REST: Constructing Rectilinear Steiner Minimum Tree via Reinforcement Learning

Jinwei Liu
CSE Department, CUHK
jwliu@cse.cuhk.edu.hk

Gengjie Chen
CSE Department, CUHK
gjchen@cse.cuhk.edu.hk

Evangeline F.Y. Young
CSE Department, CUHK
fyyoung@cse.cuhk.edu.hk

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586209

***Abstract*—Rectilinear Steiner Minimum Tree (RSMT) is the shortest way to interconnect a net's n pins using rectilinear edges only. Constructing the optimal RSMT is NP-complete and non-trivial. In this work, we design a reinforcement learning based algorithm called REST for RSMT construction. After training, REST constructs RSMT of $\leq 0.36\%$ length error on average for nets with ≤ 50 pins. The average time needed for one net is fewer than 1.9 ms, and is much faster than traditional heuristics of similar quality. This is also the first successful attempt to solve this problem using a machine learning approach.**

I. Introduction

The rectilinear Steiner minimum tree (RSMT) problem is a fundamental problem in electronic design automation (EDA) and computer science. Consider the problem of interconnecting n pins on a circuit board using rectilinear edges only, or the problem of building perpendicular roads to connect n cities. RSMT is the shortest possible solution one can come up with. In the domain of EDA, RSMT construction is especially important, and is used for net routing, wire length estimation, etc. Despite its importance, the problem has been proven to be NP-complete [1], and solving it optimally is non-trivial.

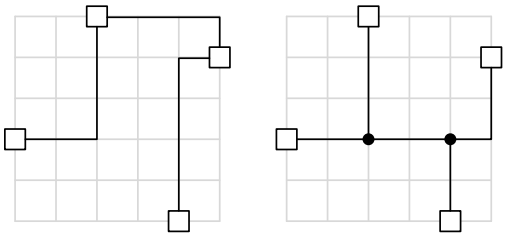
There have been lots of works studying RSMT construction related problems. In practice, it is common to use rectilinear minimum spanning tree (R-MST) to approximate RSMT, since R-MST can be efficiently constructed in $O(n \log n)$ time [2]. It is even proven that the length of an R-MST is at most $1.5\times$ of the optimal RSMT length [3]. For heuristics, Kahng et al. [4] devised an RSMT construction algorithm that computes an R-MST first, and iteratively improves its quality. This is done by constantly replacing bad edges with better ones. FLUTE [5], on the other hand, adopts a look-up table approach, and is the most efficient heuristic so far. They first solve all small instances with $degree \leq 9$, and build a look-up table. In actual use, the small instances are solved directly by the look-up table. Larger nets are first broken down into small nets that can be handled by the look-up table. The solutions of the small nets are then merged back to form that of the original net. Besides, GeoSteiner [6] is an efficient optimal algorithm that enumerates all possible full Steiner tree to form an RSMT. It is proven that an optimal RSMT can always be found by combining full Steiner trees only, which are Steiner trees with a special structure. The running time of GeoSteiner inevitably goes to exponential.

As the study of machine learning evolves, researchers are attempting to solve several hard combinatorial problems using machine learning based methodologies. This kind of approaches have shown several advantages over the traditional heuristics, e.g., shorter time for development, superior quality and speed for small to middle size instances. In [7], Vinyal et al. proposed the pointer networks that enabled neural networks to select elements from an input set to form the output. It is demonstrated that this mechanism can be utilized to tackle several combinatorial problems, in particular the NP-complete travelling salesmen problem (TSP). Unlike the supervised approach taken by the original pointer network method, Bello et al. [8] used reinforcement learning to train their model. Their method relieved the demand for massive training data, and the performance is no longer limited by the quality of the labels. Besides, Deudon et al. [9] replaced the RNN encoder of the original pointer networks with the multi-head attention encoder [10]. The advantage of the new encoder is that it treats the input as an order-invariant set. These approaches give us new ideas of solving hard combinatorial problems.

In this work, we adopt this new idea to construct RSMT. We proposed Rectilinear Edge Sequence (RES) to represent an RSMT, thus we call our method REST. We also designed an actor-critic neural network model to produce RES, and trained it using reinforcement learning. The negative length of the RSMT constructed is used as reward to encourage the model to find shorter solutions. The reward can also be redesigned to cater other needs. The main contributions of this work is as follows:

- To the best of our knowledge, this work is the first successful attempt to solve RSMT construction using a machine learning based method.
- Rectilinear edge sequence (RES) is proposed to encode an RSMT solution to bridge the gap between machine learning output and RSMT structure. We show in section II-C that it has some nice properties, making it suitable for reinforcement learning.
- An actor-critic model is designed for RSMT construction. After training, it yields solution with $\leq 0.36\%$ error for nets with ≤ 50 pins in ≤ 1.9 ms on average, which is much faster than traditional heuristics of similar quality.

