Actor-Twin Framework for Task Graph Scheduling

Narjes Nourzad
University of Southern California
Los Angeles, USA
nourzad@usc.edu

Jared Coleman
Loyola Marymount University
Los Angeles, USA
jaredcol@usc.edu

Zhongyuan Zhao
Rice University
Houston, USA
zhongyuan.zhao@rice.edu

Bhaskar Krishnamachari
University of Southern California
Los Angeles, USA
bkrishna@usc.edu

Gunjan Verma
US Army's DEVCOM Army Research Laboratory
Adelphi, Maryland, USA
gunjan.verma.civ@army.mil

Santiago Segarra
Rice University
Houston, USA
segarra@rice.edu

ABSTRACT

Task graph scheduling involves efficiently assigning computational tasks to available processors while ensuring the correctness of the result. As this problem is NP-hard and not polynomial-time approximable, traditional scheduling relies on heuristics. Although these methods can be effective, they often lack efficiency and fall short in generalizing well across different graph sizes and structures. Moreover, they are incompatible with optimization techniques that rely on backpropagation, limiting their adaptability to modern gradient-based approaches. In this paper, we present a novel *Actor-Twin* framework that integrates Multi-Branch Graph Convolutional Networks (MB-GCNs) with an Actor-Critic approach to overcome the non-differentiable nature of heuristic-based scheduling. The heart of our framework is the Actor-Twin Scheduler (*ACTS*) module, which generates a task score via the MB-GCN *actor* that is subsequently used by a heuristic for scheduling. To facilitate gradient-based training of the actor, we incorporate a differentiable *twin* component that approximates heuristic decisions. We also introduce a systematic graph representation for task-server assignments that is compatible with gradient-based optimization. Experimental results show that *Actor-Twin* consistently outperforms traditional heuristic scheduling approaches in both average and variance of makespan.

KEYWORDS

Reinforcement Learning, Graph Neural Networks, Task Scheduling

1 INTRODUCTION

Task graph scheduling, though extensively studied [10, 33, 50], remains a challenge in parallel and distributed computing systems, particularly within modern distributed analytics pipelines [15]. It involves assigning tasks to computational resources to optimize performance metrics such as total scheduling time, energy consumption, or throughput [14] while ensuring the correctness of the result. At the same time, both scheduling overhead and schedule quality are important in task graph execution. A scheduler that incurs excessive computation overhead, even if it produces high-quality schedules, is impractical. Many high-quality solvers are not viable for task graph scheduling due to their runtime. Given that this problem is NP-hard [20] and not polynomial-time approximable [7], traditional scheduling methods often rely on heuristic algorithms such as list scheduling. While these methods are valued for their simplicity, low computational complexity, and ease of implementation, they often fall short in generalizing across different scenarios. For example, they struggle with task graphs that have varying dependency patterns and server networks with diverse connectivity [8].

Graph Neural Networks (GNNs), particularly Graph Convolutional Networks (GCNs) [30], have advanced graph-based problem-solving by effectively capturing relational patterns within graph data. While GNNs generalize well across varying graph sizes and structures, they face two major challenges when applied to task scheduling. First, the NP-hard nature of data labeling restricts the use of supervised learning methods [58]. To eliminate the reliance on labeled data, reinforcement learning (RL) provides an alternative [37]. Second, compared to rule-based heuristics, GNNs struggle to directly enforce the hard constraints in task scheduling [21]. Khalil et al. [28] address this by using GNNs to guide the sequential decision-making within a heuristic under the Q-learning framework [54]. However, invoking GNNs in each iteration incurs significant overhead, defeating the very goal of accelerating task execution. Alternatively, GNNs can refine the inputs of a fast heuristic to optimize its final outputs at the cost of minimal overheads. Yet, for heuristics like Heterogeneous Earliest Finish Time (HEFT) scheduling [49], which make discrete batch decisions, direct gradient-based optimization becomes infeasible due to their non-differentiability.

To address these problems in one comprehensive solution, inspired by Zhao et al.'s [61] work, we propose the *Actor-Twin* framework (Figure 1). The core concept of our novel framework revolves around generating a task score, which is the primary function of the *GCN actor* within our Actor-Twin Scheduler (*ACTS*) module. To avoid embedding GCNs into the iterations of the heuristic, we use them to adjust the task score before calling a heuristic, which then uses this score to schedule tasks, producing an optimal or near-optimal scheduling solution. This allows GCNs to influence the final output while respecting the constraints. To better characterize the structural properties of the scheduling problem, we introduce a graph-based representation that explicitly models the relationship between tasks and servers, accounts for dependencies and communication costs, and adapts to dynamic server networks.

We construct a Directed Acyclic Graph (DAG) that integrates two distinct graphs: the task graph, representing task dependencies,

Proc. of the Adaptive and Learning Agents Workshop (ALA 2025), Avalos, Aydeniz, Müller, Mohammedalamen (eds.), May 19 – 20, 2025, Detroit, Michigan, USA, ala-workshop.github.io. 2025.

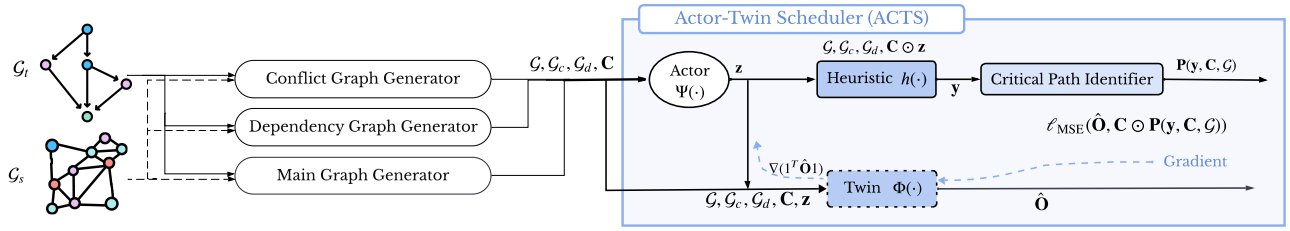


Figure 1: Proposed *Actor-Twin* Architecture. The process begins with generating the Conflict, Dependency, and Main graphs to represent task-server relationships. The actor computes task prioritization scores, while the twin network approximates expected outcomes. Together, they enable gradient-based optimization to improve scheduling performance.

and the server network graph, representing computational capabilities and communication links. Each node in this DAG represents a task-server pair, allowing us to make scheduling decisions while enforcing constraints such as execution conflicts and resource dependency. Compared to graph models in previous work [13, 14, 29], a key advantage of this representation is that it empowers GNNs to generalize to dynamic server networks with varying number of servers and dynamic connectivity topologies, without retraining. This adaptability extends our approach to mobile ad-hoc cloud and edge computing environments [57, 59], as well as real-world cloud facilities where server availability is shaped by capacity reservations, failures, upgrades, and maintenance.

In addition, to process this structured representation, we propose to model the actor using a *Multi-Branch Graph Convolutional Network* (MB-GCN) instead of a single GCN. This novel design follows modular GCN principles and multi-graph processing, where different aspects of the problem are processed separately before being merged [6, 17]. Similar to Heterogeneous GNNs [60], which assign distinct layers to different graph modalities, and Relational GNNs [38], which apply independent transformations to different edge types, MB-GCN preserves structural differences when learning representations.

