Arbitrary-size Multi-layer OARSMT RL Router Trained with Combinatorial Monte-Carlo Tree Search

Liang-Ting Chen, Hung-Ru Kuo, Yih-Lang Li, Mango C.-T. Chao
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
a123tim123@gmail.com,rose54306430@gmail.com,ylli@cs.nctu.edu.tw,mango@nycu.edu.tw

ABSTRACT

This paper presents a novel reinforcement-learning-trained router for building a multi-layer obstacle-avoiding rectilinear Steiner minimum tree (OARSMT). The router is trained by our proposed combinatorial Monte-Carlo tree search to select a proper set of Steiner points for OARSMT with only one inference. By using a Hanan-grid graph as the input and a 3D U-Net as the network architecture, the router can handle layouts with any dimensions and any routing costs between grids. The experiments on both random cases and public benchmarks demonstrate that our router can significantly outperform previous algorithmic routers and other RL routers using Alpha-Go-like or PPO-based training.

ACM Reference Format:
Liang-Ting Chen, Hung-Ru Kuo, Yih-Lang Li, Mango C.-T. Chao. 2024. Arbitrary-size Multi-layer OARSMT RL Router Trained with Combinatorial Monte-Carlo Tree Search. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656500

1 INTRODUCTION

The rectilinear Steiner minimum tree (denoted as RSMT) problem targets building a minimum-length routing tree connecting all given pins with only vertical and horizontal segments, and has been a classic NP-complete algorithmic problem. Its variant, the obstacle-avoiding rectilinear Steiner minimum tree (denoted as OARSMT) problem, adds obstacles to the routing problem and requests to find a minimum-length rectilinear Steiner tree without passing any obstacle. Compared to the conventional RSMT problem, the OARSMT problem attracts a higher interest from the EDA community especially when multiple routing layers are considered, since its problem formulation is closer to a real routing problem in an IC design, where macros, routing blockages, or pre-routed wires are often encountered and multiple routing layers are in use.

The previous algorithmic works on OARSMT can be classified into two main categories, single-layer OARSMT (*SL-OARSMT*) [1, 10, 11, 13, 18, 22] and multi-layer OARSMT (*ML-OARSMT*) [12, 14–16], depending on the number of routing layers used in the routing problem. Those previous works on SL-OARSMT can be classified into three categories based on their applied key methods. The first category is spanning-graph methods [13, 18, 22], which built a routing tree based on a spanning graph encoded with the locations of the pins and the corners of the obstacles. The second category is lookup-table methods [1], which extended the concept of the lookup-table method FLUTE for RSMT [5] to further handle obstacles. The third category is exact-algorithm methods [10, 11], which applied the concept of the exact algorithm GeoSteiner [25] for RSMT while reducing the number of full Steiner trees and obstacles to be considered.

Among previous algorithmic works on ML-OARSMT, the earliest work [12] extended a spanning-graph method to handle multiple routing layers. The later ML-OARSMT works [14–16] apply Steiner-point methods, which divided the process of finding an OARSMT into two main steps: (1) selecting a proper set of Steiner points and (2) constructing an obstacle-avoiding rectilinear minimum spanning tree (denoted as OARMST) to connect all the given pins and selected Steiner points. If the set of the selected Steiner points is optimal, the resulting spanning tree (OARMST) is an optimal Steiner tree (OARSMT). Since constructing an OARMST can be done in polynomial time, the OARSMT problem can be transferred into the problem of Steiner point selection, which is still NP-complete.

Instead of applying an algorithmic method, [4] developed a reinforcement learning (RL) framework to train a policy agent that can select proper Steiner points for building an OARSMT through repeated trials and self-learning. [4] treated the Steiner-point selection as a sequential process and asked the agent to select the best next Steiner points based on the given pins, obstacles, and previously selected Steiner points. The policy agent in [4] is a convolutional neural network trained by conventional Monte-Carlo tree search (MCTS) [2] plus UCT formula [20], which is also the policy-optimization scheme used by the well-known AlphaGo [23].

However, the RL framework in [4] has the following three major limitations. First, its agent selects one Steiner point at a time, and such a sequential process requires multiple inferences of the policy agent for obtaining all required Steiner points for one OARSMT construction, which leads to runtime inefficiency. Besides, the focus of Steiner point selection should be on finding the final selected Steiner points, not the order of Steiner points to be selected. Second, its policy agent can only handle layouts in a pre-defined size, meaning that changing the size of layouts needs to retrain a new policy agent. This inflexibility in processable layout size significantly limits the application of the router. Third, the agent in [4] can handle only SL-OARSMT, which again limits its application.

There are other recent works also applying RL techniques to solve a routing problem [3, 6, 9, 17]. [6] targeted the single-layer routing of multiple 2-pin nets and applied a collaborative game-theory model for training the agents to perform routing and rip-up & reroute. [17] trained an agent to construct a single-layer RSMT and applied an actor-critic approach with REINFORCE [26] for training. [9] trained an agent to perform single-layer PCB routing and applied MCTS for training. [3] trained an agent to perform rip-up and re-routing during detailed routing for custom circuits and applied PPO [21] for training.

In this paper, we propose a novel RL framework to train an agent that can select proper Steiner points for a multi-layer Steiner-point-based OARSMT router. Our RL framework applies an actor-critic approach with a newly proposed version of MCTS as its policy-optimization scheme, called the *combinatorial MCTS*, which can train the agent to directly infer all selected Steiner points at a time and in turn speed up the overall runtime of the OARSMT router while preserving the capability of exploring unseen solutions in a stochastically balanced way as the conventional MCTS. This combinatorial MCTS focuses on finding the final combination of Steiner points and avoids situations where the same group of Steiner points are repeatedly selected with different orders in MCTS, resulting in higher search efficiency and better training effectiveness. In addition, our agent adopts a 3D U-Net network architecture plus residual connections and can handle layouts with any length, any width, and any number of router layers. Moreover, the input layout to our trained agent is represented by a Hanan grid graph [7], and hence our router can handle different routing costs between adjacent grids, which along with the 3D Residual U-Net can significantly increase the generalizability and applicability of our ML-OARSMT router. The experimental results on both random layouts and previous public benchmarks show that our RL-based OARSMT router can significantly outperform top previous algorithmic ML-OARSMT routers on total routing cost and runtime. The experimental results also demonstrate that the proposed combinatorial MCTS can outperform other policy-optimization schemes including AlphaGo-like MCTS and PPO on both efficiency and effectiveness for training a ML-OARSMT agent and has the potential to be extended to other optimization algorithmic problems.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656500

