# Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning

**Cong Zhang**[1,*], **Wen Song**[2,*], **Zhiguang Cao**[3,†], **Jie Zhang**[1], **Puay Siew Tan**[4], **and Chi Xu**[4]

[1]Nanyang Technological University
[2]Institute of Marine Science and Technology, Shandong University, China
[3]National University of Singapore
[4]Singapore Institute of Manufacturing Technology, A*STAR
`cong030@e.ntu.edu.sg, wensong@email.sdu.edu.cn, zhiguangcao@outlook.com`
`zhangj@ntu.edu.sg, {pstan, cxu}@simtech.a-star.edu.sg`

## Abstract

Priority dispatching rule (PDR) is widely used for solving real-world Job-shop scheduling problem (JSSP). However, the design of effective PDRs is a tedious task, requiring a myriad of specialized knowledge and often delivering limited performance. In this paper, we propose to automatically learn PDRs via an end-to-end deep reinforcement learning agent. We exploit the disjunctive graph representation of JSSP, and propose a Graph Neural Network based scheme to embed the states encountered during solving. The resulting policy network is size-agnostic, effectively enabling generalization on large-scale instances. Experiments show that the agent can learn high-quality PDRs from scratch with elementary raw features, and demonstrates strong performance against the best existing PDRs. The learned policies also perform well on much larger instances that are unseen in training.

## 1 Introduction

Job-shop scheduling problem (JSSP) is a well-known combinatorial optimization problem in computer science and operations research, and is ubiquitous in many industries such as manufacturing and transportation [1, 2]. In JSSP, a number of *jobs* with predefined processing constraints (e.g. the operations are processed in order by their eligible machines) are assigned to a set of heterogeneous *machines*, to achieve the desired objective such as minimizing the makespan, flowtime, or tardiness. Due to its NP-hardness, finding exact solutions to JSSP is often impractical [3, 4], while efficiency in practice usually relies on heuristics [5, 6] or approximate methods [7].

Priority dispatching rule (PDR) [6] is a heuristic method that is widely used in real-world scheduling systems. Compared with complicated optimization methods such as mathematical programming and metaheuristics, PDR is computationally fast, intuitive and easy to implement, and naturally capable of handling uncertainties that are ubiquitous in practice [8]. Motivated by these advantages, a large number of PDRs for JSSP have been proposed in the literature [9]. However, it is commonly accepted that designing an effective PDR is very costly and time-consuming, requiring substantial domain knowledge and trial-and-error especially for complex JSSP. Moreover, performance of a PDR often varies drastically on different instances [10]. Therefore, a natural question to ask is: can we *automate* the process of designing PDR, such that it performs well on a class of JSSP instances sharing common characteristics? A number of recent works on learning algorithms for other types of combinatorial optimization problems (COPs) (see [11] for a survey) show that deep reinforcement

---

[*]Both authors contributed equally.

[†]Corresponding Author.

learning (DRL) could be an ideal technique for this purpose. However, for complex scheduling problems such as JSSP which differs structurally from other COPs and received much less attention, existing methods cannot apply [11], and it remains challenging to design effective representation and learning mechanism.

In this paper, we propose a novel DRL based method to automatically learn strong and robust PDRs for solving JSSP. Specifically, we first present a Markov Decision Process (MDP) formulation of PDR based scheduling, where the states are captured by leveraging the *disjunctive graph* representation of JSSP. Such representation effectively integrates the operation dependencies and machine status, and provides critical information for scheduling decisions. Then, we propose a Graph Neural Network (GNN) based scheme with an efficient computation strategy to encode the nodes in the disjunctive graphs to fixed dimensional embeddings. Based on this scheme, we design a size-agnostic policy network that can process JSSP instances with arbitrary size, which effectively enables training on small-sized instances and generalizing to large-scale ones. We train the network using a policy gradient algorithm to obtain high-quality PDRs, without the need of supervision. Extensive experiments on generated instances and standard benchmarks show that, the PDRs trained by our policy significantly outperform existing manually designed ones, and generalize reasonably well to instances that are much larger than those used in training.

## 2   Related Work

Lately, the idea of applying deep (reinforcement) learning as an end-to-end solution to combinatorial optimization problems has been widely explored. Most of them focus on solving routing problems (e.g. travelling salesman problem) [12, 13, 14, 15, 16, 17], graph optimization problems [18, 19], and the satisfiability problem (SAT) [20, 21, 22]. In contrast, scheduling problems which have numerous real-world applications, are relatively unexplored, especially for JSSP.

Several existing works study simple job scheduling problems, in which jobs as elementary tasks without internal operation dependencies that are essential to JSSP. In [23], a DRL agent is proposed to learn job scheduling policies for a compute cluster. A 2-D image based state representation scheme is used to capture the status of resources and jobs. In [24], DRL is employed to learn local search heuristics for solving a similar problem, where the states are represented by a Directed Acyclic Graph (DAG) describing the temporal relations among jobs in the corresponding schedule. A major limitation in these works is that, the state representation is hard-bounded by some factors (e.g. look ahead horizon, size of job queue or slot), and is not scalable to arbitrary numbers of jobs and machines (resources). This limitation is partially alleviated in [25], which also employs an image based representation but with a transfer learning method to reconstruct the trained policies on problems with different sizes. Nevertheless, policy transfer is still relatively costly and inconvenient. In contrast, our method is fully size-agnostic and the trained policy can be directly applied to solve larger problems without the need of transfer.

In [26], a DRL method is proposed for task scheduling in a cloud computing environment. GNN is used to extract embedding of each task represented as a DAG, and the policy network can scale to arbitrary number of tasks. However, the underlying problem is not JSSP and the task DAG only describes the required temporal dependencies among its subtasks. The resource information is encoded as node features, hence the number of resources is hard bounded. In contrast, our GNN performs embedding on the disjunctive graph with directed disjunctive arcs reflecting processing order on each machine, and is size-agnostic in terms of both jobs and machines. Moreover, the topology of task DAGs in [26] is static, while the disjunctive graph in our setting is dynamically evolving and highly correlated with the decisions made at each step. Similar to [26], a RL method combined with GNNs is proposed to accelerating computation in distributed system [27]. Other examples of applying GNNs to solve real life scheduling problem includes [28], where they adopted an imitation learning algorithm for solving robotic scheduling problems in manufacturing.

Research on standard JSSP is rather sparse. In [29], an imitation learning method is proposed to learn dispatching rules for JSSP, where optimal solutions to the training instances are labelled using a MIP solver. However, due to the complexity of JSSP, finding enough optimal solutions to large-scale instances for training is impractical, and only instances up to $10\ jobs \times 10\ machines$ are considered, which significantly limits the applicability. In [30], a Deep Q Network [31] based method is proposed to solve JSSP, which learns to select PDR for each machine from a pool of candidates. Though the
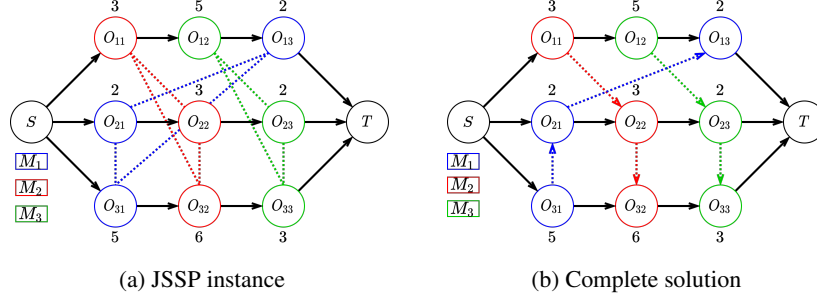
(a) JSSP instance        (b) Complete solution

Figure 1: **Disjunctive graph representation.** (a) represents a $3 \times 3$ JSSP instance. Black arrows are conjunctive arcs, and dotted lines are disjunctive arcs grouped into machine cliques with different colors. (b) is a complete solution, where all disjunctive arcs are assigned with directions.

methods in [29, 30] show better performance than traditional PDRs, they are not end-to-end, and rely on manual features to describe scheduling states, ignoring the underlying graph structure of JSSP. In contrast, our method can extract informative knowledge from low-level raw features based on the disjunctive graph, and directly makes decisions without the need of pre-defined candidate PDRs.

## 3 Preliminaries

**Job-Shop Scheduling Problem.** A standard JSSP instance consists of a set of jobs $\mathcal{J}$ and a set of machines $\mathcal{M}$. Each job $J_i \in \mathcal{J}$ must go through $n_i$ machines in $\mathcal{M}$ in a specific order denoted as $O_{i1} \to ... \to O_{in_i}$, where each element $O_{ij}$ ($1 \le j \le n_i$) is called an operation of $J_i$ with a processing time $p_{ij} \in \mathbb{N}$. The binary relation $\to$ also refers to precedence constraint. Each machine can only process one job at a time, and preemption is not allowed. To solve a JSSP instance, we need to find a *schedule*, i.e. starting time $S_{ij}$ for each operation $O_{ij}$, such that the *makespan* denoted as $C_{\max} = \max_{i,j}\{C_{ij} = S_{ij} + p_{ij}\}$ is minimized and all the constraints are satisfied. The size of a JSSP instance is denoted as $|\mathcal{J}| \times |\mathcal{M}|$.

**Disjunctive graph.** It is well-known that a JSSP instance can also be represented as a disjunctive graph [32]. Let $\mathcal{O} = \{O_{ij}|\forall i,j\} \cup \{S, T\}$ be the set of all operations, where $S$ and $T$ are dummy ones denoting the start and terminal with zero processing time. Then a disjunctive graph $G = (\mathcal{O}, \mathcal{C}, \mathcal{D})$ is a mixed graph with $\mathcal{O}$ being its vertex set. In particular, $\mathcal{C}$ is a set of directed arcs (conjunctions) representing the precedence constraints between operations of the same job; and $\mathcal{D}$ is a set of undirected arcs (disjunctions), each of which connects a pair of operations requiring the same machine for processing. Consequently, finding a solution to a JSSP instance is equivalent to fixing the direction of each disjunction, such that the resulting graph is a DAG [33]. An example of disjunctive graph for a JSSP instance and its solution are depicted in Figure 1(a) and (b), respectively.

## 4 Method

In this section, we present the rationale of our method. We first formulate Markov Decision Process model of PDR based scheduling. Then, we design an efficient method to represent the scheduling policy based on Graph Neural Network, followed by the introduction of training algorithm.

### 4.1 Markov Decision Process Formulation

The PDR based method solves a JSSP instance using $|\mathcal{O}|$ steps of consecutive decisions. At each step, a set of eligible operations (i.e. those whose precedent operation has been scheduled) are identified first. Then, a specific PDR is applied to compute a priority index for each eligible operation, and the one with the highest priority is selected for scheduling (or dispatching). However, solely deciding which operation to dispatch is not sufficient, as we also need to choose a suitable start time for it. In order to build tight schedules, it is sensible to place the operation as early as possible on the corresponding machine [29]. Once all operations are dispatched, a complete schedule is generated.

