# MALT: ML Assisted Shallow-Light Tree Construction

LIANG XIAO, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
JINWEI LIU, Hong Kong Baptist University, Hong Kong, Hong Kong
LIXIN LIU, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
QIJING WANG, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
EVANGELINE YOUNG, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
MARTIN WONG, Hong Kong Baptist University, Hong Kong, Hong Kong

Timing is a critical issue in electronic design automation (EDA). To reduce the delay of a net, an important strategy is to minimize the path lengths from the source to the sinks. However, minimizing the path lengths will inevitably sacrifice the total wirelength. To balance the two objectives, researchers use shallow-light tree (SLT) to model and optimize the problem. In this article, we introduce MALT, a novel approach that uses a neural network to guide the construction of Steiner shallow-light trees. The constructed trees are further refined by a dynamic programming-based branch merging algorithm, which improves the wirelength without sacrificing the path lengths of any sinks. Our experimental results demonstrate that the proposed framework achieves significant improvements over both state-of-the-art traditional SLT generation algorithms and existing machine learning enhanced methods.

CCS Concepts: • **Hardware** → *Wire routing*; **Physical design (EDA)**; • **Computing methodologies** → *Machine learning;*

Additional Key Words and Phrases: EDA, shallow-light tree, machine learning

**ACM Reference Format:**
Liang Xiao, Jinwei Liu, Lixin Liu, Qijing Wang, Evangeline Young, and Martin Wong. 2026. MALT: ML Assisted Shallow-Light Tree Construction. *ACM Trans. Des. Autom. Electron. Syst.* 31, 4, Article 79 (March 2026), 15 pages.
https://doi.org/10.1145/3789668

This research was partially conducted by ACCESS AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government.
Authors' Contact Information: Liang Xiao (corresponding author), Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: lxiao23@cse.cuhk.edu.hk; Jinwei Liu, Hong Kong Baptist University, Hong Kong, Hong Kong; e-mail: jinweiliu@comp.hkbu.edu.hk; Lixin Liu, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: lxliu@cse.cuhk.edu.hk; Qijing Wang, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, NT, Hong Kong; e-mail: qjwang21@cse.cuhk.edu.hk; Evangeline Young, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: fyyoung@cse.cuhk.edu.hk; Martin Wong, Hong Kong Baptist University, Hong Kong, Hong Kong; e-mail: mdfwong@hkbu.edu.hk.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).
ACM 1084-4309/2026/03-ART79
https://doi.org/10.1145/3789668

## 1 Introduction

Routing is a critical part of **electronic design automation** (**EDA**), which aims to efficiently connect all pins of a net with high quality. Among all the modules in routing, Steiner tree construction is crucial and widely studied [2, 4–6, 8, 9, 12, 21], as it provides a preliminary result to guide the subsequent steps of routing. As timing becomes increasingly critical, researchers are focusing more on minimizing wire delay in nets. Thus, in the initial routing stage, path lengths from source to sinks become another crucial optimization objective alongside total **wirelength** (**WL**).

In the direction of minimizing total WL (**lightness**), researchers try to construct **Rectilinear Steiner Minimum Tree** (**RSMT**) for nets, although it is an NP-hard problem. GeoSteiner [20] provides an optimal solution that has exponential complexity. To reduce the complexity, many efforts have been made to give an approximation, such as heuristic FLUTE [6]. ML-based works are also proposed in this direction [9, 14]. To shorten the path from source to sinks (**shallowness**), researchers attempt to construct trees for nets towards **Rectilinear Steiner Minimum Arborescence** (**RSMA**), which is the shortest WL tree under the constraint that distances from all sinks in a net to the source are equal to the Manhattan distance. To tackle the RSMA problem, which is also NP-hard, a heuristic CL algorithm [7] was proposed, providing a near-optimal solution in $O(n\log(n))$ time. To better balance total WL and source-sink path lengths, SALT [4] proposes an efficient algorithm to generate a **shallow-light tree** (**SLT**) and gives a much tighter bound compared to previous works. PD-II [1] is another algorithm proposed to trade off between WL and average source-sink path length.

Besides the heuristics mentioned above, some recent works adopt learning-based methods to optimize lightness and shallowness. In the combinatorial optimization domain, Pointer networks [18] give researchers a promising example to solve combinatorial optimization problems. Inspired by it, REST [14] adapted it into the RSMT generation task without considering source-sink path lengths. Extending the principle of Pointer networks, Yang et al. [21] proposed their **reinforcement learning** (**RL**) model which balances between total WL and the longest source-sink path to better address timing concerns. However, they only pay attention to the longest path and overlook the timing issues of the other sinks that are closer to the source. For example, in Figure 1(a), both sink $A$ and $B$ deviate from their shortest paths to the source. However, sink C dominates the longest path, so no extra optimization is done for sink $A$ and $B$ in their approach.

Since there are different heuristics that optimize lightness and shallowness, Treenet[13] trains a neural network to choose between two heuristics, SALT and PD-II, and utilize the potential of the two heuristics. However, Treenet can never outperform both algorithms.

Although SALT remains the state-of-the-art algorithm in SLT construction. We observed that it still has a lot of space for improvement. SALT corrects out-of-constraint points in a greedy way and lacks consideration of global information. Hence, we propose an algorithm called MALT that uses a machine learning model to guide the construction. Our contributions are listed below.

- We design a machine learning model to encode tree structures and identify critical points to guide SLT construction process.
- We introduce a customized reward function with a moving average baseline that can resolve our two-objective combinatorial optimization problem smoothly.
- We propose a **dynamic programming** (**DP**)-based branch merging algorithm to further improve the lightness and shallowness.
- Based on the methods we proposed, our algorithm can improve the lightness and shallowness compared to SALT, and surpass the state-of-the-art AI-assisted SLT generation method.

The rest of the paper is organized as follows. Section 2 introduces some preliminaries and related works. We introduce our ML-Assisted SLT construction algorithm in Section 3 and a DP-based

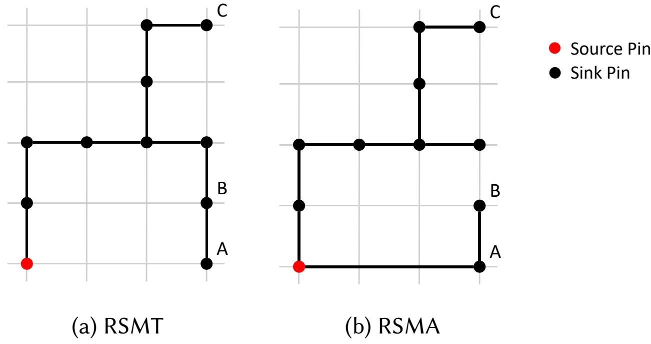


Fig. 1. RSMT and RSMA.

branch merging algorithm in Section 4. Section 5 shows our experimental results. Last but not least, we draw the conclusions in Section 6.

