NeuroSteiner: A Graph Transformer for Wirelength Estimation

Sahil Manchanda[1*], **Dana Kianfar**[2], **Markus Peschl**[2], **Romain Lepert**[2], **Michaël Defferrard**[2]

[1]Indian Institute of Technology Delhi, [2]Qualcomm AI Research
sahilm1992@gmail.com, {dkianfar,mpeschl,romain,mdeff}@qti.qualcomm.com

Abstract

A core objective of physical design is to minimize wirelength (WL) when placing chip components on a canvas. Computing the minimal wirelength (WL) of a placement requires finding rectilinear Steiner minimum trees (RSMTs), an NP-hard problem typically infeasible for modern chips comprising millions of circuit components. We propose NeuroSteiner, a neural model that distills GeoSteiner, an optimal RSMT solver, to navigate the cost–accuracy frontier of WL estimation. NeuroSteiner is trained on synthesized nets labeled by GeoSteiner, alleviating the need to train on real chip designs. Moreover, NeuroSteiner's differentiability allows to place by minimizing WL through gradient descent. On ISPD 2005 and 2019, NeuroSteiner can obtain 0.3% WL error while being 60% faster than GeoSteiner, or 0.2% and 30%.

1 Introduction

Optimizing the placement of circuit elements, e.g., standard cells, in integrated circuits is a critical step in early phases of Electronic Design Automation (EDA) (Chu 2004; Shahookar and Mazumder 1991; Kahng et al. 2011). Typically, the design pipeline starts with a set of *nets*, each of which are a logical description of the connectivity between circuit elements. Each net describes a different connectivity hypergraph and in combination they form a *netlist*, which encapsulates the overall connectivity of different elements on the chip. A placer optimizes the location of circuit elements based on power, performance, and area (PPA) objectives. One crucial objective is *wirelength* (WL), which is a common proxy for power consumption (Mirhoseini et al. 2021; Shahookar and Mazumder 1991; Caldwell et al. 1998). In principle, a WL function estimates the expected total length of wires needed to connect all nets in a routed netlist. Due to the iterative nature of (gradient-based) placement optimization, any desirable WL function needs to be fast to evaluate, accurate, and differentiable.

Among popular WL estimation methods (Shahookar and Mazumder 1991; Kahng et al. 2011), the Rectilinear Steiner Minimum Tree (RSMT) (Kahng et al. 2011) is known to be an accurate estimator of the true routed WL (Roy and Markov 2007). Specifically, solving for an RSMT consists of adding nodes—the Steiner points—to a graph such that the length of its spanning tree is minimal with respect to the L1 (Manhattan) metric. While GeoSteiner (Juhl et al. 2018), an algorithm that enumerates Steiner trees, is optimal, its runtime is exponential in the size of the problem (i.e., the net degree) since finding the optimal set of Steiner points for an arbitrary point set in a plane is NP-hard (Zhang 2016).

Several works have used efficient and heuristic approximations to the RSMT with desirable empirical performance. The Minimum Spanning Tree (MST) can be viewed as the simplest approximation because it assumes no Steiner points and incurs a runtime cost of order $\mathcal{O}(d \log d)$ for a net of degree d[1]. However, it overestimates WL by 4% on average (Wong and Chu 2008). Similarly, FLUTE (Wong and Chu 2008) uses a look-up table to quickly approximate WL without solving for any Steiner points, but is non-differentiable with respect to pin locations. Bi1S (Kahng and Robins 1992) proposes a batched-iterative approach to solving the RSMT problem. However, its performance is subpar on low-degree nets which constitute the majority of real-world netlists (Chu 2004). Another method SFP (Fallin et al. 2022) leverages CPU/GPU parallelization to accelerate the RSMT construction. Despite its high throughput, its solution quality rapidly degrades with net degree.

While these algorithms were hand-engineered to trade off accuracy for runtime, machine learning (ML) offers an automatic way to navigate this trade-off through the distillation of an optimal algorithm. Moreover, fine-tuning the neural model allows for tailoring it to the real data distribution (not the worst case). Accordingly, there has been a surge in developing ML approximations to combinatorial optimization problems on graphs (Kool, Van Hoof, and Welling 2018; Khalil et al. 2017; Bengio, Lodi, and Prouvost 2021; Gasse et al. 2019). Finally, while progress in hardware benefits all workloads, the current emphasis on ML accelerates these workloads faster: GPUs are being optimized for calculations used by neural models.

To our knowledge, REST (Liu, Chen, and Young 2021) is the only method that approached the RSMT problem with

[*]Work completed during an internship at Qualcomm AI Research.

Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

[1]The number of pins in a net is called the degree of a net.

