Towards Timing-Driven Routing: An Efficient Learning Based Geometric Approach

Liying Yang[1*], Guowei Sun[2*], Hu Ding[1†]
[1]School of Computer Science and Technology, [2]School of Data Science
University of Science and Technology of China, Anhui, China
{liynnyang, gwsun1998}@mail.ustc.edu.cn, huding@ustc.edu.cn

***Abstract*—As the rapid increasing of the circuits complexity, it is urgent to develop efficient algorithmic techniques for EDA. In this paper, we consider the routing problem which is a key part for designing high-quality chips. In particular, we combine both the max path length and total wirelength for modeling our optimization objective, since the path delay often causes timing issue that can seriously degrade the whole routing efficiency (even if the total wirelength is small). Comparing with most of the previous works that only considering wirelength, the timing-driven routing objective is much more challenging to optimize. We propose an efficient learning-based approach together with several novel insights in geometry. For moderate-degree nets, our approach can yield a better smooth trade-off between the wirelength and max path length comparing with the state-of-the-art methods. For large-degree nets, we propose an elegant and easy-to-implement geometric data structure called "data-dependent polar quadtree" in the space; using this structure, we can successfully plug our learning-based approach into a divide & merge framework and the optimization quality over the whole instance can be well preserved.**

I. Introduction

In modern chip design, both power consumption and timing are important issues. As part of the signal transmission, the interconnection delay accounts for a large proportion of the overall signal delay. So interconnection delay is an important issue that we should consider in the routing stage [1]. For a signal net with n pins, one pin is defined as the source node and the other $n-1$ pins are defined as the sinks. The length of each source-sink pair approximately reflects the delay between them. According to the analysis of Elmore delay model, we know that in order to obtain low delay, it is necessary to obtain a trade-off between **the total wirelength** and **the longest source-sink length** based on the net size and its capacitance and resistance [2, 3]. The two objectives may be in conflict with each other in some cases, so how to efficiently optimize these two objectives simultaneously is an important and challenging problem in the past decades [4].

A number of methods have been proposed for minimizing wirelength as the single objective. Given n points in $\mathbb{R}^2$, the problem of *Rectilinear Steiner Minimum Tree* ***(RSMT)*** aims to connect these n points by a network with rectilinear edges (*i.e.,* vertical and horizontal line segments) and a set of additional auxiliary points (*i.e.,* the steiner points), where the objective is to minimize the total net length (measured in the Manhattan metric) [5, 6]. Warme et al.[7] proposed an exact RSMT algorithm called GeoSteiner. Their main idea is based on concatenating all the possible full Steiner trees for a given instance. To compute an approximate RSMT, a common idea is using *Rectilinear Minimum Spanning Tree* **(R-MST)**, which can be computed in $O(nlogn)$ time and yields a 1.5-approximate RSMT [6]. Several methods have been further proposed for improving R-MST to achieve lower objective cost, such as SPAN [8] and BGA [9]. Chu and Wong developed a method FLUTE [10] that is based on a pre-computed lookup table of nets with the degree less than 9. Recently, Liu et al. [11] proposed a method REST to construct RSMT by using reinforcement learning.

Another relevant problem is called *Rectilinear Steiner Minimum Arborescence* **(RSMA)**, which aims to minimize the total wirelength with each path length from the source being equal to the Manhattan distance [12]. Córdova et al.[13] developed a heuristic algorithm called CL, which can be computed in $O(nlogn)$ time and the obtained cost is at most twice the optimal wirelength. Cong et al.[14] presented a polynomial time optimal algorithm based on the distributed RC delay model, which can be applied to any routing topology. Pan et al. [15] proposed a fast method by constructing an A-tree based on net-breaking and table lookup. It is worth noting that both the RSMT and RSMA problems are NP-complete [12, 16].

Several approximation algorithms have been studied for minimizing both wirelength and max path length. Alpert et al. [3] proposed a PD algorithm that combines the Prim and Dijkstra algorithms. Cong et al. [17] developed the BRBC algorithm, where it generates a minimum spanning tree first and then iteratively connecting a sink to the source if the sink violates the path length constraints. Khuller et al. [18] proposed a method simultaneously approximating a minimum wirelength tree and a shorest path length tree. Chen et al. [19] proposed a algorithm called SALT which has the tighter bound and better experimental results.

As the development of artificial intelligence, the learning-based approach for solving combinatorial optimization problems has also attracted a great amount of attention in EDA [20]. For example, Liu et al. [11] proposed a novel method "REST" to construct RSMT by using reinforcement

[*]First two authors contributed equally. [†]Corresponding author. The research of this work was supported in part by USTC Innovation Group Grant, CAAI-Huawei MindSpore Open Fund and Huawei-USTC Joint Innovation Project on Fundamental System Software.

learning. But their approach only optimizes the wirelength as the single objective. If we want to consider the max path length at the same time, we need a structure that can better represent the topological information of the entire data. Also most existing learning-based approaches can only handle the nets with moderate size (*e.g.,* n is no larger than 50), where the main reason is that training a reinforcement learning model requires a large amount of memory and computational cost, especially when the input net is large. In practical chip design, although a net usually has a moderate size, we may also need to deal with large nets that contain hundreds or even thousands pins in some scenarios. As the fast growing of circuit complexities in modern design, it is much harder to solve the large-degree nets and it takes most of the design time [21].

A. Problem Formulation and Our Main Results

In this paper, we propose a method that **combines machine learning and geometric algorithmic techniques** to achieve a better trade-off between wirelength and max path length for rectilinear steiner tree, and it is applicable to both small-scale and large-scale situations.

We introduce the formal definition of the problem first. Suppose the input net has n pins $V = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\} \subset \mathbb{R}^2$. Let (x_0, y_0) be the source (the output pin of a gate) and the remaining $n-1$ nodes are sinks (the input pins of other gates). Through connecting the sinks to the source by vertical and horizontal line segments and additional steiner point, we can obtain a rectilinear steiner tree T. With an input parameter $\lambda \in [0,1]$, the objective can be defined as:

$$\min_{T} \quad (1-\lambda)L(T) + \lambda R(T), \tag{1}$$

where $L(T)$ denotes the wirelength of T and $R(T)$ denotes the max path length from the sinks to the source in T. We can obtain a smooth trade-off between wirelength and the max path length (*i.e.,* the approximate pareto frontier) by adjusting the parameter λ in the objective (1).