3

Traditional manually designed PDRs compute the priority index based on the operation features. For example, the widely used Shortest Processing Time (SPT) rule selects from a set of operations the one with the smallest $p_{ij}$. In this paper, we employ DRL to automatically generate high-quality PDRs. As mentioned above, solving a JSSP instance can be viewed as a task of determining the direction of each disjunction. Therefore, we consider the dispatching decisions made by PDRs as actions of changing the disjunctive graph, and formulate the underlying MDP model as follows.

**State.** The state $s_t$ at decision step $t$ is a disjunctive graph $G(t) = (\mathcal{O}, \mathcal{C} \cup \mathcal{D}_u(t), \mathcal{D}(t))$ representing the current status of solution, where $\mathcal{D}_u(t) \subseteq \mathcal{D}$ contains all the (directed) disjunctive arcs that have been assigned a direction till $t$, and $\mathcal{D}(t) \subseteq \mathcal{D}$ includes the remaining ones. The initial state $s_0$ is the disjunctive graph representing the original JSSP instance, and the terminal state $s_T$ is a complete solution where $\mathcal{D}(T) = \emptyset$, i.e. all disjunctive arcs have been assigned a direction. For each node $O \in \mathcal{O}$, we record two features: 1) a binary indicator $I(O, s_t)$ which equals to 1 only if $O$ is scheduled in $s_t$, and 2) an integer $C_{LB}(O, s_t)$ which is the lower bound of the estimated time of completion (ETC) of $O$ in $s_t$. Note that for the scheduled operation, this lower bound is exactly its completion time. For the unscheduled operation $O_{ij}$ of job $J_i$, we recursively calculate this lower bound as $C_{LB}(O_{ij}, s_t) = C_{LB}(O_{i,j-1}, s_t) + p_{ij}$ by only considering the precedence constraints from its predecessor, i.e. $O_{i,j-1} \rightarrow O_{ij}$, and $C_{LB}(O_{ij}, s_t) = r_i + p_{ij}$ if $O_{ij}$ is the first operation of $J_i$ where $r_i$ is the release time of $J_i$.

**Action.** An action $a_t \in A_t$ is an eligible operation at decision step $t$. Given that each job can only have at most one operation ready at $t$, the maximum size of action space is $|\mathcal{J}|$, which depends on the instance being solved. During solving, $|A_t|$ becomes smaller as more jobs are completed.

**State transition.** Once PDR determines an operation $a_t$ to dispatch next, we first find the earliest feasible time period to allocate $a_t$ on the required machine. Then, we update the directions of the disjunctive arcs of that machine based on the current temporal relations, and engenders a new disjunctive graph as the new state $s_{t+1}$. An example is given in Figure 2, where action $a_4 = O_{32}$ is chosen at state $s_4$ from action space $\{O_{12}, O_{23}, O_{32}\}$. On the required machine $M_2$, we find that $O_{32}$ can be allocated in the time period before the already scheduled $O_{22}$, therefore the direction of the disjunctive arc between $O_{22}$ and $O_{32}$ is determined as $O_{32} \rightarrow O_{22}$, as shown in the new state $s_5$. Note, the starting time of $O_{22}$ is changed from 7 to 11 since $O_{32}$ is scheduled before $O_{22}$.

**Reward.** The goal is to learn to dispatch step by step such that the makespan is minimized. To this end, we design the reward function $R(s_t, a_t)$ as the quality difference between the partial solutions corresponding to the two states $s_t$ and $s_{t+1}$, i.e. $R(a_t, s_t) = H(s_t) - H(s_{t+1})$, where $H(\cdot)$ is the quality measure. Here we define it as the lower bound of the makespan $C_{\max}$, computed as $H(s_t) = \max_{i,j}\{C_{LB}(O_{ij}, s_t)\}$. For the terminal state $s_{|\mathcal{O}|}$, clearly we have $H(s_{|\mathcal{O}|}) = C_{\max}$ since all operations are scheduled. Hence when the discount factor $\gamma = 1$, the cumulative reward is $\sum_{t=0}^{|\mathcal{O}|} R(a_t, s_t) = H(s_0) - C_{\max}$. Given that $H(s_0)$ is constant, maximizing the cumulative reward coincides with minimizing the makespan.



Figure 2: **Example of state transition.** Orange nodes are operations already scheduled, and the grey node is operation selected to be scheduled at current state. Integers in bracket are starting time of scheduled operations, where unscheduled operations have unknown starting time (denoted as ?).

**Policy.** For state $s_t$, a stochastic policy $\pi(a_t|s_t)$ outputs a distribution over the actions in $A_t$. If traditional PDRs are employed as policy, then the distribution is one-hot, and the action with the highest priority has probability 1.

## 4.2 Parameterizing the Policy

The disjunctive graph in the above MDP formulation provides a holistic view of the scheduling states that comprehensively contains numerical and structural information such as operation processing time, precedence constraints, and processing order on each machine. By extracting all the state information embedded in disjunctive graphs, effective dispatching is viable. This motivates us to
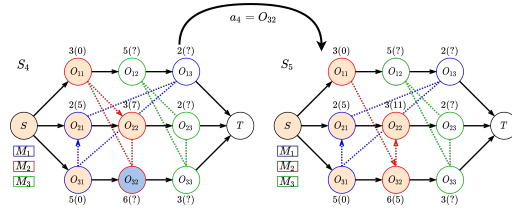
parameterize the stochastic policy $\pi(a_t|s_t)$ as a graph neural network with trainable parameter $\theta$, i.e. $\pi_\theta(a_t|s_t)$, which enables learning strong dispatching rules and size-agnostic generalization.

**Graph embedding.** Graph Neural Networks (GNN) [34] are a family of deep neural networks that can learn representation of graph-structured (non-euclidean) data, which has many applications in real life [35, 36]. It extracts feature embedding of each node in an iterative and non-linear fashion.In this paper, we adopt the Graph Isomorphism Network (GIN) [37], which is a recent GNN variant and proved to have strong discriminative power. Particularly, given a graph $\mathcal{G} = (V, E)$, GIN performs $K$ iterations of updates to compute a $p$-dimensional embeddings for each node $v \in V$, and the update at iteration $k$ is expressed as follows,

$$h_v^{(k)} = MLP_{\theta_k}^{(k)} \left( \left(1 + \epsilon^{(k)}\right) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)} \right), \tag{1}$$

where $h_v^{(k)}$ is the representation of node $v$ at iteration $k$ and $h_v^{(0)}$ refers to its raw features for input, $MLP_{\theta_k}^{(k)}$ is a Multi-Layer Perceptron (MLP) with parameter $\theta_k$ for iteration $k$ followed by batch normalization [38], $\epsilon$ is an arbitrary number that can be learned, and $\mathcal{N}(v)$ is the neighbourhood of $v$. After $K$ iterations of updates, a global representation for the entire graph can be obtained using a pooling function $L$ that takes as input the embeddings of all nodes and output a $p$-dimensional vector $h_\mathcal{G} \in \mathbb{R}^p$ for $\mathcal{G}$. Here we use average pooling, i.e. $h_\mathcal{G} = L(\{h_v^K : v \in V\}) = 1/|V| \sum_{v \in V} h_v^K$.

GIN is originally proposed for undirected graphs in [37]. However, in our case, the disjunctive graph $G(t)$ correlative to each state $s_t$ is a mixed graph with directed arcs, which describe critical characteristics such as the precedence constraints and operation sequences on machines. Therefore, we need to generalize GIN to support disjunctive graphs. A natural and straightforward strategy for this is to replace each undirected arc in $G(t)$ by two directed ones connecting the same nodes with opposite directions, resulting in a fully directed graph denoted as $G_D(t)$ [39, 40]. Then, the neighbourhood of node $v$ in Eq. (1) can be defined as $\mathcal{N}(v) = \{u|(u, v) \in E(G_D(t))\}$, where $E(\cdot)$ is the arc set of a graph, i.e. $\mathcal{N}(v)$ contains all incoming neighbours of $v$. In this way, GIN is able to operate on $G_D(t)$. An illustration is given in Figure 3(a), which is the directed version of the state $s_4$ in Figure 2. The transition to $s_5$ can be naturally achieved by removing directed arcs $(O_{32}, O_{11})$ and $(O_{22}, O_{32})$ in Figure 3(b) since the direction of the corresponding disjunctive arcs should be $O_{11} \rightarrow O_{32}$ and $O_{32} \rightarrow O_{22}$.

However, a major limitation of the above "removing-arc" strategy is that, it maintains two directed arcs for each disjunctive arc, making $G_D(t)$ too dense to be efficiently processed by GIN. This is more severe for the initial states, where for each machine the operations requiring it forms a clique that is fully connected with $|\mathcal{J}|(|\mathcal{J}| - 1)/2$ arcs in $G(t)$. To resolve this issue, we propose to "add" arcs rather than removing them, where the undirected disjunctive arcs are neglected. More specifically, we use $\tilde{G}_D(t) = (\mathcal{O}, \mathcal{C} \cup \mathcal{D}_u(t))$ as an approximation of $G_D(t)$. Along with the scheduling process, $\mathcal{D}_u(t)$ becomes larger since more directed arcs will be added to it. An illustration of this "adding-arc" strategy is shown in Figure 3(c). Clearly, this strategy leads to much sparser graphs for state representation. Finally, we define the raw features for each node $O \in \mathcal{O}$ at $s_t$ as a 2-dimensional vector $h_O^{(0)}(s_t) = (I(O, s_t), C_{LB}(O, s_t))$, and denote the node and graph embedding obtained after $K$ iterations as $h_O^{(K)}(s_t)$ and $h_\mathcal{G}(s_t)$, respectively.

**Action selection.** To select an action $a_t$ at $s_t$, we further process the extracted graph embeddings $h_O^{(K)}$ with an action selection network. In doing so, we expect to produce a probability distribution over action space from which $a_t$ can be sampled. Specifically, we first adopt an MLP to obtain a scalar sore $scr(a_t) = MLP_{\theta_\pi}([h_{a_t}^{(K)}, h_\mathcal{G}(s_t)])$ for each $a_t$, where $[,]$ means concatenation. Then, a softmax function is applied to output a distribution $P(a_t)$ over computed scores. We sample actions based on $P(a_t)$ for training. During testing, we greedily pick $a_t$ with the maximum probability.