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK14209320).



(a) R-MST (b) RSMT

Fig. 1: Rectilinear Minimum Spanning Tree vs. RSMT

The rest of the paper is organized as follows. In section II, we give a brief introduction of the problem, and discuss the rectilinear edge sequence (RES). In section III, the architecture of our actor-critic networks is introduced in detail. We discuss the experiments and results in section IV. Lastly we draw a conclusion in section V.

II. Preliminary

A. RSMT the problem

The problem of constructing RSMT or rectilinear Steiner minimum tree can be defined as follows. Given a set of points V, construct a rectilinear tree $T(U)$ connecting all points in U of minimum L_1 length, such that U is a superset of V ($V \subseteq U$). The points newly introduced in U are called *Steiner points*. The *Steiner points* help to reuse some edges and reduce the total tree length.

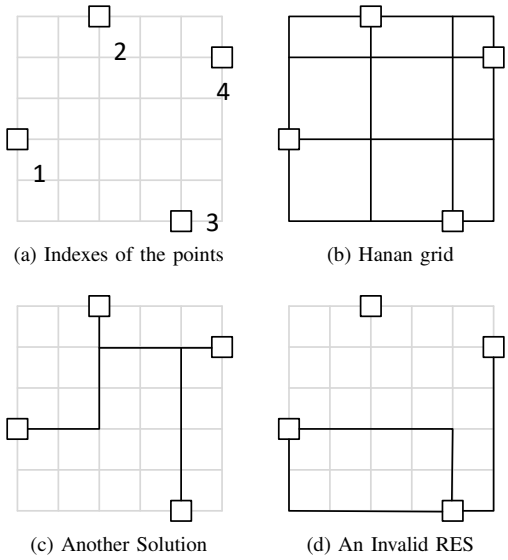
Take fig. 1 as an example. Given the 4 points in square, the shortest way to interconnect them without introducing extra *Steiner points* is by a rectilinear minimum spanning tree (R-MST). The shortest tree length in L_1 norm will be 14 as shown in fig. 1a. However, by introducing extra *Steiner points*, some edges can be reused, and a rectilinear Steiner minimum tree (RSMT) with even shorter total length can be constructed. Figure 1b illustrates such an RSMT of length 12. The two solid dots are the *Steiner points* introduced in order to construct the RSMT.

The problem of constructing RSMT for a random set of points is known to be NP-complete, but Hanan [11] has proven that an optimal RSMT can always be constructed on the Hanan grid (fig. 2b). The Hanan grid of a set of points is formed by drawing a horizontal line and a vertical line over each point in the set.

B. Sequence to RSMT

Neural networks have always being good at generating sequences, no matter it is generating a sentence or a visiting order for the travelling salesman problem (TSP). In this work, we will also use neural networks to produce sequences that can be converted to RSMT. We name this kind of sequences rectilinear edge sequence (RES).

For a given set of points $V = \{(x_1, y_1), ..., (x_n, y_n)\}$, an RES for V is a sequence of $n-1$ index pairs $((v_1, h_1), ..., (v_{n-1}, h_{n-1}))$ where $v_i, h_i \in \{1, ..., n\}$ for $i = 1, ..., n-1$. Each pair in the RES will decide a rectilinear edge that connects two points. The i^{th} pair (v_i, h_i) will connect the v_i^{th} point and the h_i^{th} point by drawing a vertical line segment over the v_i^{th} point, and a horizontal line segment over the h_i^{th} point. For example, suppose a given point set is $V = \{(0, 2), (2, 5), (4, 0), (5, 4)\}$, which is the one shown in fig. 1. The positions of the points and their indexes are shown in fig. 2a. One possible RES for V will be $res_1 = ((3, 1), (2, 1), (4, 1))$, which will give us the optimal RSMT in fig. 1b. Besides, there can be multiple optimal RES for the same set of points, for example $res_2 = ((2, 1), (2, 4), (3, 4))$ representing the RSMT in fig. 2c is also optimal. Note that the overlapping edges indicated by an RES are merged automatically, with *Steiner points* created.



(a) Indexes of the points (b) Hanan grid

(c) Another Solution (d) An Invalid RES

Fig. 2: Rectilinear Edge Sequence (RES) Explained.