Roughly speaking, we convert the rectilinear routing problem into a directed graph generation problem that can be solved via Graph Neural Network (GNN) and Reinforcement Learning (RL). Note that it is not quite realistic to directly solve large-scale instances via GNN and RL, due to not only the high computational complexity, but also the high memory usage during training [22]. In view of this, we design a novel spatial partition based divide & merge method to solve this problem. For convenience, the net with pin number ≤ 32 is called a "**moderate-degree net**", and the net with pin number > 32 is called a "**large-degree net**" (since most nets have less than 32 pins in actual industrial production, we set 32 as the threshold). The major contributions in this paper are twofold:

- For solving moderate-degree nets, we propose a novel GNN based approach that can achieve a trade-off between wirelength and max path length efficiently. The key idea relies on a conversion method between rectilinear edges and directed edges, which transforms the routing problem to a graph generation problem. Another advantage of our approach is its compatibility, where any mature GNN model can be used with the conversion method to solve the routing problem.
- To further deal with large-degree nets, we propose an elegant and easy-to-implement geometric data structure called "data-dependent polar quadtree" in the space. By using this structure, we can successfully plug our GNN model into a divide & merge framework and the optimization quality over the whole instance can be well preserved; the implementation can be also parallelized.

The rest of this paper is organized as follows. In Section II, we present the details of our neural network model to handle moderate-degree nets. In Section III, we present our divide & merge framework to deal with large-degree nets. In Section IV, we illustrate the experimental results.

II. Our Approach for Moderate-degree Nets

In this section, we present our neural network model consisting of GNN (Graph Neural Network) and RL (Reinforcement Learning) to achieve the trade-off between wirelength and max path length. Our idea is partly inspired by the "actor-critic" reinforcement learning approach REST [11]. Actually the actor-critic model is a commonly used reinforcement learning model that could date back to 1980s [23]. In REST, an RSMT is represented by an *edge sequence* of length $n-1$, where the sequence is then to be learned in the actor-critic model. Here, an "edge" is a rectilinear edge from one point to another point in the rectilinear steiner tree; the direction is always from the end point that connects a vertical segment to the end point that connects a horizontal segment (if the rectilinear edge is only a simple vertical or horizontal segment, either direction should be fine). See Fig. 1(a) for an illustration. Also these $n-1$ edges are restricted to be free of cycle (we ignore the edge directions when talking about a cycle). So these edges actually form a spanning tree of the given n points, and please see Fig. 1(b) for an illustration. These $n-1$ edges can be easily extracted one by one, and we briefly overview the idea below.

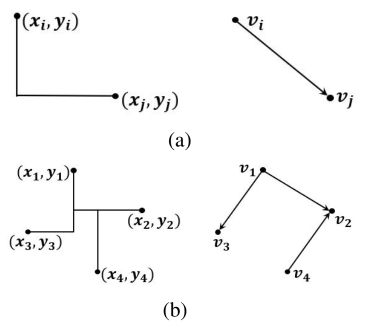


Fig. 1. Suppose in the directed graph, a vertex v_i corresponds to an input point (x_i, y_i) for $i = 1, 2 \cdots, n$. In (a), a rectilinear edge from (x_i, y_i) to (x_j, y_j) of the routing is represented by a directed edge from v_i to v_j. In (b), a toy example is illustrated, where the left is a routing result and the right is the corresponding directed graph.

Let p be an arbitrary leaf node of the rectilinear steiner tree; if it is connected to another point q by a rectilinear edge, we can directly take it; otherwise, it is connected to a steiner point s, then we can find another point r that is also connected to s (by the Hannan theory [5]), and then the two edges $\overline{ps}$ and $\overline{sr}$ can be concatenated as a rectilinear edge; we remove p from the rectilinear steiner tree and continue to extract the next rectilinear edge. So we can convert the routing result consisting of rectilinear edges to a directed graph. And we can also reversely convert the directed graph to a rectilinear steiner tree (the details are shown in Section II-A). Therefore, the routing problem can be transformed to a directed graph generation problem.

The conversion has several advantages. First, the rectilinear edges in the routing can preserve the spatial topology information. After converting it to a directed graph, we can use an appropriate graph neural network to learn the spatial topology information. Second, in the process of generating directed edges, we do not need to concern about the positions of the steiner points, since the steiner points can be automatically generated when the directed graph is constructed. However we should emphasize that realizing our approach is a non-trivial task, where **the major challenge is how to design a suitable GNN model with an efficient policy network for training the "actor".** Below, we introduce the actor-critic model [11, 23] in Section II-A, and introduce our policy network in Section II-B. Finally, we introduce a strategy for the pre-training and fine-tuning on λ to speed up training in section II-C.

A. The Actor-Critic Framework

The actor module and the critic module can be viewed as adversaries in the training process, so they jointly improve the performance of the model. Another benefit of the actor-critic framework is that it does not need to know the optimal result. This property is particularly useful to train the nets whose optimal solution is hard to compute.

