that *DAS* adds significant value to the Steiner construction, pushing its Pareto curve further left and down compared to the one from the baseline.

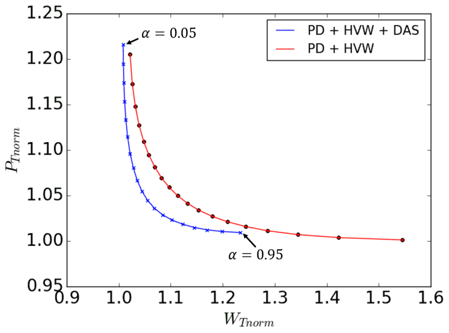

Figure 6: WL and PL tradeoff for Steiner tree constructions.

Table 4: Comparisons of the best P_{Tnorm} for (1) *PD + HVW* and (2) *PD + HVW + DAS* across different WL thresholds.

$\|V\|$	Method	WL threshold					
		1%	2%	4%	7%	10%	15%
small	(1)	1.0233	1.0241	1.0250	1.0249	1.0236	1.0202
	(2)	1.0126	1.0115	1.0097	1.0073	1.0054	1.0033
	Imp. (%)	**46.14**	**52.31**	**61.15**	**70.85**	**77.30**	**83.67**
med.	(1)	1.0786	1.0821	1.0828	1.0757	1.0649	1.0489
	(2)	1.0665	1.0629	1.0532	1.0385	1.0277	1.0168
	Imp. (%)	**15.43**	**23.30**	**35.78**	**49.07**	**57.24**	**65.58**
large	(1)	1.1637	1.1644	1.1547	1.1275	1.1026	1.0728
	(2)	1.1440	1.1347	1.1087	1.0760	1.0553	1.0357
	Imp. (%)	**12.01**	**18.07**	**29.73**	**40.36**	**46.08**	**50.93**
huge	(1)	1.2278	1.2091	1.1606	1.1107	1.0812	1.0538
	(2)	1.228	1.209	1.161	1.111	1.081	1.054
	Imp. (%)	**8.36**	**15.14**	**27.82**	**36.36**	**39.74**	**41.69**

Similarly to Table 3, Table 4 shows normalized PL across a range of permissible WL degradations for *HVW* versus *HVW+DAS*. We observe the following:

- DAS always obtains better results than HVW. Again, this makes sense since DAS starts with an HVW solution and further refines it to improve both WL and PL.
- Improvements for DAS can be quite significant, ranging from 8.36% to 83.67%.
- DAS improves results more significantly for smaller fanout nets than for larger ones. This may suggest there is still further room for improvement in Steinerization.
- Larger WL thresholds correspond to larger normalized PL improvements, which again is likely due to more freedom for the algorithm to find a solution that reduces detour cost.

6.4 Experiment III - Comparison with SALT[36]

Our final set of experiments compares the best combined flow (*PD-II* + *HVW* + *DAS*) with the results from the state-of-the-art academic Steiner tree construction, SALT[36]. SALT uses FLUTE [33] to generate its initial input and improves the initial construction to reduce PL. For nets with less than 10 terminals, FLUTE produces the optimal WL and may also produce excellent or even optimal PL, in which case running SALT is not even necessary. Hence, the cases for which FLUTE produces excellent PL are in some sense uninteresting. If FLUTE produces a good tradeoff curve, then SALT simply returns the FLUTE solution. Our approach can do something similar using the following simple metaheuristic: (1) run both FLUTE and (*PD-II* + *HVW* + *DAS*) in parallel; (2) if FLUTE is better than (*PD-II* + *HVW* + *DAS*) for both WL and PL, return the FLUTE solution, else return the (*PD-II* + *HVW* + *DAS*) solution. Essentially, the metaheuristic returns a solution identical to SALT's when the FLUTE solution is dominant. Note that for large and huge nets, the FLUTE solution almost never is dominant.

Figure 7 shows normalized WL and PL tradeoff curves for the metaheuristic flow and SALT for (a) small, (b) medium, (c) large and (d) huge nets. For small nets, SALT actually achieves better solutions than the metaheuristic until the normalized WL is about 2.3% higher than optimal.[6] However, for medium, large and huge nets, the Pareto curve for the metaheuristic outperforms the one from SALT, especially as nets increase in size. For huge nets, SALT achieves $W_{Tnorm} = 1.0370$, $P_{Tnorm} = 1.141$ for $\epsilon = 1.281$, which is its knee point in the tradeoff curve. The knee point in the metaheuristic's tradeoff curve corresponds to $W_{Tnorm} = 1.024$ and $P_{Tnorm} = 1.121$ at $\alpha = 0.35$, which achieves 35.13% WL and 14.18% PL improvements compared to SALT at its $\epsilon = 1.281$.

