Neural DAG Scheduling via One-Shot Priority Sampling

Wonseok Jeon[*] WJEON@QTI.QUALCOMM.COM
Mukul Gagrani[*] MGAGRANI@QTI.QUALCOMM.COM
Burak Bartan BBARTAN@QTI.QUALCOMM.COM
Weiliang Will Zeng WZENG@QTI.QUALCOMM.COM
Harris Teague HTEAGUE@QTI.QUALCOMM.COM
Piero Zappi PZAPPI@QTI.QUALCOMM.COM
Christopher Lott CLOTT@QTI.QUALCOMM.COM
Qualcomm AI Research[†]

Abstract

We consider the problem of scheduling operations/nodes, the dependency among which is characterized by a Directed Acyclic Graph (DAG). Due to its NP-hard nature, heuristic algorithms were traditionally used to acquire reasonably good solutions, and more recent works have proposed Machine Learning (ML) heuristics that can generalize to unseen graphs and outperform the non-ML heuristics. However, it is computationally costly to generate solutions using existing ML schedulers since they adopt the episodic reinforcement learning framework that necessitates multi-round neural network processing. We propose a novel ML scheduler that uses a *one-shot* neural network encoder to sample node priorities which are converted by list scheduling to the final schedules. Since the one-shot encoder can efficiently sample the priorities in parallel, our algorithm runs significantly faster than existing ML baselines and has comparable run time with the fast traditional heuristics. We empirically show that our algorithm generates better schedules than both non-neural and neural baselines across various real-world and synthetic scheduling tasks.

1. Introduction

The problem of scheduling operations arises across many domains, such as data centers where the incoming jobs have to be scheduled on a distributed server [14], manufacturing pipelines in the form of job shop scheduling problems (JSSP) [13], and ML compilers where the operations of a computation graph need to be scheduled on the available hardware devices [15, 26]. In all these cases, the problem may be abstracted using a directed acyclic graph (DAG) where the nodes of the graph represent the operations and the edges represent the dependency constraints between the operations and hence the problem is also referred to as DAG scheduling. The objective is to minimize the finish time (or makespan) of the DAG subject to resource and dependency constraints.

It is well known that this is an NP-hard problem [9], and practitioners have traditionally relied on heuristic methods to obtain good solutions. One of the celebrated scheduling approaches is *list scheduling* [5] where the idea is to schedule nodes as early as possible and to break ties using priorities. The priorities can be obtained via different node metrics which are computationally inexpensive such as critical-path based, shortest processing time or most operations remaining [7]. More recently,

[*]Equal contribution
[†]Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

researchers have proposed deep reinforcement learning based methods to solve scheduling problems [14, 22, 25, 26]. The scheduling policy in all the references utilize Graph Neural Networks (GNN) as an encoder to derive node embeddings. [25] proposed an auto-regressive GNN based policy for the JSSP problem which predicts the next node for scheduling given the nodes scheduled so far. [22] proposed a bi-level optimization approach which modifies the input DAG by adding multiple edges via a learned policy and then apply the critical-path heuristic on the modified DAG. One major drawback of the existing ML based schedulers is the computational cost as they require multi-round neural network processing (encoding step). The multi-round neural network processing is reflected as auto-regressive architecture [25] or bi-level optimization design [22]. This drawback limits the scalability to large graphs and the applicability to domains where solutions need to be obtained in a timely manner (e.g., scheduling computation graphs in compilers).

In this paper, we propose a novel ML scheduler that uses a *one-shot* neural network encoder to sample node priorities which are converted by list scheduling to the final schedules. Since our encoder generates node priorities with a single forward pass of a neural network and efficiently samples priorities in parallel, our algorithm runs significantly faster than existing ML baselines and has comparable run time with the fast traditional heuristics. We perform experiments on a variety of scheduling tasks which includes TPC-H benchmark and scheduling for synthetic and real-world computation graphs. We show that our method outperforms (w.r.t to the makespan) both the non-neural baselines and the neural baseline.

2. Preliminaries

2.1. Scheduling Problem

In scheduling problems, we define a DAG as a tuple $G := (\mathcal{V}, \mathcal{E}, \delta, \rho, \mu)$ with a set $\mathcal{V}$ of nodes (or vertices) and a set $\mathcal{E}$ of directed edges (or arcs). Each node $v \in \mathcal{V}$ represents an operation with $\delta(v) \geq 0$ denoting its operational duration and $\rho(v) \geq 0$ denoting the resources required to execute v. For a set $\mathcal{M}$ of machine types, each node $v \in \mathcal{V}$ has to be assigned to its own machine type $\mu(v) \in \mathcal{M}$ ($|\mathcal{M}| = 1$ and $|\mathcal{M}| > 1$ correspond to scheduling with homogeneous machines and heterogeneous ones, respectively). The set $\mathcal{E}$ of edges in the DAG G represents computational dependency among nodes. For instance, for the scheduled start time $\tau(v) \geq 0, v \in \mathcal{V}$ for each node, a directed edge $(v_1, v_2) \in \mathcal{E}, v_1, v_2 \in \mathcal{V}$, means $\tau(v_1) + \delta(v_1) \leq \tau(v_2)$, i.e., any node should be scheduled on or after all its predecessor nodes are finished. We assume that each type of machine $m \in \mathcal{M}$ has its own maximum resource limit $\lambda(m) \geq 0$, i.e., at any point of time the total amount of occupied resources for machines of type m cannot exceed $\lambda(m)$.

Let us introduce the vectorized notation $\tau = [\tau(v)]_{v \in \mathcal{V}} \in \mathbb{R}_{\geq 0}^{|\mathcal{V}|}$ of the start times with a little abuse of notation for the sake of simpler notation. We define a valid *schedule* as a vector $\tau \in \mathcal{T}$ where $\mathcal{T}$ is the set of all valid schedules (satisfying both precedence and resource constraints for given DAG G). The objective of the scheduling problem is to find $\tau^* := \arg\min_{\tau \in \mathcal{T}} C(\tau; G)$, where $C(\tau; G) := \max_{v \in \mathcal{V}} \{\tau(v) + \delta(v)\}$, the duration required to complete all operations, is the *makespan* of schedule τ.

2.2. List Scheduling

List scheduling [5] is a class of priority-based schedule algorithms that are widely adopted in practice due to their simplicity. We describe how list scheduling works as follows (See Figure in Appendix A);

(Step 1) Input a list of node priorities and set the current decision time to be zero; **(Step 2)** Find *ready nodes* that can be scheduled at the current decision time, i.e., nodes whose predecessors have finished; **(Step 3)** Schedule the ready nodes sequentially at the current decision time by following the order of node priority until either all ready nodes are scheduled or further nodes cannot be scheduled due to resource constraints; **(Step 4)** Move the decision time to the earliest finish time over all scheduled nodes which have not finished at the current decision time and repeat **(Step 2)** to **(Step 4)** until all nodes are scheduled.