## 2 Problem Definition and Related Works

### 2.1 Problem Definition

Since our algorithm is a **rectilinear Steiner tree** (**RST**) generation algorithm, we first introduce the concept of RST. An RST is a tree connecting a set of points (pins) in the plane, where all edges must be horizontal or vertical. This constraint reflects the routing requirements in VLSI design, where interconnections are typically limited to horizontal and vertical wires. Steiner points are additional vertices that can be introduced to reduce the total WL. The **Rectilinear Steiner Minimal Tree** (**RSMT**) is the RST with minimum total edge length.

For a given tree $T = \{V, E\}$ the definition of lightness $\beta$ and shallowness $\alpha$ are:

$$\begin{aligned} \alpha &= \max\{[d_T(r,v)/d_G(r,v)] | v \in V \backslash \{r\}\} \\ \beta &= w(T)/w(RSMT(V)) \end{aligned}, \tag{1}$$

where $r \in V$ is the root of the tree. In the equation, $d_T(r, v)$ is the path length from sink $v$ to the source $r$. $d_G(r, v)$ is the optimal distance to the source, where in RST, the optimal distance between $r$ and $v$ is their Manhattan distance. For an $(\bar{\alpha}, \bar{\beta})$-SLT, the distance from any pin to the source does not exceed $\bar{\alpha}$ times the optimal distance, and its total WL $w(T)$ does not exceed $\bar{\beta}$ times the optimal WL $w(RSMT(V))$ ($\alpha \leq \bar{\alpha}, \beta \leq \bar{\beta}$). For example, Figure 1(a) is a shortest WL tree, but for point $B$, its path length to the source is 6, while the optimal distance is 4. For point $A$, its path length divided by the optimal distance is $\frac{7}{3}$.. Therefore, it is a $(\frac{7}{3}, 1)$ SLT. The WL of Figure 1(b) is 11, while the shortest total WL is 10 in the RSMT. Thus, it is a $(1, \frac{11}{10})$ SLT. In this article, we use FLUTE to give a reference for the optimal WL $w(RSMT(V))$.

The input and output of our algorithm are the same as SALT [4], including three parts: the set of the positions of the pins $P \in \mathbb{R}^{n\times 2}$, the source point of the net $r \in P$, and the shallowness constraint $\epsilon$.

The output of the algorithm is a Steiner tree to connect all points, denoted by $T = \{V \in \mathbb{R}^{(n+k)\times 2}, E \in \mathbb{Z}^{(n+k-1)\times 2}\}$, where $V$ includes pins $P \in \mathbb{R}^{n\times 2}$ and the added steiner points $S \in \mathbb{R}^{k\times 2}$. $E$ is the edges in the tree. Each edge in $E$ records the index of the two points it connects. A tree is called "legal" if and only if its shallowness measure $\alpha$ satisfies the input constraint $\alpha \leq 1 + \epsilon$.

Overall, SLT construction is a bi-objective problem that optimizes the trade-off between shallowness and lightness, aiming to minimize overall WL while ensuring that the distance from each sink to the source remains close to its optimal value.

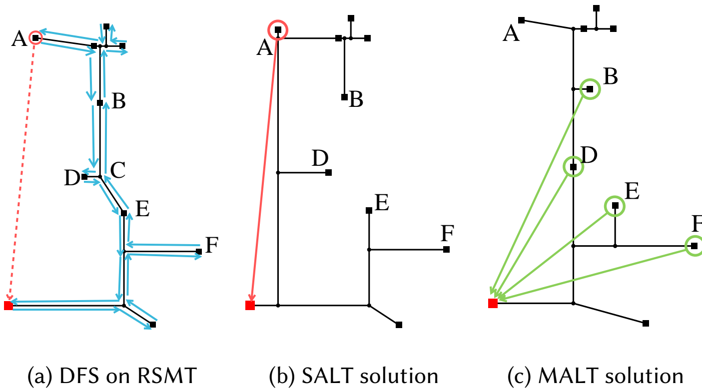


Fig. 2. An example of suboptimal DFS based detection ($\epsilon = 0.5$). (a) DFS process on an RSMT (shallowness $\alpha = 1.5995$, lightness $\beta = 1$). (b) Solution of SALT ($\alpha = 1.3867$, $\beta = 1.2435$). Illegal points detected during DFS are circled red. (c) Solution given by MALT($\alpha = 1.3316$, $\beta = 1.0826$). Points predicted by our model to be reconstructed are circled green.

## 2.2 Related Works

To construct a tree with shallowness no greater than $1 + \epsilon$, a straightforward approach is to directly connect points violating this constraint (illegal points) to the source pin. The KRY heuristic [10] implements this idea using DFS to traverse a minimum spanning tree. When finding an illegal point during the DFS, KRY connects it to the source and updates the tree structure accordingly.

SALT [4] also employs the same method to detect illegal points, but it further adapted the method to Steiner trees. Figure 2(a) illustrates such a DFS process. Starting from the source of an RSMT, SALT traverse the tree along the blue arrows to find illegal points that break the shallowness constraint. For example, in Figure 2(b), $A$ is the only detected illegal point, and it will be connected to the source directly. Note that the updated path will influence the points along the DFS traversal. For example, we might disconnect $B$ from its original parent to save WL. In that case, $B$'s only child will become its new parent. After the traversal, SALT utilizes a near-optimal RSMA algorithm to connect all the illegal points to the source, rather than using $n$ separate paths to directly connect them to the source. Additionally, SALT implements some post-processing algorithms for further improvement. For instance, in Figure 2(b), point $D$ will be connected to the $A$-source path instead of to point $E$, thereby reducing the overall WL.

From above, we see that there are mainly two stages for the two algorithms: (1) *Detect*: Both KRY and SALT use a common DFS traversal to detect the points that violate the shallowness constraint. Afterward, these points' connections to their parents are ripped up. (2) *Repair*: KRY repairs the tree by connecting all the points to the source directly, while SALT improves the process by using an RSMA to reconnect these points to the source. We summarize this two-stage SLT construction as a *Detect-Repair Framework*.

## 2.3 Limitations of Previous Works

Although the aforementioned Detect-Repair Framework shows its effectiveness in SLT construction, we find that the detect stage still has a lot of room for improvement. In some cases, it is not necessary to rip up all the illegal points to meet the shallowness constraint. Sometimes ripping up some illegal points can lead to a drastic change of the tree structure, which might be followed by a significant WL degradation. Instead, we can adjust some other parts of the tree with a smaller cost to meet the shallowness constraint, even if these parts are already legal.

