

# Spear: Optimized Dependency-Aware Task Scheduling with Deep Reinforcement Learning

Zhiming Hu, James Tu and Baochun Li  
Department of Electrical and Computer Engineering  
University of Toronto

Email: zhiming@ece.utoronto.ca, james.tu@mail.utoronto.ca, bli@ece.toronto.edu

**Abstract**—Modern data parallel frameworks, such as Apache Spark, are designed to execute complex data processing jobs that contain a large number of tasks, with dependencies between these tasks represented by a directed acyclic graph (DAG). When scheduling these tasks, the ultimate objective is to minimize the *makespan* of the schedule, which is equivalent to minimizing the job completion time. With task dependencies, however, minimizing the makespan of the schedule is non-trivial, especially when tasks in the DAG have different resource demands with respect to multiple resource types.

In this paper, we present Spear, a new scheduling framework designed to minimize the makespan of complex jobs, while considering both task dependencies and heterogeneous resource demands at the same time. Inspired by recent advances in artificial intelligence, Spear applies Monte Carlo Tree Search (MCTS) in the specific context of task scheduling, and trains a deep reinforcement learning model to guide the expansion and rollout steps in MCTS. With deep reinforcement learning, search efficiency can be significantly improved by focusing on more promising branches. With both simulations and experiments using traces from production workloads, we compare the scheduling performance of Spear with state-of-the-art job schedulers in the literature, and Spear can outperform those approaches by up to 20%. Our results have validated our claims that MCTS and deep reinforcement learning can readily be applied to optimize the scheduling of complex jobs with task dependencies.

**Keywords**-task scheduling, big data processing, reinforcement learning

## I. INTRODUCTION

Based upon data parallel frameworks such as Hive [1], Tez [2] and Spark [3], modern data analytic jobs are inherently complex in nature, consisting of a large number of inter-dependent tasks. The dependencies between tasks are represented by a directed acyclic graph (DAG). It is naturally important to scheduling these inter-dependent tasks in the best possible way, so that the total job completion time, represented by the *makespan* of the schedule, is minimized.

Fundamentally, such a problem of dependency-aware task scheduling is challenging, especially when we need to consider the heterogeneous demands for multiple types of resources. First, the DAG scheduling problem is a NP-hard problem in general settings [4], [5]. Second, thanks to different task characteristics in the DAG, heterogeneous demands for multiple types of resources are commonly

seen in real-world workloads [6], [7]. Knowledge about such resource demands from tasks in a data analytic job is necessary to pack these tasks effectively in the cluster, and ultimately to minimize the makespan of the task schedule.

In the literature, several existing approaches has been proposed to solve the dependency-aware task scheduling problem, but with caveats and limitations. Tetris [6], for example, focused on how tasks with different resource demands can be effectively packed, but failed to consider dependencies between tasks. Traditional DAG scheduling algorithms [8], [9], [10], on the other hand, considered task dependencies, but failed to take heterogeneous resource demands for multiple types of resources into account, and may produce schedules that are far from optimal [7].

As a step forward, Grandl *et al.* [7] proposed Graphene, a new cluster scheduler that considered both task dependencies and tasks with heterogeneous resource demands for multiple types of resources. However, it heavily relied upon a suite of manually-tuned parameters used to define a set of *troublesome tasks*, which affected the scheduling outcome significantly. In addition, within each group of troublesome tasks, tasks are scheduled in the descending order of their runtimes, which may not be optimal with respect to minimizing the makespan of the entire schedule.

We argue that the root cause to the potential lack of optimality in Graphene’s scheduler lies in the heuristic nature of its scheduling algorithm, which sought to prioritize tasks based on parameter settings that are defined based on empirical experience. This design may minimize the makespan for some DAGs, but may not work well for others. We believe that a new task scheduling algorithm should be designed to be disciplined and technically sound, directly examining the search space and searching for the best possible scheduling order to minimize the makespan. It goes without saying that, given a typical task DAG, the size of the search space prohibits any exhaustive search, and unpromising search directions must be pruned. The fundamental question is, therefore, how such a scheduling algorithm can operate within the confines of real-world time and resource constraints.

In this paper, inspired by recent advances in artificial intelligence, we present our design and implementation of

Spear, a new cluster scheduler that minimizes the schedule makespan as much as possible. The theoretical foundation in Spear is Monte Carlo Tree Search (MCTS), used to search for good solutions with a certain amount of budget in the solution space. In each round of search, Spear will naturally consider both task dependencies and task resource demands for multiple types of resources.

To make it efficient and practical within a vast search space, our new scheduling algorithm first limits the search space by avoiding superficial actions, such as not scheduling anything even when the cluster is idle. Inspired by AlphaZero [11], our scheduling algorithm then uses a deep reinforcement learning (DRL) model to guide the search process of MCTS, in order to pay more attention to more promising subtrees and to improve the overall search efficiency. To achieve this objective, we replace random expansion and random rollout in MCTS, and adopt a trained DRL model to choose actions like an expert, rather than choosing them randomly.

Highlights of our original contributions in this paper are as follows. *First*, to the best of our knowledge, we are the first to adopt MCTS and DRL in the design of a new cluster scheduler that takes both task dependencies and heterogeneous resource demands into account. *Second*, we propose to train a deep reinforcement learning model for the task scheduling problem, which incorporates both resource demands and dependency graphs of tasks. Our strategy to integrate MCTS and the RL model is specifically designed for the context of dependency-aware task scheduling, and is different from existing game agents such as AlphaZero. *Finally*, We have implemented MCTS in Python and deep reinforcement learning in Theano [12]. With an extensive array of simulations and trace-driven real-world experiments, we show convincing evidence that our new scheduler outperforms existing solutions (such as Tetris and Graphene) by a significant margin: the makespan of the schedule produced by Graphene is reduced by up to 20%.