It is worth mentioning that an RES is not always optimal and valid. Consider $res_3 = ((3, 1), (1, 3), (4, 3))$, it suggests the solution in fig. 2d. It is obviously not a valid solution to interconnect all points. Therefore, we enforce two requirements in theorem 1 to guarantee the validity of an RES.

Theorem 1. *Suppose* $res = ((v_1, h_1), ...(v_{n-1}, h_{n-1}))$ *is an RES for a point set* $V = \{(x_1, y_1), ..., (x_n, y_n)\}$*, if the following 2 requirements are satisfied,* res *is guaranteed to be valid, i.e. the solution defined by* res *is guaranteed to connect all points in* V *: (1)* $v_1 \neq h_1$*. (2) Exactly one of* v_i*,* h_i *is visited before, for* $i = 2, ..., n-1$*, i.e., exactly one of* v_i*,* h_i *appeared in* $\{h_1, v_1, ..., h_{i-1}, v_{i-1}\}$*.*

Proof. Let's consider the sub-tree formed by the visited points. In the first pair, we build a sub-tree with 2 distinct points connected by a rectilinear edge. In each of the following $n-2$ pairs, we will attach a new point to the sub-tree by a rectilinear edge. Thus, eventually all n points will be visited and interconnected by the RES satisfying the two requirements. □

Theorem 2. *For any point set* $V = \{(x_1, y_1), ..., (x_n, y_n)\}$*, we can always find an RES such that the tree it represents is an optimal RSMT for* V*.*

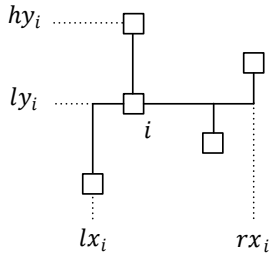


Fig. 3: The horizontal and vertical segment over point i. The left end x of its horizontal segment is lx_i, and the right end is rx_i. The lower end y of its vertical segment is ly_i, and the higher end is hy_i.

Theorem 2 ensures that an optimal RES always exists. This is actually implied by the Hanan theory [11], which states that an optimal RSMT can always be constructed on the Hanan grid (fig. 2b). Given such an RSMT, we can break it into $n-1$ rectilinear edges on the Hanan grid, each connecting two points in V. By picking an appropriate order, the edges can be written as the $n-1$ pairs of an RES.

C. Good Properties of RES

RES is not the only way to encode an RSMT solution, but there are several good properties that make RES a good choice for RSMT construction via reinforcement learning. The major advantages of RES are as follows.

1) Fixed Length Sequence: Determining the number of pairs to output is non-trivial for a neural network model. Fortunately, this will not be a problem with RES, since the length of the RES for any set of n points is always $n-1$.

2) Linear Time Evaluation: In order to learn from experiences, we need to evaluate the solutions obtained by the model on the fly to generate reward. The reward in our case is the negative of the total length. The evaluation process is often the bottleneck of reinforcement learning, as it usually requires lots of computations or even simulations. The RES can be evaluated in linear time by finding the length of the horizontal and vertical segments over each point. For each segment, we will keep track of the positions of its two ends as illustrated in fig. 3. The algorithm is summarized in algorithm 1.

Algorithm 1 RES Evaluation

Input: $V = \{(x_1, y_1), ..., (x_n, y_n)\}$,
$res = ((v_1, h_1), ..., (v_{n-1}, h_{n-1}))$
Output: $length$
1: $lx_i \leftarrow x_i$, $rx_i \leftarrow x_i$, for $i = 1, ..., n$
2: $ly_i \leftarrow y_i$, $hy_i \leftarrow y_i$, for $i = 1, ..., n$
3: **for** $i = 1$ to $n-1$ **do**
4: $(v, h) \leftarrow (v_i, h_i)$
5: $ly_v \leftarrow \min(ly_v, y_h)$, $hy_v \leftarrow \max(hy_v, y_h)$
6: $lx_h \leftarrow \min(lx_h, x_v)$, $rx_h \leftarrow \max(rx_h, x_v)$
7: $length \leftarrow \sum_{i=1}^{n}((hy_i - ly_i) + (rx_i - lx_i))$
8: **return** $length$

III. Neural Network Model

In this section we will introduce the neural network model we designed for RSMT construction. Figure 4a summarizes our model in a simplified fashion. The model we use mainly has two components - actor and critic. Both of them take a set of n point coordinates V ($V \in \mathbb{R}^{n\times 2}$) as input. The actor will take several actions to determine the elements in the RES according to a stochastic policy, i.e., each action is chosen by different probabilities produced by the model. The critic will set a baseline by predicting the expected length of the RSMT found by the current actor, so that it encourages the actor to constantly improve its performance. The general goal is to increase the probability of generating good RES(s), and decrease the probability of generating bad ones.