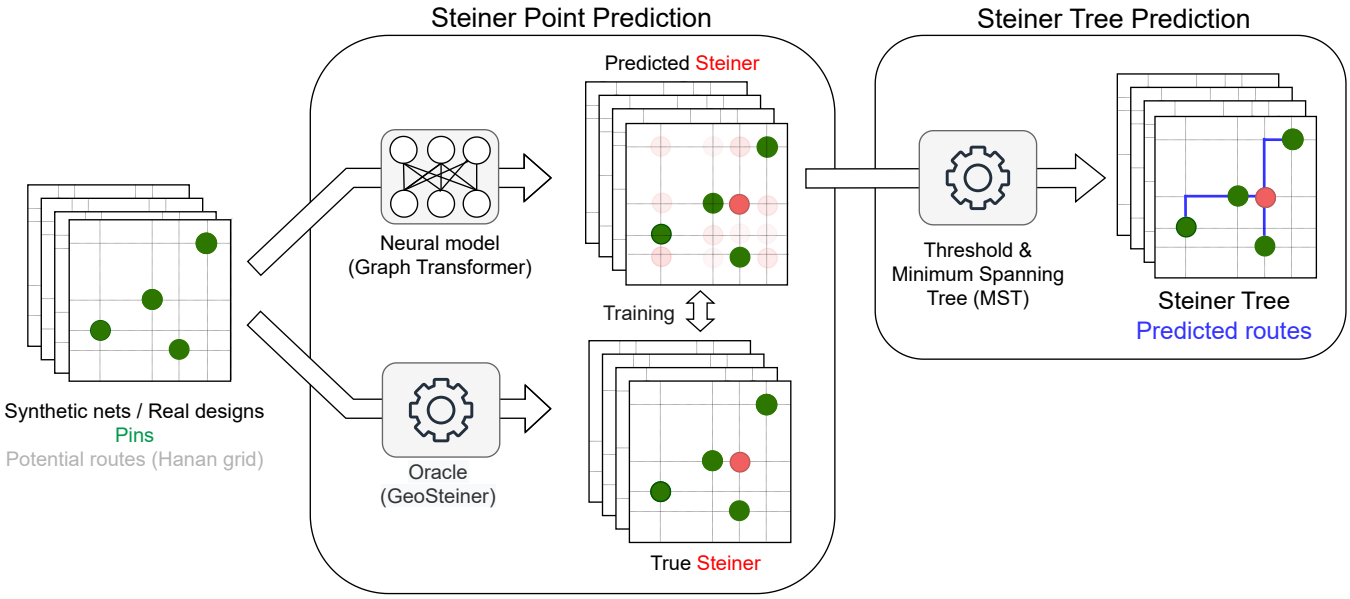


Figure 1: NeuroSteiner. Rectilinear Steiner Minimum Trees (RSMTs), hence wirelength (WL), are predicted in two steps: (1) determine Steiner points, then (2) find its Minimum Spanning Tree (MST). Our neural model predicts the probability that each node on the Hanan grid is a Steiner point. The model is trained by distilling an oracle: synthesized nets (or real designs) are labelled by GeoSteiner, an optimal RSMT solver.

ML. REST uses a reinforcement learning (RL) approach, proposing an auto-regressive model that sequentially adds edges to a tree to predict an RSMT. However, REST learns multiple models that are specialized for nets of particular degrees (number of pins in a net) instead of a single model for all input sizes. Furthermore, due to its auto-regressive nature, it generates solutions in a step-by-step sequential fashion which can increase runtime. Overall, this can lead to long runtime when predicting the wirelength of a whole netlist, consisting of a large variety of net degrees.

1.1 Contributions

To tackle the above limitations, we propose NeuroSteiner. NeuroSteiner is built on the following novel contributions:

***One-shot* supervised binary node classification task.** We formulate the prediction of Steiner points as a *supervised* binary node-classification task where each intersecting point in the Hanan grid (see Figure 1) is considered as a node in a graph. NeuroSteiner leverages a Graph Transformer (Rampášek et al. 2022) through training on labeled data from GeoSteiner (Juhl et al. 2018). The transformer mechanism enables capturing pairwise interactions between points in the Hanan Grid effectively. The Steiner point prediction is done in a *one-shot* fashion which facilitates the utilization of GPU parallelization. In contrast to REST (Liu, Chen, and Young 2021), we only predict Steiner points and not a tree. Thus, we propose a *hybrid method*, where the neural network focuses on approximating the NP-hard problem of finding Steiner point(s), while the MST calculation can be done in polynomial time. Additionally, NeuroSteiner uses a single trained model for inference and has the ability to batch nets of different sizes. This is an advantage over existing neural model REST, which relies on specialized model checkpoints for each size(degree) net and also lacks the capability to batch nets with different number of pins which hinders parallelizability.

Training on infinitely available synthetic nets. Most chip design data is proprietary, which makes learning generalizable models for chip design challenging (Kahng 2022). To tackle this, we train NeuroSteiner on a synthetically-generated labelled dataset that is cheap to obtain and infinitely large in theory. We demonstrate NeuroSteiner's capability in achieving high-quality results on real-data when it was solely trained on synthetic data. Subsequently, we illustrate that the performance of NeuroSteiner can be enhanced through fine-tuning on available *real industrial datasets* to adapt to real data distributions.

Evaluation on real-world benchmarks. Through extensive experiments on chip design benchmarks from ISPD2005 and ISPD2019, we establish that NEUROSTEINER **(1)** constructs an RSMT estimate with an error of about 0.3% when compared to the optimal solution and **(2)** predicts RSMTs for the benchmarks at 0.3 milliseconds per net on average.