## II. PRELIMINARIES AND MOTIVATION

In this section, we briefly introduce some preliminary background on Monte Carlo Tree Search (MCTS) and Reinforcement Learning (RL) in general settings, and then show an motivating example on the benefits of applying MCTS and RL in the context of cluster scheduler design.

### A. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an efficient search algorithm suitable for sequential decision making problems where the outcome is typically a win or loss; it has been applied in many Chess AIs [11] and Go AIs [13]. In this case, MCTS maintains a state tree where each node contains a unique trajectory of actions and the edges represent individual actions. Furthermore, each node also contains a value summarizing how desirable that state is with respect to the

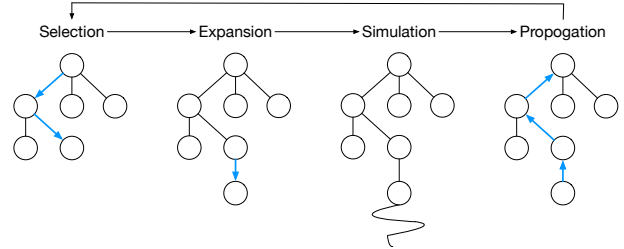


Figure 1. The steps for Monte Carlo Tree Search.

objective. When exploring the search space, MCTS utilizes this value in conjunction with information regarding prior search paths to obtain a good balance between exploiting known information and exploring unknown states.

MCTS expands the search tree by performing four steps: *Selection*, *Expansion*, *Simulation* and *Backpropagation*, shown in Fig. 1. When making a decision, MCTS will explore the search space by repeating these four steps according to how much budget is available, empirically building a tree of states and values. Then, a decision will be made by selecting the action leading to the child with the highest value.

**Selection:** In this step, MCTS will select a path of interest (sequence of moves) that will be searched further, shown in Fig. 1. This step is crucial for MCTS, as it aims to select the most promising subtree to be explored further. When traversing down the tree, the selection process tries to maintain a good balance between exploration and exploitation. Exploration entails choosing nodes that have not been explored as much with the intent of exploring new search paths that might be better. Exploitation entails choosing nodes with high value to obtain more information about promising moves. Generally, the following Upper Confidence Bound(UCB) equation is used to choose nodes when traversing down the tree.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln n}{n_i}} \quad (1)$$

where  $n_i$  is the number of visits the  $i$ -th node has received and  $w_i$  is the number of wins throughout all of simulations during the visits. Moreover,  $n$  is the total number of visits of the parent node and  $c$  represents an exploration factor that balances exploration and exploitation. In the equation, the first term is the exploitation score, describing the win rate from computed simulations. The second term is the exploration score, decreasing with the number of visits; this encourages exploring nodes that have not been explored as much.

**Expansion:** Upon selecting the path of interest and stopping at a node, the next step is to generate a new child node by performing an action from the current state. The new node is then appended to its parent and added to the search tree. With a classic MCTS approach, the action to take is selected randomly.

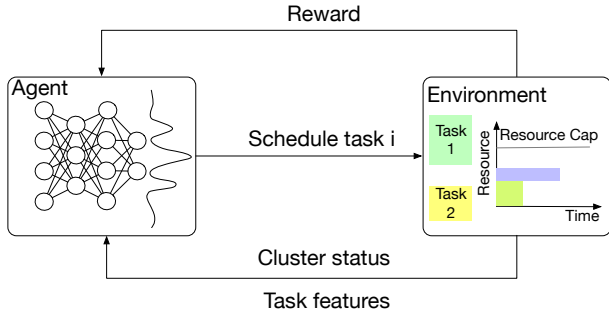


Figure 2. The framework of reinforcement learning.

**Simulation:** After expansion, there is a new node added to the search tree without any information regarding its value and it is impossible to assess how desirable the new state is. Therefore, in this step, MCTS will conduct a quick **rollout**<sup>1</sup>, simulating to the end of the game and obtaining the outcome. With a classic MCTS approach, actions are selected randomly during the rollout process.

**Backpropagation:** The results obtained from the simulation step is then backpropagated up the search tree. All ancestor nodes will have their values updated according to the outcome of the simulation.

In sum, to choose an action when playing a game, MCTS will iterate over these four steps according to the budget available, building a search tree with information on how favorable each state is. MCTS will then choose the action yielding the highest  $\frac{w_i}{n_i}$  value as the next move.

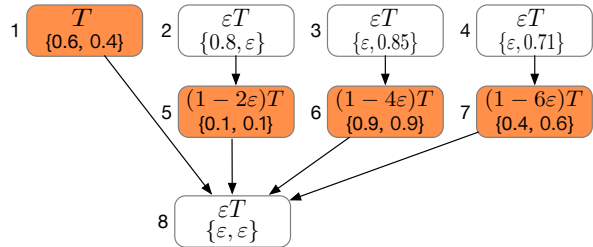
### B. Reinforcement Learning

In addition to MCTS, reinforcement learning is another framework that can be used for sequential decision-making problems. A summary is shown in Fig. 2. There are two main entities: the agent and the environment. At each time step, the agent will observe the environment, use that information and a policy to take an action, then receive a reward signal from the environment. The objective of reinforcement learning is to maximize the expected cumulative rewards  $J(\theta) = \mathbb{E}(\sum_t \gamma^t r_t)$  where  $\gamma$  is a discount factor and  $r_t$  is the reward at the  $t$ -th time slot. By observing the state  $S$ , action  $a$ , and reward  $r$  at every time step, the agent can gradually improve the policy to maximize the expected cumulative reward.

The policy  $\pi(a|S_t, \theta)$  specifically describes the probability of choosing action  $a$  given state  $S_t$  and the parameters  $\theta$ . In deep reinforcement learning, a neural network will be used as the function approximator to represent the policy mapping states and actions to probabilities. In this case,  $\theta$  denotes the parameters in the neural network as shown in Fig. 2.

To train the neural network, a policy gradient descent model can be used to find a local optima of  $\theta$  to maximize

<sup>1</sup>We use rollout and simulation interchangeably in this paper.