Since SALT optimizes shallowness and not detour cost, Figure 8 presents the same set of data but using SALT's proposed metrics. SALT dominates our method according to the shallowness metric. Thus, SALT is superior with respect to its proposed metric, while *PD-II* + *HVW* + *DAS* is superior with respect to its metric.

Finally, Table 5 compares our best recipe to SALT using the same methodology as Tables 3 and 4. Note that we use FLUTE WL as a lower bound. We observe the following:

- For small nets, and WL thresholds below 10%, SALT outperforms the proposed approach. SALT is also better on medium nets with WL thresholds below 2%. This makes sense since trees in this space will closely resemble FLUTE constructions. SALT starts with a FLUTE construction and iteratively improves it, so in the space where FLUTE obtains good trees for WL and PL, such an approach outperforms the algorithm proposed in this work. Note that the magnitude of the improvement is still small. For example, for small nets and a 1% threshold, SALT is 0.99% away from the optimal normalized path length, while our approach is 1.26% away.

[5]FLUTE constructs optimal RSMTs for nets with terminal sizes up to 9, and near-optimal RSMTs for nets with higher terminal counts.

[6]For {small, medium, large, huge} nets, FLUTE results for {55.6, 7.9, 0.03, 0}% of nets have smaller WL and PL than our results. As expected, FLUTE results are dominant for small nets, but our algorithm gives better PL for large and huge nets.

- For large and huge nets, and for medium nets with thresholds larger than 2%, the proposed approach performs better, reaching a peak of 36.46% improvement for huge nets with a 10% threshold. This is the domain for which the optimal tradeoff can be considerably different from FLUTE. These arguably form the class of more interesting instances where the tradeoff between WL and PL becomes increasingly important.
- As WL threshold increases, the improvement of our approach vs. SALT improves too, especially around the 7% and 10% WL threshold ranges. However, for large and huge nets the improvement is somewhat less at the 15% threshold.

Table 5: Comparisons of the best $P_{T norm}$ for (1) SALT and (2) *PD-II* + *HVW* + *DAS* across different WL thresholds.

$\|V\|$	Method	WL threshold					
		1%	2%	4%	7%	10%	15%
small	(1)	1.0099	1.0093	1.0082	1.0067	1.0053	1.0036
	(2)	1.0126	1.0115	1.0097	1.0073	1.0054	1.0033
	Imp. (%)	**-27.29**	**-23.85**	**-17.98**	**-8.80**	**-0.86**	**7.90**
med.	(1)	1.0652	1.0619	1.0547	1.0435	1.0337	1.0213
	(2)	1.0665	1.0629	1.0532	1.0385	1.0277	1.0168
	Imp. (%)	**-1.95**	**-1.66**	**2.76**	**11.32**	**17.63**	**21.15**
large	(1)	1.1564	1.1475	1.1261	1.0961	1.0720	1.0432
	(2)	1.1440	1.1347	1.1087	1.0760	1.0553	1.0357
	Imp. (%)	**7.91**	**8.66**	**13.77**	**20.92**	**23.09**	**17.31**
huge	(1)	1.2744	1.2574	1.2205	1.1688	1.1277	1.0763
	(2)	1.2278	1.2090	1.1606	1.1107	1.0811	1.0536
	Imp. (%)	**17.01**	**18.79**	**27.18**	**34.44**	**36.46**	**29.71**

Runtime. For the benchmarks studied, SALT's total runtime is 2762 seconds. By contrast, the *PD-II* + *HVW* + *DAS* algorithms, as implemented and optimized within a commercial EDA tool's code base, take 361 seconds in total. Thus, PD-II today runs more than 7 times faster than SALT.

Delay. Below, we show the impact of WL and PL improvement on delay. We estimate delays of nets produced by our algorithms and by SALT, based on the Elmore delay model with resistance of 37.318Ω per micron of wire, capacitance of 0.228fF per micron of wire, and 0.67fF pin capacitance per sink. For the solutions produced by our approach and SALT with WL threshold 2%, we calculate the sum of all sink delays for each net, and the average of this sum across all nets. For {small, medium, large, huge} nets, the average sum of sink delays from PD-II is lower than the average sum of sink delays from SALT by {-0.0005, 0.24, 1.54, 5.62}%. As seen with the WL and PL comparison, our algorithm has slightly larger delays for small nets and smaller delays for higher-fanout nets.