2 BACKGROUND

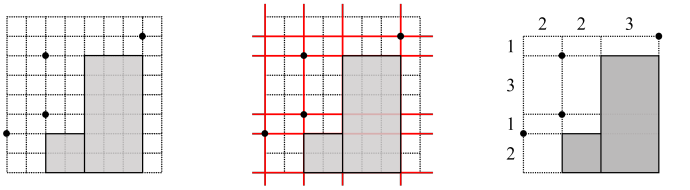
2.1 Problem Formulation of ML-OARSMT

Problem ML-OARSMT. Given a set of pins $\mathcal{P}$, a set of obstacles $\mathcal{O}$, and a via cost C_{via}, construct a multilayer rectilinear Steiner tree $\mathcal{T}$ that connects all pins in $\mathcal{P}$ using rectilinear paths, while ensuring that no edge or via intersects any obstacle in $\mathcal{O}$ and minimizing the cost of $\mathcal{T}$.

A Steiner tree in the definition means a routing tree that connects all pins with additional intermediate vertices, called Steiner points. The use of Steiner points allows the sharing of common routing segments for connecting multiple pins. Once the Steiner points are determined, the Steiner tree can be constructed by finding the obstacle-avoiding rectilinear minimum spanning tree connecting all pins and the Steiner points. In the resulting Steiner tree, a Steiner point with a degree less than 3 is considered redundant since it cannot act as an effective intermediate vertex to create the common routing segments shared by the connection of at least three pins or Steiner points. As a result, a layout with n pins needs at most $n-2$ irredundant Steiner points.

2.2 Hanan Grid Graph

A Hanan grid graph [7] is derived by intersecting horizontal and vertical cuts created at every pin and obstacle boundary. This approach, compared to the uniform grid graph where every coordinate is treated as a vertex, significantly reduces the graph's size, constraining the solution space of ML-OARSMT within a more manageable scope. The cost between adjacent vertices in this graph corresponds to the distance based on their original coordinates. Fig. 1 illustrates an example of converting a uniform 9×9 grid graph in Fig. 1(a) into a 4×5 Hanan grid graph in Fig. 1(c) through identifying all cuts (red lines) in Fig. 1(b), where black dots and gray areas represent pins and obstacles, respectively, and the routing cost of each vertical and horizontal grid line in the Hanan grid graph is listed on the sides.



(a) Uniform grid graph (b) Cuts at pins and obstacles (c) Hanan grid graph

Fig. 1: Converting a uniform grid graph into a Hanan grid graph.

In this work, the input is a layout represented by a 3D Hanan grid graph, on which a vertex can be a pin, an obstacle, or an empty location to place a Steiner point. A 3D Hanan grid graph (denoted as Hanan graph) is constructed by first consolidating all objects onto a single layer, followed by constructing a 2D Hanan grid graph for the consolidated layer. Next, each object is relocated to the new Hanan grid on its original layer.

3 PROPOSED ML-OARSMT ROUTER

3.1 Flow of Our RL Router

Fig. 2 illustrates the flow of our RL router. First, the input is an encoded 3D Hanan graph representing the layout for ML-OARSMT. Next, our trained Steiner-point selector will select proper Steiner points based on the input layout. Last, an obstacle-avoiding rectilinear minimum spanning (OARMST) router is applied to generate a minimum spanning tree connecting all the pins and irredundant Steiner points, and this minimum spanning tree is our final ML-OARSMT.

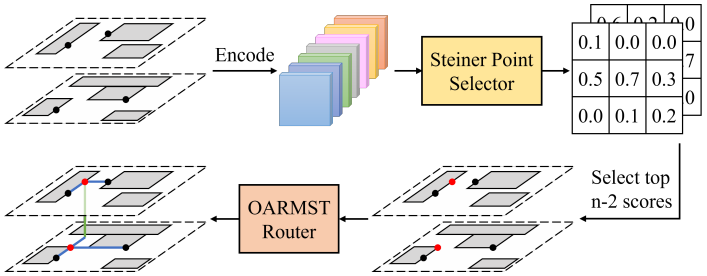


Fig. 2: Flow of our router with a trained Steiner-point selector.

The Steiner point selector here is implemented with a neural network, which takes a 3D Hanan graph with H horizontal grids, V vertical grids, and M routing layers as its input. The output of the neural network is a $H \times V \times M$ array, where each array entry represents the probability of the corresponding vertex in the Hanan graph being selected as a Steiner point. Then if there are n pins to be connected in the input layout, the vertices with the top $n-2$ highest probabilities will be selected as the Steiner points. As a result, determining all selected Steiner points only requires one inference of the neural network.

The OARMST router here follows the same algorithm in [14], which applies a Maze-router-based Prim's algorithm to construct the initial version of the minimum spanning tree connecting all the pins and selected Steiner points, removes redundant Steiner points on this initial version, and then reconstructs the minimum spanning tree connecting all the pins and the remaining irredundant Steiner points.

3.2 Key Ideas

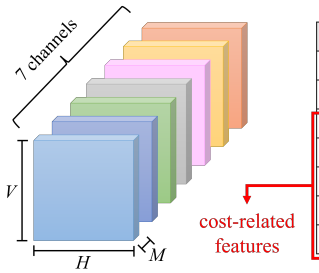
This subsection shows the key ideas for designing our RL framework. First, we plan to apply MCTS to train our agent because MCTS has been proven to be a powerful policy-optimization scheme that can effectively generate a better label of a sample based on a weaker agent. However, MCTS explores the solution space by adding one action (i.e., Steiner point) at a time while our agent, the Steiner point selector, focuses on identifying all n–2 Steiner points at once. Therefore, we need to design an MCTS-specific actor that can convert the result of the selector to a proper probability policy for determining which node should be explored during MCTS. Moreover, we also need a method to convert the result of MCTS into high-quality labels for training the selector directly.

