Received October 21, 2021, accepted November 21, 2021, date of publication November 23, 2021, date of current version December 6, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3130407

A Global DAG Task Scheduler Using Deep Reinforcement Learning and Graph Convolution Network

HYUNSUNG LEE[1], SANGWOO CHO[2], YEONGJAE JANG[2], JINKYU LEE[3], (Senior Member, IEEE), AND HONGUK WOO[3], (Member, IEEE)
[1]Kakao Corporation, Seongnam 13494, South Korea
[2]Department of Mathematics, Sungkyunkwan University, Suwon 16419, South Korea
[3]Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Honguk Woo (hwoo@skku.edu)

This work was supported in part by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant 2021-0-00875, in part by the ICT Creative Consilience Program supervised by the IITP under Grant IITP-2020-0-01821, in part by the National Research Foundation of Korea (NRF) funded by the Korean Government (MSIT) under Grant 2020M3C1C2A0108081 and Grant 2021R1A2B5B02001758, and in part by Kakao I Research Supporting Program.

ABSTRACT Parallelization of tasks and efficient utilization of processors are considered important and challenging in operating large-scale real-time systems. Recently, deep reinforcement learning (DRL) was found to provide effective solutions to various combinatorial optimization problems. In this paper, inspired by recent achievements in DRL, we employ DRL techniques for scheduling a directed acyclic graph (DAG) task in which a set of non-preemptive subtasks are specified by precedence conditions among them. We propose a DRL-based priority assignment model for scheduling a DAG task on a multiprocessor system, named `GoSu`, which adapts a graph convolution network (GCN) to process a complex interdependent task structure and minimize the makespan of a DAG task. Our proposed model makes use of both temporal and structural features in a DAG to effectively learn a priority-based scheduling policy via GCN and policy gradient methods. With comprehensive evaluations, we verify that our model shows comparable performance to several state-of-the-art DAG task scheduling algorithms, and outperforms them by 2∼3% in the slowdown of achieved makespans particularly in nontrivial system configurations where workloads are neither too small nor heavy compared to the given number of processors. We also analyze the priority assignment behaviors of our model by leveraging a regression method that imitates the learned policy of the model.

INDEX TERMS Deep reinforcement learning, graph convolution network, policy gradient learning, DAG task.

I. INTRODUCTION

In cyber-physical real-time systems, there has been an increasing demand for both high performance and strong timeliness to execute a pipeline of complex functions, e.g., autonomous driving with perception, planning and control. One of the key techniques to meet the demand is to exploit parallelism on a multiprocessor system. A directed acyclic graph (DAG) task has been used to represent the dependencies among a number of task components (subtasks) and to formulate a fine-grained parallel scheduling problem with the interdependent subtasks. Furthermore, as non-preemptive task models can avoid the overhead issue of migration and switching tasks, recently, priority-based non-preemptive scheduling for a DAG task has gained much attention [1]. The problem has been tackled by investigating scheduler techniques with priority assignments, which take a set of subtasks in a single DAG as input and produce a priority order for the subtasks and their non-deterministic execution order [1], [2].

Given a priority order, subtasks can be scheduled to run in parallel on multiple processors subject to the precedence conditions. Existing studies have developed heuristic priority assignment algorithms and analyzed their impact on the goal of minimizing the elapsed time (or makespan) required to complete all the subtasks [1]–[3], as there might be a number

The associate editor coordinating the review of this manuscript and approving it for publication was Binit Lukose.

of different priority orders, e.g., up to $n!$ where n is the number of subtasks. Due to the nature of heuristic strategies, they have not been able to establish fundamental design principles for DAG schedulers such as exploiting temporal and graph-structural spatial features under a variety of task configurations and scales, which highly affect prioritization.

In this paper, we present a learning-based priority assignment model for scheduling a single DAG task of multiple interdependent subtasks on a multiprocessor system with a non-preemptive mechanism. The model not only generates a priority order more favorable than existing approaches to achieving the goal but also automatically identifies critical temporal and graph-structural features of a DAG and systematically utilizes the features for priority assignments under various task configurations.

To do so, we use deep reinforcement learning (DRL) techniques, considering the difficulty to collect sufficient supervised labels for optimal priority assignments for DAG tasks. We also consider a large combinatorial problem space in priority assignments. Furthermore, we adapt a graph convolution network (GCN) with both forward-path and inverse-path aggregation to effectively extract temporal (e.g., execution times) and structural (e.g., precedence conditions) features from graph representations. We then present the `GoSu` (graph convolutional task scheduler) model for which the priority assignment policy using GCN-based embeddings is established through the policy gradient learning with slowdown-based rewards. In `GoSu`, each subtask is assigned a priority offline, and then during runtime, subtasks are scheduled on the basis of their priority order. This is the same as the fixed-priority (static-priority) scheduling in the field of real-time systems [4] where the highest-priority subtasks that are not yet scheduled are executed first.

The `GoSu` model consistently shows competitive performance compared to several state-of-the-art DAG scheduling heuristic methods based on fixed-priority assignments (i.e., [1], [2]) under various experimental settings. Furthermore, the model often achieves better performance than those heuristic methods particularly in nontrivial task configurations, e.g., showing a 2$\sim$3% gain in the slowdown of achieved makespans for the cases when the number of processors is 3 or 4 with moderate parallelism and when the number of processors is between 3 and 8 with high parallelism (Figure 4(b) and (c)). We also demonstrate that `GoSu` outperforms the others with a probability of up to 73% for each testing DAG task sample in several nontrivial cases (Figure 5(b) and (c)).

The main contributions of this paper are as follows.

- We present a DRL-based priority assignment model `GoSu` for scheduling a single DAG task on a multiprocessor system.
- We devise a GCN path aggregation scheme and a slowdown-based reward function specific to the objective of makespan minimization for a DAG task.
- We show performance gains by `GoSu` through comprehensive evaluations, and provide an analysis on the learned policy by projecting the `GoSu` model to a linear model via differentiable programming.

The rest of the paper is organized as follows. Section III briefly describes the DAG task scheduling problem and its objective. Section IV presents our proposed encoder-decoder model structure with GCN-based DAG embeddings and Attention-based priority assignments for subtasks in a DAG. Sections V and II provide the experiment results and analysis, and the review on related work, respectively. Finally, Section VI concludes the study. In addition, Table 1 provides a list of symbols used throughout this paper with three aspects such as DAG tasks, model training, and datasets.

TABLE 1. A list of symbols.

Symbols: DAG	Description
τ	A task with DAG $G = (V, E)$
$v_i \in V$	A node in G, a subtask of τ
x_i	Raw features of v_i (in Table 3)
C_i	The worst case execution time of v_i
$e_{ij} \in E$	An edge in G, a precedence condition
W	Total workload, $\sum_{v_i \in V} C_i$
L	Critical workload, $\sum_{v_i \in \text{CriticalPath}} C_i$
$\text{deg}_{\text{in}}(v_i)$	In-degree of v_i, $\lvert\{e_{ji} \mid e_{ji} \in E\}\rvert$
$\text{deg}_{\text{out}}(v_i)$	Out-degree of v_i, $\lvert\{e_{ij} \mid e_{ij} \in E\}\rvert$
$\mathcal{N}(v_i)$	Neighbors of v_i, $\{v_k \mid e_{ki} \in E\} \cup \{v_i\}$
$\mathcal{N}^{-1}(v_i)$	Inverse neighbors of v_i, $\{v_k \mid e_{ik} \in E\} \cup \{v_i\}$
π	A priority order for τ, a permutation
π^*	An optimal priority order for scheduling τ
$M(\pi, \tau)$	Makespan (elapsed time) for τ by π-schedule
$M_{LB}(\tau)$	The lower bound of makespan for τ
Symbols: Training	Description
J	Objective function
θ	Model parameters
W_θ	Linear transformation
σ	Nonlinear activation function
$h_i^{(k)}$	kth latent vector in GCN msg. passing
O_t	A set of priority assigned nodes at t
L_t	A set of priority unassigned nodes at t
c_t	Latent context vector of nodes at step t
$p_\theta(\pi \mid \tau)$	Prob. of priority order π for τ by θ-policy
$S(\pi, \tau)$	A slowdown reward, $-M(\pi, \tau)/M_{LB}(\tau)$
β	Baseline model parameters
$B(b, \tau)$	A baseline slowdown reward by β
s_i	Score of node v_i
Symbols: Dataset	Description
n_{child}	The number of child nodes in a DAG
n_{depth}	The number of layers in a DAG
p_{fork}	Prob. of edge generation between two nodes
p_{pert}	Prob. of perturbation
m	The number of processors in a platform

II. RELATED WORK

In the real-time system literature, fixed-priority scheduling schemes for parallel tasks have been receiving attention relatively recently in [1], [2], while many works on DAG task scheduling considered dynamic scheduling cases [3], [5], [6], [12], [13]. By fixed-priority scheduling, subtasks (or nodes) in a DAG task can be scheduled globally on all processors, and furthermore, a platform can be free from overhead issues of online scheduling in real-time systems [1], [14]. The overview of these related works is at the top of Table 2.