*Remark.* Our design of policy network has several advantages. First, unlike previous works [23, 24, 25], it is not hard bounded by the instance size ($|\mathcal{J}|$ and $|\mathcal{M}|$), since all parameters are shared across all nodes in the graph. This size-agnostic property effectively enables generalization to instances of different sizes without re-training or knowledge transferring. Second, it can potentially deal with more complex environments with dynamics and uncertainty such as job arriving on-the-fly and random machine breakdown, by adding or removing certain nodes and/or arcs from the disjunctive graphs. Finally, our model could be extended to other shop scheduling problems (e.g. flow-shop and
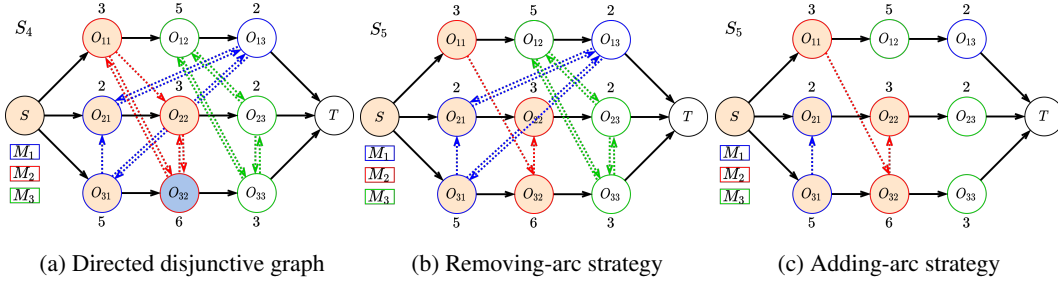
Figure 3: **Fully directed disjunctive graph, removing-arc strategy, and adding-arc strategy.** (a) is a fully directed disjunctive graph by replacing each undirected disjunctive arc with two opposite directed arcs. (b) shows the removing-arc strategy. The directed arc conflicting with the current decision is removed from the graph, i.e. arcs $(O_{32}, O_{11})$ and $(O_{22}, O_{32})$. (c) shows the adding-arc strategy. The directed arc following the current decision is added to the graph, i.e. arcs $(O_{11}, O_{32})$ and $(O_{32}, O_{22})$.

open-shop) since they can be represented by disjunctive graphs [41, 42]. While the first advantage will be demonstrated in the experiments, we plan to further explore the latter two in the future.

### 4.3 Learning Algorithm

We train the policy network using Proximal Policy Optimization (PPO) [43], which is an actor-critic algorithm. The actor refers to the policy network $\pi_\theta$ described above. The critic $v_\phi$ shares the same GIN network with the actor, and uses an MLP $MLP_{\theta_v}$ that takes input of $h_{\mathcal{G}}(s_t)$ to output a scalar to estimate the cumulative rewards at $s_t$. To boost learning, we follow the principle of PPO and generate $N$ independent trajectories, and update network parameters w.r.t cumulative gradient of $N$ estimates. Details of the training algorithm are provided in the Supplementary Material.

## 5 Experiment

We present the experimental results in this section. The evaluations are performed both on generated instances and public JSSP benchmarks.

**Datasets.** We evaluate our method on instances of various sizes. Specifically, $6 \times 6$, $10 \times 10$, $15 \times 15$, $20 \times 20$, and $30 \times 20$ instances are generated following the well-known Taillard's method [44] for training and testing. Furthermore, we demonstrate strong generalization of our method by directly testing on much larger instances with sizes $50 \times 20$ and $100 \times 20$ generated also following [44]. We also perform experiments on well-known public JSSP benchmarks, including Taillard's instances [45] generated following [44] and the DMU instances [46]. The range of operation processing times in DMU instances doubles that of Taillard's ones.

**Models and configurations.** We use fixed hyperparameters for training. For each problem size, we train the policy network for 10000 iterations, each of which contains 4 independent trajectories (i.e. instances). The model is validated on 100 instances generated on-the-fly and fixed during training. All raw features are normalized to the same scale. For the GIN layers (Eq. (1)) shared by $\pi$ and $v$, we set the number of iterations $K = 2$. We set $\epsilon$ to 0 following [37]. Each $MLP_{\theta_k}^{(k)}$ in the GIN layer has 2 hidden layers with hidden dimension 64. The action selection network $MLP_{\theta_\pi}$ and state value prediction network $MLP_{\theta_v}$ both have 2 hidden layers with hidden dimension 32. For PPO, we set the epochs of updating network to 1, the clipping parameter $\epsilon_{PPO}$ to 0.2, and the coefficient for policy loss, value function, and entropy to 2, 1, and 0.01, respectively. For training, we set the discount factor $\gamma$ to 1, and use the Adam optimizer with constant learning rate $lr = 2 \times 10^{-5}$. Other parameters follow the default settings in PyTorch [47]. The hardware we use is a machine with Intel Core i9-10940X CPU and a single Nvidia GeForce 2080Ti GPU. Our code is available.[1]

---

[1]https://github.com/zcajiayin/L2D

| Size | | SPT | MWKR | FDD/MWKR | MOPNR | Ours | Opt. Rate(%) |
|------|------|------|------|------|------|------|------|
| $6 \times 6$ | Obj. | 691.95 | 656.95 | 604.64 | 630.19 | **574.09** | |
| | Gap | 42.0% | 34.6% | 24.0% | 29.2% | **17.7%** | 100% |
| | Time(s) | **0.012** | **0.012** | **0.012** | **0.012** | 0.061 | |
| $10 \times 10$ | Obj. | 1210.98 | 1151.41 | 1102.95 | 1101.08 | **988.58** | |
| | Gap | 50.0% | 42.6% | 36.6% | 36.5% | **22.3%** | 100% |
| | Time(s) | **0.037** | 0.039 | 0.039 | **0.037** | 0.176 | |
| $15 \times 15$ | Obj. | 1890.91 | 1812.13 | 1722.73 | 1693.33 | **1504.79** | |
| | Gap | 59.2% | 52.6% | 45.1% | 42.6% | **26.7%** | 99% |
| | Time(s) | 0.113 | 0.116 | 0.117 | **0.112** | 0.435 | |
| $20 \times 20$ | Obj. | 2519.8 | 2469.19 | 2328.15 | 2263.68 | **2007.76** | |
| | Gap | 62.0% | 58.6% | 49.6% | 45.5% | **29.0%** | 4% |
| | Time(s) | 0.306 | 0.312 | 0.312 | **0.305** | 0.932 | |
| $30 \times 20$ | Obj. | 3208.69 | 3080.11 | 2883.88 | 2809.62 | **2508.27** | |
| | Gap | 65.3% | 58.7% | 48.6% | 44.7% | **29.2%** | 12% |
| | Time(s) | 0.721 | 0.731 | 0.731 | **0.720** | 1.804 | |

Table 1: **Results on instances of small and medium sizes.** "Opt. Rate": rate of instances for which OR-Tools returns optimal solution.

**Baselines.** There are hundreds of PDRs proposed for JSSP in the literature with various performance and we can not compare with them exhaustively. Therefore, we select four traditional PDRs based on their performance reported in [9], including *Shortest Processing Time* (SPT), *Most Work Remaining* (MWKR), *Most Operations Remaining* (MOPNR), and *minimum ratio of Flow Due Date to Most Work Remaining* (FDD/MWKR). SPT is one of the most widely used PDRs in research and industry, while the other three are top-performing PDRs on Taillard's benchmark as reported in [9]. Specifically, FDD/MWKR is newly developed in [9]. All baselines are implemented in Python, and the details are introduced in the Supplementary Material. For the generated instances, solutions of all methods are benchmarked with those obtained by Google OR-Tools [48], a mature and widely used exact solver based on constraint programming, with time limit of 3600 seconds for each instance. For the public benchmarks, we use the best-known solutions from the literature.[2]

## 5.1 Results on Generated Instances

We first perform training and testing on the generated instances of small to medium sizes ($6 \times 6$, $10 \times 10$, $15 \times 15$, $20 \times 20$, $30 \times 20$). For each size, we generate 100 instances randomly and report the average objective, gap to the OR-Tools solutions, and computational time of our method and baselines. The results are summarized in Table 1. We can observe from this table that the PDR learned by our method consistently outperforms all baseline PDRs by a large margin regarding all instance sizes. Performance of baseline PDRs deteriorates quickly with the increase of instance size, whereas PDR learned by our method performs stable and relatively well especially on larger instances. In terms of computational efficiency, though the inference time of our method is relatively longer than the traditional PDRs, it is still quite acceptable especially considering the significant performance boost. Compared with OR-Tools which takes 3600s on the vast majority of $20 \times 20$ and $30 \times 20$ instances, our method is much more efficient. Based on the above observations, we can conclude that our method is able to train high-quality PDRs from scratch, without the need of supervision.

Next, we evaluate the performance of our policy in terms of generalizing to large instances. More specifically, we directly use the policies trained on $20 \times 20$ and $30 \times 20$ instances to solve $50 \times 20$ and $100 \times 20$ instances. The results are summarized in Table 2, where the result for each instance size is averaged over 100 random instances. As shown in this table, both our policies trained on much smaller instances perform reasonably well on these large sized ones, and deliver solutions that are much better than those of the traditional PDRs. This observation shows that our method is able to extract knowledge from small sized instances that is also useful in solving large-scale ones, which is a desirable property for practical applications. Meanwhile, our method is computationally efficient and can provide high-quality solution for the largest instance within 30s. We can also observe that

---

[2]The best solutions for Taillard's and DMU instances can be found in http://optimizicer.com/TA.php and http://jobshop.jjvh.nl/, respectively.

| Size | | SPT | MWKR | FDD/MWKR | MOPNR | Ours (20 × 20) | Ours (30 × 20) | Opt. Rate |
|---|---|---|---|---|---|---|---|---|
| 50 × 20 | Obj. | 4469.8 | 4273.08 | 3993.45 | 3859.14 | 3581.5 | **3522.5** | |
| | Gap | 54.9% | 48.1% | 38.4% | 33.7% | 24.1% | **22.1%** | 48% |
| | Time(s) | **2.504** | 2.523 | 2.524 | **2.504** | 4.917 | 4.872 | |
| 100 × 20 | Obj. | 7516.12 | 7069.72 | 6658.17 | 6385.32 | 6175.01 | **6088.68** | |
| | Gap | 35.1% | 27.0% | 19.6% | 14.7% | 10.9% | **9.4%** | 2% |
| | Time(s) | 16.661 | 16.694 | 16.723 | **16.625** | 27.869 | 28.616 | |

Table 2: **Generalization results on large-sized instances.** "Opt. Rate": rate of instances that OR-Tools returns optimal solution.