While this approach enables the *actor* to generate improved task scores, the heuristic remains a key decision-making component. However, direct training of this system is impossible as gradients cannot propagate back to the actor through the non-differentiable heuristic. To address this, we introduce a differentiable *twin*, also an MB-GCN, that approximates the heuristic's decisions. This allows for gradient-based optimization of the actor's parameters, which leads to improvements in the overall performance. Once training is complete, the twin is removed, allowing the actor to operate independently using the learned parameters.

We performed a series of experiments to evaluate key aspects of task scheduling. Results show that *Actor-Twin* consistently outperforms HEFT in both average and variance of makespan. While *Actor-Twin* incurs a marginally higher scheduling time for small graphs, it demonstrates better performance for medium and large graphs, suggesting improved scalability with increasing task graph complexity.

In summary, this work makes the following contributions:

- We introduce a *systematic graph-based DAG* representation tailored for task-server pairs that not only enables cloud systems to handle a variable number of servers without requiring retraining but also generalizes to dynamic server connectivity.
- We propose a *Multi-Branch Graph Convolutional Network* (MB-GCN) for task scheduling, where each graph component is processed independently, preserving structural semantics and supporting adaptive task prioritization.
- We present the *Actor-Twin* framework, which integrates the *ACTS* module, combining an actor for task scoring alongside a twin-network architecture. This design bypasses non-differentiable heuristics, enabling gradient-based optimization of the actor. vspace1mm
- We demonstrate that *Actor-Twin* consistently outperforms HEFT in both the average and variance of makespan. Moreover, it achieves better scalability and efficiency for medium and large graphs despite slightly higher scheduling time on small graphs.

2 RELATED WORK

Task scheduling is an optimization challenge often addressed through single or multi-objective approaches. As it is NP-hard [20], numerous solutions have been proposed, each with strengths and limitations. In this context, both machine learning (ML) and classical algorithmic methods have been explored to enhance adaptability and improve scheduling efficiency [24, 25].

2.1 Classical Methods for Graph Scheduling

Classical methods for solving graph scheduling problems have focused on mathematical programming [2], heuristics [16], and metaheuristics [1]. Mathematical programming approaches, particularly linear and convex optimization, have been widely used to model scheduling as an optimization problem [9, 19, 51]. Even though they provide optimal solutions for small instances, they become computationally intractable as the size of the problem grows. While heuristic-based methods, such as path-planning scheduling [44] and task duplication strategies [4], provide efficient solutions, but can misinterpret dependencies or fail in large-scale graphs, leading to suboptimal results.

To address these limitations, metaheuristic techniques such as Genetic Algorithms (GAs) and Particle Swarm Optimization (PSO) have been explored [42, 56]. These methods introduce stochastic search mechanisms that improve exploration beyond traditional

heuristics. However, their high computational cost and reliance on extensive parameter tuning often limit their applicability in large-scale scheduling problems. Hybrid methods, such as GA combined with PSO [26] or Variable Neighborhood Search (VNS) [56], have demonstrated improvements in makespan and resource allocation but introduce additional computational overhead, making them impractical for large-scale systems.

Beyond these approaches, AI-driven classical methods such as Constraint Programming (CP) and Logic Programming have been applied to scheduling. CP formulates scheduling as a constraint satisfaction problem, leveraging backtracking and constraint propagation for efficient search [5, 32]. Logical reasoning frameworks, such as Boolean Satisfiability (SAT) solvers, convert scheduling constraints into Boolean formulas, enabling efficient search strategies for task ordering [18]. Their high computational complexity limits its practicality in large-scale scenarios despite offering theoretical guarantees.

Although classical methods can be effective, they often lack efficiency. An effective scheduling algorithm minimizes makespan, but an efficient one also reduces the complexity of the optimization process.

2.2 Learning Methods for Graph Scheduling

In recent years, learning-based approaches have become a popular method for solving graph scheduling problems due to their ability to generalize and adapt to complex environments. These methods are often integrated with heuristic or metaheuristic techniques to leverage the strengths of both: heuristics provide fast, often near-optimal solutions, while learning-based models improve through adaptation and feedback [39, 40, 62].

Supervised learning (SL) has been used to predict schedules based on labeled datasets, utilizing task durations and dependencies. While effective, SL methods' performance relies heavily on the availability of well-labeled, diverse, and representative datasets. For instance, SL was combined with constraint programming (CP) to improve job shop scheduling by predicting variable ordering based on prior instances [46]. However, the effectiveness of this approach remains constrained by the quality and availability of training data. In contrast, unsupervised methods like clustering algorithms offer more flexibility but face challenges in handling complex task graphs. As an example, a multi-level parallel scheduling approach was proposed by Kaur et al. [27], where task graphs are partitioned into clusters to minimize execution time. Heuristics such as MinMin were integrated with metaheuristics such as the Bacterial Foraging Optimization Algorithm (BFOA) and PSO. While communication delays were reduced by this strategy, the overhead between task clusters across resources remained a challenge.

Among learning approaches, reinforcement learning (RL) methods [47] like deep Q-Networks (DQN) [35] and policy gradient (PG) methods [48] have gained significant traction in recent years. They frame scheduling as a sequential decision problem, where models like DeepWave [45] and Decima [34] have shown success in minimizing job completion time and makespan, respectively. Nonetheless, these approaches face scalability issues due to large action spaces. In our approach, RL is used solely for task prioritization, while the heuristic handles scheduling. By offloading the scheduling step to the heuristic, the RL model operates in a reduced action space. A blend of learning methods is seen in the work by Wang et al. [52], a system that initially uses heuristics to prioritize tasks based on spatial and temporal representations, followed by a transition to deep RL for automated scheduling. Despite its improvements, it is computationally intensive and relies on feature extraction, limiting its performance across diverse scheduling environments.

In a related setting to our work, READYS [23] integrates a Graph Convolutional Network (GCN) and an actor-critic approach for scheduling in heterogeneous environments. However, unlike our approach, READYS does not account for network settings and lacks methods to mitigate the challenges typical of pure RL solutions [11]. Our approach integrates RL with a heuristic-based decision process to address these issues. Unlike previous RL-based schedulers that make direct scheduling decisions, our *Actor-Twin* framework focuses on task prioritization, allowing the heuristic to handle the final scheduling step. This maintains computational efficiency while leveraging the learning capabilities of RL.

3 ACTOR-TWIN METHODOLOGY

We now introduce our method for learning-based task prioritization, *Actor-Twin*. Our desiderata are twofold: we want a trainable, gradient-based system that adapts to diverse scheduling scenarios; we also want a computationally efficient approach that scales to large scheduling problems. Since task graph execution demands fast responses, relying on high-complexity optimal solvers is impractical. Thus, in addition to reducing makespan, we want our approach to minimize scheduling overhead. Inspired by actor-critic RL methods, our approach uses reinforcement learning to prioritize tasks without labeled data, addressing the NP-hard nature of the problem. The heuristic enforces task-specific constraints, while the twin, acting as a critic substitute, refines the actor's performance and improves generalization across diverse graph structures.

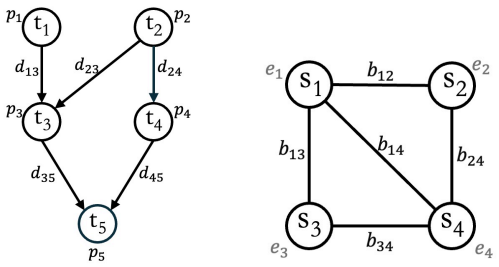
In the following subsections, we will explore these components of the *Actor-Twin* architecture in more detail, discussing the structure of the graph model and the differentiable twin mechanism and how they improve task scheduling.