For example, He *et al.* [2] proposed a simple but efficient scheduling heuristic by which nodes in the critical

TABLE 2. Summary of related studies: At the top for *DAG task scheduling*, the *"Properties"* column describes which properties each method makes explicit use of, i.e., temporal (execution time, deadline) in *"Temp."* and spatial (precedence conditions) in *"Spat."*. At the middle for *DRL-based scheduling*, the *"Target"* column describes the target application domain that each method focuses on. Our work addresses the issue of DAG task scheduling by using both temporal (O in *"Temp."*) and spatial (O in *"Spat."*) properties, and shares the similar problem specification with [1], [2] where node-level fixed-priority scheduling is explored; however, unlike those existing studies, GoSu leverages learning-based approaches through DRL and adapts GCN techniques subject to DAG representations. In addition, regarding the *"Target"* column at the middle, GoSu is a novel DRL-based DAG task scheduling model.

	Name	*Properties*		Description
		Temp.	*Spat.*	
DAG task scheduling	GEDF [5]	O	X	This work verified the efficiencies and theoretical bounds of global earliest deadline first (EDF) algorithms in DAG scheduling.
	Pathan17 [6]	O	X	This presented a two-level scheduler with the DAG-level determining the priority among DAGs and the node-level performing subtask scheduling.
	DAG-Fluid [3]	O	X	This applied fluid theory [7] for DAG scheduling, decomposing heavy nodes into smaller ones, and assigning priorities based on utilization.
	He19 [2]	X	O	In this iterative algorithm, nodes in the critical path have the highest priorities, and nodes blocking priority-assigned nodes are assigned the next highest priorities.
	Zhao20 [1]	O	O	This heuristic algorithm focuses on node-level fixed-priority DAG scheduling, similar to our problem. It employs the provider and consumer model.

	Name	*Target*	Description
DRL-based scheduling	DeepRM [8]	Cluster mgmt.	This work was the first attempt to adopt DRL for cluster management, presenting a time-slice based scheduling framework.
	Metis [9]	Cloud computing	This model supports a scheduler for long running apps in a cloud environment, enhancing scales (e.g., up to 3K tasks) with recursive MLPs.
	Decima [10]	Data cluster	This model employs structural embeddings for data parallel tasks (e.g., spark), enabling scalable, online task scheduling in a cluster environment.
	Panda [11]	Real-time system	This model casts the GFPS problem into combinatorial optimization, employing the sequential decoder structure for priority assignments on a set of periodic tasks.

path are first scheduled and then other nodes that might immediately interfere with the critical path execution are scheduled. Zhao *et al.* [1] presented a single non-preemptive DAG scheduling framework that partitions nodes in a DAG into providers (i.e., nodes in the critical path) and consumers (nodes not in the critical path), intending to exploit both parallelism and dependency conditions. Given the partitioned nodes, the highest priorities are assigned to the providers, the second-highest priorities to nodes that might block the providers, and the lowest priorities to the other nodes. These prior works concentrate on structural features of DAGs of which the representation needs to be designed according to problem specifications and task configurations. Our work shares the same problem structure with these, focusing on fixed-priority scheduling and producing a priority order to minimize the makespan of a task running on a multiprocessor system. However, unlike these works based on sophisticated engineering procedures, our DRL-based model employs GCNs to extract important features from both individual node and graph-structural data, hence automating the learning of relevant enriched features.

There exist several works on applying DRL for scheduling tasks in the various domains such as cloud computing, networks, and manufacturing systems [8]–[11], [15]–[18]. DeepRM [8] was the first attempt to learn a scheduling policy systematically through DRL. This work introduces a time-slice based framework to solve scheduling problems in cluster management. However, this is limited in scales due to no consideration on permutation-invariant properties, i.e., task sets $[\tau_1, \tau_2, \tau_3]$ and $[\tau_2, \tau_1, \tau_3]$ are treated differently. Wang *et al.* [9] presented Metis, a DRL-based scheduler for managing online long-running applications in a cloud computing platform. They tackled the device placement issue in application containers across a cluster of server machines by exploring hierarchical DRL. Mao *et al.* [10] introduced a DRL-based Scheduling for large-scale cluster task scheduling. They developed sophisticated task and subtask embedding techniques and used REINFORCE algorithm for model training, maximizing service level objectives such as cluster utilization and job completion time. Recently, Lee *et al.* [11] adopted DRL for the global fixed-priority task scheduling on a multiprocessor system using the Transformer architecture.

While DRL techniques were employed for task scheduling in these studies, as summarized at the bottom of Table 2, the applicability of DRL and GCNs for DAG task scheduling has not been fully investigated. Our work focuses on the learning-based DAG task scheduling model by not only leveraging DRL and adapting GCN techniques but also analyzing the learned priority assignment policy of the model.

With the advancement of deep learning, numerous research works based on DRL were proposed to solve combinatorial optimization problems. The pointer network [19] provided a well-structured mechanism to adopt neural networks for combinatorial problems, e.g., traveling salesman problem (TSP). It establishes the probabilistic representation of a permutation by continuously estimating the probability of selecting a next item, e.g., the next city to visit in TSP. The pointer network was extended in a DRL framework [20]. This extension has been considered effective in that it is often impossible to establish labeled datasets of reliable quality (e.g., those annotated by optimal solutions) for large-scale combinatorial problems. In [21], a learning model based on

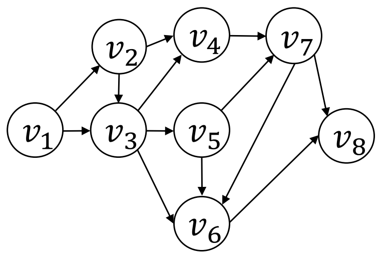


FIGURE 1. A DAG task dependency graph: For a DAG task with 8 subtasks, precedence conditions among 8 nodes $v_1, \ldots, v_8$ are represented in a DAG, where e.g., the condition by the edge from v_1 to v_2 specifies that v_2 cannot be executed before the completion of v_1.

Transformer [22] was proposed to solve TSP and several other representative combinatorial problems. In particular, a simple rollout baseline for policy gradient algorithms was introduced to facilitate model training and fast convergence. In the same vein, our work leverages DRL in discrete optimization problem spaces but concentrates on DAG task scheduling with GCNs.

III. DAG TASK SCHEDULING

In this section, we describe the DAG task scheduling problem for which global fixed-priority scheduling (GFPS) [23] schemes can be applied.

In the DAG task scheduling problem with GFPS, we consider a single DAG task that consists of n subtasks and precedence conditions among the subtasks. Regarding a system model in which a DAG task runs, we consider a multi-processor platform of m *homogeneous* processors with *non-preemptive* scheduling. We also presume that n is much larger than m, considering resource limitations. Then, GFPS with a priority order for the n subtasks is able to schedule the m highest-priority subtasks among the subtasks that have not been scheduled but satisfy the precedence conditions in each time slot upon the system of m processors. Since we focus on non-preemptive scheduling, each subtask cannot be preempted by any other subtasks once it starts its execution. The goal of this scheduling is to minimize the makespan of a DAG task. A priority order for the n subtasks corresponds to a mapping of the subtasks to distinct integers $i \in \{1, \ldots, n\}$, and it is fixed. That is, each subtask in a DAG task is assigned such i as its priority so that the precedence conditions in the DAG are satisfied and the makespan can be minimized, when the subtasks are scheduled according to the priority order on m processors in GFPS. We consider a work-conserving scheduler that intends to always utilize all available processors.

Specifically, a DAG task τ is represented in a task dependency graph $G = (V, E)$ that specifies the precedence conditions among subtasks within τ, where V denotes a set of nodes corresponding to the subtasks (as many as n) and E denotes a set of edges corresponding to the precedence conditions. Each node $v_i \in V$ represents a subtask of τ with worst-case execution time (WCET) C_i. Accordingly, we use the terms node and subtask interchangeably. Same as the DAG task scheduling formulation in the real-time system literature [1]–[3], we assume that the DAG specification including WCET C_i is known in advance. Each edge $e_{ij} \in E \subset V \times V$ corresponds to a precedence condition between two subtasks, specifying that v_j cannot run before the completion of v_i. That is, v_i is a predecessor of v_j, and v_j is a successor of v_i.

Furthermore, we use $\deg_{\text{in}}(v_i) = |\{e_{ji} | e_{ji} \in E\}|$ and $\deg_{\text{out}}(v_i) = |\{e_{ij} | e_{ij} \in E\}|$ to denote *in-degree* and *out-degree* of node v_i, respectively. In a task dependency graph, there might be more than a single source node such that $\deg_{\text{in}}(v) = 0$ or a single sink node such that $\deg_{\text{out}}(v) = 0$. In the case of multiple source nodes, we add a dummy source of WCET $= 0$ connecting to each of the multiple source nodes. Similarly, in the case of multiple sink nodes, we add a dummy sink of WCET $= 0$ connected to each of the multiple sink nodes. This allows us to consider only the DAG formation with one single source and sink. For example, Figure 1 illustrates a DAG task dependency graph with the precedence conditions among 8 nodes where v_1 and v_8 are source and sink nodes, respectively. In this example, the edge from v_1 to v_2 specifies that v_2 cannot be executed before the completion of v_1.

A *path* $\lambda = [v_{\lambda_1}, v_{\lambda_2}, \ldots, v_{\lambda_k}]$ is a finite sequence of nodes such that $(v_{\lambda_i}, v_{\lambda_{i+1}}) \in E$ for $i \in \{1, \ldots, k-1\}$. A path is called *complete path* if it contains both a source and a sink. A *path length* is the sum of the execution times of nodes in a path, i.e., $\sum_{v_i \in \lambda} C_i$. A path with the longest path length among complete paths is called *critical path*. We use L to denote the critical workload, the sum of WCETs of only the nodes in a critical path, while we use W to denote the total workload, the sum of WCETs of all nodes such that $W = \sum_{v_i \in V} C_i$.