2 Methodology

Formally, our goal is to solve the following problem:

Problem 1 (Rectilinear Steiner Minimum Tree) *Given a set of points $\mathcal{V}_P \in \mathbb{R}^2$, construct a rectilinear minimum spanning tree connecting a set of points $\mathcal{V} \in \mathbb{R}^2$, with $\mathcal{V} \supseteq \mathcal{V}_P$.*

In the above definition, the newly introduced points $\mathcal{V} \setminus \mathcal{V}_P$ are called Steiner points. Constructing the Rectilinear Steiner Mimimum Tree (RSMT) for a set of points is known to be NP-complete (Hanan 1966; Chu 2004). However, one can show that for any set $\mathcal{V}_P$, there always exists a set of optimal Steiner points on the Hanan grid (Hanan 1966), which is the union of nodes resulting from intersecting all horizontal and vertical lines passing through each point $v \in \mathcal{V}_P$. For example, Fig. 1 shows green nodes which depict the set of pins on the 2D plane and the lines crossing the green points represent the Hanan grid. Let $\mathcal{V}_H$ denote Hanan points, i.e., the set of intersecting points apart from the pins. We model the RSMT problem as a binary classification problem of the points in $\mathcal{V}_H$.

Problem 2 *Given a set of pins $\mathcal{V}_P \subseteq \mathbb{R}^2$ and corresponding Hanan points $\mathcal{V}_H \subseteq \mathbb{R}^2$, learn the parameters of a neural model for predicting Steiner points in $\mathcal{V}_H$.*

Fig. 1 shows the prediction and training flow of NeuroSteiner.

2.1 Hanan grid graph

Given a set of pin points $\mathcal{V}_P$, we first identify its Hanan points $\mathcal{V}_H$. To represent relationships among various points in a Hanan grid, we represent the grid using a graph. Let the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the graph consisting of the node set $\mathcal{V} = \mathcal{V}_H \bigcup \mathcal{V}_P$ and the set of edges $\mathcal{E}$ consisting of the Hanan grid. Formally, let $\mathcal{E} = \{e_{uv} = (u, v) \mid v \in \mathcal{N}(u)\}$ be defined as the set of neighboring edges, where neighbors are defined as follows:

Definition 1 (Neighborhood of a node) *In a graph $\mathcal{G}$, a node v is said to be a neighbor of node u, i.e., $v \in \mathcal{N}(u)$, if and only if one of the following conditions hold $\forall w \in \mathcal{V} \setminus \{u, v\}$ and $\forall \alpha, \beta \in [0, 1]$:*

$$v_x = u_x \;\; \textit{and} \;\; w_x \neq \alpha \cdot v_x + (1 - \alpha) \cdot u_x, \tag{1}$$
$$v_y = u_y \;\; \textit{and} \;\; w_y \neq \beta \cdot v_y + (1 - \beta) \cdot u_y, \tag{2}$$

where v_x refers to the x coordinate of node v and v_y refers to its y coordinate.

In simple terms, eq. 1 holds if two nodes u and v lie on the same horizontal line and no other node lies horizontally between them. Eq.2 can be viewed analogously for the vertical dimension.

2.2 Node and edge features

Apart from $\mathcal{G}$, we define a set of node and edge features containing task-relevant information to be used as input to our neural network. As *node features*, we make use of positional information and a node pin indicator variable. Formally, the input features of a node v are defined as

$$\mathbf{h}_v^0 = [v_x, v_y, I(v)]. \tag{3}$$

In the above equation, v_x and v_y denote the x and y coordinates of a node v, and $I(v)$ is an indicator function that returns 1 if $v \in \mathcal{V}_P$ and 0 otherwise. The coordinates v_x and v_y serve as positional information, which assists our model to recognize spatial dependencies between different nodes (Rampášek et al. 2022). Since our Graph Transformer model employs message-passing between different nodes to construct node embeddings, we employ the pin indicator $I(v)$ as a mechanism for messages to differentiate pin nodes $v \in \mathcal{V}_P$ from candidate Steiner nodes $v \in \mathcal{V}_H$. Let $\boldsymbol{X}^0 = \left[\mathbf{h}_v^0\right]_{v \in \mathcal{V}} \in \mathbb{R}^{|\mathcal{V}| \times 3}$ denote the input feature matrix of nodes belonging to the graph $\mathcal{G}$.

For the *edge features* of an edge $e_{uv} \in \mathcal{E}$, we define

$$\mathbf{e}_{uv} = [u_x - v_x, u_y - v_y] \tag{4}$$

to capture the displacement between two neighboring nodes on the Hanan grid.

2.3 Message-passing and Graph Transformers

From the input node features, we aim to generate a richer representation that encodes local and global structural information. Since classification of a node as a Steiner point is influenced by nodes beyond its local neighborhood, we need a model that can capture long-range dependencies. While, theoretically, message passing neural networks (MPNNs) (Ying et al. 2021; Rampášek et al. 2022) are able to learn global interactions, they require several rounds of message-passing at the risk of oversquashing and oversmoothing (Oono and Suzuki 2020; Morris et al. 2019; Ying et al. 2021). Instead, we opt for Graph Transformers, combining MPNNs with a global attention mechanism (Vaswani et al. 2017). This facilitates capturing interactions at both local and global levels (Rampášek et al. 2022).