3.1 Graph Modeling

Unified DAG Model. Task scheduling presents a unique challenge for graph-based learning as it involves managing two distinct graphs. The *task graph* represents tasks and their dependencies, while the *server network graph* captures node computational capabilities and communication rates. Examples of such graphs can be seen in Figure 2a and Figure 2b, respectively.

To facilitate cost computation and enable a graph-based machine learning model, we have constructed a Directed Acyclic Graph (DAG) from the task graph $\mathcal{G}_t(\mathcal{T}, \mathcal{E}_t)$ and the server graph $\mathcal{G}_s(\mathcal{S}, \mathcal{E}_s)$ as illustrated in Figure 3.

In our DAG model $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, each node represents a pair of task and server, e.g., (t_1, s_1), or simply $(1, 1)$, for task t_1 executed on server s_1. A directed edge $((t_i, s_m), (t_j, s_n)) \in \mathcal{E}$ connects node (t_i, s_m) to node (t_j, s_n) if there is a directed edge from t_i to t_j on the task graph and an edge between s_m and s_n on the server graph. The cost of a node (t_i, s_m) is $c_{im} = p_i/e_m$ and the cost of an edge from



(a) Task Graph **(b) Server Graph**

Figure 2: Example of a task graph and a server graph. The task graph represents the relationships between tasks and their dependencies, whereas the server graph illustrates nodes' computational capabilities and communication rates.

(t_i, s_m) to (t_j, s_n) is $c_{im,jn} = d_{ij}/b_{mn}$, as defined under the related machines model [22]. The cost matrix $\mathbf{C} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ places node costs on its diagonal elements and edge costs on the off-diagonal elements. The diagonal elements of $\mathbf{C}$ can change from time to time as the servers are occupied by different lists of tasks, even if constant communication costs are assumed. We further extend our graph model by introducing a virtual source S connected to all the nodes of zero incoming degrees and a virtual sink T connected to all the nodes of zero outgoing degrees. The costs of virtual edges are zero.

Some list scheduling methods, such as HEFT [49], do not inherently prevent a task from being assigned to multiple servers, assuming each task is assigned to a single processor without conflict checks. To address this limitation, we define a Conflict graph $\mathcal{G}_c = (\mathcal{V}, \mathcal{E}_c)$, where $((t_i, s_m), (t_i, s_n)) \in \mathcal{E}_c$ for all tasks t_i and servers s_m and s_n. This formulation ensures that task-server pairs associated with the same task conflict with each other, meaning that only one of them can be scheduled at a time. To capture sequential resource dependencies for tasks scheduled on the same server, we introduce a third graph, the Dependency Graph, $\mathcal{G}_d = (\mathcal{V}, \mathcal{E}_d)$. In this graph, an undirected edge $((t_i, s_m), (t_j, s_m)) \in \mathcal{E}_d$ exists between two task-server pairs if there is no direct path between them in $\mathcal{G}$, indicating a potential resource dependency between these tasks when assigned to the same server.

The scheduler not only guarantees conflict constraints but also ensures that scheduled nodes remain connected in our graph model, preserving task dependencies. Specifically, if a valid schedule assigns multiple tasks to the same server, it must define their execution order. To enforce this order, once a schedule is determined, we introduce additional edges, a zero-cost directed edge from (t_i, s_m) to (t_j, s_m) if the former is scheduled immediately before the latter. These edges capture sequential dependencies between tasks that share a server (depicted as directed green edge in Figure 3). With these constraints in place, we construct the residual DAG $\tilde{\mathcal{G}}$ by removing unscheduled nodes (and their corresponding edges) from $\mathcal{G}$. The total cost of a given schedule, known as the makespan, is then computed as the sum of the costs of nodes and edges along the critical path from the virtual source to the virtual sink in $\tilde{\mathcal{G}}$. The critical path, which determines the overall completion time, is the longest-cost path from the source to the sink (depicted as bold purple edges in Figure 3).

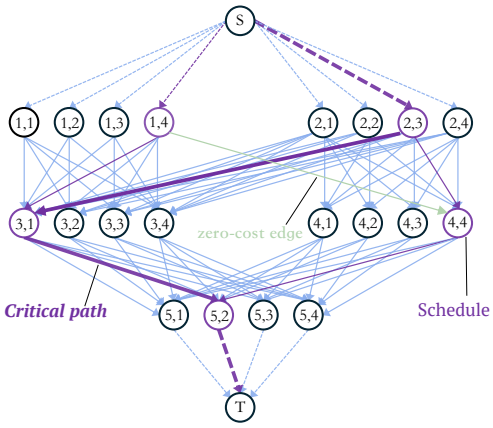


Figure 3: Proposed Graph Model. In this DAG, nodes represent task-server pairs, with edges capturing dependencies and costs reflecting processing and communication overhead. Virtual source and sink nodes handle tasks without predecessors or successors. The green directed edge indicates a resource dependency post-scheduling.

One of the main advantages of using this modeling approach, as opposed to the graph modeling described in [13, 14, 29], is its ability to generalize across diverse server network configurations and dynamically changing task graphs. This adaptability is particularly critical in decentralized environments such as mobile ad-hoc clouds, edge computing, and fog computing systems, where both computational resources and network topology can vary over time. Unlike traditional centralized cloud facilities with fixed infrastructure, these dynamic settings demand flexible scheduling approaches capable of adjusting to fluctuating resources and connectivity without necessitating retraining [3, 12].

The graph modeling discussed by Kiamari and Krishnamachari [29] and Coleman et al. [13, 14], where node features $[p_i/e_s | s \in \mathcal{S}] \in \mathbb{R}^{|\mathcal{S}|}$ for task $t_i \in \mathcal{T}$ and edge features $\left[d_{ij}/b_{mn} | s_m, s_n \in \mathcal{S}\right] \in \mathbb{R}^{|\mathcal{S}|^2}$ for tasks $t_i, t_j \in \mathcal{T}$ remain fixed, only works for systems with a constant number of servers. Consequently, modifying the number of servers in these systems requires retraining the GCN.

Graph Input Processing. Once the Unified (Main) Graph, Conflict Graph, and Dependency Graph are constructed (Algorithms 1, 2, and 3), they are fed into both the actor and the twin networks. The primary challenge here is to process these graphs in a way that preserves their unique structural constraints while enabling effective task prioritization. A standard GCN, which applies uniform message passing across all edges, cannot distinguish between task dependencies, execution conflicts, and server constraints, leading to information loss or misinterpretation of graph relationships. To mitigate this problem, we model them as a *Multi-Branch Graph Convolutional Network* (MB-GCN), where each graph is processed by a dedicated GCN branch before their representations are combined. This ensures that distinct structural constraints of the graphs

are preserved rather than collapsed into a single representation. The embeddings from these branches are then aggregated through a weighted combination, preserving the semantic roles of each graph while providing a comprehensive representation for task prioritization.

3.2 ACTS Architecture

We present a model for task graph scheduling that leverages the GDPG-Twin framework, structured into several stages to optimize task assignments and minimize makespan under the constraints of a dynamic task-server environment. This process is illustrated in Figure 1 and outlined in Algorithm 5.