For a DAG task τ of n nodes, our model is learned to generate a priority order for a system of m processors, which maps the nodes in τ,

$$[v_1, v_2, \ldots, v_n] \tag{1}$$

to a permutation of distinct integers from 1 to n such as

$$\pi = [\pi_1, \pi_2, \ldots, \pi_n]. \tag{2}$$

That is, given such π, we establish a priority-ordered node list,

$$[v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}] \tag{3}$$

which specifies that node v_{π_1} is assigned the highest priority, v_{π_2} is assigned the next highest priority, and so on for GFPS upon a system of m processors.

We use *makespan* $M(\pi, \tau)$ to denote the elapsed time for task τ by π-scheduling, which is required to complete τ when GFPS with priority order π is considered. Our model aims to find a priority order π^* that minimizes the makespan of τ, allowing efficient use of the computing resources of a system running τ.

$$\pi^* = \mathbf{argmin}_\pi M(\pi, \tau) \tag{4}$$

Notice that our model can be used scheduling a *periodic* DAG task τ with its deadline D and period T, where jobs

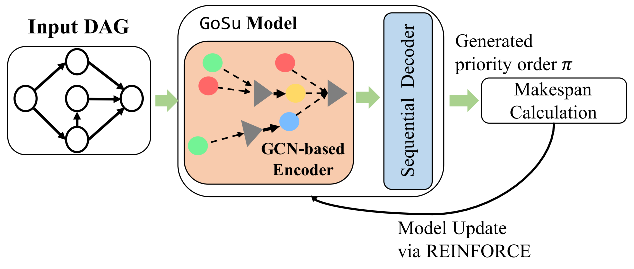


FIGURE 2. **Overall structure of GoSu: The GoSu model takes a DAG task as input, generating the embeddings for the DAG task via the GCN-based encoder, and uses the sequential decoder to produce a priority order π from the embeddings for the DAG task upon a multiprocessor system. The model is learned to minimize the makespan of running each DAG task, so calculated makespans are used as reward signals for updating the model through REINFORCE algorithm [24].**

for τ repeat regularly at the inter-release time of T. It is determined that the periodic task τ is *schedulable* if the model yields such π^* that the time constraint of τ, $M(\pi^*, \tau) \leq D$ holds. In this regard, it is also feasible to use the model for scheduling multiple DAG tasks, because for each DAG task, its WCET can be individually induced by its respective tight makespan bound.

IV. PROPOSED APPROACH

In this section, we describe our scheduling model that takes a DAG task as input and produces a priority order in GFPS for the task. Figure 2 illustrates the overall model structure with encoder and decoder modules. The modules are end-to-end trained through DRL to generate a priority order of a DAG task input, where the learning objective is to minimize the makespan of the task scheduled by the priority order. Specifically, we adapt graph learning with two-way path aggregation in the encoder to effectively extract the relational information in a DAG. We also employ a sequential selection procedure in the decoder to robustly update the ordering probability over timesteps of variable input task sizes.

In the GCN-based encoder, the raw features of individual n nodes (or n subtasks) in a DAG task τ are first processed via a feed-forward network (FFN). The structural information of τ's task dependency graph is encoded via the message passing mechanism of a GCN, and then latent vectors of the nodes are generated. With the latent vectors, the Attention-based decoder generates a priority order for the nodes of τ, establishing a probability distribution, i.e., $p_\theta(\pi|\tau)$ over priority orders π. Given the learning objective to minimize the makespan, the model with the encoder and decoder is trained in an end-to-end fashion by DRL with calculated makespans. Note that throughout this paper, a subscript θ is used to represent trainable model parameters.

The decoding for a priority order is conducted in a sequential selection procedure of n (time)-steps,

$$p_\theta(\pi|\tau) = \prod_{t=1}^{n} p_\theta(\pi_t|\pi_1, \ldots, \pi_{t-1}, \tau) \tag{5}$$

where Attention mechanism [19] is used to calculate the probability of each selection $p_\theta(\pi_t|\pi_1, \ldots, \pi_{t-1}, \tau)$. Each selection at step t assigns the priority $(n - t)$ to a selected node. In general, a problem space for DAG task scheduling can be intractably large, e.g., there are 100! different combinations in priority orders for a DAG task with 100 subtasks, so it is not effective to use a single probability distribution on whole possible orders of n subtasks. Rather, it is more feasible to use a sequential procedure to select a subtask with the next-highest priority iteratively, as have been studied in combinatorial optimization problems [19], [20].

In the following, we describe these encoding and decoding procedures based on DRL. We notate functions or modules parameterized by θ using a subscript θ. For example, an affine transform function parameterized by θ is represented in

$$\mathbf{Aff}_\theta(x) = W_\theta x + b_\theta. \tag{6}$$

Note that parameters in different modules denote different parameter sets.

It is noteworthy that while our model is intended for scheduling a single DAG task with many interdependent subtasks, it can be used for scheduling multiple parallel DAG tasks. For a DAG task, the model generates a priority order for its subtasks and yields the makespan bound by the priority order. Then, such makespan bound can be used to estimate the WCET of a DAG task. Given a set of DAG tasks where each task is associated with its respective WCET estimated by the model, it is feasible to adopt multiprocessor real-time task scheduling methods (e.g., [11], [25]).

A. GCN-BASED ENCODER

The encoder learns to represent each node in a DAG task into a fixed-sized vector. Each node has its own execution time, and a set of nodes forms a graph structure with precedence dependency. To reflect both individual node information and graph-structured information, the encoder takes a two-step procedure in that it first transforms individual raw features of each node into vectors through an FFN and then incorporates graph structure into the vectors through a GCN.

1) NODE EMBEDDING

Using an affine transform with the tanh(·) activation, the encoder transforms raw features x_i of node v_i into a latent vector representation via an FFN.

$$h_i^{(0)} = \tanh(W_\theta^{(0)} x_i + b_\theta^{(0)}) \tag{7}$$

The raw features used in our implementation are all listed in Table 3. In addition to node-level features such as execution time, the number of incoming edges, and the number of outgoing edges, we also include several graph-level features such as critical and non-critical workloads.

2) GRAPH-LEVEL EMBEDDING

The encoder iteratively updates the generated vector representations with the affine projection above through a

TABLE 3. Raw node-level features x_i of node v_i and additional graph-level features in DAG task τ: C_i denotes WCET of v_i, W denotes total workload of τ, and L denotes critical workload of τ, as described in Section III.

Node features	Description
normalized execution time	C_i/W, $\log(1+C_i)$
in-degree	$\deg_{\text{in}}(v_i)$
out-degree	$\deg_{\text{out}}(v_i)$
v_i is source or sink	{0, 1}
v_i is in critical path	{0, 1}
Graph features	Description
normalized critical workload	L/W
normalized non-critical workload	$(W-L)/W$

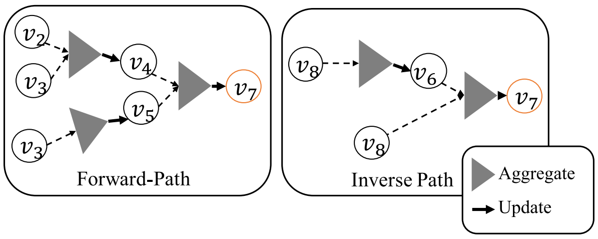


FIGURE 3. Graph path aggregation with two path types, forward- and inverse-path: This illustrates how to produce the representation of node v_7 in a DAG (in Figure 1) using the forward-path and inverse-path aggregation, where self-loops in the aggregate are omitted for simplicity.

GCN to further incorporate graph-structural features into the vector representations. In principle, the encoding process through a GCN can be seen as iterative message passings along the paths in a DAG. At step k, a latent representation $h_i^{(k)}$ of node v_i is updated by aggregating latent representations of v_i's neighbor nodes as shown in Figure 3. Note that in a GCN, v_i's neighbors include directly connected nodes as well as v_i itself. Accordingly, we define

$$\mathcal{N}(v_i) = \{v_k | e_{ki} \in E\} \cup \{v_i\} \tag{8}$$

hereafter.

Then, the GCN message passing can be formulated as

$$h_i^{(k+1)} = \textbf{Update}(\textbf{Aggregate}_\theta(\{h_j^{(k)} | v_j \in \mathcal{N}(v_i)\})) \tag{9}$$

where the **Aggregate** function accumulates messages from the neighbors of v_i, and the **Update** function takes the accumulated embedding and performs a nonlinear transformation on the embedding. The representations generated by iterating the above message passing operation K times can contain structural features in that each node is differently aggregated according to graph topology. The representations also preserve node features; at $k = 0$, $h_i^{(0)}$ in Eq. (7) is an initial node embedding from node features that are preserved during GCNs along with structural varieties embedded [26].

For aggregation, we adopt Attention [27] similar to [28]. Specifically, the encoder performs the Attention operations such as

$$\textbf{Att}_\theta(\{h_j^{(k)} | v_j \in \mathcal{N}(v_i)\}) = W_\theta \sum_{v_j \in \mathcal{N}(v_i)} \alpha_{ij} h_j^{(k)} \tag{10}$$

where α_{ij} denotes the Attention coefficient that estimates the importance of node v_j to node v_i's representation. We restrict α_{ij} to satisfy $\sum_j \alpha_{ij} = 1$ and formulate it as

$$\alpha_{ij} = \frac{\exp\left(a_\theta^T W_\theta h_i + b_\theta^T W_\theta h_j\right)}{\sum_{v_{j'} \in \mathcal{N}(v_i)} \exp\left(a_\theta^T W_\theta h_i + b_\theta^T W_\theta h_{j'}\right)} \tag{11}$$

where a and b are trainable vectors, and W is a trainable matrix.