In summary, while our approach does not uniformly outperform SALT, it does provide a superior tradeoff for the most interesting class of nets that are far from optimal in terms of PL and WL.[7]

7 CONCLUSION

This work shows that the classic *PD* spanning tree algorithm that balances between Prim's and Dijkstra's algorithm can have a bad tradeoff that ends up with both WL and PL being highly suboptimal. A new spanning tree heuristic *PD-II* is demonstrated to significantly improve both WL and total detour cost compared to *PD*. Further, this work extends the construction to Steiner tree with the *DAS* algorithm that directly improves trees according to both objectives. The algorithms are shown to be fast and practical. They are also suitable for integration into existing commercial routers, and can be applied in conjunction with any existing spanning and Steiner tree constructions for simultaneous WL and PL improvements. Compared to the recent SALT algorithm, our construction generates clear improvements according to the proposed metrics, especially for medium-size and larger nets. Future research includes (i) revisiting the still-open question of worst-case detour from a *PD* construction; (ii) learning-based estimation of the best α for any given instance (i.e., set of pin locations of a signal net); and (iii) extending the detour cost objective to encompass sink criticality, "global" radius, and other additional criteria.

[7]The PD-II algorithm has been released as part of a leading commercial tool, with demonstrated improvements of timing and wirelength.

8 ACKNOWLEDGMENTS

We thank Mr. Gengjie Chen and the other authors of SALT [36] for their kindness in performing studies with their code so that we could report comparisons with SALT. We also thank the authors of [38] for making their ePlace executable available for us to gather placement data. ABKGroup research is supported by NSF, Samsung, Qualcomm, NXP, Mentor Graphics and C-DEN.