The **actor** module uses RL to take a sequence of actions to generate the directed edges of nodes, where the intermediate states are encoded via GNN. Fig. 2(a) shows the process that the agent takes the action step by step to generate a directed graph, and eventually a rectilinear steiner tree is constructed. The initial input of the actor network is a null graph. Each step the network adds one edge to the graph and at the end a directed graph with $n-1$ edges (*i.e.,* a spanning tee of the n vertices) is generated. We regard the generation process as a sequential Markov Decision Process (MDP). In particular, we have three key components to define the MDP (the first two components are controlled by a "policy network" and we will elaborate on the details in Section II-B):

1) **State:** For each $t \in \{0, 1, \cdots, n-1\}$, we denote the the intermediate graph as G_t, which is generated at the t-th state S_t; G_t contains the information of the current routing environment. We encode the state S_t by using graph convolutional network.
2) **Action:** An "action" means adding a valid edge to the graph of the current state, where an edge is valid if and only if it does not generate a cycle. To ensure the validity, we use a mask matrix to hide the edges that cannot be connected. The set of all the valid actions form the action space. For guiding the model to choose an appropriate action from the action space, we also associate it with a "policy matrix", *i.e.,* the probability distribution of the actions to be chosen.
3) **Reward:** As the common tool of reinforcement learning, we also assign a reward to guide the trajectory. In our problem, we only assign the reward to the final state when the directed graph is already generated. Following the method of [11] , we combine the "critic" module (which will be discussed below) to set the reward.

From graph to rectilinear steiner tree. Let the returned directed graph of S_{n-1} be G_V. So we need to convert G_V to a rectilinear steiner tree (this is an inverse transformation of the directed graph representation). Each directed edge of G_V can be drawn as a rectilinear edge in the plane. But the $n-1$ rectilinear edges together may yield some intersections (see Fig. 3(b)). We can apply the classical "sweep line" algorithm [24] to detect the steiner points and remove the intersections (see Fig. 3 as an illustration). The whole time complexity of the sweep line algorithm is $O((n+n_s+n_i)\log n)$, where n_s is the number of steiner points and n_i is the number of intersections; in practice n_i is usually a small number and thus the complexity is nearly linear. Once the rectilinear steiner tree is built, we can also compute the trade-off value of the tree according to the Equation (1) in linear time and we use $\mathtt{A}(G_V)$ to denote the value.

The **critic** module predicts the expected trade-off value of the rectilinear steiner tree of the input n points; the predicted value is denoted as $\mathtt{C}(V)$. Then a 3-layer multi-head attention mechanism [25] is applied to train the critic module. The difference $L = \mathtt{C}(V) - \mathtt{A}(G_V)$ is used as the reward to improve the performance of the actor. So a lower $\mathtt{A}(G_V)$ should receive a higher reward. As we know, $n-1$ edges can connect n nodes to form a directed tree. Assuming that the probability of selecting a edge in the state S_t is p_t, then the probability of a valid trajectory of the agent is

$$p_{tra} = \prod_{t=1}^{n-1} p_t, \quad p_t = p_\theta(edge_t | S_t), \tag{2}$$

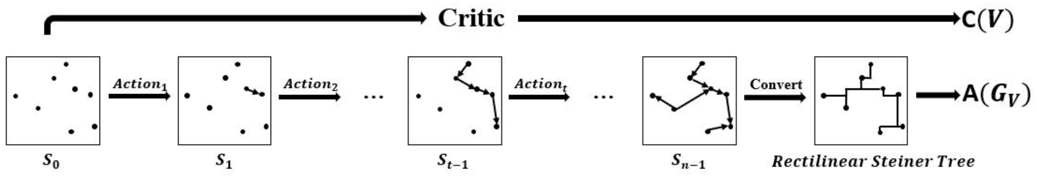
where θ is the parameter to train in the model. The loss can be defined as

$$L(\theta|V) = \sum_{tra \in J} (C(V) - A(V, tra))p_{tra}, \tag{3}$$

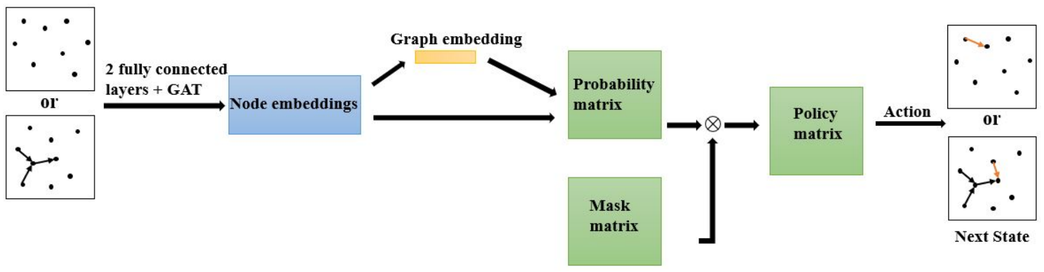
where J denotes the set of all trajectories. We train the model by the stochastic gradient descent method [26].

B. Policy Network

The policy network is the core of our actor-critic model, and it determines the "state" and "action" of the actor module. As shown in Fig. 2(b), each state S_t applies the policy network

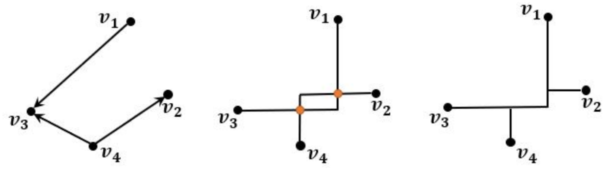


(a)



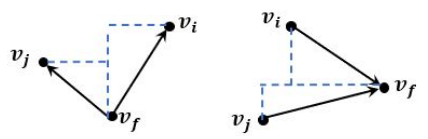
(b)

Fig. 2. (a) shows the overall rectilinear steiner tree construction process. (b) is our Policy network architecture



(a) (b) (c)

Fig. 3. Conversion from graph to rectilinear steiner tree: (a) is the directed graph, (b) is the configuration of the rectilinear edges, (c) is the tree after the adjustment by sweep line.



(a) (b)

Fig. 4. In (a), the vertices v_i and v_j are pointed out from the common vertex v_f, and they are defined as the second-order in-degree neighbors. In (b), the vertices v_i and v_j point to the common node v_f, and they are defined as the second-order out-degree neighbors.

to train the intermediate graph G_t so as to return the action a_t, which represents the new added directed edge, and then proceeds to the next state S_{t+1}. In order to predict the edge to be connected, our model embeds each graph G_t and all the nodes of state S_t in the Euclidean space $\mathbb{R}^d$ (we set $d = 128$ in our experiments).