We exploit several Attention modules (heads) simultaneously and aggregate them to get the final latent representation. This is useful since different Attention heads can give weights more relevantly to different latent representations (h). The **Aggregate** function is implemented using multi-head Attention, and is defined as

$$\begin{aligned}\textbf{Aggregate}_\theta&(\{h_j^{(k)} | v_j \in \mathcal{N}(v_i)\}) \\ &= \left(\textbf{Att}_{\theta,1} \oplus \textbf{Att}_{\theta,2} \oplus \cdots \oplus \textbf{Att}_{\theta,H}\right)\end{aligned} \tag{12}$$

where $\oplus$ concatenates vectors, and $\textbf{Att}_{\theta,1}, \ldots, \textbf{Att}_{\theta,H}$ are H-individual Attention modules. H denotes the number of distinct Attention heads per layer.

We then apply an affine transform followed by the exponential linear unit activation (**ELU**) [29] for the **Update** function. This constructs the Attention network layer for the GCN message passing in Eq. (9).

$$\begin{aligned} h_i^{(k+1)} &= \textbf{Update}(\textbf{Aggregate}(\{h_j^{(k)} | v_j \in \mathcal{N}(v_i)\})) \\ &= \textbf{ELU}\left(W_\theta \left(\textbf{Att}_{\theta,1} \oplus \cdots \oplus \textbf{Att}_{\theta,H}\right) + b_\theta\right) \end{aligned} \tag{13}$$

The result latent vectors $h_i^{(k)}$ are produced through consecutive GCN layers. We refer to those as node embeddings for $k = 3$ in our implementation.

3) TWO-WAY PATH AGGREGATION

In a DAG, an edge e_{ij} normally specifies a predecessor condition such that a node v_j cannot be executed without the termination of a node v_i. However, in the message passing, we observe that successor conditions are equally important as predecessor conditions. Thus, we adopt two types of Attention heads, incorporating them in the message-passing loop. This is similar with [22], [28] except we exploit different masks for each Attention head. Specifically, we define inverse neighbors as

$$\mathcal{N}^{-1}(v_i) = \{v_k | e_{ik} \in E\} \cup \{v_i\}. \tag{14}$$

Then, among H Attention heads, we set half of them to be forward-path graph Attention in Eq. (10) and the other half to be inverse-path graph Attention in Eq. (15).

$$\textbf{Att}_\theta(\{h_j^{(k)} | v_j \in \mathcal{N}^{-1}(v_i)\}) \tag{15}$$

This graph path aggregation in a GCN with both forward-path and inverse-path is illustrated in Figure 3. Employing Attention from the two path types enables the model to learn features on predecessor conditions as well as successor conditions in a DAG while preserving their different relations.

B. SEQUENTIAL DECODER

With the node embeddings (the final latent vectors $h_i^{(k)}$ in Eq. (9)) from the encoder, for a DAG task τ of n nodes, the decoder sequentially selects nodes to generate an n-sized priority order $\pi = [\pi_1, \pi_2, \dots, \pi_n]$. If node v_i is selected earlier than another v_j, then v_i has a higher priority than v_j for $i, j \in \{1, 2, \dots, n\}$. Thus, π corresponds to a priority-ordered node list $[v_{\pi_1}, v_{\pi_2}, \dots, v_{\pi_n}]$. In the following, we omit superscript k for h_i, as it is fixed in the decoder.

1) SEQUENTIAL NODE SELECTION

As formulated in Eq. (5), we decompose the probability function of a priority order into a product of probabilities of node selection in sequence. The decoder chooses a node to have the highest priority by sampling from probability distribution $p_\theta(\pi_1|\tau)$. It then chooses another node with the next highest priority (i.e., the highest priority among nodes not chosen yet) by sampling from $p_\theta(\pi_2|\pi_1, \tau)$. For each selection at step t, the embedding of a partial priority order $(\pi_1, \dots, \pi_{t-1}, \tau)$ is used to generate the respective conditioned probability. This procedure continues until no node is left, so it comprises n-iterative decoding.

It is observed that the order of selected nodes is not important when choosing a node v_t since those are indifferent in that they do not compete with v_t [11]. Thus, we maintain two partitions of nodes, a set of nodes already chosen O_t and a set of nodes that are not chosen yet L_t, and exploit the two sets (O_t, L_t) as context information for $t = 1, \dots, n$. The context information is used to calculate a probability distribution over nodes in L_t. In doing so, we aggregate O_t and L_t via nonlinear transformation and obtain their respective vectors $g_t^{(O)}$ and $g_t^{(L)}$ for steps $t = 1, \dots, n$,

$$g_t^{(O)} = \sigma_\theta \left(\sum\nolimits_{j \in O_t} \tanh \left(\mathbf{Aff}_\theta(h_j) \right) \right),$$
$$g_t^{(L)} = \sigma_\theta \left(\sum\nolimits_{j \in L_t} \tanh \left(\mathbf{Aff}_\theta \left(h_j\right) \right) \right) \quad (16)$$

where $\sigma(\cdot)$ is an arbitrary nonlinear activation function.

We also consider the previously selected node $v_{\pi_{t-1}}$ as an important factor to determine its next node and thus make use of its embedding $h_{\pi_{t-1}}$ when making the successive selections. This is consistent with common observation such that fixed-priority task scheduling heuristics share the same pattern that similar priorities are assigned for tasks of similar properties.

$$g_t^{(\text{last})} = \sigma_\theta(h_{\pi_{t-1}}) \quad (17)$$

Given the three vectors for the partitioned set and the previously selected node in Eq. (16) and Eq. (17), we combine them into a context vector by

$$c_t = \mathbf{Aff}_\theta(g_t^{(O)} + g_t^{(L)} + g_t^{(\text{last})}). \quad (18)$$

Then, we exploit the context vector c_t to derive a probabilistic inference for the tth subsequent node selection.

$$\begin{aligned} & p_\theta(\pi_t|\pi_1, \dots, \pi_{t-1}, \tau) \\ & \quad = \mathbf{Softmax}(C \cdot \tanh(h_i\, W_\theta\, \mathbf{Att}_\theta(c_t, \{h_j | v_j \in L_t\}))) \\ & \qquad \text{for all } v_i \in L_t. \end{aligned} \quad (19)$$

We multiply the output of $\tanh(\cdot)$ by an inverse temperature C to confine logits within the range $[-C, C]$, where C is empirically chosen. This derives the probability of priority orders.

$$p_\theta(\pi|\tau) = \prod_{t=1}^{n} p_\theta(\pi_t|\pi_1, \dots, \pi_{t-1}, \tau) \quad (20)$$

2) SAMPLING PRIORITY FROM THE DISTRIBUTION

Here we describe our strategy for sampling π_t from the distribution at step t in Eq. (19). We can conduct successive selection in a greedy way using

$$\pi_t = \mathbf{argmax} \left(p_\theta(\pi_t|\pi_1, \dots, \pi_{t-1}, \tau) \right). \quad (21)$$

It is also possible to use stochastic sampling that randomly draws π_t according to the distribution. Note that there are other sampling strategies than those simple approaches, e.g., $A*$ [30], Beam Search [31], which are computationally expensive and thus are not appropriate for large-scale problem settings.

C. MDP FORMULATION

Here, we discuss the formulation of DAG task scheduling problems from the perspective of a Markov decision process (MDP). In general, an MDP is specified by a tuple with a set of states, a set of actions, a reward function, a state transition probability, and a discount factor.

- **State**. For scheduling a DAG task through sequential selection, a state needs to include information about the subtask partitions that evolve over time (i.e., a set of priority-assigned subtasks O_t and a set of the others or priority-unassigned subtasks L_t as in Eq. (16)). As step t goes on, O_t starts with an empty set and expands to a set containing more subtasks. More formally, according to the probability of priority orders in Eq. (20), a state consists of the embeddings about a DAG task τ with n subtasks (i.e., $h_1, h_2, \dots, h_n$) and priority-assigned subtasks; a state at step t is given as

$$\left((h_1, \dots, h_n), [\pi_1, \pi_2, \dots, \pi_{t-1}] \right). \quad (22)$$

Recall that the embeddings for τ are calculated by the encoder in Section IV-A, and our model yields an n-sized priority order $\pi = [\pi_1, \pi_2, \dots, \pi_n]$. Through sequential selection on the subtasks by the decoder, at each step $t \leq n$, a partial permutation $[\pi_1, \pi_2, \dots, \pi_{t-1}]$ is generated. This corresponds to the indices of priority-assigned subtasks and is used as part of the state.

- **Action**. Upon a state in Eq. (22), the decoder calculates an action to select π_t (i.e., assigning the tth highest priority to the π_tth subtask in τ) at step t. Accordingly, a set of actions corresponds to the distinct integers from 1 to n, and a set of states corresponds to a partial permutation of the subtask embeddings.
- **Reward**. A reward is yielded according to a given objective of scheduling, i.e., minimizing the makespan for a DAG task in Eq. (4). We specifically use the slowdown metric (in Eq. (26)) to evaluate the advantage of specific priority orders over others. The details of our reward design subject to DRL training are described in Section IV-E2.
- **Transition probability**. In this MDP formulation, a transition is assumed to be deterministic in that for a state and action, the next state is determined without randomness. It is because an action for scheduling does not execute a task and it only affects the scheduling strategy and changes a permutation (a priority order) for a task. This setting is common when adopting DRL for large scale combinatorial optimization problems [20].
- **Discount factor**. Given the DAG task specification of finite n subtasks, we have *episodic* DRL with terminal states where a full permutation is obtained, and accordingly, we set the discount factor to be 1.