| Size | | SPT | MWKR | FDD/MWKR | MOPNR | Ours | Ours (20 × 20) | Ours (30 × 20) | Opt. Rate |
|---|---|---|---|---|---|---|---|---|---|
| Ta 15 × 15 | Obj. | 1902.6 | 1927.5 | 1808.6 | 1782.3 | **1547.4** | . | . | |
| | Gap | 54.8% | 56.7% | 47.1% | 45.0% | **26.0%** | . | . | 100% |
| | Time(s) | **0.111** | 0.115 | 0.117 | **0.111** | 0.447 | . | . | |
| Ta 20 × 15 | Obj. | 2253.6 | 2190.7 | 2054 | 2015.8 | **1774.7** | . | . | |
| | Gap | 65.2% | 60.7% | 50.6% | 47.7% | **30.0%** | . | . | 90% |
| | Time(s) | **0.178** | 0.183 | 0.183 | **0.178** | 0.624 | . | . | |
| Ta 20 × 20 | Obj. | 2655.8 | 2518.6 | 2387.2 | 2309.9 | **2128.1** | . | . | |
| | Gap | 64.2% | 55.7% | 47.6% | 42.8% | **31.6%** | . | . | 30% |
| | Time(s) | 0.305 | 0.311 | 0.311 | **0.304** | 0.937 | . | . | |
| Ta 30 × 15 | Obj. | 2888.4 | 2728 | 2590.8 | 2601.3 | **2378.8** | . | . | |
| | Gap | 61.6% | 52.6% | 45.0% | 45.6% | **33.0%** | . | . | 70% |
| | Time(s) | **0.383** | 0.390 | 0.392 | **0.383** | 1.114 | . | . | |
| Ta 30 × 20 | Obj. | 3234.7 | 3193.3 | 3045 | 2888.1 | **2603.9** | . | . | |
| | Gap | 66.0% | 63.9% | 56.3% | 48.2% | **33.6%** | . | . | 0% |
| | Time(s) | 0.722 | 0.730 | 0.731 | **0.720** | 1.799 | . | . | |
| Ta 50 × 15 | Obj. | 4194.7 | 3907.8 | 3736.3 | 3608 | . | 3430.2 | **3393.8** | |
| | Gap | 51.4% | 40.9% | 34.8% | 30.1% | . | 23.7% | **22.4%** | 100% |
| | Time(s) | **1.208** | 1.221 | 1.226 | 1.209 | . | 2.700 | 2.696 | |
| Ta 50 × 20 | Obj. | 4532.2 | 4375.1 | 4022.1 | 3920 | . | 3611.8 | **3593.9** | |
| | Gap | 59.5% | 53.9% | 41.5% | 37.9% | . | 27.0% | **26.5%** | 100% |
| | Time(s) | **2.507** | 2.527 | 2.523 | 2.508 | . | 4.856 | 4.883 | |
| Ta 100 × 20 | Obj. | 7564.6 | 7128.8 | 6620.7 | 6452.3 | . | 6255 | **6097.6** | |
| | Gap | 41.0% | 32.9% | 23.4% | 20.2% | . | 16.6% | **13.6%** | 100% |
| | Time(s) | 16.652 | 16.686 | 16.745 | **16.647** | . | 28.239 | 28.328 | |

Table 3: **Results on Taillard's benchmark.** "Opt. Rate": rate of instances with optimal solution.

our $20 \times 20$ policy performs only slightly worse than the $30 \times 20$ one, indicating a relatively robust generalization performance.

## 5.2 Results on Public Benchmarks

We first perform experiments on the 80 Taillard's instances, which can be classified into 8 groups according to their sizes, each with 10 instances. We train a policy for each of the 5 groups up to $30 \times 20$, while the remaining 3 groups are used for the generalization test. The results for each group are summarized in Table 3, while the detailed results for each instance can be found in the Supplementary Material. Note that the gaps are calculated using the best solutions in the literature. As shown in this table, the PDRs learned by our method still maintain good performance on the Taillard's benchmark and produce solutions significantly better than baseline PDRs, both when evaluating on the same size and when generalizing to larger size. It is interesting to see that all methods show smaller gaps on large-sized instances ($50 \times 15$, $50 \times 20$ and $100 \times 20$), which is probably because instances with larger $|\mathcal{J}|/|\mathcal{M}|$ tend to be easier to solve as noticed in [44]. But still, these instances could be hard for exact solvers due to the NP-hardness of JSSP, as OR-Tools only solves 2% of $100 \times 20$ instances optimally within the 3600s time limit, as shown in Table 2.

| Size | | SPT | MWKR | FDD/MWKR | MOPNR | Ours (30 × 20) | Opt. Rate |
|------|------|------|------|------|------|------|------|
| Dmu 30 × 20 | Obj. | 7036 | 6925 | 6827.3 | 6491.9 | **5967.4** | |
| | Gap | 65.9% | 63.2% | 60.1% | 52.0% | **39.5%** | 10% |
| | Time(s) | **0.709** | 0.718 | 0.721 | **0.709** | 1.805 | |
| Dmu 50 × 15 | Obj. | 8975.4 | 8906 | 9150.2 | 8436.5 | **8179.4** | |
| | Gap | 50.4% | 48.9% | 52.5% | 40.8% | **36.2%** | 50% |
| | Time(s) | **1.195** | 1.208 | 1.220 | 1.197 | 2.694 | |
| Dmu 50 × 20 | Obj. | 10132.8 | 9807 | 9899.6 | 9408 | **8751.6** | |
| | Gap | 62.2% | 56.4% | 57.3% | 49.6% | **38.9%** | 50% |
| | Time(s) | **2.496** | 2.516 | 2.521 | 2.500 | 4.908 | |

Table 4: **Results on DMU benchmark.** "Opt. Rate": rate of instances with optimal solution.

Next, we conduct experiments on the 80 DMU instances, which can also be classified into 8 groups according to their sizes. Here we train a policy for each of the 4 groups up to $30 \times 20$, with the remaining 4 groups as test sets for generalization. Due to limited space, we only present results of the $30 \times 20$ policy and its generalization performance on $50 \times 15$ and $50 \times 20$ instances in Table 4. We can see that our policy still outperforms baselines on these instances with reasonable time. Complete results on DMU benchmark are given in the Supplementary Material.

# 6 Conclusions and Future Work

In this paper, we present an end-to-end DRL based method to automatically learn high-quality PDRs for solving JSSP. Based on the disjunctive graph representation of JSSP, we propose an MDP formulation of the PDR based scheduling process. Then we design a size-agnostic policy network based on GNN, such that the patterns contained in the graph structure of JSSP can be effectively extracted and reused to solve instances of different sizes. Extensive experiments on generated and public benchmark instances well confirm the superiority of our method to the traditional manually designed PDRs. In the future, we plan to further enhance the performance of our method, and extend it to support other types of shop scheduling problems and complex environments with uncertainties.

## Broader Impact

Some work [49, 50] discussed the design of intelligent production systems by integrating modern AI technology. Our work, which solves a well-known problem that is ubiquitous in real-world production system, i.e. job shop scheduling, is within this scope. The automated end-to-end learning system in this work tries to free human labor from tedious work of designing effective dispatching rules for particular job shop scheduling problem. On the other side, however, this work may have some limitations. First, despite of performance improvement, it sacrifices interpretability due to the unexplainable nature of deep neural networks, whereas traditional dispatching rule based scheduling system is intuitive to human. This issue might make it untrustworthy for some applications, due to the potential risk and uncertainty. Second, highly automated and end-to-end system may conceal some details that are critical but easy to be ignored and bias human's understanding underneath.

## Acknowledgments

# References

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

[2] Kaizhou Gao, Zhiguang Cao, Le Zhang, Zhenghua Chen, Yuyan Han, and Quanke Pan. A review on swarm intelligence and evolutionary algorithms for solving flexible job shop scheduling problems. *IEEE/CAA Journal of Automatica Sinica*, 6(4):904–916, 2019.

[3] Egon Balas. An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, 13(4):517–546, 1965.

[4] V Srinivasan. A hybrid algorithm for the one machine sequencing problem to minimize total tardiness. *Naval Research Logistics Quarterly*, 18(3):317–327, 1971.

[5] Fred Glover and Manuel Laguna. Tabu search. In *Handbook of Combinatorial Optimization*, pages 2093–2229. Springer, 1998.

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

[7] Klaus Jansen, Monaldo Mastrolilli, and Roberto Solis-Oba. Approximation algorithms for flexible job shop problems. In *Latin American Symposium on Theoretical Informatics*, pages 68–77. Springer, 2000.

[8] Wen Song, Donghun Kang, Jie Zhang, Zhiguang Cao, and Hui Xi. A sampling approach for proactive project scheduling under generalized time-dependent workability uncertainty. *Journal of Artificial Intelligence Research*, 64:385–427, 2019.

[9] Veronique Sels, Nele Gheysen, and Mario Vanhoucke. A comparison of priority rules for the job shop scheduling problem under different flow time-and tardiness-related objective functions. *International Journal of Production Research*, 50(15):4255–4270, 2012.

[10] Lars Mönch, John W Fowler, and Scott J Mason. *Production planning and control for semiconductor wafer fabrication facilities: modeling, analysis, and systems*, volume 52. Springer Science & Business Media, 2012.

[11] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

[12] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 2*, pages 2692–2700. MIT Press, 2015.

[13] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9839–9849, 2018.

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

[15] Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving the travelling salesman problem. *arXiv preprint arXiv:1912.05784*, 2019.

[16] Shuang Yang Hao Lu, Xingwen Zhang. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations*, 2020.

[17] R. Zhang, A. Prokhorchuk, and J. Dauwels. Deep reinforcement learning for traveling salesman problem with time windows and rejections. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020.

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

[19] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pages 539–548, 2018.

[20] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

[21] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. Learning to solve circuit-sat: An unsupervised differentiable approach. In *International Conference on Learning Representations*, 2019.

[22] Emre Yolcu and Barnabas Poczos. Learning local search heuristics for boolean satisfiability. In *Advances in Neural Information Processing Systems*, pages 7990–8001, 2019.

[23] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[24] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pages 6278–6289, 2019.

[25] Shuai Zheng, Chetan Gupta, and Susumu Serita. Manufacturing dispatching using reinforcement and transfer learning. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2019.

[26] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Penghao Sun, Zehua Guo, Junchao Wang, Junfei Li, Julong Lan, and Yuxiang Hu. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, 2020.

[28] Z. Wang and M. Gombolay. Learning scheduling policies for multi-robot coordination with graph attention networks. *IEEE Robotics and Automation Letters*, 5(3):4509–4516, 2020.

[29] Helga Ingimundardottir and Thomas Philip Runarsson. Discovering dispatching rules from data using imitation learning: A case study for the job-shop problem. *Journal of Scheduling*, 21(4):413–428, 2018.

[30] Chun-Cheng Lin, Der-Jiunn Deng, Yen-Ling Chih, and Hsin-Ting Chiu. Smart manufacturing scheduling with edge computing using multiclass deep q network. *IEEE Transactions on Industrial Informatics*, 15(7):4276–4284, 2019.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[32] Jacek Błażewicz, Erwin Pesch, and Małgorzata Sterna. The disjunctive graph machine representation of the job shop scheduling problem. *European Journal of Operational Research*, 127(2):317–331, 2000.