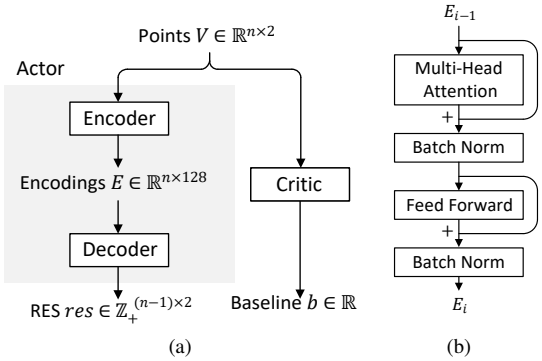


Fig. 4: (a) Simplified Structure of Our Neural Network Model; (b) The i^{th} Encoding Process.

A. Encoder

As shown in fig. 4a, our encoder will take the points as input, and produce a fixed-length encoding for each point. The dimension of each encoding is $d = 128$ in our implementation. The encodings for n points, $e_1, ..., e_n \in \mathbb{R}^d$, can be packed in an encoding matrix $E \in \mathbb{R}^{n\times d}$ ($E = [e_1, ..., e_n]^T$). The reason to encode the points before RES generation is to create better representations for the points. Before encoding, the representation of each point is just its 2D coordinates. During the encoding, the dimension is expanded to d by linear transformation. The expanded representations of different points will then exchange information for multiple rounds. The final encoding of a point will therefore contain extra information about its neighbors and the environment.

In practice, we implemented a modified multi-head attention encoder [10] to serve as our encoder. The encoder takes the 2D coordinates of the points $V \in \mathbb{R}^{n\times 2}$ (n is not fixed) as input. It first expands the dimension of the points to get the initial encodings by $E_0 = \text{BatchNorm}(VW_{emb})$, where $W_{emb} \in \mathbb{R}^{2\times d}$ is a projection matrix that can be trained. BatchNorm, a.k.a. batch normalization [12], will normalize each value in the vectors across a mini-batch. Empirically, it makes the training more stable and faster to converge.

This initial encoding is then processed by $N = 3$ identical encoding processes, each of which is illustrated in fig. 4b. The output of the i^{th} process is E_i, and $E = E_N$ will be the final output of the encoder. Each of the processes includes two sub-layers - a multi-head attention layer and a feed forward layer.

The merging arrows with + marks in fig. 4b indicate residual connections [13], i.e. point-wise addition of two inputs. Batch normalization is again used to stabilize the training.

To explain multi-head attention layer, let's first look at the single head attention,

$$\text{SingleHead}(Q, K, M) = \text{softmax}(\frac{QK^T}{\sqrt{d_s}})M$$

where $Q, K, M \in \mathbb{R}^{n \times d_s}$ ($d_s = 16$), and softmax is a function that normalize each row of its input into a probability distribution that sum up to 1. The single head attention is essentially a message passing process. We can imagine that the i^{th} row of M is the message held by the i^{th} point. Assume the result of this SingleHead is $S \in \mathbb{R}^{n \times d_s}$, the i^{th} row of S is no more than a weighted sum of M's rows. That means the information of different points are gathered to form a new representation for point i. The weight is, however, decided by Q and K.

The multi-head attention layer can then be defined as

$$\begin{aligned}\text{MultiHead}(E_i) &= \text{Concat}(S_1, ..., S_h)W_m \\ \text{where } S_j &= \text{SingleHead}(E_iW_{Q,j}, E_iW_{K,j}, E_iW_{M,j})\end{aligned}$$

where $h = 16$ is the number of heads, S_j is the output of the j^{th} single head, and the Concat function concatenates h $n \times d_s$ matrices to form one $n \times hd_s$ matrix. $W_m \in \mathbb{R}^{hd_s \times d}$ and $W_{Q,j}, W_{K,j}, W_{M,j} \in \mathbb{R}^{d \times d_s}$ are trainable parameters.

In addition, the feed forward layer applies the following function to each row of the input matrix.

$$\text{FeedForward}(x^T) = \max(0, x^TW_1 + b_1^T)W_2 + b_2^T$$

where x^T ($x \in \mathbb{R}^d$) is the input row, and $W_1 \in \mathbb{R}^{d \times d_h}$ ($d_h = 512$), $b_1 \in \mathbb{R}^{d_h}$, $W_2 \in \mathbb{R}^{d_h \times d}$ and $b_2 \in \mathbb{R}^d$ are all parameters that can be trained.