D. COMPLEXITY ANALYSIS

In the encoder, K iterations of encoding via self-attention with masking for DAG precedence conditions are performed, where K denotes the number of GCN layers. The self-attention requires $O(n^2d + nd^2)$ scalar multiplications for a single input of n subtasks and embedding dimension d. Then, the encoding complexity is $O(K \times (n^2d + nd^2))$. In the decoder, for each selection (each time-step), context information for a partitioned set $\{O_t, L_t\}$ in Eq. (16) and a subtask previously chosen is calculated in $O(nd^2)$. This is iteratively performed n times for n subtasks, requiring $O(n^2d^2)$.

Therefore, the complexity to infer a priority order for a DAG task input is $O(n^2d^2)$. Note that the number of GCN layers K is much smaller than the number of subtasks n (i.e., as in Table 5) and the embedding dimension d ($d = 64$ in our implementation).

E. DRL TRAINING

As previously explained, the priority assignment for DAG task scheduling is formulated in a probability distribution in Eq. (20), which represents a policy in DRL. In the following, we describe how to establish such a policy through a policy gradient method.

1) LEARNING OBJECTIVE

The objective function J to learn Eq. (20) is defined as

$$J = \mathbb{E}_{\pi\sim p_\theta(\tau)}\left[S(\pi,\tau)\right] = \sum_{\pi} p_\theta(\pi|\tau)S(\pi,\tau) \tag{23}$$

where $\pi \sim p_\theta(\tau)$ specifies that the priority order is sampled from a learned policy, and $S(\pi, \tau)$ denotes score values which will be explained in Section IV-E2. We update model parameters θ by the policy gradient in that differentiating the objective in Eq. (23) derives a gradient update rule as

$$\begin{aligned}\nabla_\theta J &= \sum_{\pi} \nabla_\theta p_\theta(\pi|\tau)S(\pi,\tau)\\ &= \sum_{\pi} p_\theta(\pi|\tau)\nabla_\theta \log(p_\theta(\pi|\tau))S(\pi,\tau)\\ &= \mathbb{E}_{\pi\sim p_\theta}\left[\nabla_\theta \log\left(p_\theta(\pi|\tau)\right) S\left(\pi,\tau\right)\right]\end{aligned} \tag{24}$$

using $\nabla_x f(x) = f(x)\nabla_x \log f(x)$. Specifically, we use the Monte-Carlo stochastic gradient descent method or REINFORCE algorithm [24] to estimate the gradient in Eq. (24) and its average value over a batch of tasks,

$$\nabla_\theta J = \frac{1}{|\mathcal{B}|}\sum_{\tau\in\mathcal{B}} p_\theta(\pi|\tau)\nabla_\theta \log(p_\theta(\pi|\tau))S(\pi,\tau) \tag{25}$$

where $\mathcal{B}$ is a batch list of tasks randomly sampled from a training dataset.

2) SLOWDOWN-BASED REWARD

As part of the objective function in Eq. (23), the score $S(\pi, \tau)$ is used to formulate the relevance of priority orders π for task τ. This score is calculated based on reward values [32] in DRL. Given a DAG representation for parallel task scheduling on a multiprocessor system, our model is intended to learn an optimal priority order for τ in terms of the makespan of τ. The makespan specifies the required time that elapses from the execution of τ's source node to its sink node. Specifically, we implement the slowdown-based reward (score) function $S(\pi, \tau)$ using normalized makespans for scheduling a task τ with a priority order π, i.e.,

$$S(\pi,\tau) = \frac{-M(\pi,\tau)}{M_{\mathrm{LB}}(\tau)} \tag{26}$$

where $M(\pi, \tau)$ denotes the makespan of τ by π and $M_{\mathrm{LB}}(\tau)$ denotes the lower bound of the makespan of τ by any priority order. Note that $\frac{M(\pi,\tau)}{M_{\mathrm{LB}}(\tau)}$ represents a slowdown of τ's makespan by π compared to the ideal case for τ. Since the smaller the slowdown or the makespan, the better the priority order, we define negative rewards based on slowdowns.

The lower bound of the makespan of τ upon an m-processor system is calculated by

$$M_{\mathrm{LB}}(\tau) = L + \max\left(0, \left\lceil \frac{1}{m}(W - L\cdot m)\right\rceil\right) \tag{27}$$

where L and W are the critical workload and the total workload of τ.

Claim: The makespan of a task τ is lower bounded by Eq. (27)

Proof: According to the definition of the critical path of a DAG task τ, the task requires at least the critical workload L in time to execute. During L, the m-processor system can process at most $L \cdot m$ for τ's nodes (in the case when no dependency blocks processing). In that ideal case, if τ's total workload W is no larger than $L \cdot m$, then τ can be completed within L. Otherwise, there is at least $(W - L \cdot m)$ workload not yet processed until L. It requires at least $\left\lceil \frac{1}{m}(W - L \cdot m) \right\rceil$ in time. Summing these two components establishes Eq. (27). □

Reward normalization is intended to prevent overweighing of rewards of tasks with high parallelism. Suppose that we have an optimal order π^* and another order π of lower quality for a task τ. It is observed that the less the parallelism degree of τ, the smaller the makespan difference $M(\pi^*, \tau) - M(\pi, \tau)$. At one extreme, in the case where τ has no parallelism (e.g., there is only a single path from source to sink), every priority order yields the same makespan (reward). This case is not very meaningful for our model training. For establishing the stability of model training, it is critical to define rewards to be fully dependent on the quality of priority orders, especially when the structure of DAG tasks is complex.

3) BASELINE REDUCTION

In model training, we employ an additional variance reduction technique with baseline [32]. Specifically, we exploit a greedy baseline method, similar to [21], in which a target model p_θ and another base model p_β are used. The two models share the same neural network structure but have distinct θ and β parameters. For a task τ, suppose that we obtain priority orders π and b where the former is sampled from p_θ with stochastic decoding and the latter is sampled from p_β. We then obtain a reward by b, i.e., $B(b, \tau) = \frac{-M(b,\tau)}{M_{\text{LB}}(\tau)}$ in Eq. (26). By replacing $S(\pi, \tau)$ in Eq. (24) with $S(\pi, \tau) - B(b, \tau)$, we adopt baseline reduction and hence establish the below.

$$\begin{aligned} \nabla_\theta J &= \mathbb{E}_{\pi \sim p_\theta} \left[\nabla_\theta p_\theta(\pi|\tau) \left[S(\pi, \tau) - B(b, \tau) \right] \right] \\ &= \frac{1}{|\mathcal{B}|} \sum_{\tau \in \mathcal{B}} \left[\nabla_\theta \log(p_\theta(\pi|\tau)) \frac{[M(b, \tau) - M(\pi, \tau)]}{M_{\text{LB}}(\tau)} \right] \\ & \mathbb{E}_{\pi \sim p_\theta} \left[\nabla_\theta p_\theta(\pi|\tau) B \right] = \nabla_\theta \mathbb{E}_{\pi \sim p_\theta} [B] = \nabla_\theta [B] = 0 \end{aligned} \tag{28}$$

We update the base model p_β, if a makespan calculated by p_β's priority order is statistically different from that calculated by p_θ's priority order. We perform the paired t-test on the makespans from the two models to check their difference, e.g., they are different if p-value is smaller than 0.01.

Our model training scheme is summarized in Algorithm 1.

V. EVALUATIONS

In this section, we evaluate the performance of our model.

A. EXPERIMENTAL SETTINGS

We describe the experimental settings including data generation, evaluation metrics, and the models in comparison.

Algorithm 1 Model Training of GoSu

```
// Parameter initialization
Initialize the target model $p_\theta$ with parameters $\theta$
Initialize the baseline model $p_\beta$ with parameters $\beta$
Initialize $\beta \leftarrow \theta$
// Model learning procedure
for $i \leftarrow 1, \ldots, N_{\text{train}}$ do
    $\Delta\theta \leftarrow 0$ , Sample batch $\mathcal{B}$ from training dataset $\mathcal{D}$
    for task $\tau$ in $\mathcal{B}$ do
        Sample priority order $\pi$ from $p_\theta(\cdot|\tau)$ stochastically
        Sample priority order $b$ from $p_\beta(\cdot|\tau)$ greedily
        $\Delta\theta \leftarrow \Delta\theta + \frac{1}{|\mathcal{B}|} \frac{[M(b,\tau) - M(\pi,\tau)]}{M_{\text{LB}}(\tau)} \nabla_\theta \log p_\theta(\pi|\tau)$
    end for
    // If target is better than baseline
    if paired-T Test on $M(\pi, \tau) \neq M(b, \tau)$ then
        // Update baseline
        Update $\beta \leftarrow \theta$
    end if
    Update $\theta$ with $\Delta\theta$ using Adam Optimizer
end for
```

TABLE 4. Parameters for task dataset generation with different degrees of parallelism: Unif() denotes uniform distribution.