Second, the conventional MCTS is designed to find out the best action at a current state, suitable for games where the order of actions matters, like the game of Go. However, Steiner-point selection cares more about the final combination of the selected Steiner points, not the order to be selected. Because the conventional MCTS treats the different sequences of the same group of Steiner points as different solutions, its search space in Steiner point selection can be redundant. Therefore, we need a mechanism in MCTS to avoid repeated searches on a duplicated combination of Steiner points and explore untried solutions more efficiently.

Third, we plan to design an agent that can handle a 3D layout with arbitrary size, meaning that the $H \times V \times M$ dimensions of an input Hanan graph may vary and the agent can output a 3D array of probabilities with the same $H \times V \times M$ dimensions accordingly. In other words, the neural network of the agent needs to be image-in-image-out and the training schedule needs to include layouts with different sizes and routing layers, to ensure the generalizability of the trained agent.

3.3 Feature Encoding and Neural Network

Fig. 3 shows the input feature encoding of a $H \times V \times M$ Hanan graph, where each vertex is encoded with seven features, representing whether the vertex is a pin, whether it is an obstacle, the routing cost to its immediate right, left, upstairs or downstairs vertex, and the routing cost of a via. This via cost is assumed to be the same for all vertices in a layout but its value may vary among different layouts. The five cost-related features are normalized by dividing itself by the maximum value in the entire set, ensuring that its value ranges between 0 and 1.



Channels	Explanation
Pin	1 for pin, 0 otherwise
Routable	1 for routable, 0 for an obstacle
X right	Cost to its right neighbor
X left	Cost to its left neighbor
Y up	Cost to its upstairs neighbor
Y down	Cost to its downstairs neighbor
Via cost	Cost of a via

Fig. 3: Input feature encoding of a Hanan-grid-graph represented layout.

To build an image-in-image-out neural network that can handle an arbitrary-sized 3D Hanan graph, we choose U-Net [19] with 3D convolution kernels [24] plus 3D convolutional residual blocks [8] as our network architecture. Fig. 4 illustrates the overall model architecture, all with the kernel size of $3 \times 3 \times 3$. The input is a $7 \times H \times V \times M$ array, representing the 3D Hanan graph. The output is a $H \times V \times M$ array, representing the probability of a vertex in the Hanan graph being selected as one of the final Steiner points. Each output entry here is passed through

a sigmoid activation function to ensure that every outputted value ranges between 0 and 1.

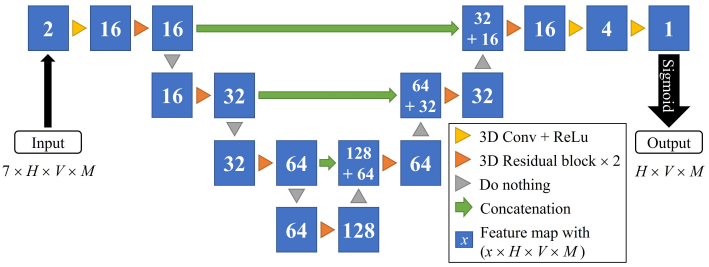


Fig. 4: U-Net-based network architecture of our agent.

3.4 Combinatorial MCTS

This subsection introduces the details of our proposed combinatorial MCTS, which still applies the Monte Carlo tree search to explore the solution space by trying one action (i.e., one Steiner point) at a time, but uses different mechanisms to sample actions for exploration and summarize the result of the entire search tree to form a label for training the Steiner-point selector that focuses on selecting a proper combination of Steiner points. In other words, our objective is to identify a proper combination of the final selected Steiner points, not the best next Steiner point to be selected as the conventional MCTS may lead to.

In a Monte-Carlo (MC) search tree, a node represents a state and an edge represents an action that transforms the edge's parent node (current state) into the edge's child node (next state). Following are the definitions of a state and an action in our Steiner-point selection problem.

State: A layout represented by a Hanan graph containing pins and obstacles, where previously selected Steiner points are also included and treated as normal pins.

Action: Select a valid vertex (node) in the current layout (state) to place a Steiner point. A valid vertex should: (1) not be an existing pin (including a selected Steiner point) or obstacle, and (2) locate at a coordinate with a lower selection priority than that of the previously placed Steiner point. A coordinate in our 3D Hanan graph is a triple (h,v,m), corresponding to the vertex located at the hth horizontal grid, vth vertical grid, and the mth routing layer. A vertex has a higher selection priority than another if the lexicographic order of its coordinate (h,v,m) is smaller than the other. Selecting an action with this selection priority can guarantee that each node in the search tree represents a unique combination of Steiner points and leads to a more compacted search space.

The construction of an MC search tree starts with a root, an initial layout with n pins, and no Steiner point placed yet. Then one action is taken at a time to expand the search tree repeatedly. A node becomes a terminal state, meaning that no child node of it will be explored, if one of the following criteria holds: (1) the node is at the $(n-2)$th level, meaning that the state contains $n-2$ selected Steiner points already, (2) the last action increases the routing cost, and (3) the routing cost stays the same for three consecutive actions. These criteria are designed to prune the search of ineffective combinations of Steiner points and in turn, can enhance the exploration efficiency.

During combinatorial MCTS, we need two critical roles: (1) an *actor* to generate a policy for determining the probability of each valid node being selected for exploration, where the summation of all the probability should be one in a policy, and (2) a *critic* to predict the final routing cost of a current node (a combination of currently selected Steiner points) after the remaining Steiner points are included later. In a conventional actor-critic RL environment, both the actor and critic are neural-network-based and trained through interaction with the environment, and the actor is usually the agent that plays the game, which is not our case. Our agent outputs a 3D probabilistic array indicating the probability of each vertex being selected as a Steiner point and the summation of all the probabilities here exceeds one since multiple vertices should be selected at a time. Such a 3D probabilistic array outputted by our agent cannot be directly used as a policy to determine which node should be explored in MCTS, and hence an additional actor on top of the agent is needed.