B. Decoder

The decoder takes the encodings E as inputs, and will generate an RES satisfying the two requirements in theorem 1. The decoder basically will decide the pairs in the RES recurrently, i.e., the previously generated pairs will serve as additional inputs for computing the next pair. The decoder achieves this by producing a stochastic policy, or in another word, computing the probability of generating different pairs given the current state. The state consists of the point set V and the existing partial RES.

More specifically, before generating any pairs, the decoder will first select a point as the starting point u_0 and mark it as visited. In the following $n-1$ steps, a new pair of the RES will be generated at each step. At time step t, rather than generating a pair (v_t, h_t) directly, the decoder will first select an *unvisited* point u_t, and then determine a *visited* point w_t and a boolean s_t simultaneously. If $s_t = 0$, $(v_t, h_t) = (u_t, w_t)$; if $s_t = 1$, the positions of u_t and w_t are swapped, and $(v_t, h_t) = (w_t, u_t)$. This intermediate step is introduced to guarantee that exactly one of h_t and v_t is visited (as required by theorem 1). Algorithm 2 provides the pseudo-code for our RES generation algorithm.

Algorithm 2 RES Generation

Input: $E = [e_1, ..., e_n]^T$
Output: res
1: $res = ()$
2: Select a starting point u_0 and mark as visited
3: **for** $t = 1$ to $n-1$ **do**
4: Select an *unvisited* point u_t
5: Select a *visited* point w_t and a boolean s_t
6: Mark u_t as *visited*
7: **if** $s_t = 0$ **then** $(v_t, h_t) = (u_t, w_t)$
8: **if** $s_t = 1$ **then** $(v_t, h_t) = (w_t, u_t)$
9: Append (v_t, h_t) to res
10: **return** res

In order to decide u_0, u_t, w_t and s_t for $t = 1, ..., n-1$, we make use of a pointing mechanism first proposed in [7]. For example, when we want to select a single *unvisited* point, the pointing mechanism takes the encodings $E = [e_1, ..., e_n]^T$ and a query vector $q \in \mathbb{R}^{d_q}$ ($d_q = 360$) as inputs, and output the probability $p \in \mathbb{R}^n$ of selecting each point. The query vector is used to differentiate between different states. The computation behind the pointing mechanism is as follows.

$$\begin{aligned}&p = \text{PTM}(E, q; \phi) = \text{softmax}(C \times \tanh(l)) \\ &\text{where } l = [l_1, l_2, ..., l_{n-1}, l_n]^T, \\ &l_i = \begin{cases} -\infty & \text{if point } i \text{ is } \textit{visited} \\ g_\phi^T \tanh(W_\phi^3 e_i + W_\phi^4 q) & \text{otherwise} \end{cases}\end{aligned}$$

where $\phi = \{g_\phi \in \mathbb{R}^{d_q}, W_\phi^3 \in \mathbb{R}^{d_q \times d}, W_\phi^4 \in \mathbb{R}^{d_q \times d_q}\}$ is a set of trainable parameters. The vector $l \in \mathbb{R}^n$ are the logits for selecting different points. The larger l_i is, the more likely point i will be selected. The logits for *visited* points are reset to $-\infty$ by a boolean mask to avoid being selected. The logit values are next clipped by $C \times \tanh(l)$ where $C = 10$, so that it will not become so biased towards selecting some certain points and get stuck in local minima easily. Lastly, the softmax function converts the logits into a probability distribution.

Using this mechanism, the decoder will first compute the probabilities of selecting each point as the starting point u_0 (given point set V) by

$$p(u_0|V) = PTM(E, 0; \phi_1) \tag{1}$$

which means vector zero is used as the query vector, and the parameter set ϕ_1 is used for computation.

After having the starting point, the decoder will decide (v_t, h_t) by deciding (u_t, w_t, s_t) at time step $t = 1, ..., n-1$. The decoder will first compute the probabilities of selecting each point as u_t (given the partial RES before t, or $res(< t)$) by

$$p(u_t|V, res(< t)) = PTM(E, q_t; \phi_2) \tag{2}$$

where a different parameter set ϕ_2 is used. The query vector q_t encoding the current state $res(< t)$, is computed recursively as follows,

$$\begin{aligned}edge_t &= W_u e_{u_t} + W_w e_{w_t} + W_v e_{v_t} + W_h e_{h_t} \\ subtree_t &= \max(subtree_{t-1}, W_{edge} edge_t) \\ q_t &= \max(0, edge_{t-1} + subtree_{t-1})\end{aligned} \tag{3}$$

where $W_u, W_w, W_v, W_h \in \mathbb{R}^{d_q \times d}$ and $W_{edge} \in \mathbb{R}^{d_q \times d_q}$ are trainable parameters. Vector $edge_t$ is a representation of the t^{th} generated edge (pair). Note that it is not redundant to have both (v_t, h_t) and (u_t, w_t) in the computation, as this can actually help the model to differentiate the role of each participating point. The representations of all edges until time step t are aggregated by point-wise maximum to form a representation of the existing sub-tree $subtree_t$ at time step t.