2.3. The Gumbel-Top-k Trick

Consider a random variable Y over a set $\mathcal{Y}$ of finite categories, where the distribution is defined by the softmax over $\mathrm{logits}(y) \in \mathbb{R}, y \in \mathcal{Y}$ (the unnormalized log-probabilities), i.e., $\Pr\{Y = y\} \propto \exp(\mathrm{logits}(y)), y \in \mathcal{Y}$, and thus

$$\Pr\{Y = y\} = \frac{\exp(\mathrm{logits}(y))}{\sum_{y' \in \mathcal{Y}} \exp(\mathrm{logits}(y'))}. \tag{1}$$

The *Gumble-Max trick* [6] is a method to sample from the categorical distributions when logits characterizing the distributions are tractable. Specifically, the trick shows that by using a random vector $Z \in \mathbb{R}^{|\mathcal{Y}|}$ where elements $Z(y), y \in \mathcal{Y}$, are sampled from i.i.d. standard Gumbel distribution, one can randomly generate a category as follows:

$$\arg\max_{y \in \mathcal{Y}} \{\mathrm{logits}(y) + Z(y)\} \sim \Pr\{Y = y\}. \tag{2}$$

More recent works [12, 21] found that the Gumbel-Max trick can be extended to *sample k categories without replacement*, which is called the *Gumbel-Top-k trick*. For the extension, they introduced $\arg\mathrm{top}^{(k)}$ which takes a real vector on $\mathcal{Y}$ and outputs a sequence of elements in $\mathcal{Y}$ that correspond to the k largest values; the output sequence of the elements should be ordered by the corresponding decreasing input values [12]. As a special case where $k = |\mathcal{Y}|$, $\arg\mathrm{top}^{(k)}$ becomes $\arg\mathrm{sort}$ in decreasing values. The Gumbel-Top-k trick generates the random sequence of elements in Y

$$[Y_1, Y_2, ..., Y_k] := \arg\mathrm{top}_{y \in \mathcal{Y}}^{(k)} \{\mathrm{logits}(y) + Z(y)\}, \tag{3}$$

and shows that the sequence is equivalent to the one from sampling k elements without replacement; note that the random vector Z is sampled *once* and *before* applying $\arg\mathrm{top}^{(k)}$. In other words, the distribution of the random sequence in Eq. (3) is shown to be described as follows [12]:

$$\Pr\{[Y_1, Y_2, ..., Y_k] = [y_1, y_2, ..., y_k]\} = \prod_{i=1}^{k} \frac{\exp(\mathrm{logits}(y_i))}{\sum_{y' \in \mathcal{Y} \setminus \{y_1, ..., y_{i-1}\}} \exp(\mathrm{logits}(y'))}. \tag{4}$$

Intuitively, Eq. (4) tells us that each element in the random sequence in Eq. (3) follows the categorical distribution that is characterized by the softmax over logits, where previously sampled categories are excluded. In this work, we use Eq. (3) to decide the priorities over the elements in $\mathcal{Y}$ when $k = |\mathcal{Y}|$, which will be elaborated in the next section.

3. DAG Scheduling with Neural Priority Sampler

Suppose we have a set $\mathcal{G} := \{G_1, G_2, ...\}$ of DAGs, where each DAG $G_i := (\mathcal{V}_i, \mathcal{E}_i, \delta_i, \rho_i, \mu_i)$ follows the definition in Section 2.1. We also assume that we have a device which is equipped with machines required for the DAGs. The learning-to-schedule algorithms by [25] and [22] aim to find out a parameterized schedule generator $\pi_\theta(\tau|G)$ (with the neural network parameter θ) that minimizes the average makespan over $\mathcal{G}$, i.e.,

$$\arg\min_\theta \mathbb{E}_{G\sim\mathcal{G}}\mathbb{E}_{\tau\sim\pi_\theta(\cdot|G)}\left[C(\tau;G)\right], \tag{5}$$

where τ is the schedule (the start times of nodes) and $C(\tau;G)$ is the makespan of τ in G as in Section 2.1.

3.1. Schedule Generator with One-Shot Priority Sampler

Using one-shot node priorities was recently proposed by [4] to solve peak memory minimization problems in DAGs. We adopt Topoformer, the graph neural network (GNN) encoder presented by Gagrani et al., as our graph encoder $\text{GNN}_{\theta_1}(G) \in (\mathbb{R}^h)^{|\mathcal{V}|}$, where h is the dimension of output embeddings for each node, and θ_1 is the neural network parameter of the encoder. We use $\text{MLP}_{\theta_2} : \mathbb{R}^h \to \mathbb{R}$ to convert the GNN's output node embeddings into *logits* over the nodes, i.e., for $\theta := (\theta_1, \theta_2)$ and $G \in \mathcal{G} = \{G_1, G_2, ...\}$,

$$\text{logits}_\theta(v;G) := \text{MLP}_{\theta_2}([\text{GNN}_{\theta_1}(G)]_v) \in \mathbb{R}, v \in \mathcal{V}. \tag{6}$$

The difference between Gagrani et al.'s algorithm and ours arises from sampling procedure using logits. Gagrani et al. considers logits of schedulable nodes at each decoding step and sequentially samples among those schedulable nodes, whereas we sample node priorities *only once at the start of decoding*. Specifically, by using i.i.d. standard Gumbel variables $Z(v) \in \mathbb{R}, v \in \mathcal{V}$, and $\arg\text{sort}$, we randomly sample a sequence of nodes from *perturbed logits*:

$$\boldsymbol{V} := [V_1, V_2, ..., V_{|\mathcal{V}|}] := \underset{v\in\mathcal{V}}{\arg\text{sort}}\{\text{logits}_\theta(v;G) + Z(v)\}. \tag{7}$$

Note that the LHS of Eq. (7) is a random sequence due to the randomness of Z and $\arg\text{sort}$ is applied in decreasing values of the perturbed logits. We then regard the sampled random sequence $\boldsymbol{V} = [V_1, V_2, ..., V_{|\mathcal{V}|}]$ as the sequence of node priorities $V_1 \succ V_2 \succ ... \succ V_{|\mathcal{V}|}$, which does not require additional computation. Due to the stochastic nature of the Gumbel-Top-k Trick described in Section 2.3, our mapping from the random sequence to the node priorities is equivalent to sampling nodes without replacement where *nodes sampled earlier are considered to be higher-priority ones*. More importantly, the trick allows us to use the tractable distribution of the random sequence, which becomes highly beneficial when optimizing the neural network. We will discuss this further in the next subsection.