In our combinatorial MCTS, both the actor and critic are built upon the agent, the Steiner-point selector, and hence the upgrade of the agent can further upgrade the actor and critic as well. Fig. 5 illustrates how the actor and the critic can transform the result of the Steiner-point selector to perform their task.

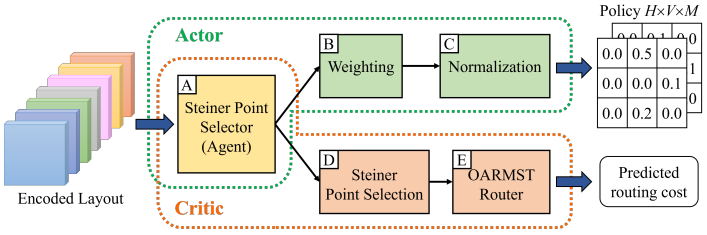


Fig. 5: Our actor and critic built upon the Steiner-point selector.

As shown in the green dash box in Fig. 5, the actor takes three steps to generate the action policy for the current state (layout). Firstly (Step-A in Fig. 5), the actor applies the Steiner-point selector to obtain the probability of each vertex v being selected at the end, called the *final selected probability* in our later discussion and denoted as $fsp(v)$, based on the current state. Second (Step-B in Fig. 5), for a valid vertex u to be evaluated at the current state, its weighted probability $p'(u)$ can be calculated by eq. (1) with w as the last selected Steiner point at the current state. For an invalid vertex, its weighted probability is zero.

$$p'_{(u)} = fsp(u) \times \prod_{w \prec v \prec u} (1 - fsp(v)) \tag{1}$$

In eq. (1), $v \prec u$ means that the selection priority of v is higher than that of u. As a result, the weighted probability $p'(u)$ means the probability that u is selected under the condition that each valid vertex v with a higher priority than the targeted vertex u and a lower priority than the last selected Steiner point w is not selected. This calculation of $p'(u)$ follows the selection priority defined for an action to be explored in our search tree. Last (Step-C in Fig. 5), the weighted probabilities for all valid vertices are normalized by dividing each value by the total sum. Then the normalized probabilities form the action policy as the actor's output.

The orange dashed box in Fig. 5 shows how the critic estimates the final routing cost of a given node (state) at the ith level, meaning that i Steiner points are already selected in this state. The critic applies the Steiner-point selector to obtain the final selected probability of each vertex, select the vertices with the top $n-2-i$ highest probability as the remaining Steiner points, perform the OARMST router to construct the minimum spanning tree connecting all pins and Steiner points, and calculate its total routing cost. This routing-cost estimation performed by the critic is similar to the procedure of applying the Steiner-point selector to generate the final OARSMT as shown in Fig. 2, except that the critic here only selects the remaining $n-2-i$ Steiner points since at most $n-2$ Steiner points are required for a n-pin layout.

Here, some notations are defined to explain the detailed process of MCTS. In an MC search tree, an edge is associated with a 2-tuple (s, a), where s is the current state and a is the action applied to s. Each edge (s, a) needs to record the following four additional pieces of information: the visit count $N(s, a)$, the prior probability $P(s, a)$, the total value $W(s, a)$, and the average value $Q(s, a)$. The actor and critic, parameterized by θ (the neural network parameters of the Steiner point selector), are denoted as p_θ and c_θ, with $p_\theta(s)$ representing the policy for state s and $p_\theta(a|s)$ indicating the selection probability of action a under the policy $p_\theta(s)$. A state may include previously placed Steiner points, treated as normal pins by the actor and critic.

Fig. 6 illustrates an exploration iteration in MCTS, including the following four steps: selection, expansion, simulation, and backpropagation.

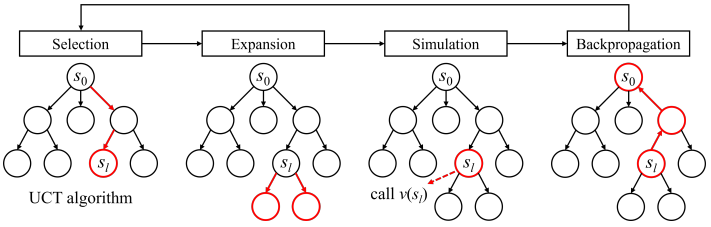


Fig. 6: Flow of one exploration iteration in MCTS.

Selection: This step starts from the root and uses the UCT formula [20] to select the next traversed edge until a leaf node s_l is reached. At each non-leaf node, the next traversed edge (s, a) is determined by maximizing

the expression $Q(s,a) + U(s,a)$, where

$$U(s,a) = P(s,a) \times \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{2}$$

This search strategy initially favors actions with a high prior probability and low visit count (high $U(s,a)$). However, as the search progresses and $N(s,a)$ increases, it shifts to favor actions with high action value $Q(s,a)$.

Expansion: The actor p_θ participates in this step. A new child node of s_l is generated for each valid Steiner point a. For each edge leading to a new child node here, the prior probability $P(s_l,a)$ is set to $p_\theta(a|s_l)$.

Simulation: The critic c_θ participates in this step. The value function $v(s_l)$ assesses the leaf node s_l with $v(s_l) = (rc_{s_0} - c_\theta(s_l))/rc_{s_0}$, which contrasts the initial routing cost rc_{s_0} with the predicted routing cost $c_\theta(s_l)$. A larger $v(s_l)$ indicates that progressing from this state is likely to reduce the routing cost more significantly.

Backpropagation: In this step, the edge data from the leaf node s_l to the root is updated, where $N(s,a)$ is incremented by one, $W(s,a)$ is incremented by the value $v(s_l)$, and $Q(s,a)$ is recalculated as $W(s,a)/N(s,a)$.

After α MCTS iterations, the action at the root $a_{\max}$ with the highest visit count $N(s_0, a_{\max})$ is executed, and the root changes to that child node. We set $\alpha = 2000$ for a $16\times16\times4$ layout, scaling it for a larger layout proportionally to the size increase. Following an executed action, the agent conducts another α iterations before executing the next action and moving to a new root. This process continues until the new root node becomes a terminal state. Subsequently, the entire MCTS process is encoded into a training label associated with the initial layout.