Actor and Twin Network. We define the *actor* $\Psi(\cdot)$ and *twin* $\Phi(\cdot)$ networks as follows:

$$\mathbf{z} = \Psi(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathrm{C}; \boldsymbol{\omega}_a), \quad \hat{\mathbf{O}} = \Phi(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathrm{C}, \mathbf{z}; \boldsymbol{\omega}_t). \tag{1}$$

The actor network generates task prioritization scores based on the MB-GCN embeddings, while the twin estimates expected scheduling outcomes. This is motivated by list-based scheduling approaches, such as HEFT and CPOP [49], that consist of two phases of task prioritization and server selection (to assign the task to the server nodes). Unlike the HEFT algorithm, which uses static prioritization, our *Actor-Twin* framework continuously adapts task priorities according to evolving task-server relationships, making it better suited for handling fluctuating system loads and closely reflecting real-world conditions.

Heuristic Scheduler. A pre-defined heuristic scheduler $h(\cdot)$ is used to generate a scheduling policy,

$$\mathbf{y} = h(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathrm{C} \odot \mathbf{z}). \tag{2}$$

The heuristic utilizes the actor's task prioritization score to determine an efficient schedule.

Twin Training. The expected outcome matrix $\mathbf{O} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ represents the cost associated with each task and is defined as

$$\mathbf{O} = \mathbb{E}_{\mathrm{C}\sim\Omega}\left[\mathrm{C} \odot \mathbf{P}(\mathbf{y}, \mathrm{C}, \mathcal{G})\right], \tag{3}$$

where $\mathbf{P}(\mathbf{y}, \mathrm{C}, \mathcal{G})$ is the critical path indicator matrix under the given schedule and graph model. Since direct gradient propagation through the heuristic is infeasible, the twin approximates the expected scheduling outcomes using an Mean Squared Error (MSE) loss function,

$$\ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O}) = \frac{1}{|\mathcal{V}|^2} \sum_{(u,v)\in\mathcal{V}} (\hat{\mathbf{O}}_{uv} - \mathbf{O}_{uv})^2. \tag{4}$$

Essentially, the twin component in our framework mirrors the heuristic decisions, allowing for smooth gradient propagation during training. This allows our model to handle non-differentiable components, addressing a key limitation of traditional methods. The twin is trained via gradient descent $\boldsymbol{\omega}_t \leftarrow \boldsymbol{\omega}_t - \alpha_t \nabla_{\boldsymbol{\omega}_t} \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O})$, where α_t is the twin's learning rate. Since $\mathbf{O}$ is an expectation over our sample space, we can further incorporate stochastic gradient descent (SGD) to simplify this equation using the following lemma.

Lemma 1. *The gradient of the Mean Squared Error (MSE) loss with respect to the twin network parameters* $\boldsymbol{\omega}_t$ *is* $\nabla_{\boldsymbol{\omega}_t} \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O}) = \nabla_{\boldsymbol{\omega}_t} \ell_{MSE}(\hat{\mathbf{O}}, \mathrm{C} \odot \mathbf{P}(\mathbf{y}, \mathrm{C}, \mathcal{G}))$ *where* $\hat{\mathbf{O}}$ *is the predicted outcome matrix and* $\mathbf{O}$ *is the expected outcome matrix.*

Proof. See Appendix A.3 □

As a consequence of this lemma, we can refine the twin update to $\boldsymbol{\omega}_t \leftarrow \boldsymbol{\omega}_t - \alpha_t \nabla_{\boldsymbol{\omega}_t} \ell_{MSE}(\hat{\mathbf{O}}, \mathrm{C} \odot \mathbf{P}(\mathbf{y}, \mathrm{C}, \mathcal{G}))$.

Actor Training. After the twin is trained, we train the actor by computing the gradient of the total outcome sum $\mathbf{1}^\top \hat{\mathbf{O}} \mathbf{1}$ with respect to the actor parameters,

$$\nabla_{\boldsymbol{\omega}_a} \mathbf{1}^\top \hat{\mathbf{O}} \mathbf{1} = \left(\nabla_{\mathbf{Z}} (\mathbf{1}^\top \hat{\mathbf{O}} \mathbf{1})\right)^\top \cdot \nabla_{\boldsymbol{\omega}_a} \Psi(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathrm{C}; \boldsymbol{\omega}_a). \tag{5}$$

The actor's parameters are updated accordingly using gradient descent $\boldsymbol{\omega}_a \leftarrow \boldsymbol{\omega}_a - \alpha_a \nabla_{\boldsymbol{\omega}_a} \mathbf{1}^\top \hat{\mathbf{O}} \mathbf{1}$, where α_a is the learning rate for the actor. To enhance exploration during training, noise can be added to the actor output $\mathbf{z}$, encouraging diverse scheduling decisions.

4 EXPERIMENTS

In this section, we empirically evaluate the performance of the proposed *Actor-Twin* scheduler against the traditional HEFT and a Random scheduler. The Random scheduler is of interest as a naive baseline to quantify the lower bound of scheduling performance, given its minimal computation overhead but suboptimal scheduling quality. Conversely, we select HEFT as our primary benchmark because it balances scheduling efficiency and makespan optimization, making it one of the few practical schedulers for real-world task graph execution.

Experimental Setup. The experiments analyze two aspects of scheduling:

(1) **Makespan**: Total time required to execute all tasks in a task graph.
(2) **Scheduling Time**: Computation time taken by the scheduler to generate a schedule for a given task graph.

These metrics provide complementary insights: makespan reflects how quickly a scheduler can complete an entire task set, while scheduling time indicates the computational overhead of generating the schedule. Notably, a low scheduling overhead does not guarantee a minimized makespan, making both metrics essential to report. We incorporate variance bars and confidence intervals (CIs) where applicable, as minimal variance is crucial for stable performance across diverse task configurations and resource conditions. CIs provide statistical significance, helping assess both mean performance and the consistency of each scheduling approach.

We perform experiments on task graphs of three sizes: Small (10 tasks), Medium (50 tasks), and Large (100 tasks). These task graphs were generated following the methodology outlined in SAGA [13]. For each graph size, we consider two dependency conditions: *Dense* (highly connected graphs, e.g., cycles dataset) and *Sparse* (lightly connected graphs, e.g., chain and in-trees datasets). For each task graph size and dependency type, we generate 10 task graphs per category, leading to a total of 60 task graphs. These graphs are split into 40 for training and 20 for testing. Each task graph is executed on a simulated system with 3 servers for small graphs, 5 servers for medium graphs, and 10 servers for large graphs.