[33] Egon Balas. Machine sequencing via disjunctive graphs: An implicit enumeration algorithm. *Operations Research*, 17(6):941–957, 1969.

[34] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[35] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[36] Mingqi Lv, Zhaoxiong Hong, Ling Chen, Tieming Chen, Tiantian Zhu, and Shouling Ji. Temporal multi-graph convolutional network for traffic flow prediction. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[37] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2018.

[38] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[39] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2020.

[40] Davide Bacciu, Federico Errica, Alessio Micheli, and Marco Podda. A gentle introduction to deep learning for graphs. *Neural Networks*, 2020.

[41] Scott J Mason and Kasin Oey. Scheduling complex job shops using disjunctive graphs: a cycle elimination procedure. *International Journal of Production Research*, 41(5):981–994, 2003.

[42] Andrea Rossi and Michele Lanzetta. Nonpermutation flow line scheduling by ant colony optimization. *AI EDAM*, 27(4):349–357, 2013.

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

[44] E Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

[45] AS Jain and S Meeran. Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, 113(2):390–434, 1999.

[46] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, 1998.

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[48] Laurent Perron and Vincent Furnon. Or-tools, 2019.

[49] Saurabh Vaidya, Prashant Ambad, and Santosh Bhosle. Industry 4.0–a glimpse. *Procedia Manufacturing*, 20:233–238, 2018.

[50] Pai Zheng, Zhiqian Sang, Ray Y Zhong, Yongkui Liu, Chao Liu, Khamdi Mubarok, Shiqiang Yu, Xun Xu, et al. Smart manufacturing systems for industry 4.0: Conceptual framework, scenarios, and future perspectives. *Frontiers of Mechanical Engineering*, 13(2):137–150, 2018.

# Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning

Cong Zhang[1], Wen Song[2], Zhiguang Cao[3], Jie Zhang[1], Puay Siew Tan[4], and Chi Xu[4]

[1]Nanyang Technological University
[2]Institute of Marine Science and Technology, Shandong University, China
[3]National University of Singapore
[4]Singapore Institute of Manufacturing Technology, A*STAR

## 1 Details of the Training Algorithm

In the paper, we adopt the Proximal Policy Optimization (PPO) algorithm [36] to train our agent. Here we provide details of our algorithm in terms of pseudo code, as shown in Algorithm 1. Similar to the original PPO in [36], we also use $N$ actors, each solves one JSSP instance drawn from a distribution $\mathbb{D}$. The difference to [36] is that, instead of sampling a batch of data, we use all data collected by the $N$ actors to perform update, i.e. line 13, 14, 15, and 19 in the psuedo code.

---

**Algorithm 1:** PPO learning to dispatch

    **Input** : actor network $\pi_\theta$ and behaviour actor network $\pi_{\theta_{old}}$, with trainable parameters $\theta_{old} = \theta$; critic network $v_\phi$ with trainable parameters $\phi$; number of training steps $U$; discounting factor $\gamma$; update epoch $K$; policy loss coefficient $c_p$; value function loss coefficient $c_v$; entropy loss coefficient $c_e$; clipping ratio $\epsilon$.

1   Initialize $\pi_\theta$, $\pi_{\theta_{old}}$, and $v_\phi$ ;
2   **for** $u = 1, 2, ..., U$ **do**
3      Draw $N$ JSSP instances from $\mathbb{D}$;
4      **for** $n = 1, 2, ..., N$ **do**
5          **for** $t = 0, 1, 2, ...$ **do**
6              Sample $a_{n,t}$ based on $\pi_{\theta_{old}}(a_{n,t}|s_{n,t})$;
7              Receive reward $r_{n,t}$ and next state $s_{n,t+1}$;
8              $\hat{A}_{n,t} = \sum_0^t \gamma^t r_{n,t} - v_\phi(s_{n,t}), r_{n,t}(\theta) = \frac{\pi_\theta(a_{n,t}|s_{n,t})}{\pi_{\theta_{old}}(a_{n,t}|s_{n,t})}$;
9              **if** $s_{n,t}$ *is terminal* **then**
10                  break;
11              **end**
12          **end**
13          $L_n^{CLIP}(\theta) = \sum_0^t \min\left(r_{n,t}(\theta)\hat{A}_{n,t}, \text{clip}\left(r_{n,t}(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_{n,t}\right)$;
14          $L_n^{VF}(\phi) = \sum_0^t \left(v_\phi\left(s_{n,t}\right) - \hat{A}_{n,t}\right)^2$;
15          $L_n^S(\theta) = \sum_0^t S(\pi_\theta(a_{n,t}|s_{n,t}))$, where $S(\cdot)$ is entropy;
16          Aggregate losses: $L_n(\theta, \phi) = c_p L_n^{CLIP}(\theta) - c_v L_n^{VF}(\phi) + c_e L_n^S(\theta)$ ;
17      **end**
18      **for** $k = 1, 2, ..., K$ **do**
19          Update $\theta, \phi$ with cumulative loss by Adam optimizer:
             $\theta, \phi = \text{argmax}\left(\sum_{n=1}^N L_n(\theta, \phi)\right)$
20      **end**
21      $\theta_{old} \leftarrow \theta$
22   **end**

---

## 2 Details of the Baselines

In this section, we show how the baseline PDRs compute the priority index for the operations. We begin with introducing the notations used in these rules, summarized as follows:

- $Z_{ij}$: the priority index of operation $O_{ij}$;
- $n_i$: the number of operations for job $J_i$;
- $Re_i$: the release time of job $J_i$ (here we assume $Re_i = 0$ for all $J_i$, i.e. all jobs are available in the beginning, but in general the jobs could have different release time);
- $p_{ij}$: the processing time of operation $O_{ij}$.

Based on the above notations, the decision principles for each baseline are given below:

- *Shortest Processing Time* (SPT): $\min\ Z_{ij} = p_{ij}$;
- *Most Work Remaining* (MWKR): $\max\ Z_{ij} = \sum_j^{n_i} p_{ij}$;
- *Minimum ratio of Flow Due Date to Most Work Remaining* (FDD/MWKR): $\min\ Z_{ij} = \left( Re_i + \sum_1^j p_{ij} \right) / \sum_j^{n_i} p_{ij}$;
- *Most Operations Remaining* (MOPNR): $\max\ Z_{ij} = n_i - j + 1$.

## 3 Result on Taillard's Benchmark

Here we present the complete results on Taillard's benchmark. In Table S.1, we report the results of training and testing on 5 groups of instances with sizes up to $30 \times 20$. As we can observe from this table, the PDRs trained by our method outperform the baselines on 92% of these instances (46 out of 50). In Table S.2, we report the generalization performance of our polices trained on $20 \times 20$ and $30 \times 20$ instances. Without training, the two trained PDRs achieve the best performance on all the 30 instances, with the $30 \times 20$ policy performing slightly better.

| Instance | | SPT | MWKR | FDD/WKR | MOPNR | Ours | UB |
|---|---|---|---|---|---|---|---|
| $15 \times 15$ | Ta01 | 1872 (52.1%) | 1786 (45.1%) | 1841 (49.6%) | 1864 (51.4%) | **1443 (17.2%)** | 1231* |
| | Ta02 | 1709 (37.4%) | 1944 (56.3%) | 1895 (52.3%) | 1680 (35.0%) | **1544 (24.1%)** | 1244* |
| | Ta03 | 2009 (64.9%) | 1947 (59.9%) | 1914 (57.1%) | 1558 (27.9%) | **1440 (18.2%)** | 1218* |
| | Ta04 | 1825 (55.3%) | 1694 (44.2%) | 1653 (40.7%) | 1755 (49.4%) | **1637 (39.3%)** | 1175* |
| | Ta05 | 2044 (67.0%) | 1892 (54.6%) | 1787 (46.0%) | **1605 (31.1%)** | 1619 (32.3%) | 1224* |
| | Ta06 | 1771 (43.1%) | 1976 (59.6%) | 1748 (41.2%) | 1815 (46.6%) | **1601 (29.3%)** | 1238* |
| | Ta07 | 2016 (64.3%) | 1961 (59.8%) | 1660 (35.3%) | 1884 (53.5%) | **1568 (27.8%)** | 1227* |
| | Ta08 | 1654 (35.9%) | 1803 (48.2%) | 1803 (48.2%) | 1839 (51.1%) | **1468 (20.6%)** | 1217* |
| | Ta09 | 1962 (54.0%) | 2215 (73.9%) | 1848 (45.1%) | 2002 (57.1%) | **1627 (27.7%)** | 1274* |
| | Ta10 | 2164 (74.4%) | 2057 (65.8%) | 1937 (56.1%) | 1821 (46.7%) | **1527 (23.0%)** | 1241* |
| $20 \times 15$ | Ta11 | 2212 (63.0%) | 2117 (56.0%) | 2101 (54.8%) | 2030 (49.6%) | **1794 (32.2%)** | 1357* |
| | Ta12 | 2414 (76.6%) | 2213 (61.9%) | 2034 (48.8%) | 2117 (54.9%) | **1805 (32.0%)** | 1367* |
| | Ta13 | 2346 (74.7%) | 2026 (50.9%) | 2141 (59.4%) | 1979 (47.4%) | **1932 (43.9%)** | 1343* |
| | Ta14 | 2109 (56.8%) | 2164 (60.9%) | 1841 (36.9%) | 2036 (51.4%) | **1664 (23.7%)** | 1345* |
| | Ta15 | 2163 (61.5%) | 2180 (62.8%) | 2187 (63.3%) | 1939 (44.8%) | **1730 (29.2%)** | 1339* |
| | Ta16 | 2232 (64.1%) | 2528 (85.9%) | 1926 (41.6%) | 1980 (45.6%) | **1710 (25.7%)** | 1360* |
| | Ta17 | 2185 (49.5%) | 2015 (37.8%) | 2093 (43.2%) | 2211 (51.2%) | **1897 (29.8%)** | 1462* |
| | Ta18 | 2267 (62.4%) | 2275 (63.0%) | 2064 (47.9%) | 1981 (41.9%) | **1794 (28.5%)** | 1396 |
| | Ta19 | 2238 (68.0%) | 2201 (65.2%) | 1958 (47.0%) | 1899 (42.6%) | **1682 (26.3%)** | 1332* |
| | Ta20 | 2370 (75.8%) | 2188 (62.3%) | 2195 (62.8%) | 1986 (47.3%) | **1739 (29.0%)** | 1348* |
| $20 \times 20$ | Ta21 | 2836 (72.7%) | 2622 (59.7%) | 2455 (49.5%) | 2320 (41.3%) | **2252 (37.1%)** | 1642* |
| | Ta22 | 2672 (67.0%) | 2554 (59.6%) | 2177 (36.1%) | 2415 (50.9%) | **2102 (31.4%)** | 1600 |
| | Ta23 | 2397 (53.9%) | 2408 (54.7%) | 2514 (61.5%) | 2194 (40.9%) | **2085 (33.9%)** | 1557 |
| | Ta24 | 2787 (69.5%) | 2553 (55.3%) | 2391 (45.4%) | 2250 (36.9%) | **2200 (33.8%)** | 1644* |
| | Ta25 | 2513 (57.6%) | 2582 (61.9%) | 2267 (42.1%) | **2146 (34.5%)** | 2201 (38.0%) | 1595 |
| | Ta26 | 2649 (61.2%) | 2506 (52.5%) | 2644 (60.9%) | 2480 (50.9%) | **2176 (32.4%)** | 1643 |
| | Ta27 | 2707 (61.1%) | 2768 (64.8%) | 2514 (49.6%) | 2298 (36.8%) | **2132 (26.9%)** | 1680 |
| | Ta28 | 2654 (65.6%) | 2370 (47.8%) | 2330 (45.4%) | 2259 (40.9%) | **2146 (33.9%)** | 1603* |
| | Ta29 | 2681 (65.0%) | 2399 (47.6%) | 2232 (37.4%) | 2367 (45.7%) | **1952 (20.1%)** | 1625 |
| | Ta30 | 2662 (68.1%) | 2424 (53.0%) | 2348 (48.2%) | 2370 (49.6%) | **2035 (28.5%)** | 1584 |