Next, a *visited* point w_t and a boolean s_t are selected simultaneously based on the existing partial RES and the u_t just selected by an extended pointing mechanism

$$p' = \mathrm{EPTM}(E, q'; \phi_3, \phi_4) = \mathrm{softmax}(C \times \tanh(l'))$$
$$\text{where } l' = [l'_{1,0}, ..., l'_{n,0}, l'_{1,1}, ..., l'_{n,1}]^T,$$
$$l'_{i,s} = \begin{cases} -\infty & \text{if point } i \text{ is } \textit{unvisited} \\ g_{\phi_3}^T \tanh(W_{\phi_3}^3 e_i + W_{\phi_3}^4 q') & \text{else if } s = 0 \\ g_{\phi_4}^T \tanh(W_{\phi_4}^3 e_i + W_{\phi_4}^4 q') & \text{else if } s = 1 \end{cases}$$

The major difference is that $EPTM$ will produce $2 \times n$ probabilities instead of n. Different parameter sets are used to compute the logits for the case when $s = 0$ and $s = 1$. We compute the probabilities of choosing each pair of w_t and s_t given V and $res(< t)$ by

$$p(w_t, s_t | V, res(< t), u_t) = EPTM(E, q'_t; \phi_3, \phi_4) \quad (4)$$

The query vector for this purpose is obtained by adding the knowledge of the u_t just selected to eq. (3):

$$q'_t = \max(0, edge_{t-1} + subtree_{t-1} + W_5 e_{u_t})$$

where $W_5 \in \mathbb{R}^{d_q \times d}$ is learned.

Note that we can use equation 1, 2 and 4 to compute the probability of generating a specific RES given a point set V and the actor parameter set θ by

$$p_\theta(res|V) = p_\theta(u_0|V) \prod_{t=1}^{n-1} p_\theta(u_t|V) p_\theta(w_t, s_t | u_t, V)$$

This will be useful when updating the parameters, as we want to increase the probability of generating good RES(s).

C. Critic and Parameters Update

The critic network is used to assist the learning process. The critic tries to set a baseline by predicting the length of the RSMT found by the actor. Assume that the actual length of the RSMT constructed by the actor is $L(V, res)$ (obtained by algorithm 1) and the critic prediction is $b(V)$, we will judge the performance of the actor for this specific point set using the advantage value computed as $a(V, res) = b(V) - L(V, res)$. This tells us how much the actor performs better than the expectation.

The critic adopts the same encoder as the actor but with different parameters, and will first encode the points into the critic encodings $E' \in \mathbf{R}^{n \times d}$. The baseline $b(V)$ will next be computed as

$$\mathrm{glimpse}(E') = \mathrm{softmax}(\tanh(E')g')^T E'$$
$$b(V) = \max(0, \mathrm{glimpse}(E')W_6 + b_6^T)W_7 + b_7$$

where $g' \in \mathbb{R}^d$, $W_6 \in \mathbb{R}^{d \times d_c}$ ($d_c = 256$), $b_6 \in \mathbb{R}^{d_c}$, $W_7 \in \mathbb{R}^{d_c \times 1}$ and $b_7 \in \mathbb{R}$ are all learnable parameters. The $\mathrm{glimpse}$ function will compute a weighted sum of the encodings according to the weights $\mathrm{softmax}(\tanh(E')g')$. It is followed by two fully connected layers to yield the baseline scalar.

So far we have introduced all the architectures of our neural network model. In the following, we will discuss our method to update the parameters of the actor θ and the parameters of the critic ψ.