3.5 Sample Generation in Combinatorial MCTS

One key property of MCTS is that it can generate direct training samples on which the agent implemented with a neural network can be fitted by supervised learning. In combinatorial MCTS, one training sample is generated after building one MC search tree on an initial layout. The input of the training sample here is the initial layout represented by a Hanan graph. The output label of the sample is the array of the final selected probability of each vertex v, denoted as $L_{fsp}(v)$. This labeled final selected probability $L_{fsp}(v)$ is calculated based on the result of MCTS on the search tree with the following equation:

$$L_{fsp}(v) = n_{sel}(v)/n_{opp}(v) \tag{3}$$

where $n_{sel}(v)$ is the total count of the vertex v actually being selected and $n_{opp}(v)$ is the total opportunities that v can be selected during the MCTS selection step. The calculated $L_{fsp}(v)$ reflects the actual probability of v being selected as a Steiner point during MCTS, where each selection is made by the UCT formula for exploring the probabilistically best action that can help to reduce the routing cost the most. In other words, a higher calculated $L_{fsp}(v)$ implies that selecting v as a Steiner point can reduce the routing cost more significantly, which is exactly what our Steiner-point selector should learn from the label of a training sample.

In the conventional MCTS, one training sample is generated at each executed node in an MC search tree by collecting the frequency of each of its child nodes being visited during MCTS as the sample's label since the objective of the conventional MCTS is to learn which vertex is probabilistically the best next Steiner point to select at a current state. On the other hand, our combinatorial MCTS collects the label and produces a training sample after the entire MC search tree is built since our objective is to learn which vertex should be probabilistically included as one of the final selected Steiner points at the end, which is a key difference compared to the conventional MCTS.

Fig. 7 shows an example of counting n_{sel} and n_{opp}, where a $4\times4\times1$ layout is used with its pins and obstacles highlighted in Fig. 7(b). Fig. 7(a) shows a search tree for this layout with s_0 as its initial state. Each edge shows the action of selecting a vertex and each node shows the already selected vertices at the state. The red arrows indicate the action selected in this selection step. After this selection is complete, the n_{sel} and n_{opp} for each vertex v are updated as shown in Fig. 7(b).

Fig. 8 illustrates the process of training our Steiner point selector with combinatorial MCTS. Initially, combinatorial MCTS is performed on random layouts to generate training samples. Once sufficient training samples are collected, the Steiner point selector is directly fitted with the collected training samples using binary cross-entropy loss, completing what is referred to as a stage. Once the selector is updated after a stage, the actor and critic in MCTS used in the next stage are upgraded accordingly, so that the selector can keep on evolving as the training stages move forward.

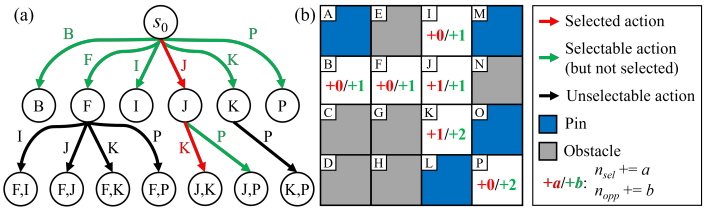


Fig. 7: Example of calculating n_{sel} and n_{opp}.

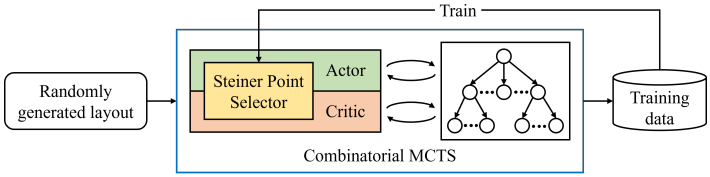


Fig. 8: Evolution of a selector with combinatorial MCTS.

3.6 Training Schedule

This subsection describes the training schedule of our Steiner point selector, including a mixed-size training process that enables the Steiner point selector to adapt to various sizes of a Hanan graph. In our training data, the $H\times V$ dimensions include 16×16, 24×24, and 32×32, each paired with a M dimension of 4, 6, 8, and 10 routing layers, resulting in 12 different layout sizes. The edge costs in these Hanan graphs range from 1 to 1000 and the via costs C_{via} range from 3 to 5. For the $16\times16\times4$ layouts, the number of obstacles ranges from 32 to 64, each with a size of either 1×3 or 1×4 placed horizontally or vertically. Obstacles can overlap to form a more complicated shape.

In each stage, 1000 random layouts of each size are generated to produce training samples through combinatorial MCTS. Next, each sample undergoes data augmentation, including rotation (0°, 90°, 180°, 270°) and reflection across the y and z axes, producing 16-fold ($4 \times 2 \times 2$) more training samples.

Fig. 9 illustrates an epoch of mixed-size training, where the Steiner-point selector is sequentially updated with a batch of 256 training samples of the same layout size. An epoch ends when all generated samples after data augmentation in a stage are included in a batch. Each stage consists of four epochs. Note that placing samples with the same layout size in a batch can achieve better computation efficiency for a GPU than that with mixed layout sizes.

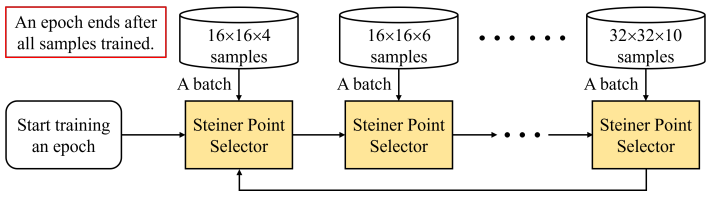


Fig. 9: An epoch in mixed-size-layout training process.

In the first four training stages, we employ curriculum learning, starting from layouts with 3 pins and progressively advancing to those with 6 pins. During this phase, we do not use the critic's predicted values to calculate the value function for a state; instead, we directly calculate the routing cost resulting from the already selected Steiner points at the state. This approach is adopted because the critic's predictions are rough in the early stages, leading to unstable training. From the fifth stage onward, layouts contain a random number of pins from 3 to 6 and the critic is used to predict routing costs. The agent used in later experiments is trained by 32 training stages with a total training time of 159 hours.