| | Instance | SPT | MWKR | FDD/WKR | MOPNR | Ours (20 × 20) | Ours (30 × 20) | UB |
|---|---|---|---|---|---|---|---|---|
| 30 × 15 | Ta31 | 2870 (62.7%) | 2590 (46.8%) | **2459 (39.4%)** | 2576 (46.0%) | 2565 (45.4%) | | 1764* |
| | Ta32 | 3097 (73.6%) | 2725 (52.7%) | 2672 (49.8%) | 2830 (58.6%) | **2388 (33.9%)** | | 1784 |
| | Ta33 | 2782 (55.3%) | 2919 (63.0%) | 2766 (54.4%) | 2746 (53.3%) | **2324 (29.8%)** | | 1791 |
| | Ta34 | 2956 (61.7%) | 2826 (54.6%) | 2669 (46.0%) | 2464 (34.8%) | **2332 (27.6%)** | | 1828* |
| | Ta35 | 2940 (46.5%) | 2791 (39.1%) | 2525 (25.8%) | 2649 (32.0%) | **2505 (24.8%)** | | 2007* |
| | Ta36 | 2933 (61.2%) | 2811 (54.5%) | 2690 (47.9%) | 2666 (46.6%) | **2497 (37.3%)** | | 1819* |
| | Ta37 | 3065 (73.1%) | 2719 (53.5%) | 2492 (40.7%) | 2584 (45.9%) | **2325 (31.3%)** | | 1771* |
| | Ta38 | 2700 (61.4%) | 2706 (61.7%) | 2425 (44.9%) | 2657 (58.8%) | **2302 (37.6%)** | | 1673* |
| | Ta39 | 2698 (50.3%) | 2592 (44.4%) | 2596 (44.6%) | **2409 (34.2%)** | 2410 (34.3%) | | 1795* |
| | Ta40 | 2843 (70.3%) | 2601 (55.8%) | 2614 (56.6%) | 2432 (45.7%) | **2140 (28.2%)** | | 1669 |
| 30 × 20 | Ta41 | 3067 (53.0%) | 3145 (56.9%) | 2991 (49.2%) | 2996 (49.4%) | **2667 (33.0%)** | | 2005 |
| | Ta42 | 3640 (87.9%) | 3394 (75.2%) | 3027 (56.3%) | 2995 (54.6%) | **2664 (37.5%)** | | 1937 |
| | Ta43 | 2843 (54.0%) | 3162 (71.3%) | 2926 (58.5%) | 2666 (44.4%) | **2431 (31.7%)** | | 1846 |
| | Ta44 | 3281 (65.8%) | 3388 (71.2%) | 3462 (74.9%) | 2845 (43.8%) | **2714 (37.1%)** | | 1979 |
| | Ta45 | 3238 (61.9%) | 3390 (69.5%) | 3245 (62.3%) | 3134 (56.7%) | **2637 (31.9%)** | | 2000 |
| | Ta46 | 3352 (67.1%) | 3268 (62.9%) | 3008 (50.0%) | 2802 (39.7%) | **2776 (38.4%)** | | 2006 |
| | Ta47 | 3197 (69.2%) | 2986 (58.1%) | 2940 (55.6%) | 2788 (47.6%) | **2476 (31.1%)** | | 1889 |
| | Ta48 | 3445 (77.9%) | 3050 (57.5%) | 2991 (54.4%) | 2822 (45.7%) | **2490 (28.5%)** | | 1937 |
| | Ta49 | 3201 (63.2%) | 3172 (61.8%) | 2865 (46.1%) | 2933 (49.6%) | **2556 (30.3%)** | | 1961 |
| | Ta50 | 3083 (60.3%) | 2978 (54.9%) | 2995 (55.7%) | 2900 (50.8%) | **2628 (36.7%)** | | 1923 |

Table S.1. **Results on Taillard's Benchmark (Part I).** The "UB" column is the best solution from literature, and "*" means the solution is optimal.

| | Instance | SPT | MWKR | FDD/WKR | MOPNR | Ours (20 × 20) | Ours (30 × 20) | UB |
|---|---|---|---|---|---|---|---|---|
| 50 × 15 | Ta01 | 4280 (55.1%) | 3899 (41.3%) | 3851 (39.5%) | 3616 (31.0%) | 3793 (37.4%) | **3599 (30.4%)** | 2760* |
| | Ta02 | 4419 (60.3%) | 3763 (36.5%) | 3734 (35.5%) | 3698 (34.2%) | 3487 (26.5%) | **3341 (21.2%)** | 2756* |
| | Ta03 | 3949 (45.3%) | 3894 (43.3%) | 3394 (24.9%) | 3402 (25.2%) | **3106 (14.3%)** | 3186 (17.3%) | 2717* |
| | Ta04 | 3977 (40.1%) | 3739 (31.7%) | 3603 (26.9%) | 3599 (26.8%) | 3322 (17.0%) | **3266 (15.0%)** | 2839* |
| | Ta05 | 4307 (60.8%) | 3782 (41.2%) | 3664 (36.8%) | 3650 (36.2%) | 3336 (24.5%) | **3232 (20.6%)** | 2679* |
| | Ta06 | 4156 (49.4%) | 3951 (42.1%) | 4016 (44.4%) | 3638 (30.8%) | 3501 (25.9%) | **3378 (21.5%)** | 2781* |
| | Ta07 | 4321 (46.8%) | 3883 (31.9%) | 3720 (26.4%) | 3705 (25.9%) | 3581 (21.7%) | **3471 (17.9%)** | 2943* |
| | Ta08 | 4090 (41.8%) | 4476 (55.1%) | 3926 (36.1%) | 3661 (26.9%) | **3454 (19.7%)** | 3732 (29.4%) | 2885* |
| | Ta09 | 4101 (54.5%) | 3751 (41.3%) | 3672 (38.3%) | 3530 (33.0%) | 3441 (29.6%) | **3381 (27.3%)** | 2655* |
| | Ta10 | 4347 (59.6%) | 3940 (44.7%) | 3783 (38.9%) | 3581 (31.5%) | **3281 (20.5%)** | 3352 (23.1%) | 2723* |
| 50 × 20 | Ta11 | 4687 (63.4%) | 4313 (50.4%) | 4142 (44.4%) | 3941 (37.4%) | 3830 (33.5%) | **3654 (27.4%)** | 2868* |
| | Ta12 | 4670 (62.8%) | 4542 (58.3%) | 3897 (35.8%) | 4025 (40.3%) | **3617 (26.1%)** | 3722 (29.7%) | 2869* |
| | Ta13 | 4415 (60.3%) | 4069 (47.7%) | 3852 (39.8%) | 3692 (34.0%) | **3397 (23.3%)** | 3536 (28.3%) | 2755* |
| | Ta14 | 4334 (60.4%) | 4176 (54.6%) | 4001 (48.1%) | 3748 (38.7%) | **3275 (21.2%)** | 3631 (34.4%) | 2702* |
| | Ta15 | 4221 (54.9%) | 4600 (68.8%) | 4062 (49.1%) | 3866 (41.9%) | 3510 (28.8%) | **3359 (23.3%)** | 2725* |
| | Ta16 | 4457 (56.7%) | 4209 (47.9%) | 3940 (38.5%) | 3846 (35.2%) | **3388 (19.1%)** | 3555 (25.0%) | 2845* |
| | Ta17 | 4420 (56.5%) | 4172 (47.7%) | 3974 (40.7%) | 3795 (34.3%) | 3848 (36.2%) | **3567 (26.3%)** | 2825* |
| | Ta18 | 4807 (72.7%) | 4428 (59.1%) | 3857 (38.5%) | 4077 (46.4%) | **3514 (26.2%)** | 3680 (32.2%) | 2784* |
| | Ta19 | 4379 (42.6%) | 4758 (54.9%) | 4349 (41.6%) | 4135 (34.6%) | 3763 (22.5%) | **3592 (17.0%)** | 3071* |
| | Ta20 | 4932 (64.7%) | 4484 (49.7%) | 4147 (38.5%) | 4075 (36.1%) | 3976 (32.8%) | **3643 (21.6%)** | 2995* |
| 100 × 20 | Ta21 | 7841 (43.5%) | 6943 (27.1%) | 6818 (24.8%) | 6601 (20.8%) | 6549 (19.9%) | **6452 (18.1%)** | 5464* |
| | Ta22 | 7655 (47.8%) | 7021 (35.5%) | 6358 (22.7%) | 6191 (19.5%) | 5884 (13.6%) | **5695 (9.9%)** | 5181* |
| | Ta23 | 7510 (34.9%) | 7381 (32.6%) | 6967 (25.1%) | 6758 (21.4%) | **6411 (15.1%)** | 6462 (16.1%) | 5568* |
| | Ta24 | 7451 (39.6%) | 6995 (31.0%) | 6381 (19.5%) | 6090 (14.1%) | 5917 (10.8%) | **5885 (10.2%)** | 5339* |
| | Ta25 | 7360 (36.5%) | 7366 (36.6%) | 6757 (25.3%) | 6611 (22.6%) | 6669 (23.7%) | **6355 (17.9%)** | 5392* |
| | Ta26 | 7909 (48.1%) | 7026 (31.5%) | 6641 (24.3%) | 6554 (22.7%) | 6337 (18.6%) | **6135 (14.8%)** | 5342* |
| | Ta27 | 7456 (37.2%) | 7502 (38.0%) | 6540 (20.3%) | 6589 (21.2%) | 6297 (15.8%) | **6056 (11.4%)** | 5436* |
| | Ta28 | 7400 (37.2%) | 6861 (27.2%) | 6750 (25.1%) | 6313 (17.0%) | 6177 (14.5%) | **6101 (13.1%)** | 5394* |
| | Ta29 | 7743 (44.5%) | 7232 (35.0%) | 6461 (20.6%) | 6665 (24.4%) | 6185 (15.4%) | **5943 (10.9%)** | 5358* |
| | Ta30 | 7321 (41.3%) | 6961 (34.3%) | 6534 (26.1%) | 6151 (18.7%) | 6124 (18.2%) | **5892 (13.7%)** | 5183* |

Table S.2. **Results on Taillard's Benchmark (Part II).** The "UB" column is the best solution from literature, and "*" means the solution is optimal.