For a specific point set V, the expected advantage of the RSMT generated by the actor is

$$J(\theta|V) = \sum_{r \in R} (b(V) - L(V, r)) p_\theta(r|V) \quad (5)$$

where R is the set of all valid RES. We use the REINFORCE algorithm [14] to compute the gradient of eq. (5) as

$$\begin{aligned} \nabla_\theta J(\theta|V) &= \sum_{r \in R} (b(V) - L(V, r)) \nabla_\theta p_\theta(r|V) \\ &= \sum_{r \in R} (b(V) - L(V, r)) (\nabla_\theta \log p_\theta(r|V)) p_\theta(r|V) \end{aligned} \quad (6)$$

In practice, we will draw B point sets $V_1, ..., V_B$ and sample a single RES for each set, i.e. $r_i \sim p_\theta(res|V_i)$ for $i = 1, ..., B$. We approximate the gradient in eq. (6) by Monte Carlo sampling, i.e., using a single trial for each point set to approximate the expectation, and take average to get the overall gradient,

$$\nabla_\theta J(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} (b(V_i) - L(V_i, r_i)) \nabla_\theta \log p_\theta(r_i|V_i) \quad (7)$$

We update the parameters of the actor using stochastic gradient ascent.

On the other hand, we train the critic to predict the actual length of the RSMT found by the actor, and minimize the mean square error as follows.

$$Loss(\psi) = \frac{1}{B} \sum_{i=1}^{B} ||b_\psi(V_i) - L(V_i, res_i)||_2^2$$

We use stochastic gradient descent to update the parameters of the critic.

IV. Experiment and Results

We implemented and trained REST with PyTorch and ran our experiments on a 64-bit Linux machine with Intel Xeon 2.2GHz CPUs and an Nvidia Titan V GPU. We train the model with random point sets from degree 3 to 50, and keep a set of parameters, a.k.a. checkpoint, for each degree. The training for degree $n+1$ will start after that of n, and will kick start using the parameters learned for degree n. This will speed up the training process, since the model trained with degree n nets is already a fairly good solver for degree $n+1$. For each degree, we will train the model for $40k$ iterations, and a mini-batch of B point sets are fed into the model for each iteration. The batch size varies for different degrees. We use a batch size of $B = 4096$ for degree 3, and will halve that at degree 10, 20

TABLE I: Average Percentage Errors (%)

Deg	GeoSt	R-MST	BGA	FLUTE (A = 3)	REST (T = 1)	FLUTE (A = 18)	REST (T = 8)
5	0.00	10.91	0.23	**0.00**	0.02	**0.00**	**0.00**
10	0.00	11.96	0.48	0.12	0.23	0.04	**0.01**
15	0.00	12.19	0.53	0.55	0.45	0.06	**0.03**
20	0.00	12.41	0.57	1.03	0.56	0.11	**0.07**
25	0.00	12.47	0.58	1.44	0.69	0.18	**0.12**
30	0.00	12.56	0.60	1.83	0.77	0.23	**0.16**
35	0.00	12.63	0.62	2.13	0.84	0.26	**0.21**
40	0.00	12.65	0.63	1.05	0.86	0.29	**0.25**
45	0.00	12.67	0.63	1.07	0.98	**0.30**	0.32
50	0.00	12.72	0.64	1.12	1.01	**0.29**	0.36

TABLE II: Time to Construct $10k$ RSMT for Each Degree (s)

Deg	GeoSt	R-MST	BGA	FLUTE (A=3)	REST (T=1)	FLUTE (A = 18)	REST (T = 8)
5	0.25	0.03	0.41	0.01	0.26	0.01	1.84
10	3.88	0.13	1.63	0.05	0.38	0.10	2.67
15	8.32	0.33	3.33	0.13	0.55	2.02	4.20
20	15.39	0.60	5.36	0.19	0.64	7.87	5.56
25	23.24	0.93	7.04	0.24	0.97	16.68	8.02
30	31.93	1.02	10.11	0.34	1.20	21.67	9.39
35	43.41	1.18	11.92	0.40	1.52	26.17	12.00
40	55.68	1.39	14.71	1.22	1.76	33.31	13.60
45	71.69	1.63	17.44	1.44	2.30	42.12	17.02
50	87.45	1.87	20.63	1.64	2.66	52.76	19.09

and 40 respectively. We use Adam optimizer [15] to train our model with an initial learning rate of 2.5×10^{-4}. The learning rate will decay by 0.96 after the training of each degree.

When testing, we will greedily pick the action assigned with the largest probability at each step to produce one RES for each net. We use a batch size of $100k/degree$ for each degree during testing. Besides, due to the stochastic nature of machine learning based methods, it is possible to further improve the quality by introducing variations. We found 8 transformations that can provide such variations without changing the RSMT solutions. The 8 transformations include rotating the point set by 0, 90, 180 and 270 degrees, with or without the x and y coordinates swapped. We utilize this feature by feeding T ($1 \leq T \leq 8$) transformed versions into the model, and pick the best solution.