4 EXPERIMENTAL RESULTS

This work was developed using Python and C++. The neural network training was implemented with Python and PyTorch, the combinatorial MCTS and OARMST router was implemented in C++, and the final integration was in Python. The machine in use is a server with a 2.10GHz Intel Xeon Silver 4310 CPU, 512 GB of RAM, an Nvidia TITAN RTX GPU, and an Nvidia A40 GPU.

4.1 Compared to Algorithmic Routers

We first compare our RL router to the best algorithmic router [14] among the previous ML-OARSMT works on randomly generated layouts. The result of [14] in the following experiments is directly generated by the executable provided by [14]. Table 1 lists the setting of the randomly generated layouts whose dimensions are specified by the size of their corresponding Hanan graph. As Table 1 shows, the $H \times V$ dimensions of a Hanan graph range from 32×32 to 512×512 and the number of layers M ranges from 4 to 10. The range of the number of pins or obstacles also increases along with the size of the layout dimensions. Note that the number of layouts in some larger-size subsets may be lower than 50K because putting more layouts into each of those subsets may make the total runtime of [14]'s router on that subset longer than 24 hours.

Table 1: Setting of each randomly generated test subset.

test subset	# layouts	H	V	M	# pins	# obstacles
T32	50,000	32	32	4~10	3~10	128~640
T64	50,000	64	64	4~10	12~40	512~2560
T128	50,000	128	128	4~10	48~160	2,048~10,240
T128_2	50,000	128	256	4~10	96~320	4,096~20,480
T256	16,000	256	256	4~10	192~640	8,192~40,960
T256_2	1,000	256	512	4~10	384~1,280	16,384~81,920
T512	360	512	512	4~10	768~2,560	32,768~163,840

Table 2 first lists the average routing cost of the routing trees resulting from [14] and our RL router, respectively, and their difference in the ratio of [14]'s result. As Table 2 shows, the average routing cost resulting from our router is always smaller than that from [14] for each of the test subsets and their difference ranges from 2.256% to 2.681%. Table 2 also lists the average improvement ratio (denoted as avg. imp. ratio), which is calculated by finding the ratio of the improvement (difference) on routing cost for each layout first and then averaging the difference ratios for all layouts. Compared to the difference ratio of the average routing cost, this average improvement ratio can avoid potential bias where the reported improvement is mainly dominated by the layouts resulting in a larger routing cost (usually with more pins). As the result shows, the average improvement ratio resulting from our router is quite close to the improvement ratio on the average routing cost, demonstrating that this routing-cost improvement is not just from the layouts requiring a larger routing cost, but from the improvement across all layouts.

Table 2: Routing-cost comparison between [14] and our router.

test subset	average routing cost			avg. imp. ratio	win rate	loss rate
	[14] (a)	ours (b)	(a-b)/a			
T32	266,253	259,115	2.681%	2.483%	64.7%	14.1%
T64	1,200,293	1,170,560	2.477%	2.473%	93.7%	6.2%
T128	4,827,819	4,714,912	2.339%	2.356%	99.5%	0.5%
T128_2	9,641,349	9,418,959	2.307%	2.328%	100.0%	0.0%
T256	19,110,178	18,676,999	2.267%	2.289%	100.0%	0.0%
T256_2	37,886,869	37,032,178	2.256%	2.273%	100.0%	0.0%
T512	75,953,445	74,212,417	2.292%	2.308%	100.0%	0.0%

Table 2 further lists the percentage of layouts in which our router results in a smaller routing cost (denoted as win rate) or a larger one (lost rate). As the result shows, our router wins on the majority of the layouts and this win rate increases as the $H \times V$ dimensions of a Hanan graph increase. When the $H \times V$ dimensions exceed 128×256, our router can outperform [14] on all generated random layouts. Note that the largest $H \times V$ dimensions of a Hanan graph and the largest number of pins used in our training are 32×32 and 6, respectively, which are both on the lower end of our test subsets. This result demonstrates that our router can consistently outperform the best algorithmic work [14] even on the layouts larger and more complicated than those used for training, which is evidence that the knowledge learned and stored in our designed 3D U-Net is generalized enough to be applied to those unexplored larger cases. In addition, the improvement ratios on the two test subsets, T128_2 and T256_2, with unequal H and V dimensions are all similar to other test subsets even though the training data contains layouts with equal H and V dimensions only, again proving the generalizability of our router.

Table 3 compares the average runtime of [14]'s router and our router for each test subset. Our runtime is further divided into the runtime for Steiner point selection with the trained agent and the total runtime including the later OARMST construction. As Table 3 shows, the average runtime of our router is slightly slower than that of [14] for the smallest test subset, T32, but then the speedup increases dramatically as the layout dimensions, number of pins, and the number of obstacles increase. This result demonstrates that our router can efficiently select all desired Steiner points and construct the corresponding OARMST, and in turn be more scalable to larger, more complicated layouts than [14]. In addition, the growth of runtime on Steiner-point selection is mild because our trained agent can output all required Steiner points with only one model inference, which is a key advantage of our designed RL router.

Table 3: Runtime comparison between [14] and our router.

test subset	[14]'s avg. runtime in sec. (a)	our avg. runtime in sec.		speedup (a/b)
		S.-point. select	total (b)	
T32	0.0055	0.0032	0.0065	0.8x
T64	0.0642	0.0091	0.0272	2.4x
T128	0.6524	0.0339	0.1172	5.6x
T128_2	2.8694	0.0672	0.2404	11.9x
T256	5.4737	0.1417	0.5101	10.7x
T256_2	91.321	0.4500	1.2244	74.6x
T512	187.09	0.8781	2.4761	75.6x

Fig. 10 further plot the average improvement ratio of our router over [14] versus the layouts with different obstacle ratios (defined as the area of obstacles over the overall layout area) for each test subset. As Figure 10 shows, the average improvement ratio in general increases as the obstacle ratio increases across all test subsets. This result demonstrates that our RL router can outperform [14] by a larger margin when the layout becomes more difficult to route.