Before presenting our overall idea, we first introduce a commonly used model Graph Attention Network (GAT) layer [27], which is efficient for collecting node information and performing embedding updates in practice. Our approach for training the policy network relies on this model. The edge weights and node embeddings are updated by aggregating neighborhood information from the graph. Formally, the update are implemented by using the Leaky Rectified LU and Softmax functions:

$$e_{ij} = \texttt{LeakyReLU}([v_i W \ v_j W] w^T); \tag{4}$$

$$v_i = \sigma\Big(\sum_{v_j \in N(v_i)} \frac{exp(e_{ij})}{\sum_{v_k \in N(v_i) exp(e_{ik})}} v_j W\Big), \tag{5}$$

where v_i is the embedding of the i-th node in $\mathbb{R}^d$, e_{ij} is attention coefficient of the edge connecting v_i and v_j, and the $N(v_i)$ is the neighbors of v_i in the graph G_t (it can be updated in each state S_t, and G_0 is a null graph with no edge connections in S_0). We also keep a node embedding matrix $X_t \in \mathbb{R}^{n\times d}$ in each state S_t, where its i-th row is the vector v_i for $1 \le i \le n$. Also, the vector $w \in \mathbb{R}^{1\times 2d}$ and matrix $W \in \mathbb{R}^{d\times d}$ are the parameters to be trained. The overall training procedure contains the following components.

(1) Initialization. In the initial state with no edge connection, each pin (node) is represented by a 3-dimensional vector. The 3-dimensions information includes the $2\mathbb{D}$ coordinate information (x, y) and the label that whether it is a source (if it is a source, it is labeled as 1, and 0 otherwise). To further embed the nodes to a higher d-dimensional space, we use a linear transformation to expand the node embedding matrix to $X_{ini} \in \mathbb{R}^{n\times d}$ (n is the number of nodes). And then we input X_{ini} to a single GAT network to obtain the embedding $X_0 \in \mathbb{R}^{n\times d}$ in the state S_0. Note that the initial graph has no edges, and to enable the information exchange in the initial training state, we assume G_0 is a virtual fully connected graph.

(2) Node embedding X_t. For the intermediate graph G_t, we compute the node embedding by **2 layers** of **3-channels** GNN. We aggregate the messages by 3-channels as follows. First, we define the matrices $A_{t,1}, A_{t,2}, A_{t,3} \in \mathbb{R}^{n\times n}$. $A_{t,1} \in \mathbb{R}^{n\times n}$ is the adjacency matrix of the graph. $A_{t,2}$ and $A_{t,3}$ are the second-order "in-degree" and "out-degree" adjacency matrices, respectively (see Fig. 4 for their illustrations). The connection relations in these adjacency matrices are closely related to the neighbor relations in the GAT model. Therefore, these adjacency matrices ($A_{t,1}, A_{t,2}$ and $A_{t,3}$) are used to determine the neighbors of the nodes, *i.e.,*$N(v_i)$ in (5), in the updating process. It should be emphasized that, comparing with using only the adjacency matrix, the second-order adjacency matrices can provide richer information for the connection, such as the overlap between their second-order neighborhoods.

For simplicity, we use $GAT_{A_{t,1}}$ to denote the training process of GAT with the connectivity information provided by $A_{t,1}$; $GAT_{A_{t,2}}$ and $GAT_{A_{t,3}}$ can be defined similarly. At the beginning of state S_t, we set $X_t^0 = X_0$ to be the input to the first layer (to avoid the common problem of over-smoothing in graph neural networks [28], we always update the node representations at the beginning of each state based on X_0). We also let $X_t^1 \in \mathbb{R}^{n\times d}$ and $X_t^2 \in \mathbb{R}^{n\times d}$ be the outputs of the first and second layers respectively. We compute these two matrices by recursion for $h = 0, 1$:

$$\begin{cases} X_{t,1}^{h+1} = GAT_{A_{t,1}}(X_t^h); \\ X_{t,2}^{h+1} = GAT_{A_{t,2}}(X_t^h); \\ X_{t,3}^{h+1} = GAT_{A_{t,3}}(X_t^h), \end{cases} \tag{6}$$

and we take their average $X_t^{h+1} = \frac{1}{3}(X_{t,1}^{h+1} + X_{t,2}^{h+1} + X_{t,3}^{h+1})$. Finally we take $X_t = X_t^2$ as the embedding of the n vertices of G_t in state S_t.

(3) Graph embedding y_t. Recall we generate the rectilinear steiner tree (*i.e.,* the graph) step by step as shown in Fig. 2(a), where each step we add an edge that connects two vertices. Let V_t be the set of vertices that are connected with at least one edge in state S_t. Obviously, $V_0 = \emptyset$ and $V_{n-1} = V$ (V is the set of input n pins). We let X_{t+} be the sub-matrix of X_t which contains the rows corresponding to the vertices of V_t, and let X_{t-} be the sub-matrix which contains the remaining rows. Finally, we obtain the d-dimensional graph embedding vector $y_t = \mathtt{tanh}(X_{t+}w_+)X_{t+} + \mathtt{tanh}(X_{t-}w_-)X_{t-}$, where $w_+ \in \mathbb{R}^{d\times 1}$ and $w_- \in \mathbb{R}^{d\times 1}$ are the parameter vectors to be trained. The vector y_t plays an important role in the following action prediction, where its intuition is to integrate the topological information of the connected nodes and the distribution information of the unconnected nodes.

(4) Action prediction P_t. We design an $n \times n$ **probability matrix** P_t for each state S_t, where each entry p_{ij}^t (in the i-th row and j-th column) represents the probability that there is a directed edge from v_i to v_j. Let $\Big[X_t[i]\ X_t[j]\Big]$ denote the $2\times d$ dimensional vector that is the concatenation of the i-th and j-th rows of X_t; we compute p_{ij}^t by embedding $\Big[X_t[i]\ X_t[j]\Big]$ and y_t into the network:

$$p_{ij}^t = tanh\Big(\big(\Big[X_t[i]\ X_t[j]\Big] * T_{i,j} + y_t * S\big) * H\Big), \tag{7}$$

where the matrices $T_{i,j} \in \mathbb{R}^{2d\times d}, S \in \mathbb{R}^{d\times d}$, and $H \in \mathbb{R}^{d\times 1}$ are all the parameters to be trained in the network.