# 4 Result on DMU Benchmark

Similar conclusion can be drawn from results on DMU benchmark. In Table S.3, we report results of training and testing on 4 groups of instances with sizes up to 30 × 20, where our method outperforms baselines over 87.5% (35 out of 40) of these instances. In Table S.4 which focuses on the generalization performance, our policies trained on 20 × 20 and 30 × 20 instances outperform the baselines on 77.5% (31 out of 40) instances.

| | Instance | SPT | MWKR | FDD/WKR | MOPNR | Ours | UB |
|---|---|---|---|---|---|---|---|
| 20 × 15 | Dmu01 | 4516 (76.2%) | 3988 (55.6%) | 3535 (37.9%) | 3882 (51.5%) | **3323 (29.7%)** | 2563 |
| | Dmu02 | 4593 (69.7%) | 4555 (68.3%) | 3847 (42.2%) | 3884 (43.5%) | **3630 (34.1%)** | 2706 |
| | Dmu03 | 4438 (62.5%) | 4117 (50.8%) | 4063 (48.8%) | 3979 (45.7%) | **3660 (34.0%)** | 2731* |
| | Dmu04 | 4533 (69.8%) | 3995 (49.7%) | 4160 (55.9%) | 4079 (52.8%) | **3816 (43.0%)** | 2669 |
| | Dmu05 | 4420 (60.8%) | 4977 (81.0%) | 4238 (54.2%) | 4116 (49.7%) | **3897 (41.8%)** | 2749* |
| | Dmu41 | 5283 (62.7%) | 5377 (65.5%) | 5187 (59.7%) | 5070 (56.1%) | **4316 (32.9%)** | 3248 |
| | Dmu42 | 5354 (57.9%) | 6076 (79.2%) | 5583 (64.7%) | 4976 (46.8%) | **4858 (43.3%)** | 3390 |
| | Dmu43 | 5328 (54.8%) | 4938 (43.5%) | 5086 (47.8%) | 5012 (45.7%) | **4887 (42.0%)** | 3441 |
| | Dmu44 | 5745 (64.7%) | 5630 (61.4%) | 5550 (59.1%) | 5213 (49.5%) | **5151 (47.7%)** | 3488 |
| | Dmu45 | 5305 (62.1%) | 5446 (66.4%) | 5414 (65.5%) | 4921 (50.4%) | **4615 (41.0%)** | 3272 |
| 20 × 20 | Dmu06 | 6230 (92.0%) | 5556 (71.3%) | 5258 (62.1%) | 4747 (46.3%) | **4358 (34.3%)** | 3244 |
| | Dmu07 | 5619 (84.5%) | 4636 (52.2%) | 4789 (57.2%) | 4367 (43.4%) | **3671 (20.5%)** | 3046 |
| | Dmu08 | 5239 (64.3%) | 5078 (59.3%) | 4817 (51.1%) | 4480 (40.5%) | **4048 (27.0%)** | 3188 |
| | Dmu09 | 4874 (57.6%) | 4519 (46.2%) | 4675 (51.2%) | 4519 (46.2%) | **4482 (45.0%)** | 3092 |
| | Dmu10 | 4808 (61.1%) | 4963 (66.3%) | 4149 (39.0%) | 4133 (38.5%) | **4021 (34.8%)** | 2984 |
| | Dmu46 | 6403 (58.7%) | 6168 (52.9%) | **5778 (43.2%)** | 6136 (52.1%) | 5876 (45.6%) | 4035 |
| | Dmu47 | 6015 (52.7%) | 6130 (55.6%) | 6058 (53.8%) | 5908 (50.0%) | **5771 (46.5%)** | 3939 |
| | Dmu48 | 5345 (42.0%) | 5701 (51.5%) | 5887 (56.4%) | 5384 (43.1%) | **5034 (33.8%)** | 3763 |
| | Dmu49 | 6072 (63.7%) | 6089 (64.1%) | 5807 (56.5%) | **5469 (47.4%)** | 5470 (47.4%) | 3710 |
| | Dmu50 | 6300 (68.9%) | 6050 (62.2%) | 5764 (54.6%) | 5380 (44.3%) | **5314 (42.5%)** | 3729 |
| 30 × 15 | Dmu11 | 5864 (71.0%) | 4961 (44.6%) | 4798 (39.9%) | 4891 (42.6%) | **4435 (29.3%)** | 3430 |
| | Dmu12 | 5966 (70.7%) | 5994 (71.5%) | 5595 (60.1%) | 4947 (41.5%) | **4864 (39.2%)** | 3495 |
| | Dmu13 | 5744 (56.0%) | 6190 (68.2%) | 5324 (44.6%) | 4979 (35.3%) | **4918 (33.6%)** | 3681* |
| | Dmu14 | 5469 (61.1%) | 5567 (64.0%) | 4830 (42.3%) | 4839 (42.6%) | **4130 (21.7%)** | 3394* |
| | Dmu15 | 5518 (65.1%) | 5299 (58.5%) | 4928 (47.4%) | 4653 (39.2%) | **4392 (31.4%)** | 3343* |
| | Dmu51 | 6538 (56.9%) | 6841 (64.2%) | 7002 (68.0%) | 6691 (60.6%) | **6241 (49.8%)** | 4167 |
| | Dmu52 | 7341 (70.3%) | 6942 (61.0%) | 6650 (54.3%) | **6591 (52.9%)** | 6714 (55.7%) | 4311 |
| | Dmu53 | 7232 (64.6%) | 7430 (69.1%) | 7170 (63.2%) | 6851 (55.9%) | **6724 (53.0%)** | 4394 |
| | Dmu54 | 7178 (64.6%) | **6461 (48.1%)** | 6767 (55.1%) | 6540 (49.9%) | 6522 (49.5%) | 4362 |
| | Dmu55 | **6212 (45.4%)** | 6844 (60.2%) | 7101 (66.3%) | 6446 (50.9%) | 6639 (55.4%) | 4271 |
| 30 × 20 | Dmu16 | 6241 (66.4%) | 5837 (55.6%) | 5948 (58.6%) | 5743 (53.1%) | **4953 (32.0%)** | 3751 |
| | Dmu17 | 6487 (70.1%) | 6610 (73.3%) | 6035 (58.2%) | 5540 (45.3%) | **5379 (41.0%)** | 3814 |
| | Dmu18 | 6978 (81.5%) | 6363 (65.5%) | 5863 (52.5%) | 5714 (48.6%) | **5100 (32.7%)** | 3844* |
| | Dmu19 | 5767 (53.1%) | 6385 (69.5%) | 5424 (43.9%) | 5223 (38.6%) | **4889 (29.8%)** | 3768 |
| | Dmu20 | 6910 (86.3%) | 6472 (74.4%) | 6444 (73.7%) | 5530 (49.1%) | **4859 (31.0%)** | 3710 |
| | Dmu56 | 7698 (55.8%) | 7930 (60.5%) | 8248 (66.9%) | 7620 (54.2%) | **7328 (48.3%)** | 4941 |
| | Dmu57 | 7746 (66.4%) | 7063 (51.7%) | 7694 (65.3%) | 7345 (57.8%) | **6704 (44.0%)** | 4655 |
| | Dmu58 | 7269 (54.4%) | 7708 (63.7%) | 7601 (61.4%) | 7216 (53.3%) | **6721 (42.8%)** | 4708 |
| | Dmu59 | 7114 (53.8%) | 7335 (58.6%) | 7490 (62.0%) | 7589 (64.1%) | **7109 (53.7%)** | 4624 |
| | Dmu60 | 8150 (71.4%) | 7547 (58.7%) | 7526 (58.3%) | 7399 (55.6%) | **6632 (39.5%)** | 4755 |

Table S.3. **Results on DMU Benchmark (Part I).** The "UB" column is the best solution from literature, and "*" means the solution is optimal.