More precisely, we use the GraphGPS (Rampášek et al. 2022) Graph Transformer. In GraphGPS, one can use any type of MPNN to aggregate information from the local neighborhood of a target node. Here, we use GINE (Xu et al. 2018), updating the node embeddings $\mathbf{h}_v^{(l)}$ at layer l by

$$\mathbf{h}_v^{(l+1)} = \mathrm{MLP}^{(l)}\left(\mathbf{h}_v^{(l)} + \sum_{u \in \mathcal{N}(v)} \phi\left(\mathbf{h}_v^{(l)} + \mathbf{e}_{vu}\right)\right), \tag{5}$$

where $\phi : \mathbb{R} \to \mathbb{R}$ is an activation function, in our case chosen to be the ReLU activation $\phi(z) = \max\{0, z\}$. The edge features $\mathbf{e}_{uv}$ are projected to the same dimension as $\mathbf{h}_v$ using an MLP layer. At a given layer l, the above equation aggregates features of neighboring nodes and features of their associated edges. This information is used to generate local embeddings of nodes at layer $l + 1$ represented by $\boldsymbol{X}_{\text{loc}}^{(l+1)} = \left[\mathbf{h}_v^{(l+1)}\right]_{v \in \mathcal{V}}$.

For the global attention scheme, we process the node embeddings at layer l as follows

$$\mathbf{Q}{=}\boldsymbol{X}^{(l)}\mathbf{W}_{\mathbf{Q}}, \qquad \mathbf{K}{=}\boldsymbol{X}^{(l)}\mathbf{W}_{\mathbf{K}}, \quad \mathbf{V}{=}\boldsymbol{X}^{(l)}\mathbf{W}_{\mathbf{V}}, \tag{6}$$

$$\boldsymbol{X}_{\text{glob}}^{(l+1)} = \mathrm{softmax}(\mathbf{A})\mathbf{V} \quad \text{where} \quad \mathbf{A}{=}\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}. \tag{7}$$

The input features $\boldsymbol{X}^{(l)}$ at layer l are projected by three learnable matrices $\mathbf{W}_{\mathbf{Q}}^l \in \mathbb{R}^{d \times d}$, $\mathbf{W}_{\mathbf{K}}^l \in \mathbb{R}^{d \times d}$, and $\mathbf{W}_{\mathbf{V}}^l \in \mathbb{R}^{d \times d}$ to the corresponding representations $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ and the attention matrix $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ captures the attention weight between all pairs of nodes.

From the local and global node representations at layer l, their representation at the next layer are given by

$$\boldsymbol{X}^{(l+1)} = \mathrm{MLP}\left(\boldsymbol{X}_{\text{loc}}^{(l+1)} + \boldsymbol{X}_{\text{glob}}^{(l+1)}\right). \tag{8}$$

With GraphGPS, we perform L rounds of local message-passing and global attention to obtain the final representation $\boldsymbol{X}^{(L)}$ of all nodes in $\mathcal{G}$. The more the layers, the more complex relationships GraphGPS can capture. Finally, embeddings pass through an MLP to obtain the logits

$$\hat{\mathbf{Y}} = \mathrm{MLP}_{\text{out}}\left(\mathbf{X}^{(L)}\right) \in \mathbb{R}^{|\mathcal{V}|}. \tag{9}$$

These logits represent the probability that a node is a Steiner point.

2.4 Training

The parameters of the model are learned by minimizing an objective that encourages the GraphGPS neural model to predict the true labels $\mathbf{Y}$. The binary cross-entropy objective is calculated as

$$-\frac{1}{N}\sum_{i=1}^{N} \mathbf{Y}_i \cdot \log \hat{\mathbf{Y}}_i + (1-\mathbf{Y}_i)\cdot \log(1-\hat{\mathbf{Y}}_i), \tag{10}$$

$\mathbf{Y} \in \{0,1\}^N$ are the true labels, $\hat{\mathbf{Y}} \in \mathbb{R}^N$ are the predicted logits, and N is the number of Steiner candidates in the Hanan grid. We note that this loss models each node as an independent Bernoulli random variable.

2.5 Inference

After training, for an unseen problem instance we (1) predict the probability of each node in the Hanan grid to be a Steiner point with our Graph Transformer, (2) classify Steiner points by thresholding[2], and (3) obtain the RSMT by finding the Minimum Spanning Tree (MST) from the original pins and predicted Steiner points. Wirelength is computed by the sum length of edges in the RSMT.

2.6 Train and inference pipelines

- **Training:** Train NeuroSteiner on synthetic data as outlined in Sec. 3.1.
- **Fine-tuning (optional):** When possible, fine-tuning on a dataset of chip designs can improve performance as shown in Tab. 5.
- **Inference:** Given an unseen netlist, predict the Steiner Points on the Hanan grid of each net. To obtain the RSMT per net, use the MST algorithm on the set of terminals and predicted Steiner points.

3 Experiments

In this section, we benchmark the performance of NeuroSteiner against several existing methods on real chip designs. We aim to answer the following questions:

[2]Nodes whose score $\hat{\mathbf{Y}}_i$ are greater than the threshold are classified as Steiner points.

Sec. 3.2 How does NeuroSteiner perform in terms of *wirelength (WL) estimation error* (accuracy) and *runtime* (cost)?

Sec. 3.3 What is the impact of training on synthetic or real nets?

Sec. 3.4 How to explore the cost–accuracy frontier with model capacity?