Finally, we use list scheduling (described in Section 2.2) with the sampled random sequence in Eq. (7) to generate a valid schedule with the start time τ:

$$\tau = \text{ListScheduling}(\boldsymbol{V};G) \in \mathbb{R}^{|\mathcal{V}|}_{\geq 0}. \tag{8}$$

Table 1: Experimental results on TPC-H datasets are described. We use bold letters to emphasize the minumim average makespan.

	TPC-H-50		TPC-H-100		TPC-H-150	
	Makespan	Time (sec)	Makespan	Time (sec)	Makespan	Time (sec)
Const. Prog.	**8629.4**	-	19278.3	-	-	-
CP	9821.3	0.008	16914.1	0.027	24429.5	0.048
SPT	12818.4	0.002	19502.7	0.008	27409.4	0.021
MOPNR	11360.1	0.011	17733.1	0.032	24871.2	0.064
PPO-BiHyb	8905.4	66.484	15192.2	149.215	22371.2	571.424
Greedy (ours)	8845.6	0.057	14981.2	0.100	22332.7	0.259
S(16) (ours)	8782.4	0.114	14972.0	0.287	22330.2	0.674
S(64) (ours)	8742.5	0.216	14968.1	0.699	22323.0	1.856
S(256) (ours)	8694.4	0.540	**14964.7**	2.270	**22320.8**	6.485

3.2. Algorithm and Practical Consideration

The objective of learning-to-schedule frameworks in Eq. (5) can be rewritten with our scheduler generator as follows:

$$\arg\min_{\theta} \mathbb{E}_{G\sim\mathcal{G}}\mathbb{E}_{\boldsymbol{V}\sim\pi_\theta(\cdot|G)}C_{\text{LS}}(\boldsymbol{V};G), \tag{9}$$

where $C_{\text{LS}}(\boldsymbol{V};G) := C(\text{ListScheduling}(\boldsymbol{V};G);G)$ is the makespan of list scheduling for given node priorities $\boldsymbol{V} = [V_1, ..., V_{|\mathcal{V}|}]$ and the graph G, and $\pi_\theta(\cdot|G)$ is the probability distribution of sampling node priorities for G. From the Gumbel-Top-k trick discussed in Section 2.3, we can get the tractable form of $\pi_\theta(\cdot|G)$ below:

$$\pi_\theta(V|G) := \Pr_\theta\{\boldsymbol{V} = [v_1, ..., v_{|\mathcal{V}|}]|G\} = \prod_{i=1}^{|\mathcal{V}|} \frac{\exp(\text{logits}_\theta(v_i;G))}{\sum_{v\in\mathcal{V}\setminus\{v_1,...,v_{i-1}\}}\exp(\text{logits}_\theta(v;G))}. \tag{10}$$

This enables us to use the following gradient descent rule using REINFORCE [23] that optimizes the objective Eq. (9) with learning rate $\alpha > 0$:

$$\theta \leftarrow \theta - \alpha\mathbb{E}_{G\sim\mathcal{G}}\mathbb{E}_{\boldsymbol{V}\sim\pi_\theta(\cdot|G)}[\nabla_\theta \log \pi_\theta(\boldsymbol{V}|G)C_{\text{LS}}(\boldsymbol{V};G)]. \tag{11}$$

Together with the above update rule, we found that the practical techniques including **Logit Norm Regularization** and **Cost Standardization** we introduce are crucial to stabilize the training and lead to significantly better performance. We describe details on those techniques together with an illustrative figure and our final algorithm in Appendix.

4. Experiments

We evaluate our neural DAG scheduler and baselines for three different scheduling tasks: DAG scheduling on **TPC-H dataset**, and scheduling on **computation graphs** (See Appendix E for details about all tasks.). We compare our algorithm with both non-neural and neural baselines. For non-neural baselines, we consider list scheduling algorithms with different node priorities (based on Critical-Path **(CP)**, Most Operations Remaining **(MOPNR)** and Shortest Processing Time **(SPT)**) [25], and a constraint programming **(Const. Prog.)** solver (CP-SAT by [17]) with 24 hours time limit. For neural baselines, we consider **PPO-BiHyb** [22], where we use the deployed model of PPO-BiHyb for TPC-H dataset and train the model for the dataset with real-world computation graphs.

For our model, we consider two types of operating modes. In **Greedy** mode, our model generates schedules with node priorities that uses $\arg\text{sort}$ and the pure logits without adding Gumbel random variables. In **Sampling** mode, we sample multiple node priorities (where the number of samples are chosen from 16, 64 and 256). We report the makespan for output schedules and the run time of each algorithm. We report the speedup metric, which is defined as the ratio of sum duration of all nodes and the makespan, in the computation graph scheduling task.

4.1. TPC-H Dataset

TPC-H dataset[1] includes DAGs that consists of industrial queries and data modifications which represent computation jobs and need to be scheduled on a homogenous machine with finite resources. Table 1 shows the performance of baselines and our model on the test set. We observe that our method obtains better average makespan than all the baselines including PPO-BiHyb on all the three instances except in TPC-50 where constraint programming achieves slightly better makespan. One more thing to note is that our method has much smaller run time compared to PPO-BiHyb. This is because our method generates node priorities via a single pass of our GNN encoder and samples priorities effectiviely by using Gumbel Top-k trick, whereas PPO-BiHyb has to run their GNN encoder multiple times and requires beam search.

4.2. Computation Graphs

We test our approach on scheduling tasks for both synthetic and real-world computation graphs of neural networks that arise in ML compilers. We consider three type of synthetic computation graphs: layered graphs [4], Erdos-Renyi and stochastic block model graphs [15] (See appendix F.3.). We show that our model significantly outperforms our baselines and describe the results in Appendix due to the space limit. We also experiment on a set of proprietary real-world computation graphs to evaluate the practical applicability of our neural scheduler. This dataset consists of computation graphs of diverse neural net architectures like classifiers, convolution nets, denoisers, etc. We observe that our method achieves superior speedup on graphs of all sizes compared to all non-neural baselines. We also compare our approach with PPO-BiHyb and constraint programming baseline in this setting. Our results show that we outperform both PPO-BiHyb and constraint programming in terms of the achieved speedup. Note that we generate better schedules and run much faster than PPO-BiHyb. The fact that we beat constraint programming shows the large search space of this problem and the effectiveness of our approach to learn to find good schedules in this large space.

[1]http://tpc.org/tpch/default5.asp

Table 2: Experimental results on real-world computation graphs are described. We use bold letters to emphasize the maximum average speedup for each test set of graphs.