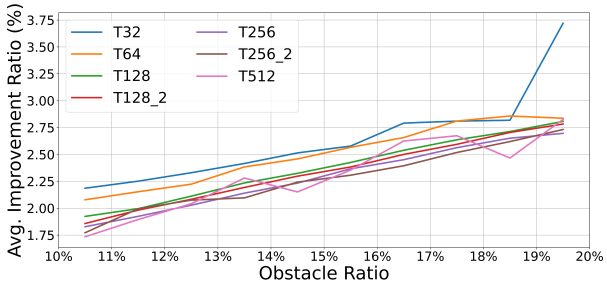


Fig. 10: Average improvement ratio against [14] vs. obstacle ratio.

Table 4 compares our RL router to three algorithmic ML-OARSMT routers developed by [12], [16] and [14], respectively, on eight public benchmark layouts, which were used in those previous works for evaluating a router's effectiveness. Note that we don't have the executables for the router from [12] or [16], and hence the reported routing costs of [12] or [16] are directly copied from their paper.

Table 4 lists the benchmark information, the routing cost resulting from each router, and the routing-cost improvement ratio achieved by our router against each of the three previous works. As the result on these eight benchmark layouts shows, our router can achieve an average routing-cost improvement ratio of 4.75%, 0.99% and 0.61% against [12], [16] and [14], respectively, demonstrating that the advantage of our RL router can be held on not only our randomly generated layouts but also the public benchmarks on which the previous works evaluated the effectiveness of their algorithmic routers.

4.2 Compared to Other RL-trained Routers

In this subsection, we compare our RL router with two RL routers trained with a different policy-optimization scheme than our combinatorial MCTS. The two RL routers for comparison apply the conventional AlphaGo-like MCTS [2] and PPO [21] for policy optimization, and are denoted as the AlphaGo-like router and the PPO router, respectively, in the following experiments. Note that both AlphaGo-like and PPO routers use actor-critic training approach, and their agents are all sequential Steiner-point selectors, meaning that the agents select one Steiner point at a time and the selected Steiner points will then become part of the inputs for selecting the next one.

In the experiments of Fig. 11, we first train and infer the three routers based on only fixed-size $24 \times 24 \times 4$ layouts, on which the training can converge faster and be easier to control. Their performance is evaluated by the ratio of the routing cost of the resulting Steiner tree built based on the selected Steiner points over that of the minimum spanning tree

Table 4: Routing-cost comparison to [12], [16] and [14] on public benchmark layouts, where via cost $C_{via} = 3$.

testcases	Hanan graph			# pins	# obstacles	total routing cost				improvement ratio against		
	H	V	M			[12] (a)	[16] (b)	[14] (c)	ours (d)	[12] (a-d)/a	[16] (b-d)/b	[14] (c-d)/c
rt1	45	44	10	25	10	4,334	4,169	4,150	4,055	6.437%	2.734%	2.289%
rt2	136	131	10	100	20	9,434	9,132	9,031	9,029	4.293%	1.128%	0.022%
rt3	294	285	10	250	50	15,569	14,750	14,757	14,556	6.507%	1.315%	1.362%
rt4	458	449	10	500	50	22,034	21,013	20,967	20,912	5.092%	0.481%	0.262%
rt5	702	707	4	1000	1000	27,890	26,970	26,864	26,828	3.808%	0.527%	0.134%
ind1	33	28	4	50	6	55,537	54,207	53,701	53,605	3.479%	1.111%	0.179%
ind2	83	191	5	200	85	12,512	12,008	12,085	12,053	3.668%	-0.375%	0.265%
ind3	221	223	9	250	13	10,973	10,555	10,491	10,453	4.739%	0.966%	0.362%
avg.	-			-	-	-	-	-	-	4.753%	0.986%	0.609%

without using any Steiner point, which is denoted as the *ST-to-MST ratio*. The lower the ST-to-MST ratio, the higher the level of the routing-cost reduction that can be achieved by the use of the selected Steiner points. Fig. 11 plots their average ST-to-MST ratio over the testing set versus the training time, where 10K layouts are randomly generated for each number of pins in the testing set. Fig. 11(a) first shows the average ST-to-MST ratio based on random layouts with 3~6 pins, which is the same range of the number of pins used for training. As Fig. 11(a) shows, the ST-to-MST ratio resulting from our router is always lower than that from the AlphaGo-like router or the PPO router at any point of the training runtime. This result demonstrates that our combinatorial MCTS is indeed a more effective policy-optimization scheme to train a high-quality Steiner-point selector for ML-OARSMT. Also, the performance of the PPO router is obviously much worse than the other two MCTS-based routers.

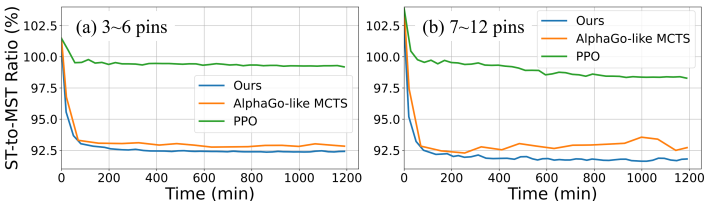


Fig. 11: ST-to-MST ratio vs. training time on $24\times24\times4$ layouts.

Compared to the AlphaGo-like router, our router can always maintain a noticeable lead after the first 100 minutes of training on Fig. 11(a), and this lead will be more significant when applying the routers to layouts with 7~12 pins, which exceeds the range of the number of pins used for training, as shown in Fig. 11(b). This result demonstrates that our proposed combinatorial MCTS can outperform conventional MCTS in building not only a more effective Steiner-point selector but also a more generalized one that can better handle cases beyond the range of training.