Dataset	n_{depth}	n_{child}	p_{fork}	p_{pert}
Low	$\mathbf{Unif}(2,5)$	$\mathbf{Unif}(2,5)$	0.5	0.05
Moderate	$\mathbf{Unif}(3,5)$	$\mathbf{Unif}(2,5)$	0.5	0.1
High	$\mathbf{Unif}(3,6)$	$\mathbf{Unif}(2,6)$	0.5	0.15

1) DATASET GENERATION

As there is no publicly available large-scale DAG task datasets with a variety of configurations in real-time task specifications, we use synthetic datasets in which each DAG task is generated according to a nested fork-join task model [33]. This is a widely adopted scheme for analyzing and generating DAG tasks [34], [35]. The task generation algorithm works as follows, similar to [35]. For each node v_i in the layer k, its child nodes v_j and edges e_{ij} are generated based on *fork* probability p_{fork} where the number of child nodes in layer $k+1$ are determined by uniform distribution n_{child}. This procedure starts from a source node and repeats for n_{depth} times, thus creating a DAG of n_{depth} layers. In addition, the edges of a node pair between the layer k and the layer $k+1$ are randomly added in the DAG based on *perturbation* probability p_{pert}. A large perturbation probability leads to a high degree of parallelism. Finally, the edges from nodes in the last layer to the sink node are added.

To perform experiments in various task configurations, we create three datasets with varying degrees of parallelism: *Low*, *Moderate* and *High*. The degree is configured by the aforementioned parameters such as fork probability p_{fork}, perturbation probability p_{pert}, the number of children n_{child}, and depth limit n_{depth}, as shown in Table 4. The characteristics of the datasets are summarized in Table 5.

TABLE 5. Characteristics of task datasets: # nodes and # edges denote the average number of nodes and edges. *L* and *W* denote the average workload of critical paths and the average total workload of DAG tasks.

Dataset	# nodes	# edges	L	W
Low	26.05	75.70	3284.23	10000
Moderate	44.80	129.47	1798.21	10000
High	83.70	339.75	1610.86	10000

2) IMPLEMENTATION

For evaluation, we implement a task scheduling simulator by which the makespan of each DAG task by a given priority order is exactly calculated upon a system of m processors. For implementing the dataset generation module and the models in comparison, we also exploit the open-source implementation provided in [1].[1] Our implementation is based on Python 3.7.9, PyTorch 1.6.0 [36], and PyTorch-geometric [37]. We train and test the models on a system of an Intel(R) Core(TM) i9-9940X processor with 160G memory, and an NVIDIA Tesla V100 GPU with CUDA 10.1. and cuDNN 7.6.0. In addition, we implement the heuristic algorithms and schedulability tests using Cython [38].

As for a multiprocessor system where DAG tasks are scheduled to run, we set its configuration, such as the number of homogeneous processors m, to be data specific and subsumed in training datasets. That is, for a model learned on specific datasets, each DAG task sampled from the datasets is configured to run upon a system of m processors during model training, and its evaluation system environment is set to have m processors. Thus, each model is evaluated in the same system environment with m processors (e.g., $m \in \{2, 3, 4, 6, 8\}$ in Figure 4) which it has been trained on.

The models in comparison are two state-of-the-art DAG task scheduling algorithms: **He19** [2] and **Zhao20** [1].

As described previously, our model aims at minimizing the makespan of individual DAG tasks. Accordingly, we measure the model performance in the slowdown ratio of an achieved makespan to its respective ideal makespan in Eq. (27) and use it as the evaluation metric.

3) MODEL HYPERPARAMETERS

The hyperparameter settings for our model are shown in Table 6. Unless otherwise mentioned, all the hyperparameters are set the same for all experiments.

We generate $10K$ datasets for each configuration. We use $8K$ samples for model training, $1K$ samples for model validation, and $1K$ samples for evaluation. In the encoder, we set the number of graph convolution layers to $K = 2$. For each graph convolution layer, we set the number of heads to 4 where two of them are forward-path Attention modules and the others are inverse-path Attention modules. We set the hidden representation layer to 64. We exploit dropout [39] with probability $p = 0.1$. We set the inverse temperature C to 5. A larger value of C makes models less exploitative. We set the batch size to 128 and use Adam Optimizer [40]

[1]https://github.com/automaticdai/research-dag-scheduling-analysis

TABLE 6. Hyperparameter settings in datasets, neural network implementation, and training for our `GoSu` model.

	Description	Values
Dataset	Num. of train samples	8000
	Num. of validation samples	1000
	Num. of test samples	1000
Network	Num. of GCN layers K	2
	Num. of Attention heads H	4
	Embedding dimension d	64
	Parameter initialization	$\mathbf{Unif}\left(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\right)$
	Dropout probability p	0.1
	Inverse temperature C	5
Training	Batch size $\vert\mathcal{B}\vert$	128
	Learning rate	10^{-4}
	Epoch	100
	Optimizer	Adam
	Gradient Clipping	$(-1, 1)$

where the learning rate sets to 0.0001. We clip gradients before model update by (−1, 1).

B. PERFORMANCE COMPARISON

For each experiment condition where a dataset is defined by a specific DAG task configuration and the number of processors, we compare the performance of our model with that of other models.

Figures 4(a), 4(b), and 4(c) show the average relative slowdown (i.e., $\frac{M(\pi,\tau)}{M_{\mathrm{LB}}(\tau)}$ in Eq. (26)) of achieved makespans with respect to various m processors ($m = \{2, 3, 4, 6, 8\}$) for low, moderate, and high parallelism datasets. As shown, our model achieves comparable performance for all the cases and outperforms He19 [2] and Zhao20 [1], with a relatively large margin of 2∼3% for the cases when $m = 3$ or $m = 4$ with the moderate parallelism datasets and when $m = 3 \sim 8$ with the high parallelism datasets. This result is consistent with our expectation on DRL-based scheduling approaches such that they learn priority assignment rules tailored for specific environment settings, achieving performance improvement compared to other heuristics upon complex conditions. Notice that the more capable a system with more processors is (e.g., larger m settings), the less performance impact a specific priority order is likely to have on less or moderate parallelism datasets. This is because most executable subtasks can run immediately regardless of their priority assigned on many available processors. Another extreme case is a single processor system where it necessarily takes the total workload W in time to complete a DAG task regardless of priority orders.

The cases where the number of processors is 3 or 4 normally correspond to nontrivial configurations in between the cases of highly capable multiprocessor systems and constrained single processor systems in our experimental settings and datasets, rendering our DRL-based model more effective and achieving better makespans and slowdowns.

Figures 5(a), 5(b), and 5(c) represent the performance in terms of the number of testing data samples for which `GoSu` performs better than another model. Note that considering that Zhao20 shows relatively better performance than He19

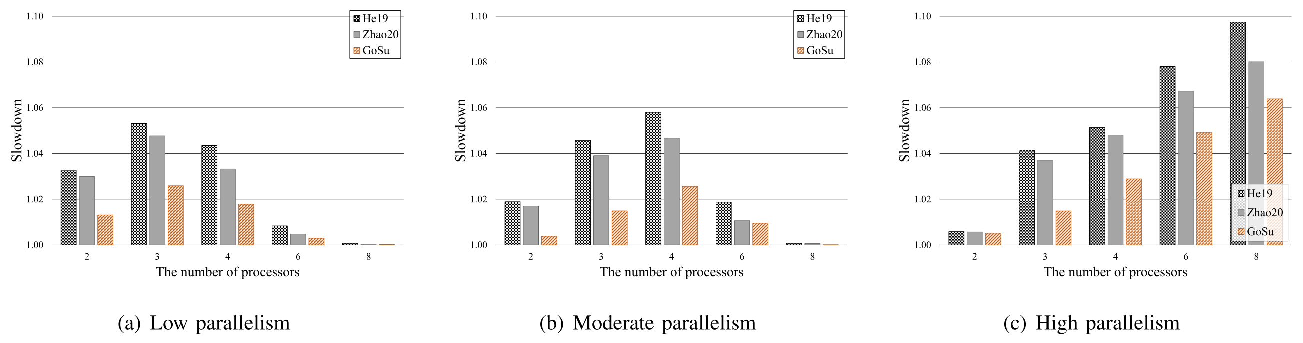


FIGURE 4. The performance in the slowdown of makespans by compared methods with respect to various system and task configurations: The Y-axis denotes the slowdown that represents the relative execution time compared to ideal execution time in Eq. (27). The lower the slowdown, the better the performance. The X-axis denotes the number of processors of a platform, representing the system configuration. The task configuration depends on the datasets of (a) low, (b) moderate, and (c) high parallelism in Table 5.