To avoid cycle, we also design a **mask matrix**, a binary $n \times n$ matrix M, to restrict the action space. For any two vertices v_i and v_j, if they belong to the same connected component, we just set the corresponding entries $M[i, j]$ and $M[j, i]$ to be 0; other entries of M are all set to be 1. We take the element-wise product of M_t and P_t to obtain a masked probability matrix, *i.e.,* the **policy matrix** (if an edge can generate a cycle, its probability to be chosen becomes 0). Then we select the edge with the highest probability in the probability matrix to connect and proceed to the next state. When all the entries of the mask matrix are 0, it implies that the rectilinear steiner tree generation has already been completed. We optimize the parameters (based on the maximum likelihood estimation) via the standard gradient descent algorithm [26].

C. Pre-training and Fine-tuning Strategy for λ

We also need to consider the parameter λ that controls the trade-off between wirelength and max path length in the objective (1). Obviously, we cannot afford to train our model for each individual $\lambda \in [0, 1]$. Instead, we can adopt a simple "pre-train+fine-tuning" strategy. Namely, we train our model (as shown in Fig. 2) for $\lambda = 0, 0.1, 0.2, \cdots, 1$; then for a given λ in the test stage, we find the pre-trained model of its nearest neighbor and perform a fine-tuning procedure which takes only a few number of iterations. For example, if $\lambda = 0.32$, we take the pre-trained model with $\lambda = 0.3$, and perform the fine-tuning.

There is also a technical trick for training these 11 pre-trained models. Instead of training them separately, we follow the "transfer training" idea of [29] to reduce the total computational complexity. When the model with $\lambda = 0$ is obtained, we can use it as the initial parameters to train the model with $\lambda = 0.1$; in practice the training time can be significantly reduced comparing with directly training the model from scratch. Similarly, we can train the other models with the order $\lambda = 0.2, 0.3, \cdots, 1$.

III. The Divide & Merge framework

In this section, we propose a novel divide & merge framework to solve large-scale instance. It is worth noting that the idea of divide & merge is common for dealing with large-scale optimization problems. **However, for our timing-driven routing problem, the objective (1) is more complicated since it takes the trade-off between two different types of costs.** In particular, it is a challenging task to keep the max path length to be small when aggregating the sub-solutions from local to global during the merging procedure. Therefore, we need to develop a new divide & merge framework that should be suitable for our problem. We introduce our "dividing" structure in Section III-A, and present our "merging" strategy in Section III-B.

A. Data-dependent Polar Quadtree

Our intuition for designing the dividing structure is as follows. In the timing problem, we need to ensure that the max path length from the sinks to the source is as short as possible. That is, it is better to have a "star" topology from the source to the sinks. Inspired by the classical Quadtree structure in computational geometry [30], we introduce a novel '**Data-dependent Polar Quadtree**" method to divide a given large-scale instant to a set of small local instances.

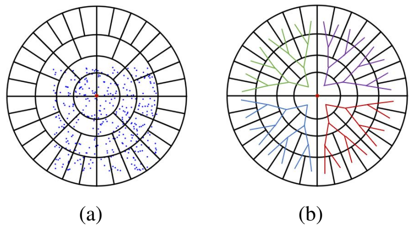


Fig. 5. (a) is the divide diagram, where the red point is the starting point, and the blue points are pins. (b) shows that these blocks can be represented by 4 complete binary trees.

The construction procedure. Without loss of the generality, we assume the source is the origin. The innermost circle is evenly divided into 4 blocks according to the angle. Then the next layer, which is a ring, is evenly divided into 8 blocks; the following layers are generated similarly, where the number of blocks is always doubled, until all the pins are covered. Please see Fig. 5(a) as an illustration. The benefit of using such a division strategy is that we can control the max path length more conveniently during the merging step (more details are shown in Section III-B). For each block, we can use the neural network model designed for moderate-degree in Section II to solve it. Note that there is an issue that is omitted here, *i.e.,* how to determine the source point in each block. We answer this question based on our merging strategy in Section III-B.

We also need to clarify some details of our dividing structure as shown in Fig. 5(a). Suppose the radius of the structure is ρ_{max}, *i.e.,* the outer radius of the largest ring which guarantees that all the pins are covered. Then we need to specify the thickness of the rings. Intuitively, **(1)** we prefer that the number of pins covered by each block should be not too large; **(2)** also, we want the number of the blocks to be as small as possible, since the total computational complexity can be high if there are too many blocks (as an extreme example, if each block has at most one pin, there will be too many merging operations and the divide & merge framework will be very inefficient). That is why we call it as a "data-dependent" structure. In practice, we can set a threshold $B > 0$ (we set $B = 30$ in our experiments), and adjust the thickness to keep that each block has at most B pins.

If the distribution of the pins is very uneven (*e.g.,* 90% of the pins locate very closely in a small local region, and the other 10% pins are scatters in the space), it is hard to find an appropriate thickness to satisfy the aforementioned conditions (1) and (2). In this case, we can build our proposed polar quadtree again in each dense block individually (as shown in Fig. 6). In other words, we build a two-level dividing structure; also we can generalize this strategy to be multi-level.

B. Bidirectional Binary Tree Search for Merging

Suppose the proposed dividing structure in Section III-A has been built, and we perform the method of Section II to solve the local instances of the blocks. Now we consider to merge these local solutions to form the global solution. In particular, we need to solve the following two issues.

(1) The merging order. A straightforward approach for merging is following the order of the binary trees as shown in Fig. 5(b). For example, we can merge the blocks from bottom to top and eventually the global solution is obtained. However, this approach can result in large wirelength or large max path length in some scenarios. To shed some light, we consider the example shown in Fig. 7, where the pins are located only on the outermost blocks and each outermost block has only one pin. If applying the straightforward merging method based on binary tree, all the pins will be directly connected to the source, which will cause the wirelength to be too long (see Fig. 7(a)). But if we directly connect the nearest blocks, the max path length will be too long (see Fig. 7(b)). Therefore, our solution needs to skip those empty blocks, connect all blocks containing pins, and ensure that the max path length is not too large (see Fig. 7(c)). We design a "**bidirectional binary tree search**" method to accomplish our goal.