	200 - 500 Node Graphs		500 - 700 Node Graphs		700 - 1000 Node Graphs	
	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)
Const. Prog.	3.267	-	3.183	-	2.497	-
CP	3.174	0.007	2.804	0.016	2.739	0.025
SPT	3.107	0.002	2.868	0.005	2.664	0.008
MOPNR	3.181	0.009	2.825	0.020	2.739	0.028
PPO-BiHyb	3.223	17.937	2.965	52.777	2.798	322.793
Greedy (ours)	3.245	0.152	3.131	0.098	2.846	0.060
S(16) (ours)	3.271	0.192	3.188	0.245	2.848	0.230
S(64) (ours)	3.278	0.263	3.199	0.456	2.856	0.606
S(256) (ours)	**3.286**	0.595	**3.207**	1.309	**2.860**	2.001

References

[1] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. *Proceedings of Machine Learning and Systems*, 2:44–57, 2020.

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

[3] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *arXiv preprint arXiv:2012.09699*, 2020.

[4] Mukul Gagrani, Corrado Rainone, Yang Yang, Harris Teague, Wonseok Jeon, Herke Van Hoof, Weiliang Will Zeng, Piero Zappi, Christopher Lott, and Roberto Bondesan. Neural topological ordering for computation graphs. *arXiv preprint arXiv:2207.05899*, 2022.

[5] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[6] Emil Julius Gumbel. *Statistical theory of extreme values and some practical applications: a series of lectures*, volume 33. 1954.

[7] Reinhard Haupt. A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16, 1989.

[8] Chaitanya K Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent. Learning TSP requires rethinking generalization. In *International Conference on Principles and Practice of Constraint Programming*, 2021.

[9] AHG Rinnooy Kan. *Machine scheduling problems: classification, complexity and computations*. Springer Science & Business Media, 2012.

[10] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

[11] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations (ICLR)*, 2019.

[12] Wouter Kool, Herke Van Hoof, and Max Welling. Stochastic beams and where to find them: the Gumbel-Top-k trick for sampling sequences without replacement. In *International Conference on Machine Learning (ICML)*, pages 3499–3508, 2019.

[13] Alan S Manne. On the job-shop scheduling problem. *Operations research*, 8(2):219–223, 1960.

[14] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*, pages 270–288. 2019.

[15] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. In *International Conference on Learning Representations (ICLR)*, 2020.

[16] Junyoung Park, Jaehyeong Chun, Sang Hun Kim, Youngkook Kim, and Jinkyoo Park. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59(11):3360–3377, 2021.

[17] Laurent Perron and Vincent Furnon. OR-tools. URL `https://developers.google.com/optimization/`.

[18] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[20] Penghao Sun, Zehua Guo, Junchao Wang, Junfei Li, Julong Lan, and Yuxiang Hu. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 3314–3320, 2021.

[21] Tim Vieira. Gumbel-Max trick and weighted reservoir sampling. 2014. URL `https://timvieira.github.io/blog/post/2014/08/01/gumbel-max-trick-andweighted-reservoir-sampling/`.

[22] Runzhong Wang, Zhigang Hua, Gan Liu, Jiayi Zhang, Junchi Yan, Feng Qi, Shuang Yang, Jun Zhou, and Xiaokang Yang. A bi-level framework for learning to solve combinatorial optimization on graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 21453–21466, 2021.

[23] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[24] Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. NeuroLKH: Combining deep learning model with Lin-Kernighan-Helsgaun heuristic for solving the traveling salesman problem. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[25] Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. Learning to dispatch for job shop scheduling via deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1621–1632, 2020.

[26] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Phothilimtha, Shen Wang, Anna Goldie, et al. Transferable graph optimizers for ML compilers. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 13844–13855, 2020.

[27] Yunfan Zhou, Xijun Li, Jinhong Luo, Mingxuan Yuan, Jia Zeng, and Jianguo Yao. Learning to optimize dag scheduling in heterogeneous environment. In *2022 23rd IEEE International Conference on Mobile Data Management (MDM)*, pages 137–146. IEEE, 2022.

Appendix A. Illustrations for List Scheduling and Our Neural DAG Scheduler

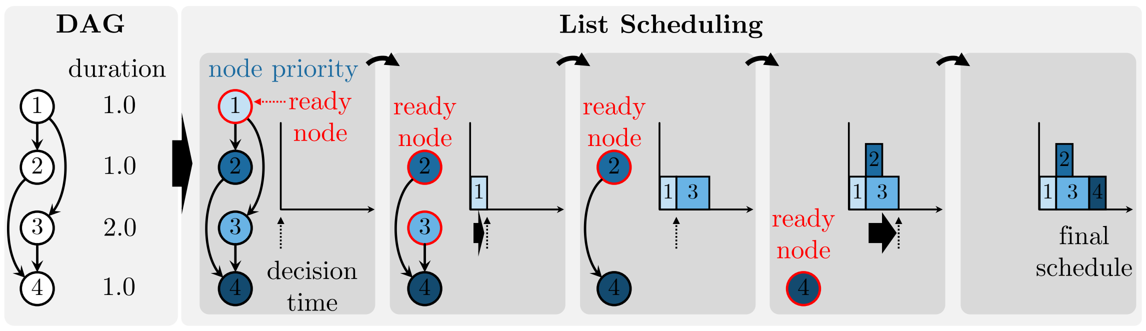

Figure 1: An example of list scheduling [5] for a 4-node DAG scheduling is described. Each node has its own duration, while resource limit is ignored for illustration purpose. List scheduling takes node priority as an input (e.g., $1 \succ 3 \succ 2 \succ 4$, brighter color implies higher priority) and schedules the higher-priority node among ready nodes earlier than the other nodes. After all ready nodes are scheduled, we move the decision time until a new set of ready nodes is found. We repeat these steps until we schedule all nodes.

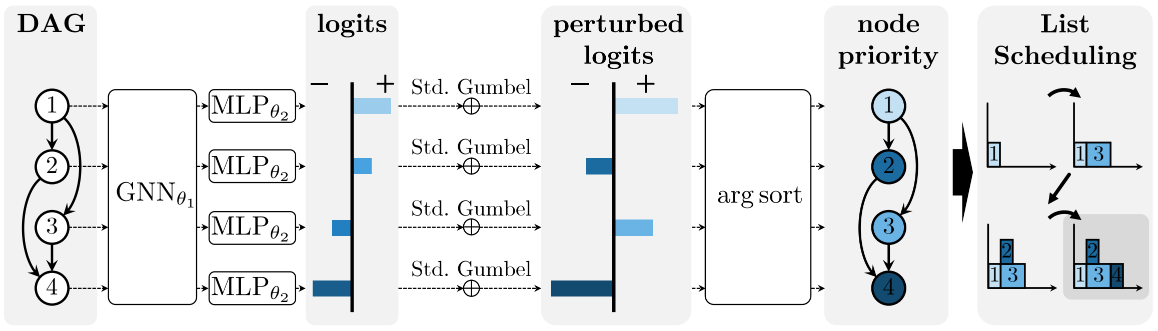

Figure 2: Our neural scheduler for DAGs works as follows: **(Step 1)** Generate logits for all nodes from a DAG by using a GNN encoder followed by an MLP. **(Step 2)** Perturb logits by adding i.i.d. Gumbel random variables. **(Step 3)** Take argsort over perturbed logits to define node priorities. Higher priority nodes have brighter colors. **(Step 4)** Use list scheduling to generate a schedule.