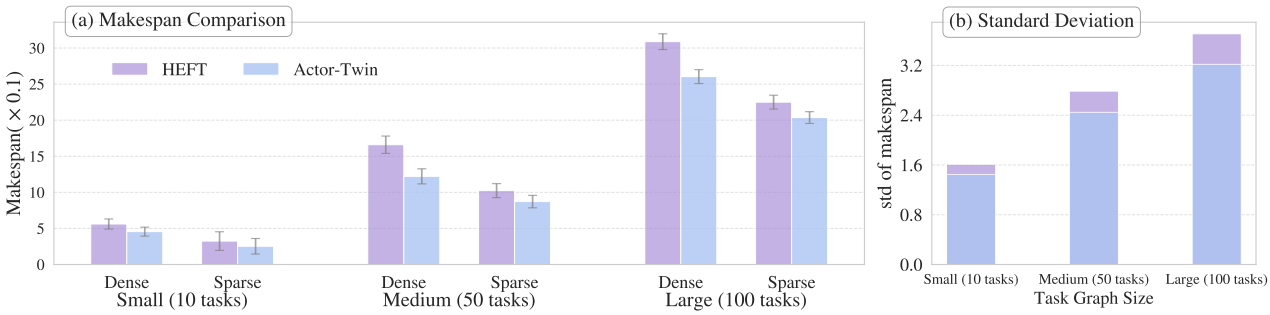


Figure 4: Makespan comparison between HEFT and *Actor-Twin* across different task graph sizes and dependency structures. (a) Mean Makespan comparison: *Actor-Twin* achieves lower makespan in most cases, especially for larger, denser graphs, while HEFT shows greater variability with wider confidence intervals across runs. (b) Standard Deviation Comparison: HEFT shows greater variability in makespan, particularly for larger and denser graphs, whereas *Actor-Twin* achieves more stable scheduling behavior across different task structures.

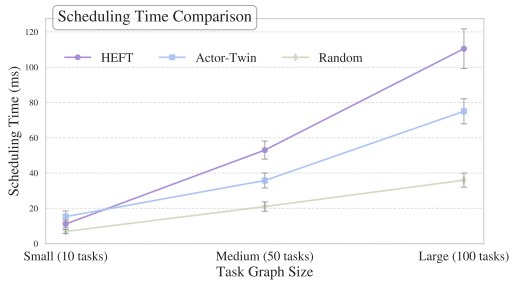


Figure 5: Scheduling Time Comparison. *Actor-Twin* scales more efficiently than HEFT, maintaining lower scheduling times for larger graphs, while HEFT's performance degrades due to its sorting overhead. The Random Scheduler has the lowest scheduling time due to minimal computational overhead

Results and Analysis. Figure 4a presents the makespan results for HEFT and Actor-Twin across different graph sizes and densities. For each DAG type, we compute the makespan and report it with a 95% confidence interval (CI) to indicate statistical significance and variability. *Actor-Twin* consistently achieves a lower makespan than HEFT, particularly in larger task graphs and dense configurations, where HEFT's reliance on static priorities leads to inefficiencies. Moreover, HEFT exhibits wider confidence intervals, especially in dense graphs and larger task sizes, suggesting greater variability due to differences in task dependencies and execution conditions. In contrast, confidence intervals of *Actor-Twin* tend to be narrower across most cases, indicating more consistent scheduling performance.

Figure 4b presents standard deviation bars of makespan across different task graph sizes and dependency structures. HEFT tends to show greater variability, particularly in larger and denser task graphs, indicating that its performance is more affected by task dependencies and execution conditions. In contrast, *Actor-Twin* consistently results in lower variance, implying a more stable scheduling behavior between different scenarios.

Figure 5 represents the raw computation time required by each scheduler to make scheduling decisions (independent of network conditions). HEFT starts off faster for small graphs due to its simplistic design and minimal overhead; however, its performance degrades as the graph size increases. In contrast, *Actor-Twin*, although incurring slightly higher overhead for small graphs, scales more efficiently and outperforms HEFT on medium and large graphs. The twin-network design of *Actor-Twin* ensures scalability while maintaining low scheduling times for larger graphs. This speedup is likely due to several factors. In heuristics like HEFT, task prioritization is explicitly determined using a computed metric (e.g., upward rank), requiring $O(n \log n)$ complexity for sorting. In contrast, *Actor-Twin* learns an implicit ordering through the MB-GCN's representations and policy network, eliminating the need for explicit sorting at inference time. Additionally, the model's learned representations allow *Actor-Twin* to generalize across graphs, avoiding redundant recomputation. The Random Scheduler shows the lowest scheduling time across all task sizes, as it does not perform dependency-aware computations, resulting in minimal overhead. However, this comes at the cost of higher makespan due to its lack of structured decision-making [29].

5 CONCLUSION AND FUTURE WORK

In this work, we introduced the *Actor-Twin* framework, a reinforcement learning-based approach for task scheduling that integrates graph-based modeling with heuristic optimization. By introducing a Unified DAG Model and processing it through a Multi-Branch GCN (MB-GCN), our method captures task dependencies, execution conflicts, and resource constraints while preserving structural semantics. The core concept of our framework revolves around generating a task score, which is the primary function of the actor within our Actor-Twin Scheduler (*ACTS*) module. Our framework further enables gradient-based optimization by leveraging a differentiable twin network to approximate the heuristic's scheduling behavior. Unlike prior RL-based schedulers [34, 45], which directly optimize scheduling decisions and suffer from large action spaces, our method decouples task prioritization (via the actor) from execution decisions (via the twin/heuristic). Empirical results show

that *Actor-Twin* consistently outperforms HEFT in makespan, particularly for larger and denser task graphs. The learned scheduling policy not only scales efficiently but also achieves lower overhead and variance, leading to more stable scheduling decisions across diverse task structures.

Moving forward, we plan to extend our evaluation by incorporating additional learning-based schedulers to better contextualize the advantages of our approach. Furthermore, we plan to compare *Actor-Twin* with other algorithms, particularly from Coleman et al.'s [13] parametric scheduler work. Another key direction is to leverage the PISA framework [15] to identify adversarial instances where our approach significantly outperforms HEFT, and vice versa. Given the NP-hard nature of task scheduling, a single approach is unlikely to be universally optimal. These comparisons will provide a deeper understanding of *Actor-Twin* 's comparative strengths and limitations.

ACKNOWLEDGMENTS

This work was supported by Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196.

REFERENCES

[1] Sourav Kanti Addya, Ashok Kumar Turuk, Bibhudatta Sahoo, Mahasweta Sarkar, and Sanjay Kumar Biswash. 2017. Simulated annealing based VM placement strategy to maximize the profit for Cloud Service Providers. *Engineering science and technology, an international journal* 20, 4 (2017), 1249–1259.

[2] Yossi Azar and Amir Epstein. 2005. Convex programming for scheduling unrelated parallel machines. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. 331–337.

[3] Enzo Baccarelli, Michele Scarpiniti, and Alireza Momenzadeh. 2019. EcoMobiFog – Design and Dynamic Optimization of a 5G Mobile-Fog-Cloud Multi-Tier Ecosystem for the Real-Time Distributed Execution of Stream Applications. *arXiv preprint arXiv:1906.07578* (2019). https://arxiv.org/abs/1906.07578

[4] Nirmeen A Bahnasawy, Magdy A Koutb, Mervat Mosa, and Fatma Omara. 2011. A new algorithm for static task scheduling for heterogeneous distributed computing systems. *African Journal of Mathematics and Computer Science Research* 4, 6 (2011), 221–234.

[5] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. 2001. *Constraint-based scheduling: applying constraint programming to scheduling problems*. Vol. 39. Springer Science & Business Media.

[6] Eda Bayram, Dorina Thanou, Elif Vural, and Pascal Frossard. 2020. Mask combination of multi-layer graphs for global structure inference. *IEEE Transactions on Signal and Information Processing over Networks* 6 (2020), 394–406.

[7] Abbas Bazzi and Ashkan Norouzi-Fard. 2015. Towards tight lower bounds for scheduling problems. In *Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*. Springer, 118–129.