| | Instance | SPT | MWKR | FDD/WKR | MOPNR | Ours (20 × 20) | Ours (30 × 20) | UB |
|---|---|---|---|---|---|---|---|---|
| 40 × 15 | Dmu21 | 7400 (68.9%) | 6314 (44.2%) | 6416 (46.5%) | 6048 (38.1%) | 5559 (26.9%) | **5317 (21.4%)** | 4380* |
| | Dmu22 | 7353 (55.6%) | 6980 (47.7%) | 6645 (40.6%) | 6351 (34.4%) | 5929 (25.5%) | **5534 (17.1%)** | 4725* |
| | Dmu23 | 7262 (55.6%) | 6472 (38.6%) | 6781 (45.3%) | 6004 (28.6%) | 5681 (21.7%) | **5620 (20.4%)** | 4668* |
| | Dmu24 | 6799 (46.3%) | 7079 (52.3%) | 6582 (41.6%) | 6155 (32.4%) | **5479 (17.9%)** | 5753 (23.8%) | 4648* |
| | Dmu25 | 6428 (54.4%) | 6042 (45.1%) | 5756 (38.2%) | 5365 (28.8%) | 4825 (15.9%) | **4775 (14.7%)** | 4164* |
| | Dmu61 | **7817 (51.1%)** | 8734 (68.9%) | 8757 (69.3%) | 8076 (56.1%) | 8053 (55.7%) | 8203 (58.6%) | 5172 |
| | Dmu62 | **7759 (47.4%)** | 8262 (56.9%) | 8082 (53.5%) | 8253 (56.8%) | 8415 (59.8%) | 8091 (53.7%) | 5265 |
| | Dmu63 | 8296 (55.8%) | 8364 (57.0%) | 8384 (57.4%) | 8417 (58.0%) | 8330 (56.4%) | **8031 (50.8%)** | 5326 |
| | Dmu64 | 8444 (60.8%) | 8406 (60.1%) | 8490 (61.7%) | 8161 (55.4%) | 7916 (50.8%) | **7738 (47.4%)** | 5250 |
| | Dmu65 | 8454 (62.9%) | 8189 (57.8%) | 8307 (60.1%) | 8225 (58.5%) | 8093 (55.9%) | **7577 (46.0%)** | 5190 |
| 40 × 20 | Dmu26 | 7766 (67.1%) | 7107 (52.9%) | 7240 (55.8%) | 6236 (34.2%) | **5908 (27.1%)** | 5946 (28.0%) | 4647* |
| | Dmu27 | 7501 (54.7%) | 7313 (50.8%) | 6965 (43.7%) | 6936 (43.1%) | 6542 (34.9%) | **6418 (32.4%)** | 4848* |
| | Dmu28 | 8621 (83.7%) | 8194 (74.6%) | 6516 (38.9%) | 6714 (43.1%) | 6272 (33.7%) | **5986 (27.6%)** | 4692* |
| | Dmu29 | 8052 (71.6%) | 7448 (58.8%) | 6971 (48.6%) | 6990 (49.0%) | 6169 (31.5%) | **6051 (29.0%)** | 4691* |
| | Dmu30 | 7372 (55.8%) | 7890 (66.7%) | 6910 (46.0%) | 6869 (45.2%) | 6022 (27.3%) | **5988 (26.5%)** | 4732* |
| | Dmu66 | 8971 (56.9%) | 8966 (56.8%) | 9606 (68.0%) | 8726 (52.6%) | 8547 (49.5%) | **8475 (48.2%)** | 5717 |
| | Dmu67 | 9096 (56.5%) | 9306 (60.1%) | 9103 (56.6%) | 9372 (61.2%) | **8791 (51.2%)** | 8832 (51.9%) | 5813 |
| | Dmu68 | 9265 (60.5%) | 9445 (63.6%) | 9431 (63.4%) | 8722 (51.1%) | 9117 (57.9%) | **8693 (50.6%)** | 5773 |
| | Dmu69 | 9215 (61.4%) | 9450 (65.5%) | 9951 (74.3%) | 8697 (52.3%) | 9130 (59.9%) | **8634 (51.2%)** | 5709 |
| | Dmu70 | 9522 (61.7%) | 9490 (61.1%) | 9416 (59.9%) | 9445 (60.4%) | **8601 (46.1%)** | 8735 (48.3%) | 5889 |

| | | | | | | | | UB |
|---|---|---|---|---|---|---|---|---|
| | Dmu31 | 8869 (57.3%) | 8147 (44.5%) | 7899 (40.1%) | 7192 (27.5%) | 7191 (27.5%) | **7156 (26.9%)** | 5640* |
| | Dmu32 | 7814 (31.8%) | 8004 (35.0%) | 7316 (23.4%) | 7267 (22.6%) | 6938 (17.1%) | **6506 (9.8%)** | 5927* |
| | Dmu33 | 8114 (41.7%) | 7710 (34.6%) | 7262 (26.8%) | 7069 (23.4%) | 6480 (13.1%) | **6192 (8.1%)** | 5728* |
| 50 × 15 | Dmu34 | 7625 (41.6%) | 7709 (43.2%) | 7725 (43.5%) | 6919 (28.5%) | 6661 (23.7%) | **6257 (16.2%)** | 5385* |
| | Dmu35 | 8626 (53.1%) | 7617 (35.2%) | 7099 (26.0%) | 7033 (24.8%) | 6417 (13.9%) | **6302 (11.8%)** | 5635* |
| | Dmu71 | 9594 (53.9%) | 9978 (60.1%) | 10889 (74.7%) | **9514 (52.6%)** | 9950 (59.6%) | 9797 (57.2%) | 6233 |
| | Dmu72 | **9882 (52.4%)** | 10135 (56.3%) | 11602 (79.0%) | 10063 (55.2%) | 10401 (60.4%) | 9926 (53.1%) | 6483 |
| | Dmu73 | 9953 (61.5%) | 9721 (57.7%) | 10212 (65.7%) | **9615 (56.0%)** | 10080 (63.6%) | 9933 (61.2%) | 6163 |
| | Dmu74 | 9866 (58.6%) | 10086 (62.2%) | 10659 (71.4%) | **9536 (53.3%)** | 10445 (67.9%) | 9833 (58.1%) | 6220 |
| | Dmu75 | **9411 (51.9%)** | 9953 (60.6%) | 10839 (74.9%) | 10157 (63.9%) | 9937 (60.4%) | 9892 (59.6%) | 6197 |
| | Dmu36 | 9911 (76.3%) | 8090 (43.9%) | 8084 (43.8%) | 7703 (37.0%) | **7213 (28.3%)** | 7470 (32.9%) | 5621* |
| | Dmu37 | 8917 (52.4%) | 9685 (65.5%) | 9433 (61.2%) | 7844 (34.1%) | 7765 (32.7%) | **7296 (24.7%)** | 5851* |
| | Dmu38 | 9384 (64.3%) | 8414 (47.3%) | 8428 (47.5%) | 8398 (47.0%) | 7429 (30.0%) | **7410 (29.7%)** | 5713* |
| 50 × 20 | Dmu39 | 9221 (60.4%) | 9266 (61.2%) | 8177 (42.3%) | 7969 (38.7%) | 7168 (24.7%) | **6827 (18.8%)** | 5747* |
| | Dmu40 | 9406 (68.7%) | 8261 (48.1%) | 7773 (39.4%) | 8173 (46.5%) | 7757 (39.1%) | **7325 (31.3%)** | 5577* |
| | Dmu76 | 11677 (71.4%) | 10571 (55.2%) | 11576 (69.9%) | 11019 (61.7%) | 10322 (51.5%) | **9698 (42.3%)** | 6813 |
| | Dmu77 | **10401 (52.5%)** | 11148 (63.4%) | 11910 (74.6%) | 10577 (55.0%) | 10729 (57.3%) | 10693 (56.7%) | 6822 |
| | Dmu78 | 10585 (56.4%) | 10540 (55.7%) | 11464 (69.3%) | 10989 (62.3%) | 10742 (58.7%) | **9986 (47.5%)** | 6770 |
| | Dmu79 | 11115 (59.5%) | 11201 (60.7%) | 11035 (58.3%) | **10729 (53.9%)** | 10993 (57.7%) | 10936 (56.9%) | 6970 |
| | Dmu80 | 10711 (60.2%) | 10894 (62.9%) | 11116 (66.3%) | 10679 (59.7%) | 10041 (50.2%) | **9875 (47.7%)** | 6686 |

Table S.4. **Results on DMU Benchmark (Part II).** The "UB" column is the best solution from literature, and "*" means the solution is optimal.

## 5 Training Curve

We show training curves for all problems in Figure.1. The problem sizes are $\{6 \times 6, 10 \times 10, 15 \times 15, 20 \times 15, 20 \times 20, 30 \times 15, 30 \times 20\}$ respectively. In each curve, after learning on every 200 totally new instances, the averaged performance (makespan) over these 200 instances is plotted. The training time for each problem is: 0.95h ( for 1a), 2.6h (for 1b), 6.2h (for 1c), 11.6h (for 1d), 20.3h (for 1e), 8.7h (1f), 11.6h (1g), 13.5h (1h), and 20.3h (1i) respectively.
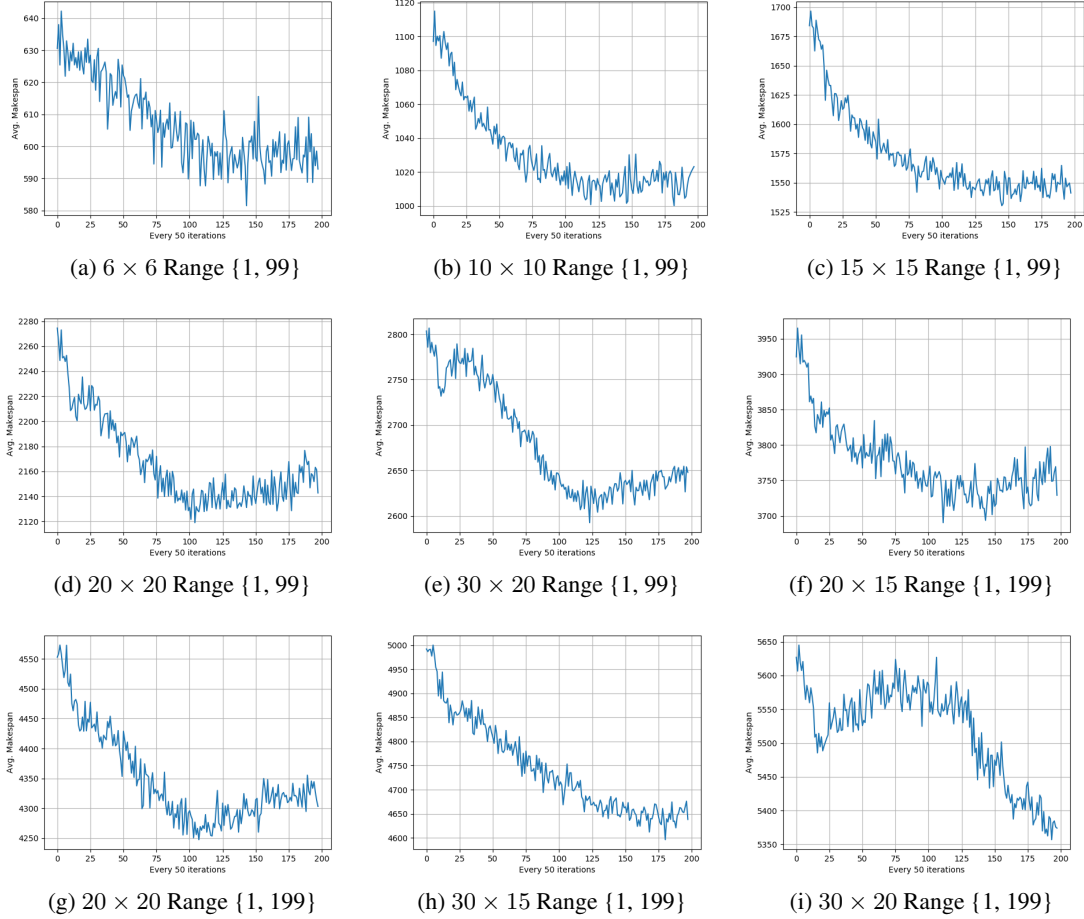
Figure 1: **Training curves for all problems.** The scale of processing time of each problem is given in braces, e.g. $\{1, 99\}$ indicating the scale of processing time is a integer uniformly distributed in range from 1 to 99. Sizes $20 \times 20$ and $30 \times 20$ have 2 different scales $\{1, 99\}$ and $\{1, 199\}$ respectively.