3.1 Setup

To get a sense of the cost–accuracy frontier, we evaluate NeuroSteiner at two capacities: *small* with $L = 10$ layers and *large* with $L = 20$. At both capacities, our GraphGPS model features hidden representations of size $d = 64$ and one attention head. On our setup, our large model is trained with a batch size of 12 while the small model can afford 16. We train it with the Adam optimizer (Kingma and Ba 2014) with a learning rate of 10^{-4} and an L2 decay of 10^{-5}. We use a prediction threshold of 0.3 which was decided based upon a held out synthetically generated dataset. All experiments are run on a Linux-based workstation made of an Intel Xeon W-2225 with 32 GB of memory and an NVIDIA GeForce RTX 3080 with 10 GB of memory. We implemented our neural network in PyTorch and used a C++ implementation of MST (shininglion 2015).

Synthetic nets The synthetic nets used for training are generated by sampling N points (as pins) uniformly at random in the 2D plane, where $N \sim \mathrm{Uniform}(5, 30)$. We trained our model on 50 million such nets. Labels, i.e., the optimal set of Steiner points for these samples, were obtained using GeoSteiner (Juhl et al. 2018).

Table 1: Netlists used for evaluation. This comprises of nets that have degree greater than 2 and less than or equal to 64.

dataset	netlist	#nets	degree(# pins)	
			mean	median
ISPD2019 (isp 2019)	test1	1,199	10.93	4
	test2	30,471	7.31	4
	test3	3,498	5.49	4
	test4	60,104	3.86	3
	test5	5,467	5.89	4
	test6	76,169	7.32	4
	test7	152,171	7.33	4
	test8	228,146	7.33	4
	test9	380,151	7.34	4
	test10	380,151	7.34	4
ISPD2005 (Nam et al. 2005)	adaptec1	101,003	6.75	4
	adaptec2	98,138	7.19	4
	adaptec3	177,997	7.04	4
	adaptec4	179,689	6.70	4
	bigblue1	114,750	6.83	4
	bigblue2	203,712	6.15	4
	bigblue3	292,528	7.17	4

.

Real nets We evaluate performance on real netlists from the ISPD 2005 (Nam et al. 2005) and 2019 (isp 2019) benchmarks. These netlists are made from about 1,000 to 400,000

nets. After filtering for nets with degree larger than 2 and smaller than or equal to 64, the median net degree is 4 and the mean degree is about 7. Statistics about these netlists are shown in Table 1.

Baselines. We benchmark NeuroSteiner against several methods which are neural as well as non-neural.

- **Non-neural:** We compare with an optimal RSMT solver *GeoSteiner (GST)* (Juhl et al. 2018), the plain *Minimum Spanning Tree (MST)* as the simplest approximation, and *Bi1S* (Kahng and Robins 1992) as a representative heuristic method. GeoSteiner gives a lower-bound on WL estimation—MST an upper-bound.
- **Neural:** We compare with *REST* (Liu, Chen, and Young 2021) as a neural alternative. It is a deep reinforcement learning method that comes in two variants: $T = 1$ and $T = 8$, where T is the number of performed augmentations (i.e., $T = 8$ applies rotations and flips to all nets), a hyper-parameter to be set at inference time. T trades off accuracy with cost: setting $T = 1$ results in faster but less accurate predictions.

We use the source-code released by the authors of REST in PyTorch, and standard implementations of Bi1S, GeoSteiner, and MST (shininglion 2015) in C++.

Evaluation Metric. To evaluate the quality of a method, we report the percentage wire length estimation error (%) obtained by each method against the optimal result obtained by GeoSteiner. This is a standard practice followed by existing works (Liu, Chen, and Young 2021; Kahng and Robins 1992).

3.2 Results

In this section, we report the results when NeuroSteiner was trained only on synthetic nets.

WL estimation error. Table 2 shows the WL estimation error obtained by different methods against the minimal WL (given by GeoSteiner). While NeuroSteiner doesn't reach the performance of well-engineered heuristic methods, it improves upon the faster variant of REST ($T = 1$), the only neural alternative. While REST with augmentations ($T = 8$) achieves significantly better performance than our large model, it targets a corner of the cost–accuracy frontier because this variant performs 8 independent forward passes each with an augmented version of the input and chooses the best (lowest WL) one post-inference.

When it comes to WL estimation in chip placement optimization, we expect errors within the 1% range to be sufficiently low. Hence we focused on achieving a fast model below the 1% error mark. From the estimation error $\%$ observed in Table 2, it is evident that NeuroSteiner, which is a *one-shot* model trained on *different degree nets*, achieves a high-quality wire length estimation. This can be attributed to its attention mechanism which captures pair-wise relationships between all possible points on the Hanan grid. Further, we highlight that NeuroSteiner was able to achieve high-quality performance without being exposed to any real chip designs, thereby demonstrating the strengths of training only on synthetic data which is available in abundance.

Table 2: Wirelength (WL) estimation error (%) against the optimal (GST). The reported error is the average across all nets, per netlist and overall. Lower is better. The training of NeuroSteiner for this experiment was performed on synthetic nets only.