We introduce some necessary notations first. For any block x, we use $\texttt{Par}(x)$ to denote its parent in the binary tree of Fig. 5(b). Denote by $\texttt{LNei}(x)$ (*resp.,* $\texttt{RNei}(x)$) the left (*resp.,* right) neighbor of x in the same ring. We always use x_0 to denote the block containing the source.

The idea of our proposed bidirectional binary tree search method is as follows. The merging starts from a non-empty block, say x, in the outermost layer. W.l.o.g., we assume x is the left child of $\texttt{Par}(x)$. If $\texttt{Par}(x)$ is non-empty, we directly connect x to $\texttt{Par}(x)$; otherwise, we search a non-empty block by two directions simultaneously. In each of the left and right sides, we conduct a zigzag searching path (as shown in Fig. 8), and we connect x to the earliest non-empty block found in these two directions. For example, in the left side, we check the blocks in the order of $\texttt{LNei}(x)$, $\texttt{Par}(\texttt{LNei}(x))$, $\texttt{LNei}\big(\texttt{LNei}(x)\big)$, and so on so forth until a non-empty block is found. However, we cannot allow the searching path to be too long, since it may result in a large max path length as the example shown in Fig. 7(b). So we terminate the searching paths within a pre-specified bounding length, and renew the search in the higher layer (*i.e.,* the layer of $\texttt{Par}(x)$); in the worst case, if we cannot find a non-empty block in any layer, we directly connect x to x_0. The benefit of conducting such bi-direction searching strategy is that it can efficiently skip empty blocks, which is beneficial to optimize the trade-off objective. After performing this strategy for each block, we can achieve the merging order for all the blocks in the binary tree from bottom to top.

(2) Determine the connection points and source points. After the connection between the blocks is determined, we

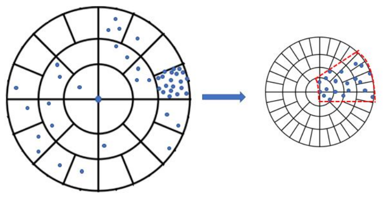


Fig. 6. Two-level dividing structure.

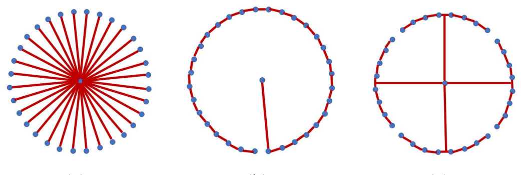


Fig. 7. In (a), the wirelength is too long. In (b), the max path length is too long. (c) is a trade-off between wirelength and max path length.

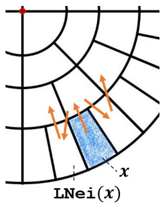


Fig. 8. An illustration of the bi-direction searching strategy.

need to find the proper **connection points** to be merged and the **source point** of each block. Intuitively, we want the connection points to be as close as possible. We consider two different cases. Case 1: three blocks are jointly connected as Fig. 9(a). Assume that the intersection point of the three blocks is p. We take its nearest points in the three blocks (measured by Manhattan distance), say p_u, p_v, p_w. Then, we connect them to a newly added steiner point. We also set p_u and p_v to be the source points of the corresponding two children blocks respectively. Case 2: two blocks are connected (they may be in the same layer, or the third block is empty as shown in Fig. 9(b)). We directly select the two nearest points at Manhattan distance in the two blocks for connection, say p_s and p_e (see Fig. 9(b) for a illustration). Then we pick p_e as the source point for the child block.

Remark 1: Another advantage of our framework is that it can be easily realized in a parallel fashion, that is, the blocks can be handled in parallel and thus the total running time can be reduced significantly.

IV. Experiments

We implement and train our neural network model designed for moderate-degree nets both using both Pytorch and MindSpore, and run our experiments on a 64-bit Linux machine with two NVIDIA RTX A6000 GPU. And we implement our divide & merge framework in C++. Our algorithms are denoted as "OURS". The codes of CL [13], KRY [18], PD [3], BONN [31] are implemented in C++; for SALT [19], FLUTE [10, 32] and REST [11], we use their open source codes. These algorithms are used for comparison in our experiments. Let l_{wire} and l_{max} respectively denote the obtained wirelength and max path length of an algorithm. Since FLUTE and CL are commonly used representative algorithms for optimizing wirelength and optimizing max path length respectively, we in particular take their obtained values $l_{wire}(FLUTE)$ and $l_{max}(CL)$ as the baselines. Then we evaluate the performance of the algorithm by the ratios $\frac{l_{wire}}{l_{wire}(FLUTE)}$ and $\frac{l_{max}}{l_{max}(CL)}$.

A. Results on Moderate-degree Nets

We compare the experimental performances of our neural network model in Section II and the existing algorithms for the nets with the degree n no larger than 32. We train our models from degree 4 to 32, using randomly generated training data as the previous articles [9, 11, 32]. Each degree is trained on two GPUs by using the Distributed DataParallell (DDP) training approach [33] with the optimization algorithm Adam [26].

For each degree, 10^5 randomly generated instances are used as the testing data, and we report their average results in Fig. 10. We can see that our method achieves better results than existing methods, both in terms of wirelength and max path length. We also conduct the experiment on the ICCAD 2015 benchmark [34], and the results are shown in Fig. 11. The results are similar with those of random data, and our method still outperform the other algorithms.

Note that SALT is the best of the baseline algorithms in Fig. 10 and Fig. 11. To have a more complete comparison with SALT, we also compare the running time with SALT in Table I; we can see that their running times are roughly the same, but our algorithm outperforms SALT in terms of the wirelength and max path length as shown in Fig. 10 & 11.