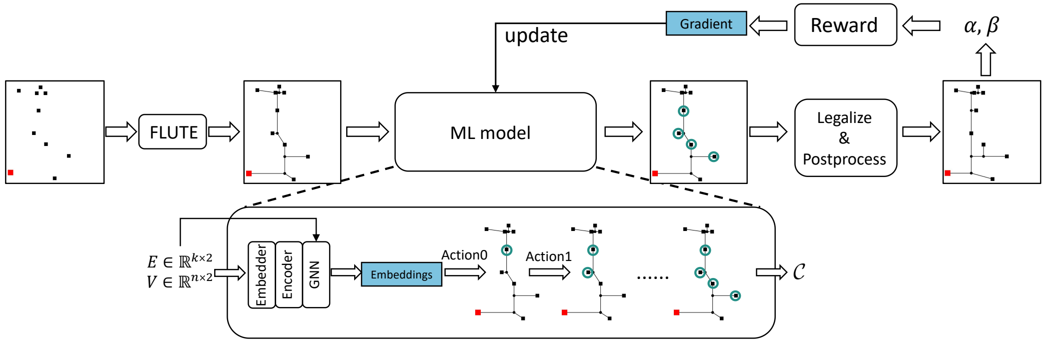


Fig. 3. Overall flow. Given the RSMT generated by FLUTE, our machine learning model encodes the tree structure to node embeddings. An encoder and GNN further learn node relationships. Our point selector then sequentially adds points to the Critical Set $\mathcal{C}$. After legalization and post-processing, we finally get a SLT. The lightness $\beta$ shallowness $\alpha$ is then used to compute the reward and the gradient.

Figure 2 is an example of a 10-pin net from our benchmark. In Figure 2(b), the only illegal point in the tree is point $A$. Then in SALT, it is ripped up from the connection to its parent node, and will be reconnected to the source. The result tree (as shown in Figure 2(b)) will have shallowness $\alpha = 1.3867$, lightness $\beta = 1.2435$. However, we can also rip up points $\{B, D, E, F\}$ instead as shown in Figure 2(c). By ripping up these four points, and using an RSMA to connect them to the source point, the final tree will have shallowness $\alpha = 1.3316$ and lightness $\beta = 1.0826$, which is much shallower and much lighter than the result of SALT shown in Figure 2(b). In this example, reconstructing point $A$'s connection would significantly alter the original RSMT structure, while reconstructing the connection of $\{B, D, E, F\}$ disturbs the original RSMT structure less but also meets the constraint $\epsilon = 0.5$. Therefore, compared to reconstructing point $A$'s connection, choosing the four points is a better solution. Note that there are multiple solutions that are better than reconstructing $A$'s connection. For example, selecting points $\{D, E, F\}$ may achieve the same effect as $\{B, D, E, F\}$.

We now understand that when reconstructing an illegal light tree to meet the shallowness constraint, multiple solutions exist. Among these solutions, reconstructing the connections of certain points may be more beneficial than others. By checking many cases, we discovered that the example shown here is only one of many situations where illegal point detection by DFS is suboptimal. This makes it even more difficult to design a single heuristic to optimize.

To determine a good set of points for reconstruction, merely identifying illegal points is insufficient. We need to comprehensively consider the relationship of multiple points. Machine learning is widely recognized as an effective approach for automatically extracting global features and relationships. Therefore, we employ a learning-based model to predict a better set of points for connection reconstruction. For convenience, we refer to this set of points as the "**Critical Set**".

## 3 ML Guided SLT Construction

For an $n$ pin net, the number of all possible Critical Sets is $O(2^n)$, and for one net, the optimal SLT can result from more than one critical set like the example in Section 2.3. Therefore, it's not feasible to enumerate all possibilities and find the "best" choice to teach the model by supervised learning. Instead, we adopt RL to handle this combinatorial optimization problem.

### 3.1 Overall Flow

The overall flow of MALT is shown in Figure 3. We first utilize FLUTE to generate an RSMT. In this figure, $n$ is the number of points in RSMT, including pins and Steiner points, and $k$ is the

number of edges. We then predict a Critical Set $\mathcal{C}$ based on it. After obtaining the Critical Set $\mathcal{C}$, we apply the legalization and post-processing step introduced in Section 3.4 to obtain SLTs with their shallowness $\alpha$ and lightness $\beta$, and use the reward strategy introduced in Section 3.5 to train our predictor. The model will update its parameters to maximize the probability of generating a high reward result. During inference, an extra optimization introduced in Section 4 is used to further improve the quality.

In the prediction step, we model the process of the Critical Set $\mathcal{C}$ as a sequence prediction problem. Specifically, we select pins one by one, forming a sequence in descending order of importance $\mathcal{P} = (P_1, P_2, \dots, P_n)$, which is a permutation of point indices from 1 to $n$. We use the points with a higher importance rank than the source pin as our Critical Set, which can be formulated as $\mathcal{C} = \{P_s \mid s < h; P_h \text{ is source point}\}$. At step $i$, our model predicts the selection probability $p_i(V_j)$ for each unselected pin $V_j$. In the training phase, we use Monte Carlo sampling to randomly sample from the probability distribution $p$ for exploration and sequentially generate the importance sequence $\mathcal{P}$. In the inference phase, we select the point with the highest probability.

This decision process can also be seen as a Markov decision process [15], where each state includes the input of the model and the sequence of the selected points. An action is to select a new point to the sequence. Since we can only evaluate the shallowness and lightness of the tree after deciding the whole critical set, the reward is given only once for the final step. More specifically, it is computed as a weighted sum of shallowness and lightness with negative weights.

The key step in the process is predicting the probability $p_i(V_j)$ of selecting pins. We use the model introduced in Section 3.2 to extract the features of the given pins, and use the selector in Section 3.3 to iteratively generate probabilities at each step.

### 3.2 Embedder and Encoder

The embedder's inputs are the positions of points, $V \in \mathbb{R}^{n\times 2}$ and edges $E \in \mathbb{R}^{k\times 2}$, where $n$ is the number of nodes and $k$ is the number of edges. Our feature extractor first extracts important node features from $V$ and $E$. Specifically, we select four features for point $v \in V$: $\{x, y, d_T(r, v), (1 + \epsilon)d_G(r, v)\}$, where $x, y$ are coordinates of points, $d_T(r, v)$ is the current path length to the source, $d_G(r, v)$ is the Manhattan distance to the source, and $(1 + \epsilon)d_G(r, v)$ is the limit of the path length. Afterward, our embedder will use two fully connected layers to convert these four features into the hidden space $m$. The output of the embedder is $F \in \mathbb{R}^{n\times m}$. In our implementation, we set the hidden dimension $m$ to 64.