REFERENCES

[1] C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng and D. Karger, "Prim-Dijkstra Tradeoffs for Improved Performance-driven Routing Tree Design", *IEEE TCAD* 14(7) (1995), pp. 890-896.
[2] ITRS 2013 Edition Report - Interconnect, *https://www.semiconductors.org/clie ntuploads/Research_Technology/ITRS/2013/2013Interconnect.pdf*, 2013.
[3] S. K. Rao, P. Sadayappan, F. K. Hwang and P. W. Shor, "The Rectilinear Steiner Arborescence Problem", *Algorithmica* 7(2) (1992), pp. 277-88.
[4] C. J. Alpert, *Personal Communication*, Nov. 2016.
[5] R. C. Prim, "Shortest Connecting Networks and Some Generalizations", *Bell System Tech. J.* 36 (1957), pp. 1389-1401.
[6] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik* 1 (1959), pp. 269-271.
[7] J. M. Ho, G. Vijayan and C. K. Wong, "New Algorithms for the Rectilinear Steiner Tree Problem", *IEEE TCAD* 9(2) (1990), pp. 185-193.
[8] J. B. Kruskal Jr., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proc. Amer. Math. Soc.* 7(1) (1956), pp. 48-50.
[9] C. J. Alpert, A. B. Kahng, C. N. Sze and Q. Wang, "Timing-driven Steiner Trees are (Practically) Free", *Proc. DAC*, 2006, pp. 389-392.
[10] J. Cong, A. B. Kahng, G. Robins and M. Sarrafzadeh, "Provably Good Performance-driven Global Routing", *IEEE TCAD* 11(6) (1992), pp. 739-752.
[11] G. Kortsarz and D. Peleg, "Approximating Shallow-light Trees", *Proc. SODA*, 1997, pp. 103-110.
[12] S. Khuller, B. Raghavachari and N. Young, "Balancing Minimum Spanning Trees and Shortest-path Trees", *Proc. SODA*, 1993, pp. 243-250.
[13] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh and C.K. Wong, "Performance-driven Global Routing for Cell Based ICs", *Proc. ICCD*, 1991, pp. 170-173.
[14] A. Lim, S.-W. Cheng and C.-T. Wu, "Performance Oriented Rectilinear Steiner Trees", *Proc. DAC*, 1993, pp. 171-175.
[15] A. B. Kahng and G. Robins, "A New Class of Iterative Steiner Tree Heuristics with Good Performance", *IEEE TCAD* 11(7) (1992), pp. 893-902.
[16] M. Borah, R. M. Owens and M. J. Irwin, "An Edge-based Heuristic for Steiner Routing", *IEEE TCAD* 13(12) (1994), pp. 1563-1568.
[17] W. Shi and C. Su, "The Rectilinear Steiner Arborescence Problem is NP-complete", *SIAM J. Comp.* 35(3) (2006), pp. 729-740.
[18] J. Cong, K. S. Leung and D. Zhou, "Performance-driven Interconnect Design Based on Distributed RC Delay Model", *Proc. DAC*, 1993, pp. 606-611.
[19] A. B. Kahng and G. Robins, *On Optimal Interconnections for VLSI*, Kluwer Academic Publishers, 1995.
[20] M. Hanan, "On Steiner's Problem with Rectilinear Distance", *SIAM J. Appl. Math.* 14(2) (1966), pp. 255-265.
[21] L. Scheffer, Bookshelf RMST code, http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/RSMT/RMST/.
[22] L. J. Guibas and J. Stolfi, "On Computing All Northeast Nearest Neighbors in the L1 Metric", *Information Processing Letters* 17 (1983), pp. 219-223.
[23] A. Naamad, D. T. Lee and W.-L. Hsu, "On the Maximum Empty Rectangle Problem", *Discrete Applied Mathematics* 8 (1984), pp. 267-277.
[24] J. Griffith, G. Robins, J. S. Salowe and T. Zhang, "Closing the Gap: Near-optimal Steiner Trees in Polynomial Time", *Proc. TCAD*, 1994, pp. 1351-1365.
[25] L. He, S. Yao, W. Deng, J. Chen and L. Chao, "Interconnect Routing Methods of Integrated Circuit Designs", *US Patent 8386984*, Feb. 2013.
[26] S. Bose, "Methods and Systems for Placement and Routing", *US Patent 8332793*, Dec. 2012.
[27] R. F. Hentschke, M. de Oliveira Johann, J. Narasimhan and R. A. de Luz Reis, "Methods and Apparatus for Providing Flexible Timing-driven Routing Trees", *US Patent 8095904*, Jan. 2012.
[28] G. M. Furnish, M. J. LeBrun and S. Bose, "Tunneling as a Boundary Congestion Relief Mechanism", *US Patent 7921393*, Apr. 2011.
[29] G. M. Furnish, M. J. LeBrun and S. Bose, "Node Spreading Via Artificial Density Enhancement to Reduce Routing Congestion", *US Patent 7921392*, Apr. 2011.
[30] P. Saxena, V. Khandelwal, C. Qiao, P-H. Ho, J. C. Lin and M. A. Iyer, "Interconnect-driven Physical Synthesis using Persistent Virtual Routing", *US Patent 7853915*, Dec. 2010.
[31] C. J. Alpert, J. Hu and P. H. Villarrubia, "Practical Methodology for Early Buffer and Wire Resource Allocation", *US Patent 6996512*, Feb. 2006.
[32] C. J. Alpert, R. G. Gandham, J. Hu, S. T. Quay and A. J. Sullivan, "Apparatus and Method for Determining Buffered Steiner Trees for Complex Circuits", *US Patent 6591411*, Jul. 2003.
[33] C. Chu and Y.C. Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design", *IEEE TCAD* 27(1) (2008), pp.70-83.
[34] M. Elkin and S. Solomon, "Steiner Shallow-light Trees are Exponentially Lighter than Spanning Ones", *SIAM J. Comp.* 44(4) (2015), pp. 996-1025.
[35] R. Scheifele, "Steiner Trees with Bounded RC-delay", *Algorithmica* 78(1) (2017), pp. 86-109.
[36] G. Chen, P. Tu and E. F. Y. Young, "SALT: Provably Good Routing Topology by a Novel Steiner Shallow-Light Tree Algorithm", *Proc. ICCAD*, 2017.
[37] N. Viswanathan, C. J. Alpert, C. C. N. Sze, Z. Li and Y. Wei, "The DAC 2012 Routability-driven Placement Contest and Benchmark Suite", *Proc. DAC*, 2012, pp. 774-782.
[38] J. Lu, H. Zhuang, P. Chen, H. Chang, C.-C. Chang, Y.-C. Wong, L. Sha, D. Huang, Y. Luo, C.-C. Teng and C.-K. Cheng, "ePlace-MS: Electrostatics based Placement for Mixed-Size Circuits", *IEEE TCAD* 34(5) (2015), pp. 685-698.