[8] Jakub Beránek, Stanislav Böhm, and Vojtěch Cima. 2022. Analysis of workflow schedulers in simulated distributed environments. *The Journal of Supercomputing* 78, 13 (2022), 15154–15180.

[9] Pierre Bonami, Andrea Lodi, Andrea Tramontani, and Sven Wiese. 2015. On mathematical programming with indicator constraints. *Mathematical programming* 151 (2015), 191–223.

[10] Vincent Boudet. 2001. *Heterogeneous task scheduling: a survey*. Ph.D. Dissertation. Laboratoire de l'informatique du parallélisme.

[11] Tim Brys. 2016. Reinforcement Learning with Heuristic Information. *Dissertationm Vrije Universiteit Brussel* (2016).

[12] Weiwei Chen, Chin-Tau Lea, and Kenli Li. 2017. Dynamic Resource Allocation in Ad-Hoc Mobile Cloud Computing. In *2017 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 1–6. https://doi.org/10.1109/WCNC.2017.7925613

[13] Jared Coleman, Ravi Vivek Agrawal, Ebrahim Hirani, and Bhaskar Krishnamachari. 2024. Parameterized Task Graph Scheduling Algorithm for Comparing Algorithmic Components. *arXiv preprint arXiv:2403.07112* (2024).

[14] Jared Coleman, Mehrdad Kiamari, Lillian Clark, Daniel D'Souza, and Bhaskar Krishnamachari. 2022. Graph convolutional network-based scheduler for distributing computation in the internet of robotic things. In *MILCOM 2022-2022 IEEE Military Communications Conference (MILCOM)*. IEEE, 1070–1075.

[15] Jared Coleman and Bhaskar Krishnamachari. 2024. Comparing Task Graph Scheduling Algorithms: An Adversarial Approach. *arXiv preprint arXiv:2403.07120* (2024).

[16] Yanyan Dai and Xiangli Zhang. 2014. A synthesized heuristic task scheduling algorithm. *The Scientific World Journal* 2014, 1 (2014), 465702.

[17] Nima Dehmamy, Albert-László Barabási, and Rose Yu. 2019. Understanding the representation power of graph neural networks in learning graph topology. *Advances in Neural Information Processing Systems* 32 (2019).

[18] Nicolai Fiege and Peter Zipf. 2023. BLOOP: Boolean Satisfiability-based Optimized Loop Pipelining. *ACM Transactions on Reconfigurable Technology and Systems* 16, 3 (2023), 1–32.

[19] Christodoulos A Floudas and Xiaoxia Lin. 2005. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research* 139 (2005), 131–162.

[20] Michael R Garey, David S Johnson, et al. 1990. A Guide to the Theory of NP-Completeness. *Computers and intractability* (1990), 37–79.

[21] Diana Gomes, Frederik Ruelens, Kyriakos Efthymiadis, Ann Nowe, and Peter Vrancx. [n.d.]. When are graph neural networks better than structure-agnostic methods?. In *I Can't Believe It's Not Better Workshop: Understanding Deep Learning Through Empirical Falsification*.

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

[23] Nathan Grinsztajn, Olivier Beaumont, Emmanuel Jeannot, and Philippe Preux. 2021. Readys: A reinforcement learning based strategy for heterogeneous dynamic scheduling. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 70–81.

[24] Mirsaeid Hosseini Shirvani. 2024. A survey study on task scheduling schemes for workflow executions in cloud computing environment: classification and challenges. *The Journal of Supercomputing* 80, 7 (2024), 9384–9437.

[25] Essam H Houssein, Ahmed G Gad, Yaser M Wazery, and Ponnuthurai Nagaratnam Suganthan. 2021. Task scheduling in cloud computing based on meta-heuristics: review, taxonomy, open challenges, and future trends. *Swarm and Evolutionary Computation* 62 (2021), 100841.

[26] Habib Izadkhah. 2019. Learning based genetic algorithm for task graph scheduling. *Applied Computational Intelligence and Soft Computing* 2019, 1 (2019), 6543957.

[27] Mandeep Kaur, Sanjay Kadam, and Naeem Hannoon. 2022. Multi-level parallel scheduling of dependent-tasks using graph-partitioning and hybrid approaches over edge-cloud. *Soft Computing* 26, 11 (2022), 5347–5362.

[28] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*. 6348–6358.

[29] Mehrdad Kiamari and Bhaskar Krishnamachari. 2021. GCNScheduler: Scheduling Distributed Computing Applications using Graph Convolutional Networks. arXiv:2110.11552 [cs.DC]

[30] Thomas N Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*.

[31] Vijay R. Konda and John N. Tsitsiklis. 2000. Actor-Critic Algorithms. In *Proceedings of the 13th International Conference on Neural Information Processing Systems*.

[32] Feng Kong and Dong Dou. 2021. Resource-constrained project scheduling problem under multiple time constraints. *Journal of Construction Engineering and Management* 147, 2 (2021), 04020170.

[33] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)* 31, 4 (1999), 406–471.

[34] Hongzi Mao, Malte Schwarzkopf, Shravan Venkatakrishnan, Ziliang Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.

[35] Volodymyr Mnih. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[36] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on Machine Learning*.

[37] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E Taylor, and Peter Stone. 2020. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research* 21, 181 (2020), 1–50.

[38] Hao Peng, Ruitong Zhang, Yingtong Dou, Renyu Yang, Jingyi Zhang, and Philip S Yu. 2021. Reinforced neighborhood selection guided multi-relational graph neural networks. *ACM Transactions on Information Systems (TOIS)* 40, 4 (2021), 1–46.

[39] Yaoyao Ping, Yongkui Liu, Lin Zhang, Lihui Wang, and Xun Xu. 2023. Sequence generation for multi-task scheduling in cloud manufacturing with deep reinforcement learning. *Journal of manufacturing systems* 67 (2023), 315–337.

[40] Shubham Suresh Pol and Avtar Singh. 2021. Task scheduling algorithms in cloud computing: a survey. In *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*. IEEE, 244–249.

[41] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural*

networks 20, 1 (2008), 61–80.

[42] Henrique Yoshikazu Shishido, Júlio Cezar Estrella, Claudio Fabiano Motta Toledo, and Marcio Silva Arantes. 2018. Genetic-based algorithms applied to a workflow scheduling algorithm with security and deadline constraints in clouds. *Computers & Electrical Engineering* 69 (2018), 378–394.

[43] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic policy gradient algorithms. *International Conference on Machine Learning* (2014), 387–395.

[44] Jiri Stastny, Vladislav Skorpil, Zoltan Balogh, and Richard Klein. 2021. Job shop scheduling problem optimization by means of graph-based algorithm. *Applied Sciences* 11, 4 (2021), 1921.

[45] Penghao Sun, Zehua Guo, Junchao Wang, Junfei Li, Julong Lan, and Yuxiang Hu. 2021. 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*. 3314–3320.

[46] Yuan Sun, Su Nguyen, Dhananjay Thiruvady, Xiaodong Li, Andreas T Ernst, and Uwe Aickelin. 2024. Enhancing constraint programming via supervised learning for job shop scheduling. *Knowledge-Based Systems* 293 (2024), 111698.

[47] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction* (2 ed.). MIT Press.

[48] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems* 12 (1999).

[49] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.

[50] Kagan Tumer and John Lawson. 2009. Coordinating learning agents for multiple resource job scheduling. In *International Workshop on Adaptive and Learning Agents*. Springer, 123–140.