Appendix B. Algorithmic details

Logit Norm Regularization. When [4] used their GNN encoder (Topoformer) to define the model distribution, the standardization over logits was used to bound the range of logits (See Section C.2 in [4]). Specifically for the mean m and standard deviation s over $\text{logits}(v), v \in \mathcal{V}$, and a

Algorithm 1: Neural DAG Scheduler via One-Shot Priority Sampling

```
Input: A set $\mathcal{G} = \{G_1, G_2, ...\}$ of the training graphs, a node priority sampler $\pi_\theta$, learning rate
       $\alpha > 0$, a regularization coefficient $c_{\text{logits}} > 0$.
for each epoch do
    for each $G \in \mathcal{G}$ do
        Sample a batch of node priorities $\boldsymbol{V}^{(1)}, \boldsymbol{V}^{(2)}, ..., \boldsymbol{V}^{(N)} \sim \pi_\theta(\cdot|G)$.
        Convert the priorities to valid schedules by list scheduling.
        Evaluate makespan for all sampled schedules.
        Standardize makespan $\bar{C}_n \leftarrow \overline{C}(\boldsymbol{V}^{(n)}; \boldsymbol{V}^{(1)}, ..., \boldsymbol{V}^{(N)}, G), n = 1, ..., N$, by Eq. (13).
        Compute REINFORCE gradient $g_{\text{REINFORCE}} \leftarrow \frac{1}{N}\sum_{n=1}^{N} \nabla_\theta \log \pi_\theta(\boldsymbol{V}^{(n)}|G)\bar{C}_n$.
        Compute logit norm gradient $g_{\text{logits}} \leftarrow c_{\text{logits}} \times \nabla_\theta L_{\text{logits}}(\theta; G)$ with Eq. (12).
        Update $\theta$ by using gradient descent: $\theta \leftarrow \theta - \alpha(g_{\text{REINFORCE}} + g_{\text{logits}})$.
    end
end
Output: $\pi_\theta$
```

scalar hyperparameter $c > 0$, the standardized logits $\overline{\text{logits}}(v) := c \times (\text{logits}(n) - m)/s$ is used to define the model's probability of topological ordering. However, we empirically observe that such standardization over logits leads to poor performance. We believe this is because standardization significantly restricts the model's representation capability. While a rigorous proof for our argument may be complicated, we can easily show that this is true for binary random variables. Specifically with the above definition of standardized logits and a random variable $X \in \{0, 1\}$, one can only represent $(\Pr\{X = 0\}, \Pr\{X = 1\}) = (\frac{1}{1+\exp(2c)}, \frac{\exp(2c)}{1+\exp(2c)})$ or $(\frac{\exp(2c)}{1+\exp(2c)}, \frac{1}{1+\exp(2c)})$ (See Appendix C.). Note that the constant c is a hyperparameter and assumed to be fixed during training, and the example implies that only two distributions can be described for any given c, which supports our claim.

After removing the standardization of logits due to the above observation, however, we still observe instability in training caused by the unbounded logits. Therefore, we introduce the *logit norm regularizer* that minimizes

$$L_{\text{logits}}(\theta; G) := \frac{1}{|\mathcal{V}|}\sum_{v \in \mathcal{V}} \text{logits}_\theta(v; G)^2 \tag{12}$$

by gradient descent, together with the aforementioned REINFORCE objective. Intuitively, the norm regularizer allows our model to have the logits around the origin so that we can avoid numerical errors due to the unbounded logits while the sufficient amount of flexibility is still maintained for choosing logits. We empirically found that this highly stabilizes training and improves the performance and hence we apply the regularizer in all of our experiments.

Cost Standardization. [4] stored the best-performing model so far and used its cost as the policy gradient baseline. This so-called greedy baseline was introduced due to the empirical performance of [11]'s algorithm on routing problems. However, if the scale of makespan varies significantly across different training graphs, the model trained with the greedy baseline can easily overfit a small subset of training graphs, which may lead to poor performance on test graphs. Also, the greedy baseline requires additional resources since the intermediate models should be stored and evaluated during training. To address these issues, we use cost standardization that has been widely adopted in policy-gradient algorithms, e.g., PPO [18, 19]. Specifically for a given graph G during a training iteration, we first sample multiple node priorities, i.e., $\boldsymbol{V}^{(1)}, ..., \boldsymbol{V}^{(N)} \sim \pi_\theta(\cdot|G)$, and evaluate makespan $C_{\text{LS}}(\boldsymbol{V}^{(1)}; G), ..., C_{\text{LS}}(\boldsymbol{V}^{(N)}; G)$. During the policy gradient update, we use the standardized makespan across the samples, i.e.,

$$\overline{C}(\boldsymbol{V}; \boldsymbol{V}^{(1)}, ..., \boldsymbol{V}^{(N)}, G) := \frac{C_{\text{LS}}(\boldsymbol{V}; G) - \text{mean}_{n=1,...,N}[C_{\text{LS}}(\boldsymbol{V}^{(n)}; G)]}{\max\{\text{std}_{n=1,...,N}[C_{\text{LS}}(\boldsymbol{V}^{(n)}; G)], \epsilon\}}, \tag{13}$$

where $\epsilon > 0$ is used to clip the standard deviation in denominator for numerical stability.

The final algorithm with additional stabilization ideas is summarized in **Algorithm 1**. We train our model with multiple training epochs over $\mathcal{G}$ where a single epoch considers each graph in $\mathcal{G}$ once. For each graph $G \in \mathcal{G}$, we randomly sample N priorities and use REINFORCE to update the model parameter θ while regularizing the norm of logits by Eq. (12). We multiply with a constant $c_{\text{logits}} > 0$ the regularization loss to balance between REINFORCE loss and regularization. We set $c_{\text{logits}} = 0.001$ and observe that it empirically works well in all of our experiments.