Algorithms	Scheduling Order	Time
Our Approach	2-3-6-5-4-1-7-8	2T
Graphene	4-2-3-7-6-1-5-8	3T
Tetris	1-3-2-6-5-4-7-8	3T
Critical path	1-2-5-3-6-4-7-8	3T

Figure 3. The motivating example. The tasks will be executed in parallel if possible given the order.

the expected cumulative rewards. The gradient of the  $J(\theta)$  can be calculated as follows [14]:

$$\nabla J(\theta) = \mathbb{E}_\pi \left[ \sum_a q_\pi(S_t, a) \nabla_{\theta} \pi(a|S_t, \theta) \right], \quad (2)$$

where  $q_\pi(S_t, a)$  is the cumulative reward resulting from taking action  $a$  in state  $S_t$  then proceeding according to the current policy  $\pi$ . Therefore, the right-hand side of the equation is the sum of the accumulated rewards over the actions weighted by the probability of selecting the actions under policy  $\pi$ . For stochastic policies where it is not feasible to compute or tabulate the cumulative reward, Monte Carlo simulations can be used to empirically compute the average cumulative reward  $G_t$ , an unbiased estimator of  $q_\pi(S_t, a)$ . Consequently,  $G_t$  will also be used to calculate the gradient update step as follows:

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t \frac{\nabla_{\theta} \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}. \quad (3)$$

In this equation, the vector  $\frac{\nabla_{\theta} \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}$  represents the direction  $\theta$  should take to increase  $\pi(A_t|S_t, \theta)$ , the probability of choosing  $A_t$  at  $S_t$ .  $G_t$  specifies the magnitude of the gradient step. In subsequent simulations, the probability of choosing  $A_t$  given  $S_t$  will increase proportionally to  $G_t$ . The learning process effectively reinforces promising actions to maximize the expected cumulative reward.

### C. The Motivating Example

We target typical jobs in the modern big data frameworks like Spark, Hive and Tez; these jobs are often modeled as a DAG. Within these DAGs, nodes represent tasks and the edges specify dependencies between the tasks. As the tasks run different programs on various data sets, it is common that they differ both in the running times and

in resource demands [6], [7]. For instance, the resource demands of reduce tasks are normally higher than map tasks. Furthermore, these tasks often have multidimensional resource demands, for example requesting various amounts of CPU and memory resources. By being aware of these resource demands, we can greatly improve the efficiency of packing tasks in the resource cluster [6], [7]. Our objective is to reduce the makespans of such jobs by considering the graph topology, running times, and resource demands.

To demonstrate the effectiveness of our approach, we show a motivating example in Fig. 3. In this figure, we can see a job with eight tasks. In each task, the top number denotes the runtime of the task and the bottom vector shows the resource demands for CPU and memory, respectively. We assume the total amount of CPU and memory available in the cluster are both 1.0. Here  $\varepsilon$  is a very small positive number close to 0. Given this DAG, the scheduling results and the makespans in the four algorithms are both presented. The makespan of our approach is  $2T$ , while the makespans of the three other algorithms are both  $3T$ . The superior performance of our approach is mainly due to the following two reasons. First, we consider both the packing efficiency as well as the dependencies of the tasks during scheduling. Second, we explore many promising scheduling options and find the best scheduling decision. However, greedy packing algorithms like Tetris do not consider the dependencies while critical path algorithms do not consider the resource demands for multiple resources. Graphene [7] is a scheduler that considers both and tries to schedule the troublesome tasks first. Although Graphene identifies the troublesome tasks in red and tries to schedule them first, the poor performance results from Graphene scheduling the troublesome tasks only by runtime. In this case, it will try to schedule the four troublesome tasks in the order of task 1, task 5, task 6 and task 7. Graphene employs two scheduling strategies to pack these tasks in the resource time space, forward scheduling, which begins placing the tasks from the bottom of the time horizon, and backward scheduling, which begins placing the tasks from the top of the time horizon. It then chooses the best scheduling result out of these two strategies. Backward scheduling returns a better result for this case and yields a scheduling order of task 7, task 6, task 1 and task 5.

### III. DESIGN

In this section, we demonstrate our design of combining Monte Carlo Tree Search (MCTS) and deep reinforcement learning (DRL) to solve the dependency-aware task scheduling problem.

#### A. Overview

In Spear, we first map our problem framework to MCTS and make adjustments to improve the search efficiency. Furthermore, we adopt deep reinforcement learning (DRL)

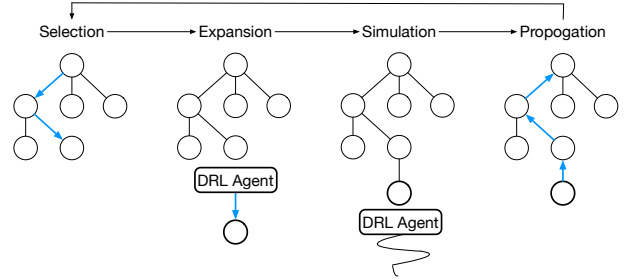


Figure 4. The overview of Spear.

to guide the search and improve the performance of MCTS. The overview of Spear is shown in Fig. 4. As shown in the figure, the DRL agent can choose an action leading to the next state during expansion and rollout, whereas the default MCTS strategy uses a random policy during these steps. There are two main reasons for integrating DRL into MCTS. First, as the agent is trained for dependency-aware task scheduling, it can expand the tree to states that typically result in lower makespans. In other words, it can focus on the more promising nodes and explore more promising sections of the search space. Second, in the rollout step, the agent can assign states a more accurate value representing the expected makespan from that state. By utilizing DRL, we can more efficiently explore the search space without increasing the runtime of MCTS.