[51] Michael H Veatch. 2020. *Linear and convex optimization: A Mathematical Approach*. John Wiley & Sons.

[52] Haoyu Wang, Zetian Liu, and Haiying Shen. 2022. Machine learning feature based job scheduling for distributed machine learning clusters. *IEEE/ACM Transactions on Networking* 31, 1 (2022), 58–73.

[53] Huijun Wang and Oliver Sinnen. 2018. List-scheduling versus cluster-scheduling. *IEEE Transactions on Parallel and Distributed Systems* 29, 8 (2018), 1736–1749.

[54] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.

[55] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[56] Yun Wen, Hua Xu, and Jiadong Yang. 2011. A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system. *Information Sciences* 181, 3 (2011), 567–581.

[57] Minxian Xu, Qiheng Zhou, Huaming Wu, Weiwei Lin, Kejiang Ye, and Chengzhong Xu. 2022. PDMA: Probabilistic service migration approach for delay-aware and mobility-aware mobile edge computing. *Software: Practice and Experience* 52, 2 (2022), 394–414.

[58] Artur Yakimovich, Anaël Beaugnon, Yi Huang, and Elif Ozkirimli. 2021. Labels in a haystack: Approaches beyond supervised learning in biomedical applications. *Patterns* 2, 12 (2021).

[59] Ibrar Yaqoob, Ejaz Ahmed, Abdullah Gani, Salimah Mokhtar, Muhammad Imran, and Sghaier Guizani. 2016. Mobile ad hoc cloud: A survey. *Wireless Communications and Mobile Computing* 16, 16 (2016), 2572–2589.

[60] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. 2019. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 793–803.

[61] Zhongyuan Zhao, Ananthram Swami, and Santiago Segarra. 2022. Graph-based deterministic policy gradient for repetitive combinatorial optimization problems. In *The Eleventh International Conference on Learning Representations*.

[62] Jian Zhou, Lianyu Zheng, and Wei Fan. 2024. Multirobot collaborative task dynamic scheduling based on multiagent reinforcement learning with heuristic graph convolution considering robot service performance. *Journal of Manufacturing Systems* 72 (2024), 122–141.

A APPENDIX

A.1 Preliminaries

List-scheduling. The list-scheduling method is widely adopted in heuristic scheduling algorithms [53]. List-scheduling algorithms typically involve the following steps: (1) Computing a priority for each task such that every task has a higher priority than its dependents. (2) Greedily scheduling tasks in order of their computed priorities (from highest to lowest) to run on the node that minimizes a predefined cost function. Variations of list-scheduling algorithms mainly differ in their prioritization functions, cost functions, and strategies for inserting tasks into existing schedules [13].

- *The HEFT Algorithm.* Topcuoglu et al. [49] proposed HEFT with three ranking approaches: upward, downward, and level ranking, each labeling tasks differently, leading to varying performance. HEFT minimizes makespan by scheduling tasks on the processor with the earliest finish time (EFT), aiming to achieve the shortest possible makespan (the time that the final task in the DAG is completed).

While HEFT provides an efficient heuristic for task scheduling, it relies on predefined ranking strategies that do not adapt dynamically. We expand on these limitations in Section 3 and propose an alternative learning-based approach.

Actor-Critic Methods. One of the foundational frameworks in reinforcement learning (RL) that combine the benefits of policy-based and value-based approaches is Actor-Critic methods [31]. This hybrid strategy consists of two main components: the actor, which learns to select actions, and the critic, which evaluates the actions taken by the actor by computing the value function. The actor updates the policy in the direction that maximizes the expected reward, as informed by the critic's temporal difference error. Formally, the actor updates the policy parameters θ by applying the gradient of the policy function π_θ to maximize the expected reward J, guided by the TD error δ_t

$$\nabla_\theta J \approx \mathbb{E}[\delta_t \nabla_\theta \log \pi_\theta(s_t, a_t)].$$

The dual learning process, which simultaneously optimizes the policy (actor) and the value estimation (critic), facilitates more stable convergence compared to approaches that optimize these components alone [31, 36]. Moreover, unlike Q-learning [55], which processes actions sequentially, policy gradient methods [43] are more adept at managing the expansive and dynamic action spaces typical of networked environments.

While Actor-Critic methods are widely used in RL for sequential decision-making, we adapt this structure for task prioritization. To tailor it to our problem, instead of using a traditional critic, we introduce a differentiable twin that approximates heuristic outcomes and enables gradient-based optimization.

Graph Neural Networks. Graph Neural Networks (GNNs) [41] are designed to process data that inherently form graphs, efficiently capturing complex relational patterns between entities. Central to GNNs is the message-passing mechanism, which allows nodes to exchange and integrate information across their local neighborhoods.

- *Graph Convolutional Neural Network.* By processing data through L layers, Graph Convolutional Neural Networks (GCNs) transform initial node features into more abstract representations. Starting with initial features $S^{(0)} = S$ on graph $\mathcal{G}$, the network's output $Z = S^{(L)}$ is derived by

$$S^{(l)} = \sigma^{(l)}\left(S^{(l-1)}\Theta_0^{(l)} + \mathcal{L}S^{(l-1)}\Theta_1^{(l)}\right), \quad l \in \{1, \ldots, L\}.$$

  where $\mathcal{L}$ represents the normalized Laplacian matrix that embeds the graph structure into the learning process, $\Theta_0^{(l)}$ and $\Theta_1^{(l)}$ are layer-specific trainable parameters, and $\sigma^{(l)}$ is the activation function.

A.2 Algorithms

Algorithm 1 Unified Graph

Input: $\mathcal{G}_t(\mathcal{T}, \mathcal{E}_t), \mathcal{G}_s(\mathcal{S}, \mathcal{E}_s)$
Output: $\mathcal{G}(\mathcal{V}, \mathcal{E})$
$\mathcal{V} \leftarrow \{(t_i, s_m) \mid t_i \in \mathcal{T}, s_m \in \mathcal{S}\}$
$\mathcal{E} \leftarrow \emptyset$
for each $(t_i, t_j) \in \mathcal{E}_t$ **do**
  for each $(s_m, s_n) \in \mathcal{E}_s$ **do**
    if $(t_i, s_m) \in \mathcal{V}$ and $(t_j, s_n) \in \mathcal{V}$ **then**
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{((t_i, s_m), (t_j, s_n))\}$
    end if
  end for
end for

Algorithm 2 Conflict Graph

Input: $\mathcal{G}_t(\mathcal{T}, \mathcal{E}_t), \mathcal{G}_s(\mathcal{S}, \mathcal{E}_s)$
Output: $\mathcal{G}_c(\mathcal{V}, \mathcal{E}_c)$
$\mathcal{V} \leftarrow \{(t_i, s_m) \mid t_i \in \mathcal{T}, s_m \in \mathcal{S}\}$ ▷ Nodes represent task-server pairs
$\mathcal{E}_c \leftarrow \{((t_i, s_m), (t_i, s_n)) \mid t_i \in \mathcal{T}, s_m, s_n \in \mathcal{S}, s_m \neq s_n\}$ ▷ Conflict edges