Appendix C. Probability Distribution with Standardized Logits

Suppose our model has logits $l_0 := \text{logits}(0)$ and $l_1 := \text{logits}(1)$. One can easily show that the mean and standard deviation of logits are equal to $m = \frac{l_0+l_1}{2}$ and $s = \sqrt{\frac{l_0^2+l_1^2}{2} - m^2} = \frac{|l_0-l_1|}{2}$, respectively. Now for the hyperparameter $c > 0$ that is assumed to be fixed, consider the standardized logits

$$\bar{l}_0 := \overline{\text{logits}}(0) := c \times \frac{l_0 - m}{s}, \bar{l}_1 := \overline{\text{logits}}(1) := c \times \frac{l_1 - m}{s} \tag{14}$$

and model a binary random variable $X \in \{0, 1\}$ by using them. The probability distribution using the softmax and the above standardized logits becomes

$$\Pr\{X = 1\} = \frac{\exp(\bar{l}_1)}{\exp(\bar{l}_0) + \exp(\bar{l}_1)} = \frac{1}{1 + \exp(\bar{l}_0 - \bar{l}_1)} = 1 - \Pr\{X = 0\}. \tag{15}$$

Since the difference between $\bar{l}_0$ and $\bar{l}_1$ is

$$\bar{l}_0 - \bar{l}_1 = c \times \frac{l_0 - m}{s} - c \times \frac{l_1 - m}{s} = c \times \frac{l_0 - l_1}{s} = 2c \times \frac{l_0 - l_1}{|l_0 - l_1|}, \tag{16}$$

we have

$$\Pr\{X = 1\} = \begin{cases} \frac{1}{1+\exp(2c)}, & \text{if } l_0 > l_1, \\ \frac{\exp(2c)}{1+\exp(2c)}, & \text{otherwise.} \end{cases}$$

and $\Pr\{X = 0\} = 1 - \Pr\{X = 1\}$.

Appendix D. Related Works

ML for Combinatorial Optimization. The idea of using ML to solve DAG scheduling fits into the broader theme of using ML for Combinatorial Optimization (CO) which has received attention

recently [2]. Most of the works in the literature in ML for CO use RL to learn a policy to select actions for reward maximization which is set to be a direct function of the problem objective. The policy can be an end-to-end policy whose actions correspond to the the solution of the CO problem [8, 10, 11, 25, 26] or the policy can augment a traditional heuristic/solver of the problem to find better solutions [1, 15, 22, 24].
End-to-End ML Schedulers. In the context of the scheduling problems, [25] and [16] proposed an end-to-end GNN based policy to solve the JSSP problem which is a special case of DAG scheduling. Their policy is auto-regressive which selects the nodes to be scheduled iteratively and at each iteration they run the GNN on the modified disjunctive graph to get new node embeddings. Thus, they are required to run the GNN encoder $|N|$ times which is prohibitive for large graphs. In order to address the complexity of auto-regressive policies for large graphs encountered in compiler workflows, [26] came up with the idea of iterative refinement which refines the generated schedule by running their GNN policy K times where K is a hyper-parameter. [20], [14] and [27] consider the problem of scheduling jobs on data clusters and provide end to end deep RL solutions to solve data center scheduling under various settings.
Hybrid Schedulers. [22] propose a bi-level optimization approach for DAG scheduling where they learn a policy which modifies the input DAG by adding edges and then use the critical-path based list scheduling method on the modified DAG to obtain a schedule for the original problem. The authors allow upto K edges to be added to the DAG (where K is a hyper-parameter) and their policy can run the GNN K times to get to the final schedule. [15] proposed a neural augmented genetic algorithm for scheduling in compilers. They used a GNN policy to learn the parameters of the mutant distribution which was used by the genetic algorithm to find good schedules.

In contrast to these works, our method generates the node priorities end-to-end requiring only a single pass of our GNN encoder and uses list scheduling to obtain the final schedule. This makes our approach more efficient and scalable compared to prior works.

Appendix E. Scheduling Tasks

In this subsection, we describe how the different tasks that we consider fit into our scheduling framework described in section 2.1.
Job Shop Scheduling Problem (JSSP). JSSP is a special case of DAG scheduling where a set of N_J jobs need to be scheduled on N_M machines in $\mathcal{M}$. Each job consists of a sequence of operations that must go through N_M machines in a specific order. This task can be modeled in the framework of section 2.1 by setting $\mathcal{M} = \{1, 2, \ldots, N_M\}$, we have one machine of each type i.e. $\lambda(m) = 1, \forall m \in \mathcal{M}$, each node has a machine type it can be scheduled on i.e. $\mu(v) \in \mathcal{M}$, each node occupies the machine it is scheduled on i.e. $\rho(v) = 1, \forall v \in \mathcal{V}$.
DAG Scheduling on TPC-H. In this task we need to schedule nodes with single machine type (homogeneous case) i.e. $\mathcal{M} = 0$ and all the nodes have same machine type i.e. $\mu(v) = 0, \forall v \in \mathcal{V}$. We assume the same setting as in [22] for experiments and set the available resource of machine 0 to $\lambda(0) = 6000$. Node v occupies $\rho(v) \in \mathbb{N}$ with $\rho(v) \leq 6000$ resources where $\mathbb{N}$ is the set of positive integers.
Computation Graph Scheduling. In this task we need to schedule computation operations of a DAG on a hardware with 3 types of machines so $\mathcal{M} = \{0, 1, 2\}$. We assume that we have 1 machine of type 0 and type 1 whereas 4 machines of type 2 i.e. $\lambda(0) = \lambda(1) = 1, \lambda(2) = 4$. This hardware setting is inspired from the structure of real ML accelerators which have multiple machines (threads)

of different types available for data processing. A node v can be scheduled on one machine of its machine type $\mu(v) \in \mathcal{M}$ i.e., the resource required $\rho(v) = 1, \forall v \in \mathcal{V}$.

Appendix F. Dataset

F.1. Synthetic JSSP Instances

We generate synthetic JSSP instances by using the code deployed by [25][2]. For reproducibility of this dataset, we fix our seed as 0 and sample 150 instances for all combinations of $N_j \in \{25, 50\}, N_m \in \{10, 20, 30\}$, respectively. We use the first 100 instances as training instances and the last 50 instances as test instances.

F.2. TPC-H Dataset

We use TPC-H dataset generated by [22][3]. The dataset for each of TPC-50, TPC-100, TPC-150 experiments consists of 50 training graphs and 10 test graphs. The average number of nodes for TPC-50, TPC-100 and TPC-150 dataset are 467.2, 929.8 and 1384.5 respectively.

F.3. Synthetic Computation Graph Dataset

Layered graphs. Layered graphs were introduced in [4] as a model to generate synthetic graphs which have similar structure to the computation graph of a neural network. We use the default parameters specified in [4] to generate the graphs. In order to generate the node duration, we first sample the memory size $m(v)$ of each node $v \in \mathcal{V}$ and then use the following affine model to generate the duration $\delta(v)$ for node $v \in \mathcal{V}$,

$$\delta(v) = \text{round}(100 \times m(v)) + 1. \tag{17}$$

We sample $m(v)$ by first sampling the value from Gaussian Mixture Model (GMM) $m(v) \sim \text{GMM}(\mathbf{w}, \mu, \sigma)\big|_{\mathbb{R}_+}$ and projecting it to non-negative values. We use a mixture of four Gaussians and set the means to $(\text{mean}_1, \text{mean}_2, \text{mean}_3, \text{mean}_4) = (0.5, 1, 3, 5)$ and standard deviations to $(\text{std}_1, \text{std}_2, \text{std}_3, \text{std}_4) = (0.5, 1, 1, 1)$.