The structure of our encoder [17] is shown in Figure 4(b). It consists of three encoder blocks. In each block, we first use multi-head attention to capture the relationship between points, following a feed forward module. We add a batch normalization after the attention and the feed-forward module to increase stability. Additionally, we incorporate residual connections for these two modules. Residual connections sum the input and output, as denoted by the symbol + in Figure 4(b). The function of the multi-head attention block is given as follows:

$$\begin{aligned} &\text{SingleHead}(Q, K, M) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_s}}\right)M \\ &S_j = \text{SingleHead}(F_i W_{Q,j}, F_i W_{K,j}, F_i W_{M,j}) \\ &F_M = \text{Multi-Head}(F_i) = \text{Concat}(S_1, \dots, S_h)W_m \end{aligned}, \tag{2}$$

where $F_i \in \mathbb{R}^{n\times m}$ is the input feature of the $i$th encoder block. For single head $j$, the input feature $F_i$ is first projected to matrices $Q, K, M$ with learnable weight $W_{Q,j}, W_{K,j}, W_{M,j} \in \mathbb{R}^{m\times d_s}$ ($d_s = 16$), then $S_j \in \mathbb{R}^{n\times d_s}$ is calculated with $Q, K, M$. Multi-head attention concatenates outputs of $h$ single

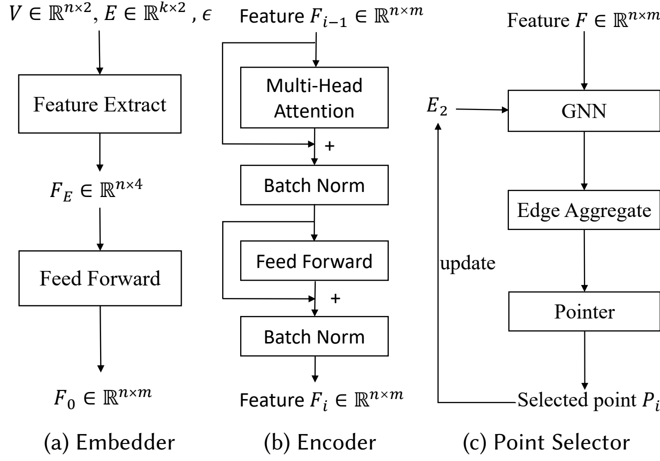


Fig. 4. Model structures.

heads and multiplies it by a learnable weight $W_m \in \mathbb{R}^{hd_s\times m}$. The output of the multi-head attention, $F_M \in \mathbb{R}^{n\times m}$, retains the same shape as the input.

There is a feed forward block following each multi-head attention block, it is applied to each row of $F_M$:

$$\text{FeedForward}(x^T) = \max(0, x^T W_1 + b_1^T)W_2 + b_2^T, \tag{3}$$

where $x^T$ ($x \in \mathbb{R}^m$) is a input row of $F_M$, and $W_1 \in \mathbb{R}^{m\times d_h}$ ($d_h = 128$), $b_1 \in \mathbb{R}^{d_h}$, $W_2 \in \mathbb{R}^{d_h\times m}$ and $b_2 \in \mathbb{R}^m$ are all trainable parameters.

After the encoder, we use a **graph neural network** (**GNN**) to integrate information from a node's neighbors. The neighbors are given by the edges in the RSMT. In our implementation, we use three GNN layers, which means that information from points within three hops are gathered. The type of GNN we use is **Graph Convolutional Network** (**GCN**) [11] , which can be formulated as follows.

$$h_u^{(l+1)} = \text{UPDATE}^{(l)}\left(h_u^{(l)}, \text{AGG}^{(l)}\left(h_v^{(l)}, \forall v \in \mathcal{N}(u)\right)\right), \tag{4}$$

where $h_u^{(l)}$ denotes the feature of node $u$ at the $l$th layer. $\mathcal{N}(u)$ is the neighbor nodes of node $u$. AGG is a message aggregating method that sums up all neighbors' features and normalizes the sum by dividing by $\sqrt{|\mathcal{N}(u)|}$. UPDATE is a feed forward function.

### 3.3 Point Selector

The structure of our point selector is illustrated in Figure 4(c). The core module is the pointer, which selects a point according to node embeddings. After each selection, we use a GNN to update the embeddings. Its input is the encoder's output $F_i \in \mathbb{R}^{n\times m}$ and edges $E_2$, which contain edges connecting the source and selected points. We also use an edge aggregation operation before the pointer to enhance features, which concatenates points' features with their parents' features (the parent relationship is given by the RSMT tree structure). The enhanced feature for point $i$ is $e_i = [x_i, x_{pa}]$, where $x_i, x_{pa} \in \mathbb{R}^m$ are the features of the $i$th point and its parent node after the GNN. By using the RSMT structure in edge aggregation, we merge some information of the original RSMT.

Then comes the pointer, which can choose one point according to the points' features. Its details are provided as follows:

$$
\begin{aligned}
p &= \text{softmax}(C \times \tanh(l)) \\
\text{where } l &= [l_1, l_2, ..., l_n]^T, \\
l_i &= \begin{cases} -\infty & \text{if point } i \text{ is selected} \\ w^T \tanh(W_1 e_i + W_2 q) & \text{otherwise} \end{cases},
\end{aligned} \tag{5}
$$

where $p \in \mathbb{R}^n$ is the points' probability of being selected, $W_1 \in \mathbb{R}^{d_q \times 2m}, W_2 \in \mathbb{R}^{d_q \times 2m}, w \in \mathbb{R}^{d_q}$ are all trainable parameters ($d_q = 64$). $C$ is a constant 10. This formulation incorporates information from both the query $q \in \mathbb{R}^{2m}$ and the embedding $e_i \in \mathbb{R}^{2m}$. The score $l_i$ for feature $i$ is computed by applying the weight vector $w$ to the combined representation. Finally, a softmax function is applied to convert these scores into probabilities for point selection. By setting the scores of points have been selected to negative infinity, we can filter these points and prevent revisits. The query $q$ is initially a one vector. In subsequent selections, it becomes the node embedding $e$ of the point previously chosen, allowing the network to incorporate the context of prior selections.

In the training phase, we use Monte Carlo sampling to randomly sample from the probability distribution $p$ for exploration and sequentially generate the importance sequence $\mathcal{P}$. In the inference phase, we select the point with the highest probability.

### 3.4 Legalization

After predicting importance sequence $\mathcal{P}$ and its corresponding Critical Set $\mathcal{C}$, we proceed to the Legalize and Postprocess phase. In this stage, Points in the Critical Set will be directly connected to the source, so their distance to the source point is marked as the optimal distance. Some shallowness violations will be fixed in this process, but not all. To ensure the legality of our tree, we will traverse the RSMT. If any points still violate the shallowness constraint, we disconnect them from their parent nodes as well. These identified illegal points, along with those in the critical set $\mathcal{C}$, are then connected to the source using the near-optimal RSMA algorithm CL [7]. Finally, we use the same post-process as SALT [4] after this legalization.