Fig. 12 plots the results on training and inferencing the three routers on only $32\times32\times4$ layouts, where a similar trend can be observed as in Fig. 11 except that our lead against AlphaGo-like router becomes more significant on either 3~6-pin or 7~12-pin layouts as the training time passes. This result shows that our proposed combinatorial MCTS owns a better edge against the conventional MCTS when the size of layouts increases. More importantly, the agents of both PPO and AlphaGo-like routers are sequential ones, and hence their runtime is much longer than ours since they need n–2 times of model inferencing to sequentially obtain $n-2$ Steiner points on a n-pin layout while our router requires only one model inferencing. The average speedup of our router against AlphaGo-like router is 1.67X and 3.54X for 3~6-pin and 7~12-pin $32 \times 32 \times 4$ layouts, respectively. In addition, the average runtime of producing a training sample with our combinatorial MCTS is 1.16 seconds, which is 3.48X faster than that of the conventional MCTS. This is because the combinatorial MCTS only explores a node with a lower search priority than the previously explored node, resulting in a smaller number of nodes being expanded and simulated during one iteration of MCTS. Such an arrangement on the node exploration can help not only avoid duplicated combinations of final selected Steiner points but also increase the search efficiency.

5 CONCLUSION

In this paper, we proposed the combinatorial MCTS, which can train a Steiner-point selector to directly select the final combination of Steiner points with only one model inference and achieve better efficiency and effectiveness on solution-space exploration and agent training compared to the conventional MCTS. Our RL router can handle layouts with arbitrary size and multiple routing layers and significantly outperform previous algorithmic routers on random layouts and public benchmarks.

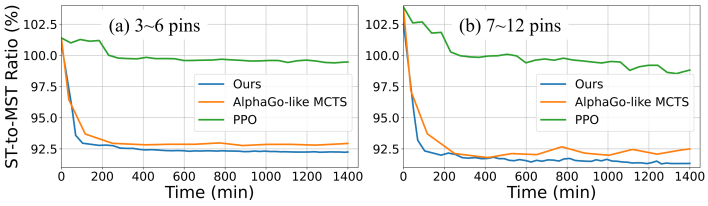


Fig. 12: ST-to-MST ratio vs. training time on $32\times32\times4$ layouts.

REFERENCES

[1] Gaurav Ajwani et al. 2010. FOARS: FLUTE based obstacle-avoiding rectilinear Steiner tree construction. In *Proc. of ISPD '19*. 27–34.

[2] Guillaume Chaslot et al. 2008. Monte-carlo tree search: A new framework for game ai. In *Proc. of AAAI AIIDE*, Vol. 4. 216–217.

[3] Hao Chen et al. 2023. Reinforcement Learning Guided Detailed Routing for Custom Circuits. In *Proc. of ISPD 2023*. ACM, 26–34.

[4] Po-Yan Chen et al. 2022. A Reinforcement Learning Agent for Obstacle-Avoiding Rectilinear Steiner Tree Construction. In *Proc. of ISPD 2022*. ACM, 107–115.

[5] Chris Chu et al. 2007. FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design. *IEEE TCAD* 27, 1 (2007), 70–83.

[6] Upma Gandhi et al. 2019. A reinforcement learning-based framework for solving physical design routing problem in the absence of large test sets. In *2019 ACM/IEEE MLCAD*. IEEE, 1–6.

[7] Maurice Hanan. 1966. On Steiner's problem with rectilinear distance. *SIAM J. Appl. Math.* 14, 2 (1966), 255–265.

[8] Kaiming He et al. 2016. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*. 770–778.

[9] Youbiao He et al. 2022. Circuit Routing Using Monte Carlo Tree Search and Deep Reinforcement Learning. In *2022 Int. Symp. VLSI-DAT*. 1–5.

[10] Tao Huang et al. 2011. On the construction of optimal obstacle-avoiding rectilinear Steiner minimum trees. *IEEE TCAD* 30, 5 (2011), 718–731.

[11] Tao Huang et al. 2013. ObSteiner: an exact algorithm for the construction of rectilinear Steiner minimum trees in the presence of complex rectilinear obstacles. *IEEE TCAD* 32, 6 (2013), 882–893.

[12] Chung-Wei Lin et al. 2008. Multilayer obstacle-avoiding rectilinear Steiner tree construction based on spanning graphs. *IEEE TCAD* 27, 11 (2008), 2007–2016.

[13] Chung-Wei Lin et al. 2008. Obstacle-avoiding rectilinear Steiner tree construction based on spanning graphs. *IEEE TCAD* 27, 4 (2008), 643–653.

[14] Kuen-Wey Lin et al. 2018. A maze routing-based methodology with bounded exploration and path-assessed retracing for constrained multilayer obstacle-avoiding rectilinear Steiner tree construction. *TODAES* 23, 4 (2018), 1–26.

[15] Chih-Hung Liu et al. 2012. Obstacle-avoiding rectilinear Steiner tree construction: A Steiner-point-based algorithm. *IEEE TCAD* 31, 7 (2012), 1050–1060.

[16] Chih-Hung Liu et al. 2014. Efficient Multilayer Obstacle-Avoiding Rectilinear Steiner Tree Construction Based on Geometric Reduction. *IEEE TCAD* 33, 12 (2014), 1928–1941.

[17] Jinwei Liu et al. 2021. REST: Constructing Rectilinear Steiner Minimum Tree via Reinforcement Learning. In *2021 58th ACM/IEEE DAC*. 1135–1140.

[18] Jieyi Long et al. 2008. EBOARST: An efficient edge-based obstacle-avoiding rectilinear Steiner tree construction algorithm. *IEEE TCAD* 27, 12 (2008), 2169–2182.

[19] Olaf Ronneberger et al. 2015. U-net: Convolutional networks for biomedical image segmentation. In *Proc. of MICCAI 2015*. Springer, 234–241.

[20] Christopher D Rosin. 2011. Multi-armed bandits with episode context. *Ann. Math. Artif. Intell.* 61, 3 (2011), 203–230.

[21] John Schulman et al. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[22] Zion Shen et al. 2005. Efficient rectilinear Steiner tree construction with rectilinear blockages. In *ICCD 2005*. IEEE, 38–44.

[23] David Silver et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[24] Du Tran et al. 2015. Learning spatiotemporal features with 3d convolutional networks. In *ICCV 2015*. 4489–4497.

[25] David M Warme et al. 2000. Exact algorithms for plane Steiner tree problems: A computational study. In *Advances in Steiner trees*. Springer, 81–116.

[26] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8 (1992), 229–256.