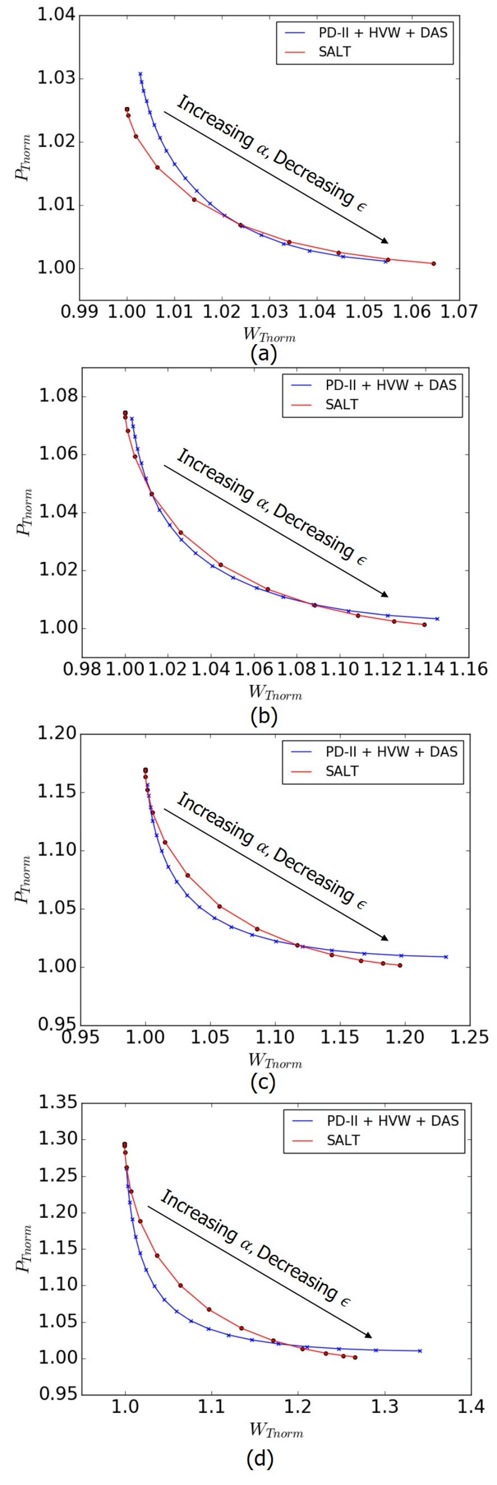

Figure 7: Normalized WL and PL for our metaheuristic and SALT on nets with $|V|$ = (a) 4 to 7, (b) 8 to 15, (c) 16 to 31 and (d) 32+.

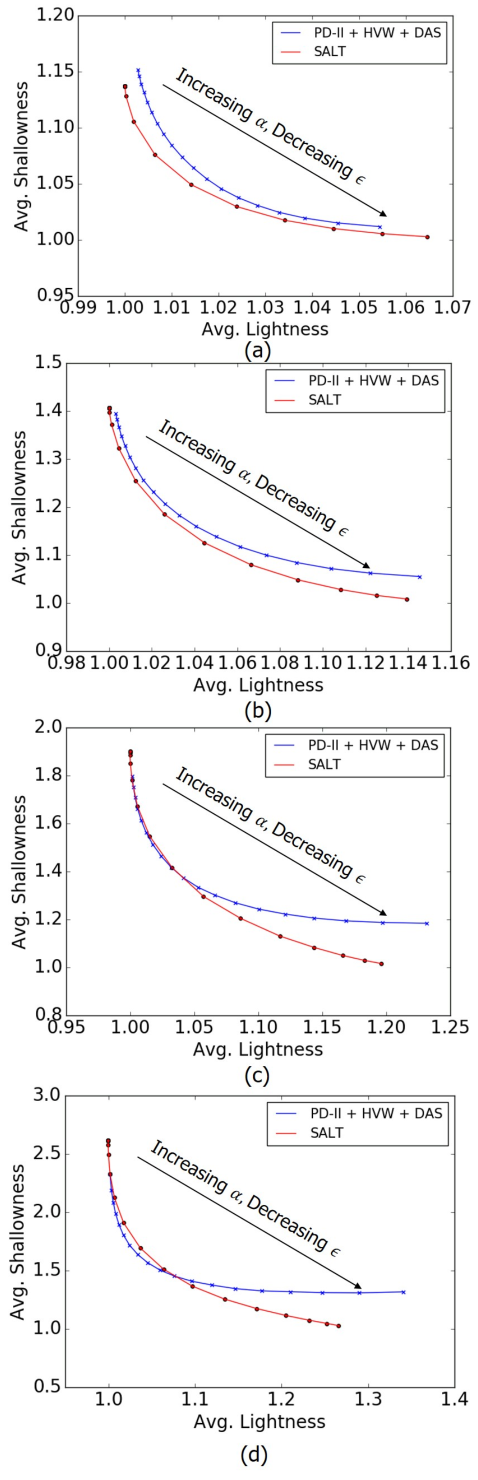

Figure 8: Average shallowness and lightness for our metaheuristic and SALT on nets with $|V|$ = (a) 4 to 7, (b) 8 to 15, (c) 16 to 31 and (d) 32+.