Our approach integrates efficiently with SALT's existing framework, as SALT already employs DFS traversal to identify constraint-violating points and uses RSMA to reconnect them. Our enhancement is straightforward to implement, merely extending this reconnection process by pre-selecting pins from the Critical Set $\mathcal{C}$ for source connection. This can be viewed as a practical plug-in module where our ML model identifies the Critical Set to guide the SLT construction.

### 3.5 Reward Function

In one step of training, for net $i$, the relationship of our objectives $\alpha_i, \beta_i$ and the Critical Set $\mathcal{C}$ can be formulated as:

$$
\alpha_i, \beta_i = Eval(T(V_i), \mathcal{C}) = Eval(T(V_i), model_\theta(T(V_i))), \tag{6}
$$

where $Eval$ is the legalization, post-process, and evaluation stages. $V_i$ is the set of net $i$'s nodes. $T(V_i)$ is an RSMT generated by FLUTE [6]. $\alpha_i, \beta_i$ are then used in the reward function and to calculate the gradient of the model.

A commonly used RL framework is the Actor-Critic framework. This framework uses the following function to calculate an advantage $J$:

$$
J(\theta \mid V) = \frac{1}{B} \sum_{i=1}^{B} (b(V_i) - L(r_i, V_i) p(r_i \mid V; \theta), \tag{7}
$$

where $B$ is batch size, $b_i$ is a baseline given by the critic model for current input $V$, $L(r_i, V_i)$ is the performance that need to be minimize, and $p(r_i \mid V; \theta)$ is the probability of the taken action $r_i$ with the model parameter $\theta$ and input $V$. In the equation, $b(V_i) - L(r_i, V_i)$ is the reward for case $i$. By maximizing $J$, we can get the gradient for our parameter $\theta$. When the reward is positive, the gradient updates model parameters in the direction that increases the probability $p$, and conversely, when the reward is negative, the gradient pushes parameters toward reducing the probability of taking that action.

In our task, the action we take is determining an importance sequence $\mathcal{P}$. Then using the probability chain rule, we can define the action probability $p(r_i \mid V; \theta)$ as follows, which is the product of conditional probabilities for each selection in the sequence:

$$p(\mathcal{P} \mid V; \theta) = \prod_{s=1}^{n} p(P_s \mid P_1, \ldots, P_{s-1}, V; \theta) \quad . \tag{8}$$

Usually, people train a critic model to predict a baseline. However, in our task, the final tree structure between nets can be very different. For some nets, the RSMT generated by FLUTE may happen to be optimal in shallowness, while some others need many modifications to meet the shallowness constraint. This brings an extra challenge to designing a critic model to predict a baseline.

In our model, we train 150 steps for every batch of nets. For step $t$, we use an exponential moving average strategy to dynamically update the baseline and the performance $L$ as follows, which is formulated as follows:

$$\begin{gathered} \overline{\alpha} = \frac{1}{B}\sum_{i=1}^{B} \tilde{\alpha}_i,\ \overline{\beta} = \frac{1}{B}\sum_{i=1}^{B} \tilde{\beta}_i,\ \gamma = \frac{\overline{\beta} - 1}{\overline{\beta} - 1 + \overline{\alpha} - 1} \quad , \\ L_{t,i} = \gamma \alpha_{t,i} + (1-\gamma)\beta_{t,i}\ ,\ b_{t,i} = \lambda b_{t-1,i} + (1-\lambda) L_{t-1,i} \end{gathered} \tag{9}$$

where $B$ is the batch size. $\beta_{t,i}$ and $\alpha_{t,i}$ denote the WL and shallowness of net $i$ at step $t$. $\tilde{\beta}_i$ and $\tilde{\alpha}_i$ denote the WL and shallowness of the reference results of SALT. $b_{t,i}$ represents the baseline, which is a weighted combination of the last step performance and the moving baseline. The initial baseline $b_{0,i}$ is the performance of the model at the beginning of the batch. $\lambda$ is the discount factor ($\lambda = 0.9$).

Note that when calculating performance $L$, we dynamically change weight $\gamma$ for different batches. The intuition behind, that is, given as follows. For large $\epsilon$ (loose shallowness constraint) and small net, WL is near optimal, ripping up a point can have a high risk of disturbing the original tree structure, which may lead to WL degradation, and vice versa. Thus, we adopt $\gamma$ to calculate $L$, which balances the two objectives and considers the fact that the optimal value of $\alpha$ and $\beta$ is 1.

With the moving baseline and the performance defined in Equation (9), we can rewrite Equation (7) to

$$J_t = \frac{1}{B}\sum_{i=1}^{B} (b_{t,i} - L_{t,i}) p(\mathcal{P}_i | V_i; \theta). \tag{10}$$

Therefore, we can define an advantage function as shown in Equation (10), which enables our model to evaluate current performance against a moving baseline that incorporates historical performance metrics. When performance improves relative to this historical baseline, the model strengthens the probability of selecting similar actions, while performance degradation leads to parameter updates that reduce the likelihood of taking such actions.

## 4 DP-based Branch Merging Algorithm

In this section, we introduce an algorithm to further improve the shallowness and lightness of a tree. We observe that sometimes we can optimize a tree by merging some branches. For example, by

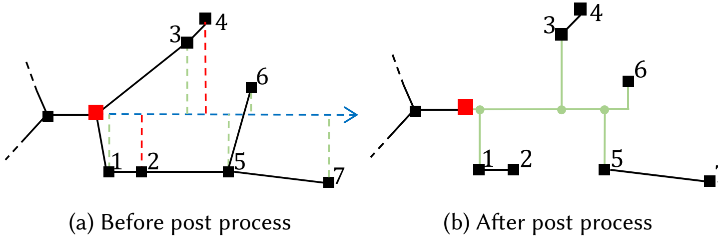


Fig. 5. An example of searching on right direction. (a) A suboptimal tree. (b) The modified tree after the post-process. The green part are generated by our DP.

merging the top right branch (containing point 3, 4) and the branches of (1, 2, 5, 6, 7) in Figure 5(a), we have the chance to further optimize the lightness and shallowness as shown Figure 5(b). In light of this, we developed a simple yet effective heuristic with DP to merge branches in a tree. Its general idea is that we construct trunks from the source and try to connect points to their projection on the trunk if WL can be reduced. Note that we construct trunks in four directions, including {*left*, *top*, *right*, *and bottom*}.