#### B. The Resource-Time Space and State Space

We model the resource cluster as a resource-time space for a fixed period of time. Each resource dimension can be expressed as a separate rectangle with the width representing the capacity and the height denoting the time span. Multidimensional resource clusters can be represented as an array of rectangles, one for each resource type. When tasks are scheduled into the cluster, portions of the unoccupied space will be filled depending on the resource demands of the task. When the cluster is processed for a certain number of time steps, the resource-time space will shift accordingly. The state of the resource-time space will be fed as input to the agent (MCTS or DRL) to make decisions.

Furthermore, the agent must also maintain a list of tasks with their dependencies met and are available for scheduling. Given a list of available tasks, each representing a possible action, we define our action space to minimize the breadth of the search space. If we allow the scheduler to select any subset of the  $n$  ready tasks, the number of possible actions will be  $2^n$ . Instead, the set of actions are  $\{-1, 1, 2, \dots, n-1, n\}$  where action  $a = i$  ( $i \neq -1$ ) means scheduling  $i$ -th task and action  $a = -1$  means processing the tasks in the cluster. When a scheduling action takes place, the state of the cluster and the list of available tasks will be updated, however time will not move forward. The agent will continue to choose the scheduling actions until it chooses the *processing action*. If the agent chooses the *processing*

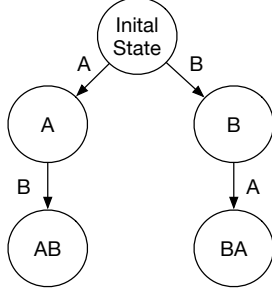


Figure 5. An example of state tree. The left branch schedules task A before task B and the right branch schedules task B first.

action, time will move forward and the resource cluster will shift forward on the time axis. Evidently, by decoupling the set of possible scheduling decisions, the action space has been substantially reduced to  $n + 1$  actions, which will significantly reduce the search space for MCTS and the training time for reinforcement learning.

### C. An Improved Monte Carlo Tree Search Algorithm

In MCTS, we build a state tree when making scheduling decisions. In the tree, each node denotes one state, defined by a unique history of actions. This is because given the same initial state, we can always reach the same state given the same sequence of actions. Each edge represents an action changing one state to another. As mentioned previously, these actions can represent scheduling a task or processing the cluster. When processing the cluster, we will only proceed until at least one task finishes, since no new information arrives prior. This adaptation attempts to minimize the depth of the search tree. A simple example for the state tree is shown in Fig. 5.

Given a certain amount of budget for the number of iterations, we iterate over the four steps: selection, expansion, simulation and backpropagation to collect data on subsequent states. After utilizing all of the budget, we then choose the next move based on the exploitation score in Eq. (1). The selected action will point to a child node which will become the new root node. This process is carried out for each decision and will be repeated until a terminal node is reached and all the tasks in the DAG are complete.

**Selection:** Starting from the current root node, we will use the equation in Eq. (1) to traverse down the search tree. The most important issue in this step is how to decide the exploration constant  $c$ . In our case, the first term in Eq. (1) is the expected makespan if we choose to move to the node in the next step. As the value of the second term in the equation is between zero and one,  $c$  must be comparable with the exploitation score to effectively balance between the exploiting promising actions and exploring new search paths. We set the value of  $c$  in the same order of the makespan of the DAG.

**Expansion:** After stopping at a node of interest, we

move one step further by taking an unexplored action and adding a new node to the search tree. By default, the set of actions available include scheduling any of the ready tasks or processing the tasks in the cluster. However, many of these possibilities are redundant and do not have to be evaluated.

First, if there are no tasks in the cluster, then the *processing action* is redundant and we do not need to consider it during expansion. Second, we only consider the tasks that can be scheduled to start before the earliest finish time of tasks in the cluster. Otherwise, the scheduler may choose to process the cluster and gain information from the completed task without losing a scheduling opportunity. These two filters significantly limit the breadth of the search space.

Furthermore, in traditional MCTS the expansion node is chosen **randomly**. However, we find out that a resource-aware DRL agent trained to minimize the makespan of a DAG is more effective than randomly selecting a child. By assigning probabilities to the possible actions, the DRL agent effectively sorts the actions by how promising they are with respect to minimizing the makespan. Therefore, when expanding the search tree, the DRL agent will be able to choose the best unexplored node. As a result, we can focus on more promising subtrees instead of a randomly selected one.

**Simulation:** In the simulation step, we simulate from the expanded node until termination, returning the negative of the makespan as the value. Instead of **randomly** choosing the actions until we reach a terminal state, we also adopt the trained DRL model to replace the random policy. In this case, our DRL model will simulate the DAG scheduling problem with expertise and provide a more meaningful estimation of the makespan as opposed to random simulations which may return misleading results. This is especially effective for larger DAGs.

**Backpropagation:** The value of the simulation step will be backpropagated back up the search tree and all ancestors of the expanded node will have their values updated. For each node, the value is updated to be the **maximum** of current value<sup>2</sup> and new value. For each node, we also keep track of the **average** of all relevant simulations to use as a tiebreaker during the selection step.

**The Budget:** In addition to the adjustments we have described so far, we also design an appropriate decay function for the budget at each level. In the DAG scheduling problem, as we traverse down the search tree, the search space decreases exponentially. Consequently, we will reduce the budget as well. Our strategy is to make the available budget inversely proportional to the depth of the current node. Additionally, we also guarantee a minimum budget for the deeper nodes to ensure a sufficient amount of information is gathered. Therefore, the budget available at each step will

<sup>2</sup>The value is the negative of the expected makespan.

be:

$$\max\left(\frac{b_{initial}}{d_i}, b_{min}\right) \quad (4)$$

where  $d_i$  is the depth of the current node and  $b_{initial}$ ,  $b_{min}$  denote the initial and minimum budget.

#### D. Our Reinforcement Learning Model

In our reinforcement learning model for a dynamic DAG scheduling problem, a neural network will take as input the list of ready tasks and the state of the cluster, then output a scheduling action. To fully explain the design of the model, we will describe how we map the DAG and cluster states to the input, the action space to the output, and how we design the rewards for training.