	Non-neural		Neural			
model→ / netlist↓	MST	Bi1S	REST		NeuroSteiner	
			$T=1$	$T=8$	small	large
test1	7.298	0.067	0.423	0.108	0.715	0.503
test2	7.760	0.045	0.224	0.044	0.324	0.237
test3	7.940	0.047	0.298	0.026	0.237	0.132
test4	4.033	0.015	0.073	0.007	0.387	0.289
test5	5.913	0.028	0.247	0.027	0.777	0.557
test6	8.167	0.046	0.218	0.044	0.333	0.243
test7	7.937	0.050	0.224	0.044	0.330	0.239
test8	7.875	0.047	0.216	0.041	0.317	0.229
test9	7.942	0.048	0.223	0.044	0.328	0.237
test10	7.832	0.047	0.220	0.042	0.322	0.230
adaptec1	7.271	0.060	0.330	0.046	0.298	0.206
adaptec2	7.133	0.064	0.385	0.065	0.349	0.250
adaptec3	6.921	0.054	0.412	0.068	0.318	0.252
adaptec4	7.097	0.051	0.396	0.067	0.289	0.238
bigblue1	7.195	0.063	0.443	0.053	0.291	0.190
bigblue2	6.676	0.054	0.261	0.040	0.264	0.176
bigblue3	7.451	0.063	0.440	0.068	0.319	0.247
overall	7.430	0.052	0.294	0.050	0.317	0.232

In Table 2 we observe variations in NeuroSteiner's performance between the two evaluation datasets ISPD 2019 (test1-10) and ISPD 2015 (adaptec1-4 and bigblue1-3). The most prominent outliers are test1 and test5. We attribute the higher errors to a shift in net degree distributions. As observed in Table 1, netlists test1, test3 and test5 have considerably fewer nets than other netlists as well as different mean and median statistics. This is significant because the WL error of our models increases with net degree as shown in Table 3. We believe this is why NeuroSteiner exhibits higher error on test1 (which has *more* high-degree nets than average) than test4 (which has *fewer* high-degree nets than average). While test5 has a lower mean degree than test1, it has few nets overall and therefore the WL error is dominated by its higher-degree nets.

We also study the estimation error per net degree in Table 3. We observe that the error grows with net degree for all methods. Both NeuroSteiner and REST with $T = 8$ match Bi1S on nets of small degree, which dominate real netlists (as shown in Table 1).

Runtime. Previously, we examined the quality of different methods on wire length estimation task. In order to understand their computational efficiency, we now investigate their running time. Table 4 shows the time required by different methods to estimate WL. The duration of initial data processing is not included as it is shared amongst all methods. For NeuroSteiner, we report the total runtime: the for-

Table 3: Wirelength estimation error (%), averaged over *adaptec3* nets, grouped by degree. NeuroSteiner was trained only on synthetic data for this experiment.

model→ / degree↓	Non-neural		Neural			
	MST	Bi1S	REST		NeuroSteiner	
			$T=1$	$T=8$	small	large
3–9	6.79	0.03	0.25	0.02	0.07	0.03
10–19	7.38	0.17	0.58	0.07	0.83	0.57
20–29	8.26	0.25	1.22	0.30	1.94	1.59
30–39	8.42	0.26	1.82	0.54	2.79	2.46
40–49	7.32	0.20	3.47	1.11	3.72	4.21
50–59	7.12	0.18	4.74	1.75	4.50	5.14
60–64	7.21	0.19	5.36	2.12	4.40	5.65

Table 4: Total runtime per netlist in seconds, and the mean per net in microseconds. Lower is better.

model→ / netlist↓	Non-neural			Neural			
	GST	MST	Bi1S	REST		NeuroSteiner	
				$T=1$	$T=8$	small	large
test1	0.93	0.02	2.05	98.01	111.59	0.55	1.00
test2	15.38	0.41	14.20	112.61	133.72	9.50	16.61
test3	0.37	0.03	1.43	83.71	94.66	0.47	0.81
test4	2.10	0.75	8.14	50.91	60.45	2.73	3.84
test5	0.41	0.05	1.71	81.30	92.09	0.57	0.93
test6	81.30	1.29	72.04	115.22	144.70	21.45	38.54
test7	162.16	3.16	150.72	117.99	158.01	44.74	77.18
test8	115.93	4.43	155.37	117.09	171.28	69.91	123.97
test9	415.69	7.50	324.28	116.06	189.92	111.91	207.50
test10	412.05	6.00	329.49	116.09	190.06	110.67	196.39
adaptec1	54.73	1.80	44.15	109.32	139.14	25.35	47.30
adaptec2	75.10	1.88	97.23	111.92	143.32	40.13	72.44
adaptec3	133.63	2.46	148.02	113.30	154.88	67.31	124.73
adaptec4	133.84	2.36	137.12	114.16	157.15	67.77	122.22
bigblue1	61.60	2.03	47.87	111.55	143.39	29.07	51.81
bigblue2	99.50	3.59	135.59	114.17	155.60	48.20	88.56
bigblue3	196.52	5.42	232.36	115.69	174.56	91.84	169.87
per net	770	17	790	720	970	300	540

ward pass of the neural model followed by the MST algorithm. We note that the difference between the runtimes of REST that we report and the ones the authors report (Liu, Chen, and Young 2021) is due to potential differences in model loading and batching. Namely, since REST uses different model checkpoints for different degrees, we evaluate different degree nets on different checkpoints, which have been provided for every 5 degrees by the authors. We then report the inference runtime of REST for sequentially processing batches of same-degree nets on the corresponding checkpoint for the closest multiple of 5. This emphasizes a drawback of REST; it lacks the capability to effectively leverage batching for data points of varying sizes(degrees). On the contrary, NeuroSteiner does not face such limitation and can batch nets of different sizes.