B. Results on Large-degree Nets

We compare the algorithms on randomly generated large-degree nets ($n = 50, 100, 1000$, each degree has 100 instances) in this section. The results are shown in Fig. 12. For $n = 50$ and 100, our method can achieve comparable results with SALT, and significantly outperform the other algorithms. For $n = 1000$, our method outperforms other algorithms except for SALT. But our method takes much less running time for large-scale nets than SALT. For example, our method took $70.33s$ for processing 100 instances with $n = 1000$, which is roughly only half of the running time of SALT (it took $137.8s$).

V. Conclusion

We study the time-driven routing problem which is a challenging task in EDA. We propose an efficient neural network model and a novel divide & merge framework for

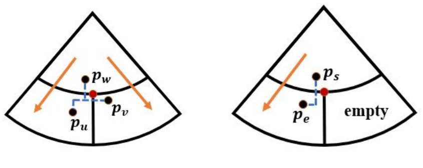
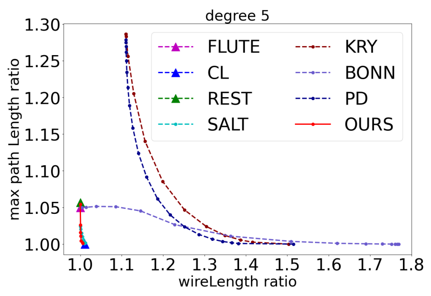


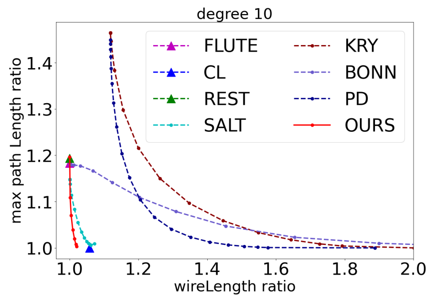
Fig. 9. The red point is the intersection point p of the three blocks. (a) shows the case of three blocks being jointly connected and (b) shows the case of only two blocks being connected.

TABLE I
Average running time over $10k$ test instances (s)

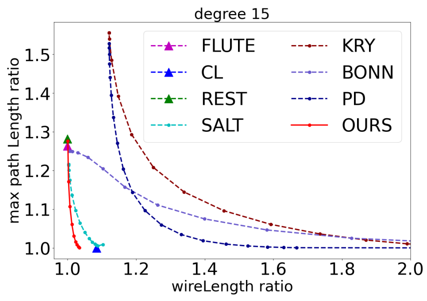
	degree 5	degree 10	degree 15	degree 20	degree 25	degree 30
SALT	0.26	0.77	2.15	5.34	10.15	16.93
OURS	0.28	0.85	2.22	4.71	8.82	14.72



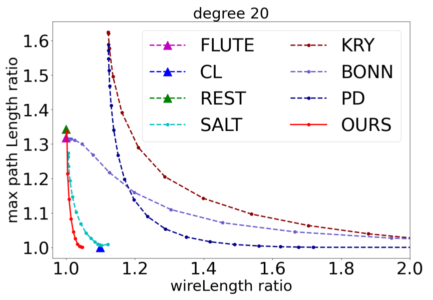
(a)



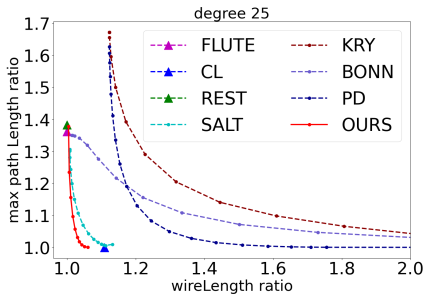
(b)



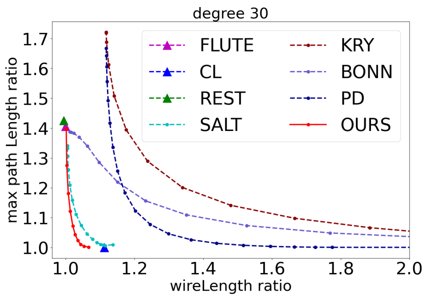
(c)



(d)

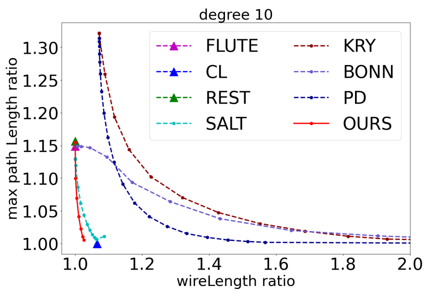


(e)

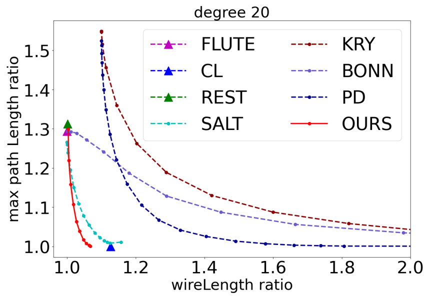


(f)

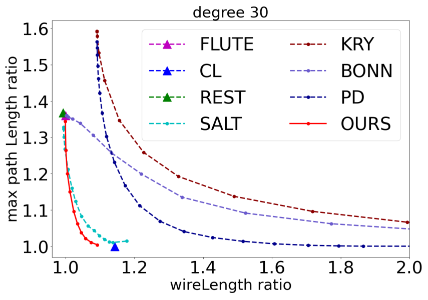
Fig. 10. (a) to (f) show the comparison results of the neural network model on the generated data with degree 5, 10, 15, 20, 25, and 30, respectively.