**States:** First, we model resource time space in the cluster and the task demands as rectangles. However, if we only take the ready tasks into consideration, we can only obtain suboptimal performance like Tetris [6]. To resolve this issue, we propose to incorporate more features to decide whether the task is important for reducing the makespan of the DAG. In Spear, we first adopt the **b-level** as one of our features, which are widely used in the DAG scheduling literature to prioritize the tasks in a DAG [8], [9], [10], [15]. The b-level is the length of longest path from the exit node to the current node (inclusive). Therefore, the upper bound of the b-level of a node is the critical path in the DAG. If the algorithm is based on the b-level, the scheduling algorithm is scheduling the tasks based on the critical path of the DAG. Moreover, the number of children of each ready task is normally used to break the tie when the values of b-level are the same. Therefore, we also include the **number of children** as one of the features.

As the b-level only captures the information about the running time of the tasks over the path, we also adopt one other feature: **b-load**, which accumulates the load of the tasks along the corresponding path. Here, the load is calculated by the product of the task runtime and the resource demands for a certain type of resource of the task. Therefore, we will calculate the b-load for both CPU and memory in our case, respectively. With these features (b-level, the number of children, b-load (CPU), b-load (memory)), our reinforcement learning model produces results superior to a model where we don't incorporate graph related features. As a result, the DRL model can easily surpass the heuristic approaches like Tetris and Shortest Job First (SJF).

**Actions:** The action space is the same with what we mentioned in Sec. III-B. We only have  $n + 1$  actions for a list of  $n$  ready tasks. Note the set of ready tasks in the DAG will be updated during the scheduling. Moreover, each time when the DRL agent is called to take an action, it will draw one action from the distribution of the actions in the output layer. When the *processing action* is selected, the tasks in the cluster will be processed for one time slot.

**Rewards:** To obtain the makespan of the scheduling, the agent will receive -1 reward each time the *processing action* is selected, during which the real executions will start for one time slot. Therefore, the total accumulated reward over time after all the tasks are complete, which is also the objective of the DRL model, equals the negative of the DAG makespan.

#### IV. IMPLEMENTATION

To approximate a scheduling policy, Spear uses a 3 hidden layer neural network with widths of 256, 32, and 32 respectively. At each hidden layer, a linear rectifier activation function will be used. At the output layer, a softmax function will be used, outputting a series of probabilities for the action space. During training, we use mini-batch gradient descent. For each example in the mini-batch, we simulate 20 times and average the trajectories to obtain the baseline. Since simulations are a major bottleneck for run time, we use multiprocessing to compute simulations for each batch example in parallel.

Prior to reinforcement learning training, we initialize our network by using supervised training. It is necessary to teach the network to imitate a greedy heuristic approach such as the critical path algorithm in our case, otherwise, simulations with a completely random network result in extremely long and meaningless trajectories that do not reflect the consequences of performing a certain action.

In both supervised and reinforcement learning, we used rmsprop optimization with learning rate  $\alpha = 10^{-4}$ ,  $\rho = 0.9$ , and  $\epsilon = 10^{-9}$ .

In the MCTS implementation, since the unscaled exploration score ranges from 0 to 1, we scale it by an estimate of the makespan produced by a simulation using a greedy packing algorithm to match the exploitation score. When updating node values in backpropagation, Spear keeps track of both the average and maximum values discovered in rollouts. To determine the exploitation score when selecting children nodes, Spear will prioritize paths with a better maximum value and use the average value as a tiebreaker. In this case the UCB score can be expressed as:

$$\max_i + c\sqrt{\frac{\ln n}{n_i}} \quad (5)$$

Here  $\max_i$  denotes the maximum of relevant rollouts.

#### V. EVALUATION

We are now ready to show the performance of Spear in comparison with existing work, using an extensive array of simulations and experiments.

##### A. Experimental Settings

**Baselines:** The most important baseline is Graphene [7], since it is the state-of-the-art approach in the literature for dependency-aware task scheduling problems. We have implemented Graphene from scratch, and used it as our

baseline. We also compare Spear with other algorithms such as Tetris [6], Shortest Job First (SJF) and largest Critical Path (CP). Moreover, as Spear combines Monte Carlo Tree Search (MCTS) and deep reinforcement learning (DRL), we also present the performance of MCTS.

**Workloads:** For the simulation, the number of tasks in each directed acyclic graph (DAG) is 100. The width of the DAG is between 2 and 5. The runtime of the tasks and the resource demands of the tasks both follow normal distributions where the max task runtime is  $20t$  and the max resource demand is  $20r$ . The budget of MCTS and Spear is set to 1000 in cases without further specifications.

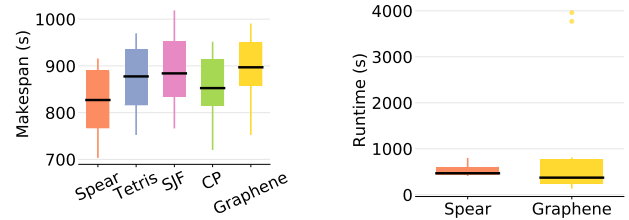
In our experiments, we adopt a workload from a production cluster, which includes 99 MapReduce jobs generated by Hive SQL query on production data. Of those jobs, the max number of map tasks and reduce tasks are 29 and 38. The mean runtime of the map tasks varies from 23 seconds to 186 seconds. The mean runtime of the reduce tasks varies from 17 seconds to 141 seconds.

**Settings:** During the training of the deep reinforcement learning (DRL), the time horizon is set to be  $20t$  and the graph size is set to 25. The total number of resource slots in the cluster is  $20r$ . Moreover, the max number of ready tasks that can be fed into the neural network at any time is 15. If there are more ready tasks, the remaining tasks will be placed in a backlog queue until the network can accommodate more ready tasks. The thresholds for the runtimes of the troublesome tasks in Graphene are 0.2, 0.4, 0.6 and 0.8. When scheduling each DAG, Graphene will try those four different parameters to determine the best scheduling results.