Without loss of generality, we take *right* direction as an example to describe our algorithm as shown in Figure 5(a). Suppose the source point is $r$ and the point sequence we search along the trunk (denoted by blue arrow) is $\mathcal{V} = [v_1, v_2, \dots, v_n]$ (can be obtained by sorting in ascending order according to their x coordinates), then we can search along the trunk from $v_1$ to $v_n$ as follows:

$$\begin{aligned} gain^{(i)} &= D(v_i, pred(v_i)) - |v_i.y - r.y| \\ sum^{(i)} &= sum^{(i-1)} + \max(0, gain^{(i)}) \\ reduction &= \max_{i \in [1,n]} \{sum^{(i)} - (v_i.x - r.x)\} \end{aligned}, \tag{11}$$

where $i$ is the index of the searching sequence, $pred(v_i)$ is $v_i$'s parent, and $D(u, v)$ is the Manhattan distance between $u$ and $v$. When $gain^{(i)}$ is positive, $v_i$ is closer to the trunk compared to its parent, marked by green lines in Figure 5(a) (Conversely, red lines denote negative gains). Then, we sum up all positive gains along the trunk, and calculate the final wire reduction by considering WL spent on the trunk $v_i.x - r.x$. Suppose the DP achieves an optimal reduction in index $k$ (like point "6" in Figure 5), then we directly connect $v_k$ to the source, and connect all pins $\{v_j | j < k, gain^{(j)} > 0\}$ to the trunk. Finally, we can obtain a new tree as Figure 5(b).

Pins on the newly constructed branches always have the Manhattan distance to the source, and we only confirm the transformation when there is some WL reduction. So when this algorithm is triggered on a net, it can guarantee to return a lighter tree with no degradation on shallowness.

## 5 Experiments

We implement our machine learning model and our DP algorithm using Python and run our experiments on a 64-bit Linux workstation with Intel Xeon Gold 6326 CPUs (2.90 GHz) and NVIDIA A800 GPUs. We use the same superblue benchmark [19] as SALT [4] and TreeNet [13]. Benchmark details are shown in Table 1. On small nets, trees generated by SALT are already good enough, and the improvement space is small. Therefore, in this section, we only use our DP method on these small nets and use our machine-learning model on nets with 8+ pins. We used a 4-fold cross-validation strategy. i.e., when testing one quarter, three other quarters of the data are used for training.

Table 1. Statistics of Superblue Benchmark

| | Small | Medium | Large | Huge |
|---|---|---|---|---|
| $\|V\|$ | 4–7 | 8–15 | 16–31 | 32+ |
| #nets | 533,029 | 128,463 | 46,486 | 20,853 |

Table 2. Detailed Experimental Results Compared with SALT

| $\epsilon$ | | 0.0500 | | 0.1000 | | 0.2000 | | 0.4000 | | 0.8000 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\|V\|$ | Method | $\alpha$ | $\beta$ | $\alpha$ | $\beta$ | $\alpha$ | $\beta$ | $\alpha$ | $\beta$ | $\alpha$ | $\beta$ |
| Small | SALT | 1.0016 | 1.0072 | 1.0041 | 1.0063 | 1.0098 | 1.0047 | 1.0210 | 1.0024 | 1.0359 | 1.0006 |
| | Ours | **1.0015** | **1.0069** | **1.0039** | **1.0062** | **1.0095** | **1.0047** | **1.0206** | **1.0024** | **1.0357** | **1.0006** |
| | Imp.(%) | 6.1789 | 3.9177 | 4.9880 | 2.5308 | 3.5850 | 1.1898 | 1.7963 | 0.3675 | 0.5874 | 0.2951 |
| Medium | SALT | 1.0116 | 1.0607 | 1.0296 | 1.0522 | 1.0676 | 1.0388 | 1.1343 | 1.0220 | 1.2261 | 1.0077 |
| | Ours | **1.0087** | **1.0585** | **1.0240** | **1.0510** | **1.0591** | **1.0378** | **1.1218** | **1.0215** | **1.2207** | **1.0072** |
| | Imp.(%) | 25.2254 | 3.5095 | 19.0142 | 2.3866 | 12.6349 | 2.5278 | 9.3104 | 2.2961 | 2.4253 | 6.3635 |
| Large | SALT | 1.0206 | 1.1083 | 1.0523 | 1.0933 | 1.1188 | 1.0703 | 1.2354 | 1.0419 | 1.4125 | 1.0168 |
| | Ours | **1.0160** | **1.1060** | **1.0437** | **1.0916** | **1.1051** | **1.0684** | **1.2166** | **1.0401** | **1.3989** | **1.0153** |
| | Imp.(%) | 22.1669 | 2.1590 | 16.3820 | 1.8139 | 11.5160 | 2.6561 | 8.0256 | 4.4907 | 3.2865 | 9.1526 |
| Huge | SALT | 1.0317 | 1.1379 | 1.0730 | 1.1165 | 1.1502 | 1.0852 | 1.2839 | 1.0498 | 1.4925 | 1.0195 |
| | Ours | **1.0268** | **1.1373** | **1.0658** | **1.1160** | **1.1398** | **1.0836** | **1.2660** | **1.0477** | **1.4725** | **1.0185** |
| | Imp.(%) | 15.3184 | 0.4488 | 9.8948 | 0.4435 | 6.9188 | 1.8306 | 6.3120 | 4.1273 | 4.0725 | 5.2790 |
| All | SALT | 1.0075 | 1.0377 | 1.0188 | 1.0324 | 1.0426 | 1.0242 | 1.0850 | 1.0139 | 1.1464 | 1.0051 |
| | Ours | **1.0060** | **1.0367** | **1.0161** | **1.0318** | **1.0383** | **1.0236** | **1.0788** | **1.0135** | **1.1424** | **1.0048** |
| | Imp.(%) | 19.8778 | 2.5603 | 14.7355 | 1.8975 | 10.1362 | 2.3328 | 7.2601 | 3.3162 | 2.7157 | 7.1474 |

### 5.1 Quality Analysis

In the domain of SLT construction [1, 3, 4, 10, 13, 16], SALT, PD-II [1], and TreeNet represent the state-of-the-art approaches. Since our algorithm gives SALT a guide for SLT construction, we will primarily compare our approach with SALT. We also compare with TreeNet since it is the leading AI-enhanced SLT construction method. According to the experimental results reported in TreeNet, TreeNet outperforms both SALT and PD-II. Therefore, we do not include a comparison with PD-II [1].

In Table 2, we test our algorithm with different shallowness constraints $\epsilon$ on the benchmark. Since the optimal values of shallowness $\alpha$ and lightness $\beta$ are all 1, we subtract 1 before comparing. For example, 1.05 to 1.04 is a 20% improvement, i.e., $((1.05-1)-(1.04-1))/(1.05-1) = 20\%$. On average, our model can improve shallowness ranging from 2.72% to 19.88% and improve lightness from 1.90% to 7.15% compared to SALT. Improvement on small nets is not as significant because they are already near-optimal (This is especially true for 4-degree nets, which are already (1, 1) SLT in SALT's results), but improvement on larger nets is obvious.