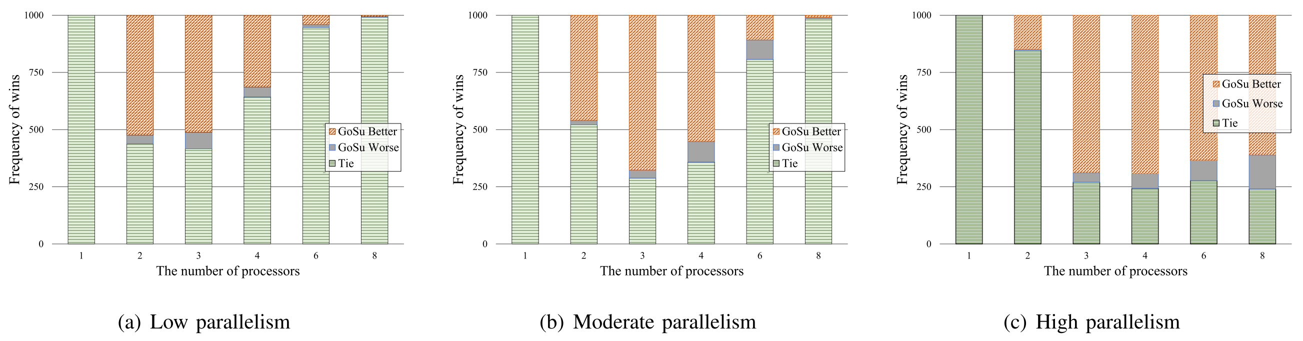


FIGURE 5. The performance comparison in terms of frequency of wins (the number of task instances for which each method yields the shortest makespan.) for Figure 4: The Y-axis represents the frequency of wins by a method. The pink-colored bar denotes the number of testing data samples for which `GoSu` outperforms another state-of-the-art method (Zhao20) on the whole testing dataset of 1000 samples. The gray-colored bar denotes the opposite case. The green-colored bar denotes the tie case. The length difference of the long pink-colored bars (`GoSu`) and short gray-colored bars (Zhao20) indicates the performance gain of `GoSu`. The X-axis denotes the number of processors of a platform, representing the system configuration. The system and task configurations are the same as those in Figure 4.

in our experiments, we include only the comparison with Zhao20. The data samples for each configuration are divided into three portions. The pink-colored bar indicates the portion of samples for which `GoSu`'s priority order outperforms (i.e., yielding a tighter makespan than) that of Zhao20 by at least 1% margin in terms of the slowdown in achieved makespans. The gray-colored bar indicates the portion of the opposite case samples. The green-colored bar indicates the portion of the other samples that tie.

It is consistently observed that `GoSu` performs more competitively in nontrivial configurations; the pink-colored bar increases when the number of processors is either 3 or 4, and its portion is up to 73% in the case when $m = 3$ with the moderate and high parallelism datasets. It is interesting that significant performance improvement is made with the systems of 2 and 3 processors for the low parallelism datasets and with the systems of a wider range of processor numbers, from 3 to 8, for the high parallelism datasets. This is because, for the low parallelism datasets, a system of more than 3 processors is likely to enable most executable tasks to run immediately and concurrently, thereby rendering specific priority orders less influential in terms of reducing makespans. For the high parallelism datasets, on the other hand, the same system has much opportunity to minimize makespans of individual tasks by priority orders that are appropriately chosen. This result demonstrates the benefit of `GoSu` performing steadily in various configurations including nontrivial cases. Through the paired-T tests, we also confirmed that except for the cases of $m = 8$ with the low and moderate parallelism datasets and $m = 2$ with the high parallelism datasets, our model shows statistically much better performance than the others.

C. COMPARISON OF GCN SCHEMES

In Table 7, we compare the models with different GCN schemes in terms of achieved slowdowns to verify the effect of our GCN processing with both forward-path and inverse-path aggregation. For comparison, GCNs are differently set to have only forward-path aggregation, only inverse-path aggregation, or both, while all the other model hyperparameters are set to the same.

For the low parallelism dataset, all the models show similar performance. However, as the complexity of parallel tasks grows, it is observed that the inverse-path aggregation becomes beneficial. For the moderate parallelism dataset, the model with only forward-path aggregation performs worse than the models with only inverse-path aggregation or both.

TABLE 7. The performance in the slowdown by the models with different GCN schemes: The number of processors is 4, and Both (GoSu) corresponds to the proposed GoSu model that has forward-path and inverse-path aggregation in GCN processing. The other models are set to consider either of two path types only.

	Low	Moderate	High
Forward-Path Only	1.016	1.033	1.043
Inverse-Path Only	1.017	1.025	1.034
Both (GoSu)	1.017	1.024	**1.029**

For the high parallelism dataset, the model with both performs better than the others. The inverse-path aggregation for node v allows the model to embed the information of nodes blocked by v to the representation of v via message passing. This enables a better scheduling policy for large-scale and highly parallel DAGs.

D. CHARACTERISTICS OF LEARNED POLICY

To inspect what kind of policy the GoSu model learns through DRL, we conduct a regression-based experiment by employing differential programming techniques [41], [42]. It turns out that the experiment results allow us to identify which properties the model learns to value more. Specifically, we construct a linear regression model to produce a pointwise score s_i for a node (subtask) v_i and then fit the regression model to render a score list $[s_1, s_2, \dots, s_n]$ consistent with the ranking of a priority order π generated by GoSu. That is, the linear model is fitted to $[s_{\pi_1}, s_{\pi_2}, \dots, s_{\pi_n}]$ which is correctly sorted in the descending order with π using Fast-Soft-Sort [41].

For simple analysis, we selectively use the following raw features of individual nodes as input: execution time, out-degree, in-degree, and whether a node is in the critical path (is-critical). A linear model $s_i = wx_i + b$ is learned where each element of w is set to be in [0, 1] and b in [0, 10] using clipping. We pose strong L1 regularization 3.0 on w to make the weights of irrelevant properties be zeros, thus having only weights of significance for scheduling decisions [43], [44].

Figure 6 shows the weight values w of the linear model with respect to different parallelism datasets. If an input feature has a large weight, it can be interpreted as an important factor for the priority assignment policy of GoSu. We observe that the out-degree is most critical for all task configurations. This is consistent with the effect of the inverse-path aggregation in Section V-C, showing that a larger out-degree means the node is more likely to be blocking other nodes and considered important. However, the in-degree does not play an important role, and its weight becomes zero in all task configurations. Another important observation is that the GoSu's policy varies depending on task configurations. As the parallelism increases, the weight of the is-critical feature increases while those of the out-degree and the execution time decrease. This indicates the adaptability of the DRL-based model for a variety of task configurations. Note that we view that the linear model correctly imitates GoSu since the linear model yields similar but slightly worse (about 6%) performance than GoSu due to its structural limitations and lack of features.

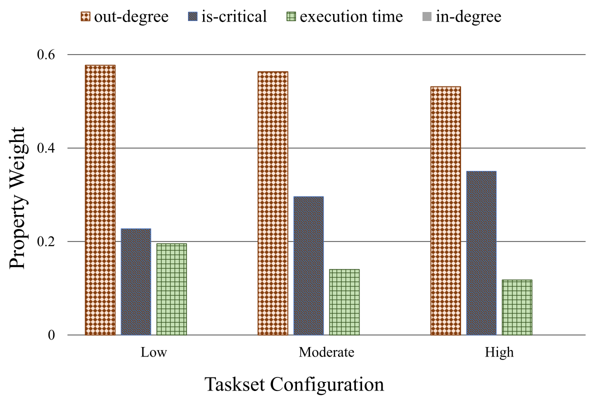


FIGURE 6. Characteristics of learned policy in GoSu: The Y-axis denotes the normalized weight of features in the linear model with respect to different task configurations on the X-axis. The higher property weights (e.g., $w \in [0, 1]$ of a linear model $wx_i + b$) of features can be interpreted as having a greater impact on scheduling. This significance of features varies across different task configurations. GoSu can learn an appropriate scheduling policy without exhaustive feature engineering.

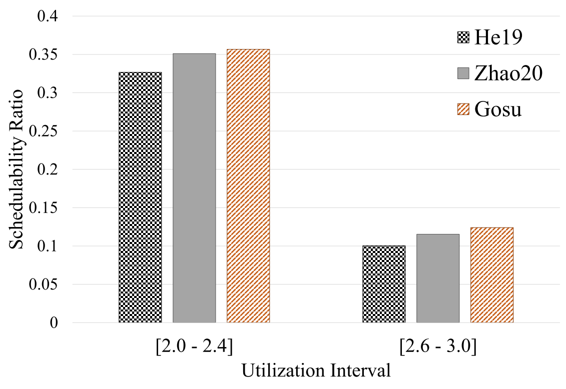


FIGURE 7. The performance comparison of multi-DAG scheduling: The Y-axis denotes the achieved schedulability ratio of testing task sets by three methods in comparison with respect to the utilization on the X-axis. The utilization interval denotes a total utilization range of a task set $\mathcal{T}$, i.e., $\sum_{\tau_i \in \mathcal{T}} \frac{C_i}{T_i}$ where C_i and T_i are the WCET and period of a task τ_i, respectively. A higher schedulability ratio means more task sets of DAG tasks are scheduled, indicating better performance.

E. SCHEDULING MULTI-DAG TASKS

Here, we discuss how to extend our approach for scheduling multi-DAG tasks [35]. Given a periodic DAG task τ with deadline D and period T, we calculate the makespan of τ using the priority order (i.e., π^* in Eq. (4)) produced by our model and set it as τ's WCET. In this way, we induce WCET estimates of multi-DAG tasks. Then, it is possible to adopt priority assignment methods in real-time task scheduling (e.g., deadline monotonic (DM)) to schedule multi-DAG tasks.

In Figure 7, we shows the performance in schedulability ratio of multi-DAG tasks by three methods calculating WCETs using different single DAG scheduling approaches. We adopt the same DM algorithm and the same schedulability test in [1] for the three methods. The schedulability ratio represents ratio between the number of schedulable task sets and the number of tested task sets, where each task set consists of 12 individual DAG tasks in this experiment. We set the number of processors to 4 and test 5000 task sets (samples). As shown, GoSu achieves better performance in schedulability ratio than the other methods. This result implies that tighter makespan bounds induced by GoSu can lead to better performance in scheduling multi-tasks. The performance gain is relatively larger when the utilization is high. This is consistent with the benefits of GoSu particularly for nontrivial cases, as the same scheduling and schedulability test methods are used.