Overall, we will try to answer three important questions: 1) What is the performance of Spear compared to the state-of-the-art approaches? 2) What is the performance of pure MCTS? 3) Why should we adopt DRL to improve the MCTS?

## B. Simulation Results

1) *Makespan:* First, we compare the makespans of Spear and the other four algorithms. In each round, we generate 10 DAGs and each DAG has 100 tasks with various runtimes and heterogeneous resource demands for CPU and memory, respectively. The results are consistent in different rounds. The initial budget of Spear is set to 1000 and will decay, being inversely proportional to the search depth. To avoid having a budget that is too small, the minimum budget is set to 100. The results are shown in Fig. 6(a). In this figure, we can see that Spear outperforms all the baseline algorithms. The average makespans of the five algorithms are 820.1, 869.8, 890.2, 849.0 and 896.6. More specifically, it outperforms Graphene in 90% of the cases. Moreover, in 30% of the cases, it reduces the makespan of Graphene solutions by more than 100 seconds. The main reason for the improvement is that we can directly search for optimal



(a) The makespan of the algorithms. (b) The runtime of our algorithm and Graphene.

Figure 6. The performance of Spear regarding makespan and runtime.

solutions while the DRL agent guides us to more promising branches. However, the major issue in Graphene is that after partitioning the DAG into four groups, the tasks in each group are greedily sorted in descending order by runtimes, and the resource demands are not considered in this stage. On the other hand, heuristics like Tetris and shortest job first (SJF) do not consider the dependencies. A greedy algorithm prioritizing tasks with high critical paths (CP) performs better than the three other baselines in this simulation. However, there is still room for improvements as the algorithm does not utilize the resource demands and cluster state to deliver efficient packing.

We further examine the runtime of Spear in the above case in Fig. 6(b). The time measurements are conducted on a Macbook Pro laptop with 2.6GHz Intel duo core and 16 GB of main memory. Given the budget of 1000 and the minimum budget of 100, the median runtime of Spear is similar with Graphene, which is around 500 seconds. The average runtime of Spear and Graphene is around 500 seconds and 1000 seconds, respectively. In some cases, Graphene will take significantly more time to finish scheduling. For Spear, most of the time is allocated to build the MCTS search tree. The RL agent in Spear does not have a significant impact on runtime and only slows down the scheduling process by a negligible amount. In the practice we can reduce the budget to 100, which can return the scheduling results within seconds and still achieve a good result; this will be shown later. We can also use multiprocessing techniques to further reduce the time as MCTS can easily be parallelized [16].

2) *Using DRL in Spear:* We have investigated the performance of the pure MCTS approach to see why we need to integrate the DRL model into Spear. First, we show how the performance of MCTS changes with the amount of budget, which is the number of iterations to build the state tree before making each decision. Here we change the initial budget for each case and the minimum budget will always be 5 to make sure that MCTS can still work. For each setting, we generate 100 DAGs, each with 100 tasks. The average makespans of the 100 jobs with different settings are shown in Fig. 7(a). In this figure, we can clearly see that the makespan decreases with the amount of budget given to the MCTS.

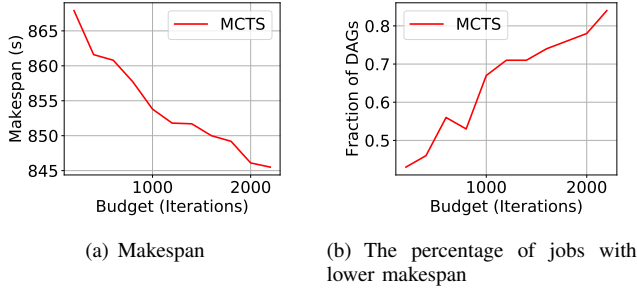


Figure 7. How the performance of MCTS changes with the budget.

To evaluate the results in each setting, we also display the percentage of jobs where MCTS will surpass Tetris. The results are in Fig. 7(b). As we can see in this figure, even when the budget is as low as 600, MCTS can outperform Tetris in 56% of cases. With a budget of 1000, MCTS performs better 67% of the time. In the case with the 2200 budget, MCTS is better 84% of the time. Therefore, we can see that the performance of MCTS steadily improves with more budget. However, the budget cannot be too small for MCTS. In this figure, when the budget is 500 or less, Tetris is the better algorithm in over half of the test examples. Later on, we will show that Spear can have surpass other algorithms with a much smaller budget by utilizing a trained policy network to guide MCTS.

Although more budget will lead to better results, the tradeoff is increased runtime overheads. To quantify the effects of budget on runtime, we also present the runtimes of the MCTS approach with different graph sizes and different amount of budget in Table. I. The server used for the time measurement is an VM instance on Google Cloud with 24 Intel Hashwell CPU core and 22 GB of main memory. In this table, we can see that the runtimes of MCTS grow with the graph size and the amount of budget. With larger graph sizes, rollouts will be longer and the search space will be deeper. With more budget, more iterations of selection, expansion, simulation, and back-propagation will take place. Therefore, both larger budget and graph size will increase the run time.

So the questions is, can DRL reach a good balance between the runtime and the performance by focusing on more promising branches? Therefore, we further examine the performance of Spear and see whether it can reduce the budget needed for the MCTS approach to decrease runtime.

Table I  
THE RUNTIME OF MCTS ONLY APPROACH ON DIFFERENT SCALES (S).