(a)

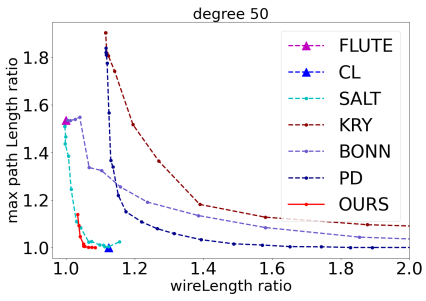


(b)

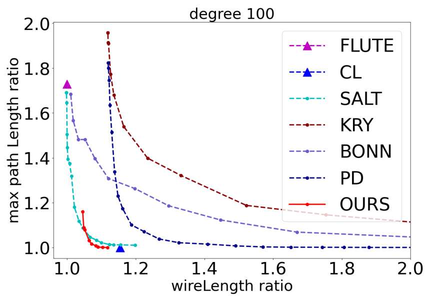


(c)

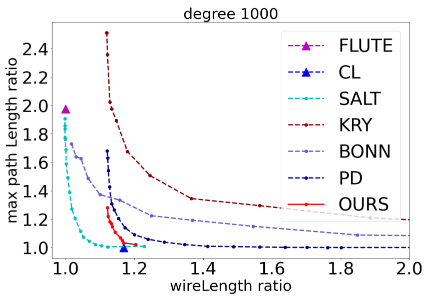
Fig. 11. (a) to (c) show the comparison results of the neural network model on the ICCAD 2015 benchmark with degree 10, 20, and 30, respectively.



(a)



(b)



(c)

Fig. 12. (a) to (c) show the comparison results of the divide & merge framework with degree 50, 100 and 1000, respectively.

solving moderate-degree and large-degree nets, respectively. Comparing with several existing algorithms, the experimental results suggest that our methods can achieve more promising trade-off between the wirelength and the max path length. In future, we can further consider other realistic factors, such as congestion and heat dissipation, for the routing problem.

References

[1] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI physical design: from graph partitioning to timing closure*. Springer, 2011, vol. 312.

[2] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of applied physics*, vol. 19, no. 1, pp. 55–63, 1948.

[3] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 7, pp. 890–896, 1995.

[4] G. Posser, E. F. Young, S. Held, Y.-L. Li, and D. Z. Pan, "Challenges and approaches in vlsi routing," in *Proceedings of the 2022 International Symposium on Physical Design*, 2022, pp. 185–192.

[5] M. Hanan, "On steiner's problem with rectilinear distance," *SIAM Journal on Applied mathematics*, vol. 14, no. 2, pp. 255–265, 1966.

[6] F. K. Hwang, "On steiner minimal trees with rectilinear distance," *SIAM journal on Applied Mathematics*, vol. 30, no. 1, pp. 104–114, 1976.

[7] D. M. Warme, P. Winter, and M. Zachariasen, "Exact algorithms for plane steiner tree problems: A computational study," in *Advances in Steiner trees*. Springer, 2000, pp. 81–116.

[8] H. Zhou, "Efficient steiner tree construction based on spanning graphs," in *Proceedings of the 2003 international symposium on Physical design*, 2003, pp. 152–157.

[9] A. B. Kahng, I. I. Măndoiu, and A. Z. Zelikovsky, "Highly scalable algorithms for rectilinear and octilinear steiner trees," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, 2003, pp. 827–833.

[10] C. Chu and Y.-C. Wong, "Flute: Fast lookup table based rectilinear steiner minimal tree algorithm for vlsi design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70–83, 2007.

[11] J. Liu, G. Chen, and E. F. Young, "Rest: Constructing rectilinear steiner minimum tree via reinforcement learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1135–1140.

[12] W. Shi and C. Su, "The rectilinear steiner arborescence problem is np-complete." in *SODA*, 2000, pp. 780–787.

[13] J. Córdova and Y.-H. Lee, "A heuristic algorithm for the rectilinear steiner arborescence problem," *CIS Department, Univ. of Florida, Tech. Rep. TR-94-025*, 1994.

[14] J. Cong, K.-S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed rc delay model," in *Proceedings of the 30th International Design Automation Conference*, 1993, pp. 606–611.

[15] M. Pan, C. Chu, and P. Patra, "A novel performance-driven topology design algorithm," in *2007 Asia and South Pacific Design Automation Conference*. IEEE, 2007, pp. 244–249.

[16] M. R. Garey and D. S. Johnson, "The rectilinear steiner tree problem is np-complete," *SIAM Journal on Applied Mathematics*, vol. 32, no. 4, pp. 826–834, 1977.

[17] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh, and C.-K. Wong, "Provably good performance-driven global routing," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 11, no. 6, pp. 739–752, 1992.

[18] S. Khuller, B. Raghavachari, and N. Young, "Balancing minimum spanning trees and shortest-path trees," *Algorithmica*, vol. 14, no. 4, pp. 305–321, 1995.

[19] G. Chen and E. F. Young, "Salt: provably good routing topology by a novel steiner shallow-light tree algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1217–1230, 2019.

[20] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong *et al.*, "Machine learning for electronic design automation: A survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 5, pp. 1–46, 2021.

[21] Z. Guo, F. Gu, and Y. Lin, "Gpu-accelerated rectilinear steiner tree generation," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[22] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[23] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 834–846, 1983.

[24] K. Mehlhorn and S. Näher, "Implementation of a sweep line algorithm for the straight\& line segment intersection problem," 1994.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[27] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[28] K. Oono and T. Suzuki, "Graph neural networks exponentially lose expressive power for node classification," *arXiv preprint arXiv:1905.10947*, 2019.

[29] K. Li, T. Zhang, and R. Wang, "Deep reinforcement learning for multiobjective optimization," *IEEE transactions on cybernetics*, vol. 51, no. 6, pp. 3103–3114, 2020.

[30] d. B. Mark, C. Otfried, v. K. Marc, and O. Mark, *Computational geometry algorithms and applications*. Spinger, 2008.

[31] R. Scheifele, "Steiner trees with bounded rc-delay," *Algorithmica*, vol. 78, pp. 86–109, 2017.

[32] Y.-C. Wong and C. Chu, "A scalable and accurate rectilinear steiner minimal tree algorithm," in *2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, 2008, pp. 29–34.

[33] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[34] M.-C. Kim, J. Hu, J. Li, and N. Viswanathan, "Iccad-2015 cad contest in incremental timing-driven placement and benchmark suite," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 921–926.