Algorithm 3 Dependency Graph

Input: $\mathcal{G}(\mathcal{V}, \mathcal{E}), \mathcal{G}_t(\mathcal{T}, \mathcal{E}_t), \mathcal{G}_s(\mathcal{S}, \mathcal{E}_s)$
Output:$\mathcal{G}_d(\mathcal{V}, \mathcal{E}_d)$
$\mathcal{V} \leftarrow \{(t_i, s_m) \mid t_i \in \mathcal{T}, s_m \in \mathcal{S}\}$
$\mathcal{E}_d \leftarrow \emptyset$
for each pair $((t_i, s_m), (t_j, s_m)) \in \mathcal{V} \times \mathcal{V}$ **do**
  if $(t_i, s_m) \neq (t_j, s_m)$ and there is no path between (t_i, s_m) and (t_j, s_m) in $\mathcal{G}$ **then**
    $\mathcal{E}_d \leftarrow \mathcal{E}_d \cup \{((t_i, s_m), (t_j, s_m))\}$ ▷ Add dependency edge
  end if
end for

Algorithm 4 ACTS ▷ for a mini-batch

Input: $\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathbf{C}, h(\cdot), \alpha_a, \alpha_t, B, \epsilon$
$Q_a = \emptyset, Q_t = \emptyset$ ▷ Clear gradient buffers
for $b \in \{1, \ldots, B\}$ **do**
  $\mathbf{z} = \Psi(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathbf{C}; \boldsymbol{\omega_a})$
  $\mathbf{z}_j = \mathbf{z} + \mathbf{N}_j, \mathbf{N}_j \in \mathbb{U}(-\epsilon, \epsilon)$ ▷ Random policy sampling
  $\mathbf{y} = h(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathbf{C} \odot \mathbf{z}^{(j)})$
  $\hat{\mathbf{O}} = \Phi(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathbf{C}, \mathbf{z}_j; \boldsymbol{\omega_t})$
  $\nabla_{\boldsymbol{\omega_t}} \ell_{\text{MSE}}(\hat{\mathbf{O}}, \mathbf{C} \odot \mathbf{P}(\mathbf{y}, \mathbf{C}, \mathcal{G}))$ ▷ Estimate gradient for twin
  $\nabla_{\boldsymbol{\omega_a}} \mathbf{1}^\top \hat{\mathbf{O}} \mathbf{1} = \left(\nabla_{\mathbf{z}} (\mathbf{1}^\top \hat{\mathbf{O}} \mathbf{1})\right)^\top \cdot \nabla_{\boldsymbol{\omega_a}} \Psi(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathbf{C}; \boldsymbol{\omega_a})$ ▷ Estimate gradient for actor
  $Q_a \leftarrow Q_a \cup \{\nabla_{\boldsymbol{\omega_a}} \mathbf{1}^\top \hat{\mathbf{O}} \mathbf{1}\}$
  $Q_t \leftarrow Q_t \cup \{\nabla_{\boldsymbol{\omega_t}} \ell_{\text{MSE}}(\hat{\mathbf{O}}, \mathbf{C} \odot \mathbf{P}(\mathbf{y}, \mathbf{C}, \mathcal{G}))\}$
end for
$\boldsymbol{\omega_t} \leftarrow \boldsymbol{\omega_t} - \alpha_t \mathbb{E}_{Q_t} \left[\nabla_{\boldsymbol{\omega_t}} \ell_{\text{MSE}}(\hat{\mathbf{O}}, \mathbf{C} \odot \mathbf{P}(\mathbf{y}, \mathbf{C}, \mathcal{G}))\right]$
$\boldsymbol{\omega_a} \leftarrow \boldsymbol{\omega_a} - \alpha_a \mathbb{E}_{Q_a} \left[\nabla_{\boldsymbol{\omega_a}} \mathbf{1}^\top \hat{\mathbf{O}} \mathbf{1}\right]$

Algorithm 5 Actor-Twin Task Graph Scheduling

Input: $\Omega^{G_t}, \Omega^{G_s}, h(.), \alpha_a, \alpha_t, E, B, \epsilon$
for $e \in \{1, 2, \ldots, E\}$ **do**
  Draw $\mathcal{G}_t(\mathcal{V}, \mathcal{E}_t) \in \Omega^{G_t}, \mathcal{G}_s(\mathcal{V}, \mathcal{E}_s) \in \Omega^{G_s}$ ▷ Draw data from training dataset
  $\mathcal{G}$ = CREATE_UNIFIED_GRAPH$(\mathcal{G}_t, \mathcal{G}_s)$
  $\mathcal{G}_c$ = CREATE_CONFLICT_GRAPH$(\mathcal{G}_t, \mathcal{G}_s)$
  $\mathcal{G}_d$ = CREATE_DEPENDENCY_GRAPH$(\mathcal{G}_t, \mathcal{G}_s)$
  $\mathbf{C}$ = CALCULATE_COST_MATRIX$(\mathcal{G}_t, \mathcal{G}_s)$
  ACTS$(\mathcal{G}, \mathcal{G}_c, \mathcal{G}_d, \mathbf{C}, h, \alpha_a, \alpha_t, B, \epsilon)$
end for

A.3 Proof of Lemma 1

Proof. Given the MSE loss function (Equation 4), the gradient with respect to the twin network parameters, $\boldsymbol{\omega_t}$, is:

$$\frac{\partial \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O})}{\partial \boldsymbol{\omega_t}} = \frac{\partial \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O})}{\partial \hat{\mathbf{O}}} \frac{\partial \hat{\mathbf{O}}}{\partial \boldsymbol{\omega_t}}$$

Substituting $\mathbf{O}$ from Equation 3:

$$\frac{\partial \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O})}{\partial \boldsymbol{\omega_t}} = \frac{2}{|\mathcal{V}|^2} \left(\hat{\mathbf{O}} - \mathbb{E}_{\mathbf{C} \sim \Omega}[\mathbf{C} \odot \mathbf{P}(\mathbf{y}, \mathbf{C}, \mathcal{G})]\right)^\top \frac{\partial \hat{\mathbf{O}}}{\partial \boldsymbol{\omega_t}}$$

By linearity of expectation:

$$\mathbb{E}_{\mathbf{C} \sim \Omega} \left[\frac{2}{|\mathcal{V}|^2} \left(\hat{\mathbf{O}} - \mathbf{C} \odot \mathbf{P}(\mathbf{y}, \mathbf{C}, \mathcal{G})\right)^\top \frac{\partial \hat{\mathbf{O}}}{\partial \boldsymbol{\omega_t}}\right]$$

Thus, the stochastic gradient estimation is:

$$\nabla_{\boldsymbol{\omega_t}} \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O}) = \mathbb{E}_{\mathbf{C} \sim \Omega} \left[\widehat{\frac{\partial \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O})}{\partial \boldsymbol{\omega_t}}}\right]$$

where

$$\widehat{\frac{\partial \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O})}{\partial \boldsymbol{\omega_t}}} = \frac{2}{|\mathcal{V}|^2} \left(\hat{\mathbf{O}} - \mathbf{C} \odot \mathbf{P}(\mathbf{y}, \mathbf{C}, \mathcal{G})\right)^\top \frac{\partial \hat{\mathbf{O}}}{\partial \boldsymbol{\omega_t}}.$$

Finally,

$$\nabla_{\boldsymbol{\omega_t}} \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{O}) = \nabla_{\boldsymbol{\omega_t}} \ell_{MSE}(\hat{\mathbf{O}}, \mathbf{C} \odot \mathbf{P}(\mathbf{y}, \mathbf{C}, \mathcal{G})).$$

□