	20	30	40	50	60	70	80	90
50	0.18	0.36	0.57	0.87	1.26	1.64	2.09	2.64
100	0.33	0.58	0.89	1.29	1.75	2.13	2.81	3.36
500	1.74	2.88	4.05	5.61	6.84	8.52	9.86	11.91
1000	3.23	5.52	7.80	10.54	12.96	16.05	18.99	21.00

We further compare the performance of MCTS, Spear (with the help of deep reinforcement learning), and three

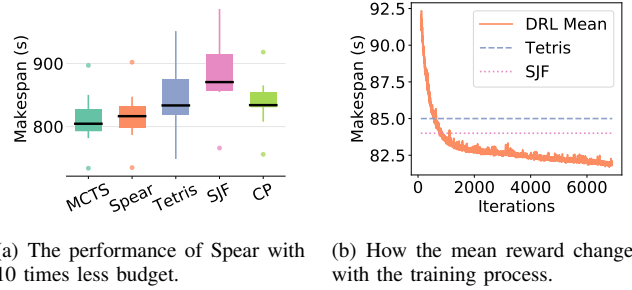


Figure 8. The DRL in Spear.

other baseline algorithms Tetris, SJF and CP in Fig. 8(a). In this experiments, we test 10 graphs and each graph has the graph size of 100. Both the task runtimes and the resource demands follow the normal distribution. The average makespans of the five algorithms over the ten graphs are 810.8, 816.7, 843.9, 884.5 and 837.9. As we can see, MCTS and Spear perform the best, followed by Tetris, CP and SJF. Among which, Spear performs similarly with the MCTS approach. However, the budget for the MCTS is 1000 and the budget for Spear is only 100. In other words, we can achieve the same level of performance with only 10% of the budget by adopting RL in Spear. In this case, the runtime of our approach is also reduced by *six times* comparing with the MCTS approach.

3) *The Learning Curve of the DRL Agent:* We display the learning curve of our DRL training in Fig. 8(b). Our training set consists of 144 randomly generated examples, each with 25 tasks. We trained the reinforcement learning agent for 7000 epochs. In each epoch, we simulate each example 20 times and average the trajectories to compute the baseline. In this figure, we show the number of epochs versus the mean makespan calculated from all trajectories over all examples. Evidently, the mean makespan which is also the negative of the reward, steadily decreases with the number of iterations. The performance finally surpasses Tetris and SJF after around 900 iterations. This trained neural network is used in all the experiments of Spear.

### C. Experimental Results

On top of simulations, we also conduct the experiments based on traces collected from production clusters running hive workloads. We first extract the task runtimes and the graph topologies of MapReduce jobs in the trace. As are only interested in the tasks with dependencies, we filtered out the jobs with no more than 5 map tasks or 5 reduce tasks. Afterwards, we obtain 99 MapReduce jobs in total for the experiment dataset.

We first show some task characteristics in Fig. 9(a) and Fig. 9(b). In the first figure, it shows the number of tasks in the map stages and reduce stages, respectively. The median number of map tasks in the jobs is 14 and the median of reduce tasks is 17. For the task runtime, the median task



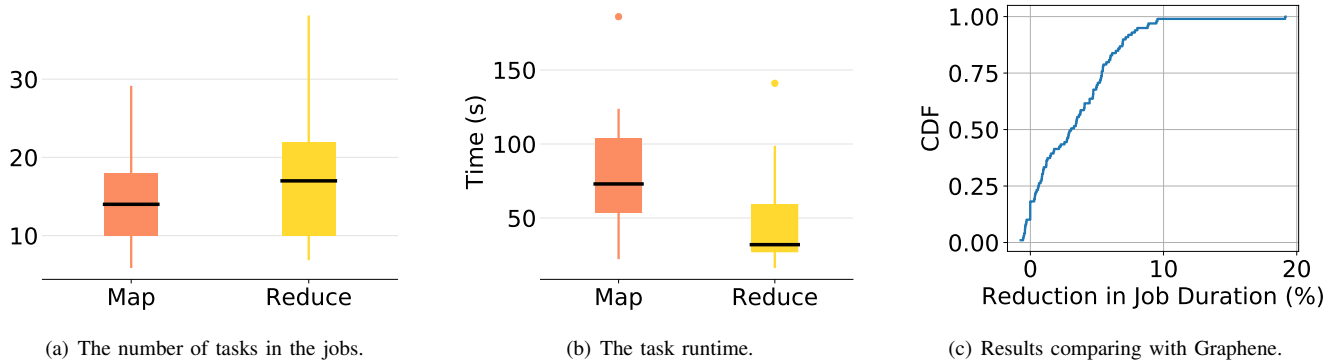


Figure 9. Results on the production workloads.

runtimes in the map stage and reduce stage are 73 and 32, respectively. As shown, both the number of tasks and the task runtimes differ significantly, providing a lot of variation for the experiments.

Here, we only set the initial budget of Spear to be 100 and the minimum budget to be 50. We take the trace as the input and obtain the results as shown in Fig. 9(c). In this figure, the reduction in job duration is calculated by  $\frac{\text{Makespan of Graphene} - \text{Makespan of Spear}}{\text{Makespan of Graphene}}$ . We can see that Spear performs no worse than Graphene in 90% of the jobs. In some cases, it can reduce the makespan of the Graphene by up to around 20%. The positive results obtained from our approach is mainly attributed to Spear’s ability to search and evaluate many scheduling decisions before committing to one action. Furthermore, the trained policy network will guide Spear’s tree exploration, guiding it towards promising paths.

## VI. RELATED WORK

In this section, we will discuss the most relevant works in task scheduling, which can be classified into several different categories depending on whether they consider dependencies and varying resource demands for multiple resources.

The most closely related paper is Graphene [7], which considers both dependencies among tasks as well as the varying resource demands for multiple resources. However, the performance of Graphene heavily relies on the parameters which define the set of *troublesome tasks*, which are not robust enough for different jobs and graph topologies. Moreover, when scheduling the troublesome tasks, they are placed in descending order of their runtimes, which may result in poor performance as it ignores the resource demands for multiple resources in this step. The dependencies and packing were also investigated in job-shop related papers [17], [18], [19]. But the problem setting of the job-shop problem is different from ours. In the job-shop problems, there is only one type of resource while Spear is designed for multiple resource dimensions.