We sample the machine type $\mu(v)$ of node v such that $\Pr(\mu(v) = j) \propto \lambda(j)$ i.e. the probability that a node has machine type j is proportional to the available resources $\lambda(j)$ for machine type j. In our computational graph experiments we used $\lambda(0) = \lambda(1) = 1, \lambda(2) = 4$ which leads to the following multi-nomial distribution for $\mu(v)$:

$$\Pr(\mu(v) = j) = \begin{cases} 1/6, & \text{if } j = 0, \\ 1/6, & \text{if } j = 1, \\ 2/3, & \text{if } j = 2. \end{cases} \tag{18}$$

Erdos-Renyi and Stochastic Block Model. These two graph distributions are well-known families of random undirected graphs and were used in [15] for their experiments. We set the probability of an edge between any two nodes as $p = 0.05$ for the Erdos-Renyi graphs. We use the following

[2] https://github.com/zcaicaros/L2D/blob/main/DataGen/generate_data.py

[3] https://github.com/Thinklab-SJTU/PPO-BiHyb/tree/main/dag_data/tpch

parameters for the stochastic block model: the number of communities $k = 4$, the probability for an edge between two nodes of same community $p_{in} = 0.3$, the probability for an edge between two nodes of different communities $p_{out} = 0.001$. We use the procedure described in [15] to convert the instances from these two graph families into a DAG. We use the same distribution to generate node duration and their machine type as in layered graph for these two graph distributions as well.

The results for 1000-node graphs are shown in the main part of our work, and 500-node graph results are in Appendix H.

Appendix G. Baseline Algorithms

G.1. Priority-Based List Scheduling Baselines

We consider classical list scheduling baselines that use the following values as node priorities [22, 25]:
Critical-Path (CP). The priority for each node is determined by the length of the critical path from the node to the target node in a DAG.
Most OPeratioNs Remaining (MOPNR). The priority for each node is computed using the number of operations remaining on the longest path from the node to the target node in a DAG.
Shortest Processing Time (SPT). The priority for each node is computed using the inverse of the processing time of the node.

G.2. Constraint Programming Baseline

Constraint programming models are commonly used to solve scheduling problems in the literature. We have developed constraint programming formulations and carried out numerical simulations to provide a baseline comparison for our methods.

1. **Constraint programming for computation graph scheduling.** The developed constraint programming problem has start times of jobs and machine assignments as discrete variables. The constraints consist of precedence constraints, correct machine assignment constraints, and constraints for enforcing no overlap of tasks if they are assigned to the same machine.
2. **Constraint programming for DAG scheduling with TPC-H dataset.** The constraint programming problem for the TPC-H dataset experiments has only the start times of jobs as variables. The constraints ensure that the precedence relations in a DAG are satisfied, and the total resource usage does not exceed the given resource budget. In addition, the task durations for this dataset have been scaled by 1000 and rounded to the nearest integer. The reason for this step is that the CP-SAT solver only works with integer valued data, and the duration values for this dataset are of floating point precision. We have solved the constraint programming problem using the scaled and rounded duration values. We have then divided the resulting makespan by 1000. This rounding step introduces an error of at most 0.0005 seconds per node. Overall, for a 1000-node graph, this leads to an error of at most 0.5 seconds in the worst case. This is negligibly small since the makespan values for this dataset are in the order of few thousand seconds.

We have implemented these formulations using Google OR-Tools and solved them using the CP-SAT solver ([17]). Differently from other baselines, the constraint programming solver is guaranteed to find the optimal schedule given enough time. Due to practical reasons, we have set a time limit of 24 hours for the solver in the simulations. In most of the experiments, the time limit was reached before finding the optimal solution. In these cases, we have reported the best result obtained prior to the moment of time-out. Across all experiments, the solver has found the optimal solution for only 3 graphs in the TPC-H-50 dataset. For these 3 jobs, the amounts of time that the solver has run to find the optimal solutions are 2429, 21949, 39274 seconds, which are still significantly larger than our algorithms' run time.

We have observed that the memory requirements of the solver for large graphs could exceed the available memory in our servers. The graphs in TPC-H-150 fall into this category, where the experiments resulted in out-of-memory errors. We have not reported results for these graphs in Table 1.

In the experiments, we have provided initial feasible schedules for the solver. This is sometimes referred to as solution hinting or warm start. This speeds up the solver considerably. Initial feasible schedules could be obtained in different ways. One way is to generate a topological order of the DAG and map it to a sequential schedule without any parallelization. Another way is to run list scheduling and initialize the solver using the output of the list scheduling algorithm. We have experimented with both options. We have found that using the output of list scheduling leads to a better initialization for the solver.

G.3. Neural Baselines

In this section we provide how our neural baselines are used. For all the tasks, we use a machine with a single GPU (Nvidia Tesla V-100) with 32 GB memory that is also used to train and evaluate our model.

L2D. While the original implementation of L2D uses a new set of training graphs for every training iteration, we consider more practical scenario where the number of training graphs is restricted. We use the aforementioned 100 training graphs and 50 test graphs for each JSSP instance. We use 4 environments (that was proposed by [25]), which ends up with training 4 graphs per training iteration. We train the L2D model for 3000 iterations over 5 different seeds and report the one that performs the best. Note that we tested learning rates 2×10^{-5} (the default learning rate in their implementation) and 1×10^{-4} for L2D and the learning rate 1×10^{-4} works much better, and thus we report the results for this case.

PPO-BiHyb. For the experiment with TPC-H dataset, we use the pretrained model[4] deployed by the author. We confirmed that the results reported in their paper are reproducible and evaluate the run time in our machine settings. For computation graph scheduling, we train PPO-BiHyb for 40 epochs (where 1 epoch is defined to train all training graphs in the dataset once).

Appendix H. Additional Results

H.1. JSSP

We evaluate our model on randomly generated JSSP instances from [25] which are defined by number of jobs N_j and number of machines N_m. We summarize the results for $(N_j, N_m) = (25, 20), (25, 30), (50, 20)$ in Table 3. For each case, we train our algorithm for 100 training graphs and evaluate it for 50 unseen test graphs. We also train the neural baseline, L2D [25], with the same training and test graphs. Note that this is different from the original training of L2D since L2D generates a new set of training graphs for every training iteration. The results show that our algorithm works well across different numbers of machines and also outperforms both neural and non-neural

[4] `https://github.com/Thinklab-SJTU/PPO-BiHyb/tree/main/pretrained`

Table 3: Experiment results on synthetic JSSP instances are described. We use bold letters to emphasize the minimum average makespan for each JSSP instance.