In Table 4, we observe that in most cases NeuroSteiner is faster than REST, Bi1S and GST. We also observe a constant runtime overhead for REST, which we attribute to the sequential nature of degree-based batching and prediction. On the other hand, for NeuroSteiner, we divide nets into 5 groups of similar degrees. For each group, we use an optimized batch size, fitting as many nets as possible into a single forward pass. Further, from Table 4, we observe that MST has the lowest running time. However this result should be observed in conjunction with Table 2, which indicates that its estimation quality is significantly inferior by orders of magnitude compared to all other methods across all datasets.

3.3 Fine-tuning on real nets

Table 5: **NeuroSteiner's performance when training on synthetic data, real data, or both:** Wire length error (%) when training only on real data (*adaptec4*), training only on *synthetic* data, or training on *synthetic* nets and fine-tuning on *adaptec4* which is referred to as *both* in the table. Lower is better.

model	test netlist	training netlists		
		adaptec4	*synthetic*	*both*
NeuroSteiner (large)	adaptec1	0.351	0.194	0.159
NeuroSteiner (small)	adaptec3	0.533	0.318	0.290

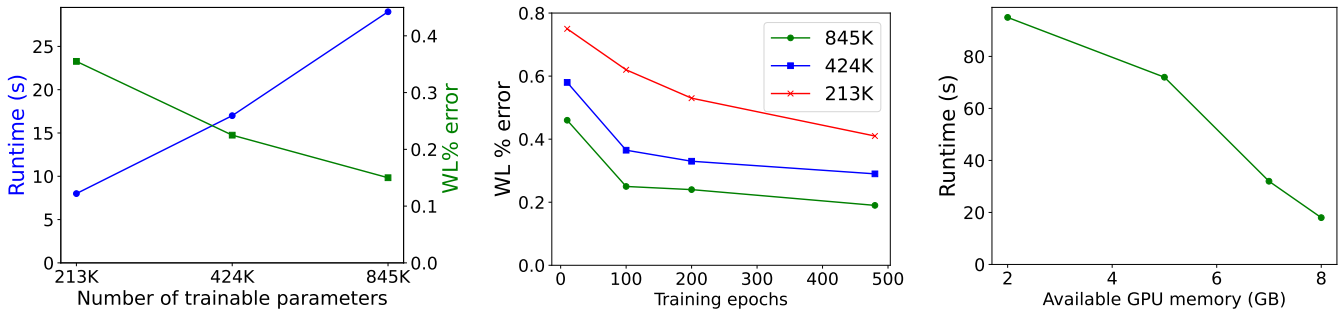
In section 3.2, we showed that by training on synthetic nets, NeuroSteiner achieves good results on real-world benchmarks without further fine-tuning on real-data, alleviating the need to train on scarce real designs. When such data is available then fine-tuning on them can improve accuracy (Yue et al. 2022) as pins on real-world nets are rarely distributed uniformly on the 2D canvas as our synthetic data generator assumes. Real-world chip designs are scarce in the public domain (Kahng 2022, 2023) even though some datasets exist (Nam et al. 2005; isp 2019).

Table 5 shows the WL estimation error for different training datasets[3]. Training on synthetic nets is better than training on a limited set of real nets. Large neural networks such as ours can easily overfit on small training sets. This emphasizes the strength of training on synthetic nets which can be generated at will (see also Figure 2b). However, training on synthetic and fine-tuning on real nets shows significant improvements on test netlists. This suggests that while synthetic data is sufficient for learning the fundamentals, it is fine-tuning that tailors to the real data distribution without overfitting. As a result, we expect training on cheap synthetic nets then fine-tuning on scarce real nets to further improve on the performance reported in Table 2.

3.4 Scaling properties

In this section, we examine how various hyper-parameters, such as model capacity, increased training data, and the

[3]These are not comparable to the errors reported in Table 2.



(a) Trading WL error (%) for runtime by controlling the model capacity. These models were trained on synthetic nets, fine-tuned on *adaptec4*, and evaluated on *adaptec1*.

(b) Training on more synthetic nets reduces WL error, at all model capacities. Evaluated on *adaptec1*(real net).

(c) Thanks to batching, more GPU memory leads to lower runtime. Runtime reported for the 424K parameter model on *adaptec1*.

Figure 2: Scaling properties.

availability of GPU memory, influence the performance of NeuroSteiner.

Cost–accuracy trade-off. Figure 2a shows the runtime and WL estimation error obtained by NeuroSteiner for different numbers of trainable parameters. As this number increase, the error decreases and the runtime increases. Since NeuroSteiner is a *one-shot* model capable of handling all net degrees, tuning the number of trainable parameters allows the user to trade-off cost and accuracy in a simple way.

More synthetic training data. In Figure 2b we observe how training on more *synthetic nets* reduces the WL error on unseen *real nets* till convergence.

Parallelism. As NeuroSteiner, unlike REST, can process nets of different degree simultaneously, increasing GPU memory allows for more nets to be processed simultaneously, irrespective of a netlist's degree distribution. Figure 2c shows that the runtime of NeuroSteiner indeed decreases linearly as GPU memory increases.

4 Conclusion

We introduced NeuroSteiner, a Graph Transformer neural model, to predict the Steiner points on a Hanan grid graph in *one shot*. Wirelength (WL) is then estimated through the construction of Rectilinear Steiner Minimum Trees (RSMTs).