There are lots of related works regarding dependency-aware task scheduling that don’t consider the varying re-

source demands [8], [9], [10], [20] and a survey is available in [15]. Normally, they use some node or graph related metrics to prioritize certain tasks and then schedule these tasks to the earliest available processor. As shown in our results, the performance of these approaches can be further improved because considering the varying multidimensional resource demands of tasks is critical for efficient cluster utilization.

Some related works do not consider the dependencies among the tasks [6], [21], [22], [23], [24]. These approaches schedule the tasks in a DAG level by level, which will naturally result in a sub-optimal performance. For instance, in Tetris [6], it explicitly considers the varying multidimensional resource demands of tasks in a DAG but ignores the dependencies among the tasks. We extensively compared our approach with Tetris in this paper.

Monte Carlo Tree Search (MCTS) and Deep Reinforcement Learning (DRL) have also been applied in scheduling and related decision making problems. [25] and [26] applied MCTS only for job-shop problems. Reinforcement learning was adopted in DeepRM [27] to minimize the average job slow down for a group of independent jobs where dependencies do not exist. Moreover, AlphaGo[28], AlphaGo Zero [13] and AlphaZero [11] both employed MCTS and RL to create an AI for Go, Chess and other games.

## VII. CONCLUDING REMARKS

In this paper, we propose to combine deep reinforcement learning(DRL) and Monte Carlo Tree Search to minimize the makespan of the DAGs in big data processing systems. We first design a neural network as a function approximator to represent a scheduling policy capable of choosing scheduling actions given the cluster state and ready tasks. We then train the network to minimize the makespan for the DAG scheduling problem. For MCTS, in addition to mapping it to our problem, we implemented several heuristics to greatly reduce the search space. After that, we utilize the trained DRL model by integrating the policy into the expansion step and rollout step in MCTS, replacing the default random policy. This way, the policy will select more

promising actions for minimizing the makespan and we can significantly improve the searching efficiency by focusing more on promising branches of the search tree. Both our simulation and experimental results demonstrate Spear's ability to outperform other modern heuristics, reducing the makespan of production workloads by up to 20%.

#### REFERENCES

- [1] "Hive," <https://hive.apache.org/>, accessed: 2018-05-23.
- [2] "Tez," <https://tez.apache.org/index.html>, accessed: 2018-05-23.
- [3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proc. USENIX NSDI*, 2012.
- [4] M. Mastrolilli and O. Svensson, "(Acyclic) Job Shops are Hard to Approximate," in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, 2008, pp. 583–592.
- [5] —, "Improved Bounds for Flow Shop Scheduling," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2009, pp. 677–688.
- [6] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource Packing for Cluster Schedulers," in *Proc. ACM SIGCOMM*, 2014.
- [7] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters," in *Proc. of OSDI*, 2016.
- [8] T. L. Adam, K. M. Chandy, and J. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.
- [9] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors," in *Second IEEE International Symposium on Parallel Architectures, Algorithms, and Networks.*, 1996, pp. 207–213.
- [10] A. Gerasoulis and T. Yang, *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors*. Rutgers University, Department of Computer Science, Laboratory for Computer Science Research, 1991.
- [11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv preprint arXiv:1712.01815*, 2017.
- [12] Theano Development Team, "Theano: A Python Framework for Fast Computation of Mathematical Expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: <http://arxiv.org/abs/1605.02688>
- [13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the Game of Go without Human Knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.
- [14] R. S. Sutton, A. G. Barto *et al.*, *Reinforcement Learning: An Introduction*. MIT press, 1998.
- [15] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.
- [16] G. M.-B. Chaslot, M. H. Winands, and H. J. van Den Herik, "Parallel Monte-Carlo Tree Search," in *International Conference on Computers and Games*. Springer, 2008, pp. 60–71.
- [17] A. Czumaj and C. Scheideler, "A New Algorithm Approach to the General Lovász Local Lemma with Applications to Scheduling and Satisfiability Problems," in *Proc. of the thirty-second annual ACM symposium on Theory of computing*, 2000, pp. 38–47.
- [18] L. A. Goldberg, M. Paterson, A. Srinivasan, and E. Sweedyk, "Better Approximation Guarantees for Job-Shop Scheduling," *SIAM Journal on Discrete Mathematics*, vol. 14, no. 1, pp. 67–92, 2001.
- [19] D. B. Shmoys, C. Stein, and J. Wein, "Improved Approximation Algorithms for Shop Scheduling Problems," *SIAM Journal on Computing*, vol. 23, no. 3, pp. 617–632, 1994.
- [20] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996.
- [21] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica, "Low Latency Geo-Distributed Data Analytics," in *Proc. ACM SIGCOMM*, 2015.
- [22] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "CLARINET: WAN-Aware Optimization for Analytics Queries," in *Proc. USENIX OSDI*, 2016.
- [23] C.-C. Hung, L. Golubchik, and M. Yu, "Scheduling Jobs Across Geo-distributed Datacenters," in *Proc. ACM SoCC*, 2015.
- [24] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, and M. Zhang, "Wide-Area Analytics with Multiple Resources," in *Proc. of the ACM Thirteenth EuroSys Conference*, 2018.
- [25] T. P. Runarsson, M. Schoenauer, and M. Sebag, "Pilot, Rollout and Monte Carlo Tree Search Methods for Job Shop Scheduling," in *Learning and Intelligent Optimization*. Springer, 2012, pp. 160–174.
- [26] T.-Y. Wu, I.-C. Wu, and C.-C. Liang, "Multi-Objective Flexible Job Shop Scheduling Problem Based on Monte-Carlo Tree Search," in *Technologies and Applications of Artificial Intelligence (TAAI), 2013 Conference on*. IEEE, pp. 73–78.
- [27] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource Management with Deep Reinforcement Learning," in *Proc. of the 15th ACM Workshop on Hot Topics in Networks*, 2016.
- [28] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *nature*, vol. 529, no. 7587, p. 484, 2016.