	$(N_j, N_m) = (25, 20)$		$(N_j, N_m) = (25, 30)$		$(N_j, N_m) = (50, 20)$	
	Makespan	Time (sec)	Makespan	Time (sec)	Makespan	Time (sec)
CP	2120.24	0.009	2588.72	0.017	3290.80	0.024
SPT	2265.88	0.002	2739.20	0.005	3548.20	0.008
MOPNR	2115.86	0.012	2625.96	0.023	3278.82	0.030
L2D	2253.94	1.245	2799.00	2.850	3452.70	3.711
Greedy (ours)	2032.70	0.021	2512.40	0.031	3108.56	0.049
S(16) (ours)	1970.98	0.054	2452.64	0.085	3032.44	0.138
S(64) (ours)	1948.76	0.127	2427.30	0.294	3009.08	0.469
S(256) (ours)	**1932.42**	0.514	**2411.68**	0.909	**2997.10**	1.527

Table 4: Experiment results on synthetic JSSP instances ($N_j = 25$).

	$(N_j, N_m) = (25, 10)$		$(N_j, N_m) = (25, 20)$		$(N_j, N_m) = (25, 30)$	
	Makespan	Time (sec)	Makespan	Time (sec)	Makespan	Time (sec)
CP	1673.56	0.005	2120.24	0.009	2588.72	0.017
SPT	1807.90	0.001	2265.88	0.002	2739.20	0.005
MOPNR	1656.26	0.007	2115.86	0.012	2625.96	0.023
L2D	1725.58	0.577	2253.94	1.245	2799.00	2.850
Greedy (ours)	1578.58	0.018	2032.70	0.021	2512.40	0.031
S(16) (ours)	1540.92	0.045	1970.98	0.054	2452.64	0.085
S(64) (ours)	1527.40	0.092	1948.76	0.127	2427.30	0.294
S(256) (ours)	**1519.16**	0.321	**1932.42**	0.514	**2411.68**	0.909

baselines. Due to one-shot decoding scheme in our model, the running time of our model is shorter than L2D. We observe that L2D ends up with worse performance than non-neural baselines although we train L2D until convergence. We think L2D does not generalize well with the limited number of training graphs, while our algorithm generalizes well with the same dataset.

We describe empirical results for $(N_j, N_m) = (25, 10), (25, 20), (25, 30)$ in Table 4 and $(N_j, N_m) = (50, 10), (50, 20)$ in Table 5. For broader settings, we can see that our neural scheduler outperforms our baselines.

H.2. Computation Graphs

We describe empirical results for both 500 and 1000 node synthetic computation graphs in Table 6. Over all sizes and graph distributions, our algorithm achieves better speedup within a short time.

Table 5: Experiment results on synthetic JSSP instances ($N_j = 50$).

	$(N_j, N_m) = (50, 10)$		$(N_j, N_m) = (50, 20)$	
	Makespan	Time (sec)	Makespan	Time (sec)
CP	2903.14	0.012	3290.80	0.024
SPT	3111.74	0.003	3548.20	0.008
MOPNR	2897.60	0.016	3278.82	0.030
L2D	2964.96	1.084	3452.70	3.711
Greedy (ours)	2814.58	0.024	3108.56	0.049
S(16) (ours)	2800.32	0.064	3032.44	0.138
S(64) (ours)	2797.84	0.165	3009.08	0.469
S(256) (ours)	**2796.98**	0.644	**2997.10**	1.527

Table 6: Experiment results on synthetic graph datasets are described.

		Layered Graph		Erdos-Renyi		Stoc. Block Model	
		SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)
500 Node Graph	CP	4.314	0.025	4.973	0.015	4.664	0.018
	SPT	4.214	0.004	4.480	0.002	4.416	0.003
	MOPNR	4.428	0.033	5.033	0.018	4.715	0.023
	Greedy (ours)	4.513	0.042	5.113	0.017	4.835	0.021
	S(16) (ours)	4.558	0.129	5.152	0.082	4.901	0.082
	S(64) (ours)	4.585	0.269	5.168	0.182	4.935	0.185
	S(256) (ours)	**4.603**	0.803	**5.178**	0.545	**4.956**	0.539
1000 Node Graph	CP	4.580	0.058	5.049	0.055	4.701	0.055
	SPT	4.526	0.013	4.541	0.008	4.473	0.007
	MOPNR	4.745	0.075	5.112	0.068	4.761	0.068
	Greedy (ours)	4.819	0.094	5.194	0.046	4.866	0.043
	S(16) (ours)	4.848	0.311	5.211	0.214	4.916	0.198
	S(64) (ours)	4.872	0.750	5.227	0.542	4.944	0.477
	S(256) (ours)	**4.889**	2.418	**5.239**	1.799	**4.965**	1.533

Appendix I. Training and Model details

I.1. Training

In this section we provide the training details of our model on the different scheduling tasks that we consider. For all the tasks we trained our model on a machine with a single GPU (Nvidia Tesla V-100) with 32 GB memory. We used Adam optimizer for training on all the tasks.

JSSP. In this case we train our model on each JSSP instance with 100 training graphs and test it on 50 unseen graphs. We train our model for 25 epochs with 5 random seeds and pick the best performing model. We use the number of samples $N = 1000$.

TPC-H Dataset. We train our model on each TPC instance for 100 epochs with 10 random seeds and pick the best performing model. We use the number of samples $N = 2000$.

Computation Graphs. For each synthetic graph distribution, we consider the graph size equal to either 500 or 1000 and generate a training set of 3000 graphs and 300 unseen test graphs. For the real-world graphs, our dataset consists of 92 train graphs and 23 test graphs. We train for our model for 20 epochs with 5 random seeds and pick the best performing model. We use the number of samples $N = 1000$ for both synthetic and real computation graphs.

I.2. Model architecture

We use topoformer encoder with the same hyperparameters as described in [4]. We use $c_{\text{logits}} = 0.001$ and $\epsilon = 0.1$ for clipping the denominator in Eq. (13).

I.3. Input features

We use the following input node features $\boldsymbol{x}_v$ for node $v \in \mathcal{V}$:

- Node duration $\delta(v)$
- Node resource requirement $\rho(v)$
- One hot representation of node machine type $\mu(v)$
- The critical path duration from v to the target node and critical path duration from source node to v

Note that for tasks where $\rho(v) = 1, \forall v$ we ignore $\rho(v)$ from the input feature. We normalize each entry i of the input features across the nodes so that the features are in between 0 and 1 as follows:

$$\boldsymbol{x}_v[i] = \frac{\boldsymbol{x}_v[i]}{\max_{v' \in \mathcal{V}} \boldsymbol{x}_{v'}[i]}$$

In addition, we also augment the node features with the Laplacian positional encodings [3] of dimension 20 by computing it on the undirected version of the DAG. We pass the input feature $\boldsymbol{x}_v$ through a linear layer to obtain the initial embedding of node v for the topoformer encoder.