TreeNet is also a work that enhances SALT using AI. So we also compare with it in Table 3 using the same evaluation method as TreeNet. In the table, "WL deg." represents maximum WL degradation. It means that we search through different shallowness constraints $\epsilon$ to obtain the best result that satisfies the WL degradation constraint. This constraint ensures that the resulting trees' WL doesn't deviate too much from the shortest WL. The result of SALT is slightly different from the result in TreeNet, because we don't know the searching strategy of TreeNet, and our searching strategy may be different. Generally, a more refined search process towards a given WL constraint will result in shallower trees. We search the results with $\epsilon = 0, 0.05, 0.05 \times 1.5^1, 0.05 \times 1.5^2, \dots, 0.05 \times 1.5^{14}$. The results show that we can outperform TreeNet on most degrees and WL degradation constraints. In total, we can outperform SALT by a range of 2.026% to 11.14%, and outperform Treenet by 0.019% to 6.026%.

Table 3. Comparison of Shallowness Under Different WL Constraints

| $\|V\|$ | Method | WL deg. | | | | |
|---|---|---|---|---|---|---|
| | | 0.00 | 0.05 | 0.10 | 0.15 | 0.20 |
| Small (4-7 pins) | SALT | 1.0462 | 1.0216 | 1.0078 | 1.0022 | 1.0006 |
| | Treenet | 1.0461 | **1.0210** | **1.0074** | 1.0021 | 1.0005 |
| | Ours | **1.0460** | 1.0211 | 1.0074 | **1.0020** | **1.0005** |
| | Imp.(%) | 0.3781 | 2.0937 | 4.9801 | 10.2974 | 22.2411 |
| | Imp.*(%) | 0.1123 | -0.4845 | -0.0405 | 5.2143 | 0.8121 |
| Med. (8-15 pins) | SALT | 1.3457 | 1.1776 | 1.0838 | 1.0391 | 1.0181 |
| | Treenet | 1.3435 | 1.1689 | 1.0790 | 1.0370 | 1.0172 |
| | Ours | **1.3433** | **1.1658** | **1.0756** | **1.0346** | **1.0158** |
| | Imp.(%) | 0.6971 | 6.6287 | 9.7666 | 11.6671 | 13.0070 |
| | Imp.*(%) | 0.0723 | 1.8424 | 4.3297 | 6.5581 | 8.2224 |
| Large (16-31 pins) | SALT | 1.7983 | 1.3550 | 1.1568 | 1.0727 | 1.0358 |
| | Treenet | **1.7755** | 1.3339 | 1.1481 | 1.0690 | 1.0341 |
| | Ours | 1.7764 | **1.3243** | **1.1419** | **1.0655** | **1.0322** |
| | Imp.(%) | 2.7460 | 8.6292 | 9.5302 | 10.0106 | 10.1444 |
| | Imp.*(%) | -0.1114 | 2.8651 | 4.2151 | 5.1355 | 5.6437 |
| Huge (32+ pins) | SALT | 2.0126 | 1.4401 | 1.2084 | 1.0987 | 1.0466 |
| | Treenet | 1.9793 | 1.4152 | 1.1975 | 1.0941 | 1.0444 |
| | Ours | **1.9783** | **1.3994** | **1.1894** | **1.0907** | **1.0428** |
| | Imp.(%) | 3.3802 | 9.2569 | 9.1144 | 8.1004 | 8.0861 |
| | Imp.*(%) | 0.0978 | 3.8102 | 4.1081 | 3.5670 | 3.5856 |
| All | SALT | 1.2532 | 1.1175 | 1.0524 | 1.0236 | 1.0110 |
| | Treenet | 1.2481 | 1.1114 | 1.0495 | 1.0223 | 1.0104 |
| | Ours | **1.2481** | **1.1088** | **1.0476** | **1.0212** | **1.0098** |
| | Imp.(%) | 2.0261 | 7.3846 | 9.1248 | 10.2683 | 11.1447 |
| | Imp.*(%) | 0.0190 | 2.3244 | 3.8210 | 5.1436 | 6.0260 |

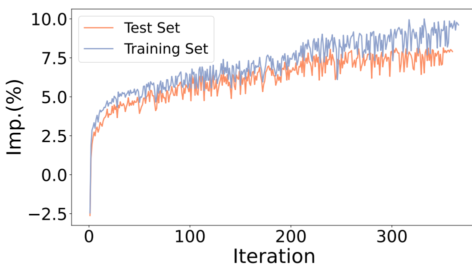


Fig. 6. Training process.

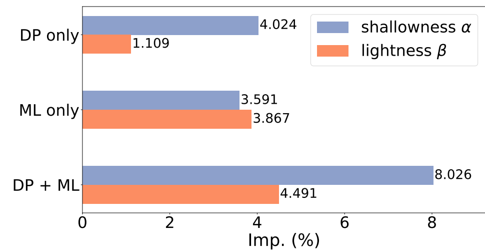


Fig. 7. Ablation study.

## 5.2 Effectiveness of Proposed Reward

In this section, we demonstrate the effectiveness of our reward function (introduced in Section 3.5) through an analysis of the training process. Using a four-fold cross-validation strategy, we present results from one of the four training folds in Figure 6, which illustrates how solution quality improves throughout the training iterations. We mix-train on nets of all degrees in the training set. We train our model batch by batch. Specifically, we train on one batch of data (2,000 nets in

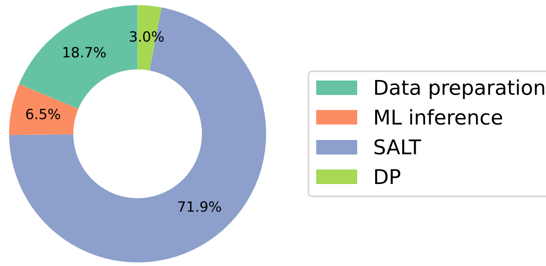


Fig. 8. Runtime breakdown.

$\epsilon = 0.05$ $\epsilon = 0.1$ $\epsilon = 0.3$ $\epsilon = 0.5$

RSMT

$\beta = 1, \alpha = 3.0255$ $\beta = 1, \alpha = 1.2315$ $\beta = 1, \alpha = 1.4217$ $\beta = 1, \alpha = 1.6825$

SALT

$\beta = 1.4594, \alpha = 1.0497$ $\beta = 1.326, \alpha = 1.0911$ $\beta = 1.3103, \alpha = 1.2687$ $\beta = 1.2915, \alpha = 1.4353$

MALT

$\beta = 1.2693, \alpha = 1.0260$ $\beta = 1.026, \alpha = 1.0911$ $\beta = 1.005, \alpha = 1.2657$ $\beta = 1.0016, \alpha = 1.3901$

Fig. 9. Qualitative results. Illegal points detected by SALT are circled red, and the critical set predicted by MALT is circled blue.

our implementation) for 150 steps, after which we move to another batch. We test our model at the initial stage and then test it every 40 batches with $\epsilon = 0.4$ on the benchmark. The improvement in the figure is the improvement of the two objectives. Starting from a near-random Critical Set prediction, the total improvement of WL and shallowness on test set increases from a negative

value to about 7.5% (The DP introduced in Section 4 is not used on this result), which shows the effectiveness of our reward strategy.