VI. CONCLUSION

In this work, we presented GoSu, a DRL-based priority assignment model for DAG task scheduling, which adapts graph learning to utilize the graph structure of a DAG task. On graph embedding results, our model performs a sequential decoding procedure to obtain a permutation for subtasks in a DAG task. That permutation represents a priority order for task scheduling to minimize the makespan of the DAG task. Through extensive experiments, we demonstrated that GoSu achieves robust performance in the slowdown of DAG tasks, compared to other state-of-the-art priority assignment heuristics. We also showed the adaptability of our DRL-based approach for various configurations by leveraging the rank regression mimicking the policy of GoSu.

The direction of our future work is to develop an integrated learning approach of hierarchical DRL and GCN techniques for large-scale virtual application management problems. As network service chains consist of many virtual network functions, they can be analyzed through GCNs to be mapped on underlying network infrastructures with heterogeneous resources in data centers. While exploiting the structural similarity in the use of graph representation learning between DAG task scheduling and virtual network mapping, the latter has scalability issues on underlying large networks. Hierarchical learning techniques in DRL can be explored for those issues.

REFERENCES

[1] S. Zhao, X. Dai, I. Bate, A. Burns, and W. Chang, "DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2020, pp. 128–140.

[2] Q. He, X. Jiang, N. Guan, and Z. Guo, "Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2283–2295, Oct. 2019.

[3] F. Guan, J. Qiao, and Y. Han, "DAG-fluid: A real-time scheduling algorithm for DAGs," *IEEE Trans. Comput.*, vol. 70, no. 3, pp. 471–482, Mar. 2021.

[4] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns, "A review of priority assignment in real-time systems," *J. Syst. Archit.*, vol. 65, pp. 64–82, Apr. 2016.

[5] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global EDF scheduling for parallel real-time tasks," *Real-Time Syst.*, vol. 51, no. 4, pp. 395–439, Jul. 2015.

[6] R. Pathan, P. Voudouris, and P. Stenström, "Scheduling parallel real-time recurrent tasks on multicore platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 915–928, Apr. 2018.

[7] Y. Nazarathy and G. Weiss, "A fluid approach to job shop scheduling: Theory, software and experimentation," *J. Scheduling*, vol. 13, pp. 509–529, Nov. 2009.

[8] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, Nov. 2016, pp. 50–56.

[9] L. Wang, Q. Weng, W. Wang, C. Chen, and B. Li, "Metis: Learning to schedule long-running applications in shared container clusters at scale," in *Proc. SC Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–17.

[10] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 270–288.

[11] H. Lee, J. Lee, I. Yeom, and H. Woo, "Panda: Reinforcement learning-based priority assignment for multi-processor real-time scheduling," *IEEE Access*, vol. 8, pp. 185570–185583, 2020.

[12] J. Kwon, K.-W. Kim, S. Paik, J. Lee, and C.-G. Lee, "Multicore scheduling of parallel real-time tasks with multiple parallelization options," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2015, pp. 232–244.

[13] P. Chen, W. Liu, X. Jiang, Q. He, and N. Guan, "Timing-anomaly free dynamic scheduling of conditional DAG tasks on multi-core systems," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, pp. 1–19, Oct. 2019.

[14] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of DAGs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3242–3252, Dec. 2014.

[15] S. Chilukuri and D. Pesch, "RECCE: Deep reinforcement learning for joint routing and scheduling in time-constrained wireless networks," *IEEE Access*, vol. 9, pp. 132053–132063, 2021.

[16] S. Sheng, P. Chen, Z. Chen, L. Wu, and Y. Yao, "Deep reinforcement learning-based task scheduling in IoT edge computing," *Sensors*, vol. 21, no. 5, p. 1666, Feb. 2021.

[17] L. Zhou, L. Zhang, and B. K. P. Horn, "Deep reinforcement learning-based dynamic scheduling in smart manufacturing," *Proc. CIRP*, vol. 93, pp. 383–388, Jan. 2020.

[18] J. Park, J. Chun, S. Kim, Y. Kim, and J. Park, "Learning to schedule job-shop problems: Representation and policy learning using graph neural network and reinforcement learning," *Int. J. Prod. Res.*, vol. 59, pp. 1–18, Jan. 2021.

[19] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Proc. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2015, pp. 2692–2700.

[20] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," 2016, *arXiv:1611.09940*.

[21] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2019.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[23] J. Fonseca, G. Nelissen, and V. Nélis, "Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling," *Real-Time Syst.*, vol. 55, no. 2, pp. 387–432, Apr. 2019.

[24] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 229–256, 1992.

[25] N. C. Audsley, "On priority assignment in fixed priority scheduling," *Inf. Process. Lett.*, vol. 79, no. 1, pp. 39–44, May 2001.

[26] W. L. Hamilton, "Graph representation learning," in *Synthesis Lectures on Artificial Intelligence and Machine Learning*. San Rafael, CA, USA: Morgan & Claypool, 2020.

[27] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–15.

[28] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2018, pp. 1–12.

[29] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (ELUs)," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–14.

[30] S. J. Russell and P. Norvig, *Artificial Intelligence—A Modern Approach*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.

[31] V. Steinbiss, B. Tran, and H. Ney, "Improvements in beam search," in *Proc. Conf. Spoken Lang. Process.*, 1994, pp. 1–4.

[32] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1998.

[33] L. Nyman and M. Laakso, ''Notes on the history of fork and join,'' *IEEE Ann. Hist. Comput.*, vol. 38, no. 3, pp. 84–87, Jul. 2016.

[34] B. Peng, N. Fisher, and M. Bertogna, ''Explicit preemption placement for real-time conditional code,'' in *Proc. 26th Euromicro Conf. Real-Time Syst.*, Jul. 2014, pp. 177–188.

[35] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, ''Response-time analysis of conditional DAG tasks in multiprocessor systems,'' in *Proc. 27th Euromicro Conf. Real-Time Syst.*, Jul. 2015, pp. 211–221.

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, and A. Desmaison, ''PyTorch: An imperative style, high-performance deep learning library,'' in *Proc. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2019, pp. 8024–8035.

[37] M. Fey and J. E. Lenssen, ''Fast graph representation learning with PyTorch geometric,'' 2019, *arXiv:1903.02428*.

[38] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, ''Cython: The best of both worlds,'' *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 31–39, 2011, doi: 10.1109/MCSE.2010.118.

[39] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, ''Improving neural networks by preventing co-adaptation of feature detectors,'' 2012, *arXiv:1207.0580*.

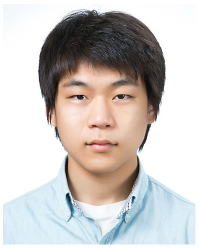
[40] D. P. Kingma and J. Ba, ''Adam: A method for stochastic optimization,'' in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–15.

[41] M. Blondel, O. Teboul, Q. Berthet, and J. Djolonga, ''Fast differentiable sorting and ranking,'' in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2020, pp. 950–959.

[42] A. Grover, E. Wang, A. Zweig, and S. Ermon, ''Stochastic optimization of sorting networks via continuous relaxations,'' in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2019.

[43] R. A. Johnson and D. W. Wichern, *Applied Multivariate Statistical Analysis*. London, U.K.: Pearson, 2002.

[44] C. M. Bishop, ''Pattern recognition and machine learning,'' in *Information Science and Statistics*, 5th ed. New York, NY, USA: Springer, 2007.



HYUNSUNG LEE was born in Seoul, South Korea. He received the B.S. degree in computer engineering from Sungkyunkwan University, Suwon, in 2019, where he is currently pursuing the master's degree in electrical and computer engineering. His research interests include recommendation systems, cluster orchestration, and reinforcement learning.

SANGWOO CHO is currently pursuing the bachelor's degree with the Department of Mathematics, Sungkyunkwan University, Suwon, South Korea. His research interests include multi armed bandit, theoretical machine learning, and differential programming.

YEONGJAE JANG received the B.S. degree in mathematics from Sungkyunkwan University, Suwon, South Korea, in 2017, where he is currently pursuing the degree with the Department of Mathematics. Since April 2021, he has been working as a Data Engineer. His research interests include theoretical machine learning and service optimization.

JINKYU LEE (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea, in 2004, 2006, and 2011, respectively. In 2014, he joined the Department of Computer Science and Engineering, Sungkyunkwan University (SKKU), Republic of Korea, where he is currently an Associate Professor. He has been a Visiting Scholar/Research Fellow with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA, from 2011 to 2014. His research interests include system design and analysis with timing guarantees, QoS support, and resource management in real-time embedded systems, mobile systems, and cyber-physical systems. He won the Best Student Paper Award from the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), in 2011, and the Best Paper Award from the 33rd IEEE Real-Time Systems Symposium (RTSS), in 2012.

HONGUK WOO (Member, IEEE) was born in Seoul, South Korea. He received the B.S. degree in computer science from Korea University, Seoul, in 1995, and the M.S. and Ph.D. degrees in computer sciences from The University of Texas at Austin, Austin, TX, USA, in 2002 and 2008, respectively.

From 2008 to 2018, he worked at Samsung Research, Samsung Electronics, as a Principal Engineer and the Vice President. Since 2018, he has been an Assistant Professor with the Department of Computer Science and Engineering, Sungkyunkwan University, Suwon, South Korea. His research interests include intelligent application, data-driven monitoring, cloud computing, and networked cyber-physical systems.