We compare REST with GeoSteiner (optimal), an efficient implementation of R-MST [16], BGA [4], FLUTE [5] with default setting ($A = 3$) and the most accurate setting ($A = 18$) as mentioned in their work. We test the algorithms on $10k$ randomly generated point sets for each degree. We show the results for every 5 degree in table I, and their runnings in table II. REST is almost as fast as FLUTE ($A = 3$) when choosing $T = 1$, and outperforms both BGA and FLUTE ($A = 18$) w.r.t. both quality and speed by choosing $T = 8$. The process that REST builds an RSMT for a degree 10 net from scratch is illustrated in fig. 5. The solid point at time step 0 is the point chosen as the starting point. We also provide an example solution for a degree 50 net in fig. 6.

Besides, we also test our model on the benchmarks of ICCAD 2019 global routing contest [17] (ispd18_test{1-10}, ispd19_test{1-10}). We test 1.72 million nets having degree 3 to 100. We use the corresponding checkpoints for nets of degree 3 to 50, and use the degree 50 checkpoint for degree 51 to 100. It takes our algorithm ($T = 1$) 66.60s to construct all RSMTs, and the total length is just $1.008\times$ optimal length.

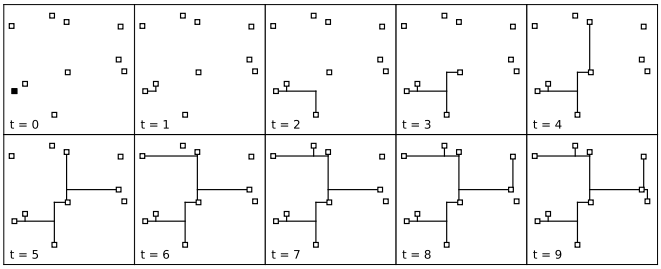


Fig. 5: Building an RSMT for a Degree 10 Net by REST

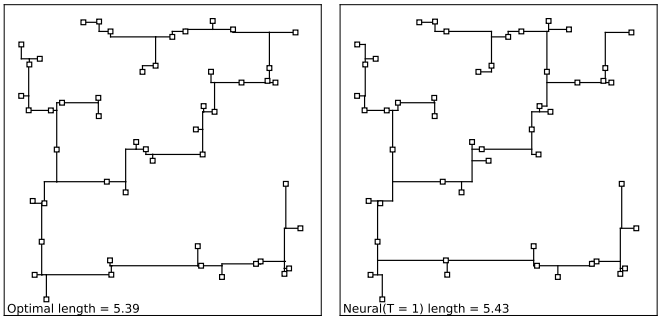


Fig. 6: Example Solution for a Degree 50 Net

V. Conclusions

In this work, we proposed REST, a machine learning based algorithm for RSMT construction. We trained it using reinforcment learning. After training, REST produces competitive RSMT solutions for small to medium size nets in terms of both quality and running time.

References

[1] M. R. Garey and D. S. Johnson, "The rectilinear steiner tree problem is np-complete," *SIAM Journal on Applied Mathematics*, 1977.

[2] F. K. Hwang, "An o (n log n) algorithm for rectilinear minimal spanning trees," *Journal of the ACM (JACM)*, 1979.

[3] F. K. Hwang, "On steiner minimal trees with rectilinear distance," *SIAM journal on Applied Mathematics*, 1976.

[4] A. B. Kahng *et al.*, "Highly scalable algorithms for rectilinear and octilinear steiner trees," in *Proc. ASPDAC*, 2003.

[5] Y.-C. Wong and C. Chu, "A scalable and accurate rectilinear steiner minimal tree algorithm," in *2008 IEEE VLSI-DAT*, 2008.

[6] D. M. Warme and J. S. Salowe, *Spanning trees in hypergraphs with applications to Steiner trees*. University of Virginia, 1998.

[7] O. Vinyals *et al.*, "Pointer networks," in *NIPS*, 2015.

[8] I. Bello *et al.*, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[9] M. Deudon *et al.*, "Learning heuristics for the tsp by policy gradient," in *CPAIOR*, Springer, 2018.

[10] A. Vaswani *et al.*, "Attention is all you need," in *NIPS*, 2017.

[11] M. Hanan, "On steiner's problem with rectilinear distance," *SIAM Journal on Applied Mathematics*, 1966.

[12] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[13] K. He *et al.*, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016.

[14] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, 1992.

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

[16] "An R-MST implementation." https://github.com/shininglion/rectilinear_spanning_graph.

[17] S. Dolgov *et al.*, "2019 CAD Contest: LEF/DEF based global routing."