Discussion Although NeuroSteiner achieves high performance at faster runtime when compared to GeoSteiner, we acknowledge that heuristics, such as SFP, for computing the RSMT can achieve competitive performance and prediction speed through efficient implementations. This is common in the ML for combinatorial optimizaiton literature (Joshi et al. 2020). However, algorithms, such as GeoSteiner, Bi1S, or SFP make discrete choices in order to generate Steiner points, impairing differentiability and often fixing a single tradeoff in terms of speed and accuracy. In this work, we aim to advance the frontier of ML-based methods for approximating the RSMT. We show that, compared to the state-of-the-art neural baseline REST, NeuroSteiner is able to achieve a competitive solution quality while requiring only a single neural model. This has potential advantages in terms of inference speed and memory requirements. Furthermore, we show that NeuroSteiner can be fine-tuned to a task-specific data distribution (Section 3.3), allowing for further performance improvements when iterating through netlist revisions.

Future Work To simplify modeling, we treated each Steiner point independently during training and inference. Since the Steiner point problem is inherently combinatorial, future work could explore joint or conditional Steiner point models to further enhance the quality of predictions. Furthermore, our proposed method first predicts the Steiner points and then computes MST to obtain wirelength. An alternative to this approach is to use a neural network to directly predict the wirelength of RSMT by treating it as a regression task. This alternative promises faster execution time and an end-to-end differentiable path from wirelength to the input points.

References

2019. *ISPD '19: Proceedings of the 2019 International Symposium on Physical Design*. Association for Computing Machinery. ISBN 9781450362535.

Bengio, Y.; Lodi, A.; and Prouvost, A. 2021. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2): 405–421.

Caldwell, A. E.; Kahng, A. B.; Mantik, S.; Markov, I. L.; and Zelikovsky, A. 1998. On wirelength estimations for row-based placement. In *Proceedings of the 1998 international symposium on Physical design*, 4–11.

Chu, C. 2004. FLUTE: Fast lookup table based technique for RSMT construction and wirelength estimation.

Fallin, A.; Kothari, A.; He, J.; Yanez, C.; Pingali, K.; et al. 2022. A Simple, Fast, and GPU-friendly Steiner-Tree Heuristic. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact combinatorial optimization with graph con-

volutional neural networks. *Advances in neural information processing systems*, 32.

Hanan, M. 1966. On Steiner's problem with rectilinear distance. *SIAM Journal on Applied mathematics*, 14(2): 255–265.

Joshi, C. K.; Cappart, Q.; Rousseau, L.-M.; and Laurent, T. 2020. Learning the travelling salesperson problem requires rethinking generalization. *arXiv preprint arXiv:2006.07054*.

Juhl, D.; Warme, D. M.; Winter, P.; and Zachariasen, M. 2018. The GeoSteiner software package for computing Steiner trees in the plane: an updated computational study. *Mathematical Programming Computation*.

Kahng, A. B. 2022. A Mixed Open-Source and Proprietary EDA Commons for Education and Prototyping. In *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 1–6.

Kahng, A. B. 2023. Machine Learning for CAD/EDA: The Road Ahead. *IEEE Design & Test*, 40(1): 8–16.

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

Kahng, A. B.; and Robins, G. 1992. A new class of iterative Steiner tree heuristics with good performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(7): 893–902.

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30.

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

Kool, W.; Van Hoof, H.; and Welling, M. 2018. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*.

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

Mirhoseini, A.; Goldie, A.; Yazgan, M.; Jiang, J. W.; Songhori, E.; Wang, S.; Johnson, E.; Pathak, O.; Nazi, A.; et al. 2021. A graph placement methodology for fast chip design. *Nature*.

Morris, C.; Ritzert, M.; Fey, M.; Hamilton, W. L.; Lenssen, J. E.; et al. 2019. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*.

Nam, G.-J.; Alpert, C. J.; Villarrubia, P.; et al. 2005. The ISPD2005 Placement Contest and Benchmark Suite. In *Proceedings of the 2005 International Symposium on Physical Design*. Association for Computing Machinery.

Oono, K.; and Suzuki, T. 2020. Graph neural networks exponentially lose expressive power for node classification. *ICLR2020*, 8.

Rampášek, L.; Galkin, M.; Dwivedi, V. P.; Luu, A. T.; Wolf, G.; and Beaini, D. 2022. Recipe for a general, powerful, scalable graph transformer. *Advances in Neural Information Processing Systems*, 35.

Roy, J. A.; and Markov, I. L. 2007. Seeing the Forest and the Trees: Steiner Wirelength Optimization in Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Shahookar, K.; and Mazumder, P. 1991. VLSI cell placement techniques. *ACM Computing Surveys (CSUR)*, 23(2): 143–220.

shininglion. 2015. An RSMT implementation. https://github.com/shininglion/rectilinear_spanning_graph.

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

Wong, Y.-C.; and Chu, C. 2008. A scalable and accurate rectilinear Steiner minimal tree algorithm. In *2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 29–34. IEEE.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*.

Ying, C.; Cai, T.; Luo, S.; Zheng, S.; Ke, G.; He, D.; Shen, Y.; and Liu, T.-Y. 2021. Do transformers really perform badly for graph representation? *Advances in Neural Information Processing Systems*.

Yue, S.; Songhori, E. M.; Jiang, J. W.; Boyd, T.; Goldie, A.; et al. 2022. Scalability and generalization of circuit training for chip floorplanning. In *Proceedings of the 2022 International Symposium on Physical Design*.

Zhang, Z. 2016. Rectilinear Steiner Tree Construction.