### 5.3 Ablation Study

We conduct ablation studies on large degree nets with $\epsilon = 0.4$ under three configurations: (1) DP only, (2) ML only, and (3) use both DP and ML model. Figure 7 shows the improvement achieved by different configurations. These results further demonstrate the effectiveness of both our ML model and DP algorithm.

### 5.4 Runtime Breakdown

The runtime breakdown of our framework is shown in Figure 8. We evaluated the entire superblue benchmark with batch size 12,000 and $\epsilon = 0.1$, and analyzed the runtime distribution across different components. The total runtime is 153.2 seconds. Data preprocessing for the machine learning model accounts for 18.7% of the runtime. In this phase, we precompute the four features mentioned in Section 3.2 and organize the input into batches. ML inference requires 5% of the runtime to predict Critical Sets for nets. The preprocessing step is significantly slower because it runs on CPU, while the ML model executes on GPU. The DP algorithm introduced in Section 4 consumes only 3% of the runtime.

As described in Section 3.4, the legalization and post-processing step is a straightforward adaptation of SALT. This component dominates the runtime at 71.9%. Since our model provides guidance to SALT for SLT construction, some runtime overhead compared to SALT is expected. Overall, the algorithms introduced in this article require only 28.1% of the total runtime, demonstrating their computational efficiency.

### 5.5 Qualitative Results

Figure 9 gives four qualitative results with different constraints $\epsilon = 0.05, 0.1, 0.3, 0.5$. The three lines are the result trees of FLUTE, SALT, and our algorithm, respectively. Illegal points detected by SALT are circled in red, and the Critical Set predicted by our model is circled in blue.

## 6 Conclusions

In this work, we design an ML-assisted SLT construction framework, MALT. We modeled the SLT construction as a detect-repair process. Compared to previous work which only used DFS to passively detect points that violate the constraint, we observe the suboptimality of this detection strategy and propose a machine learning model to detect a better "Critical Set" to guide the tree reconstruction process. We also develop a DP-based branch merging algorithm to further improve the quality. Experimental results show that our algorithm can achieve high solution quality in the SLT problem. In the future, we plan to further optimize considering more realistic factors, such as critical paths and congestion.

## References

[1] Charles J Alpert, Wing-Kai Chow, Kwangsoo Han, Andrew B Kahng, Zhuo Li, Derong Liu, and Sriram Venkatesh. 2018. Prim-dijkstra revisited: Achieving superior timing-driven routing trees. In *Proceedings of the 2018 International Symposium on Physical Design*. 10–17.

[2] Sanjeev Arora. 1998. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM* 45, 5 (Sept. 1998), 753–782. DOI: https://doi.org/10.1145/290179.290180

[3] B. Awerbuch, A. Baratz, and David Peleg. 1992. Efficient broadcast and light-weight spanners. *Manuscript* (01 1992).

[4] Gengjie Chen and Evangeline F. Y. Young. 2020. SALT: Provably good routing topology by a novel steiner shallow-light tree algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 6 (2020), 1217–1230. DOI: https://doi.org/10.1109/TCAD.2019.2894653

[5] Wing-Kai Chow, Liang Li, Evangeline F. Y. Young, and Chiu-Wing Sham. 2014. Obstacle-avoiding rectilinear Steiner tree construction in sequential and parallel approach. *Integration VLSI Journal* 47, 1 (Jan. 2014), 105–114. DOI: https://doi.org/10.1016/j.vlsi.2013.08.001

[6] Chris Chu and Yiu-Chung Wong. 2008. FLUTE: Fast lookup table based rectilinear steiner minimal tree algorithm for VLSI design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 1 (2008), 70–83. DOI: https://doi.org/10.1109/TCAD.2007.907068

[7] J. Córdova and Y.-H. Lee. 1994. *A Heuristic Algorithm for the Rectilinear Steiner Arborescence Problem.* Technical Report TR-94-025. Department of Computer and Information Science, University of Florida, Gainesville, FL, USA.

[8] Zizheng Guo, Feng Gu, and Yibo Lin. 2022. GPU-accelerated rectilinear steiner tree generation. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 53, 9 pages.

[9] Andrew B. Kahng, Robert R. Nerem, Yusu Wang, and Chien-Yi Yang. 2024. NN-steiner: A mixed neural-algorithmic approach for the rectilinear steiner minimum tree problem. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'24/IAAI'24/EAAI'24)*. AAAI Press, Article 1452, 9 pages. DOI: https://doi.org/10.1609/aaai.v38i12.29200

[10] S. Khuller, B. Raghavachari, and N. Young. 1995. Balancing minimum spanning trees and shortest-path trees. *Algorithmica* 14, 4 (1995), 305–321.

[11] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[12] Liang Li, Zaichen Qian, and Evangeline F. Y. Young. 2009. Generation of optimal obstacle-avoiding rectilinear steiner minimum tree. In *Proceedings of the 2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*. 21–25.

[13] Wei Li, Yuxiao Qu, Gengjie Chen, Yuzhe Ma, and Bei Yu. 2021. TreeNet: Deep point cloud embedding for routing tree construction. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) *(ASPDAC '21)*. Association for Computing Machinery, New York, NY, USA, 164–169. DOI: https://doi.org/10.1145/3394885.3431566

[14] Jinwei Liu, Gengjie Chen, and Evangeline F.Y. Young. 2021. REST: Constructing rectilinear steiner minimum tree via reinforcement learning. In *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1135–1140. https://doi.org/10.1109/DAC18074.2021.9586209

[15] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., USA.

[16] Rudolf Scheifele. 2015. Steiner trees with bounded RC-delay. In *Proceedings of the Approximation and Online Algorithms*, Evripidis Bampis and Ola Svensson (Eds.). Springer International Publishing, Cham, 224–235.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. 2017. *Attention Is All You Need.* https://doi.org/10.48550/arXiv.1706.03762

[18] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Proceedings of the Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc.

[19] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. 2012. The DAC 2012 routability-driven placement contest and benchmark suite. In *Proceedings of the DAC Design Automation Conference 2012*. 774–782. DOI: https://doi.org/10.1145/2228360.2228500

[20] David M Warme, Pawel Winter, and Martin Zachariasen. 2000. Exact algorithms for plane steiner tree problems: A computational study. In *Proceedings of the Advances in Steiner Trees*. Springer, 81–116.

[21] Liying Yang, Guowei Sun, and Hu Ding. 2023. Towards timing-driven routing: An efficient learning based geometric approach. In *Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. DOI: https://doi.org/10.1109/ICCAD57390.2023.10323981


Received 18 June 2025; revised 25 November 2025; accepted 4 January 2026