Learning to Optimize Permutation Flow Shop Scheduling via Graph-Based Imitation Learning

Longkang Li[1,2], **Siyuan Liang**[3], **Zihao Zhu**[1], **Chris Ding**[1], **Hongyuan Zha**[1], **Baoyuan Wu**[1*]

[1]The School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China
[2]Mohamed bin Zayed University of Artificial Intelligence, UAE
[3]National University of Singapore, Singapore

Abstract

The permutation flow shop scheduling (PFSS), aiming at finding the optimal permutation of jobs, is widely used in manufacturing systems. When solving large-scale PFSS problems, traditional optimization algorithms such as heuristics could hardly meet the demands of both solution accuracy and computational efficiency, thus learning-based methods have recently garnered more attention. Some work attempts to solve the problems by reinforcement learning methods, which suffer from slow convergence issues during training and are still not accurate enough regarding the solutions. To that end, we propose to train the model via expert-driven imitation learning, which accelerates convergence more stably and accurately. Moreover, in order to extract better feature representations of input jobs, we incorporate the graph structure as the encoder. The extensive experiments reveal that our proposed model obtains significant promotion and presents excellent generalizability in large-scale problems with up to 1000 jobs. Compared to the state-of-the-art reinforcement learning method, our model's network parameters are reduced to only 37% of theirs, and the solution gap of our model towards the expert solutions decreases from 6.8% to 1.3% on average. The code is available at: https://github.com/longkangli/PFSS-IL.

1 Introduction

Scheduling is a synthesized process of arranging, controlling, and optimizing jobs and workload in a manufacturing system. Here we utilize machine learning to solve a frequently encountered optimization problem: the permutation flow shop scheduling (PFSS) (Sharma and Sharma 2021), which is about sequentially processing several jobs on a series of machines. The PFSS is one of the most comprehensively studied scheduling problems, with broad applications in production and planning (Ribas, Leisten, and Framiñan 2010), intuitive designs (Alfaro-Fernández et al. 2020), and transportation (Soukhal, Oulamara, and Martineau 2005).

Figure 1 gives an example and overview of a 4-job 3-machine PFSS problem. One machine corresponds to one operation, and the PFSS strictly requires all the jobs to be processed in the same order of operations. Typically, the goal of the PFSS is to find the optimal permutation
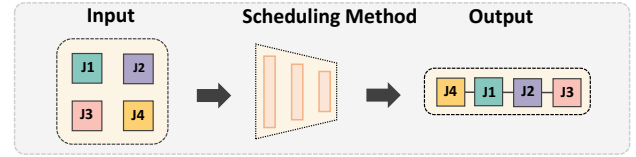
(a) The overview of the PFSS.
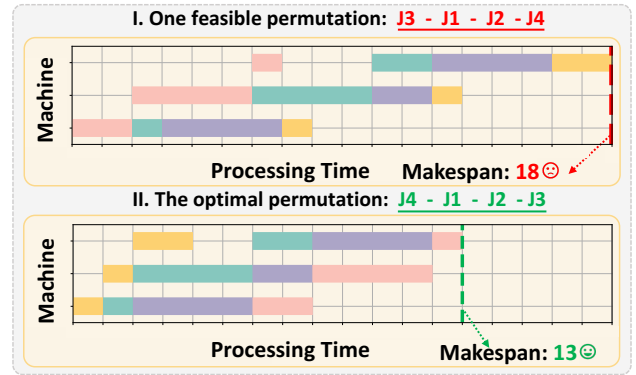
(b) Gantt charts of different permutations.

Figure 1: An overview of a 4-job (J1∼J4) 3-machine (M1∼M3) PFSS problem. One color corresponds to one job. All jobs follow the same operational order M1-M2-M3.

of the jobs, which minimizes the makespan. Makespan describes the total processing time from the start to the end for scheduling a sequence of jobs. When there is only 1 machine, this PFSS problem can be trivially solved. As for a n-job m-machine PFSS problem ($m \geq 2$), the number of feasible solutions should be $n!$. The PFSS thus can be considered as a combinatorial optimization (CO) problem with the NP-hard property (Lenstra and Kan 1981).

Recently, machine learning has been widely used in combinatorial optimization problems (Bengio, Lodi, and Prouvost 2020; Li et al. 2022). One popular model is pointer network (PN) (Vinyals, Fortunato, and Jaitly 2015), which uses the Long Short Term Memory (LSTM) as an encoder and the attention mechanism as a decoder (or pointer) to select a member of the input as the output. The PN can generalize a small pre-trained model to arbitrarily large instances. The current state-of-the-art learning-based method for solving the PFSS utilizes the PN and training the network via actor-

critic as a paradigm of reinforcement learning (RL) (Pan, Dong, and Han 2020). Besides the learning-based methods, there are traditional optimization algorithms for solving the PFSS, including the mathematical models and the heuristics. Mathematical models (Ríos-Mercado and Bard 1998) are exact methods that are guaranteed to obtain optimal solutions. However, it takes a tremendous amount of time to solve due to the NP-hardness of the PFSS. The heuristics (Sharma and Sharma 2021) allow getting a feasible or sometimes quasi-optimal solution within a limited time but still encounter the trade-off between the solution effectiveness and computational efficiency.

Motivations. For solving large-scale PFSS problems in practical scenarios, traditional optimization algorithms could hardly satisfy the demands of both solution accuracy and computational efficiency, thus the learning-based methods attract more attention. However, the state-of-the-art RL method (Pan, Dong, and Han 2020) requires heavy networks (an actor-network and a critic network) and takes a long time for training until the network converges. To that end, we train the model via expert-driven imitation learning (IL) (Monteiro et al. 2019), which leads to more stable and accurate convergences. Moreover, in order to extract better feature representations of input jobs, we incorporate the graph structure, the Gated Graph ConvNets (GGCN) (Bresson and Laurent 2018), for obtaining better job feature representations. More technical motivations for using graph encoder and attention decoder, and how our model pre-trained on a small-sized job can be generalized to arbitrarily larger-sized jobs, will be explained in Section 4.

To sum up, we make the following contributions in this paper: **1)** To the best of our knowledge, we are the first to solve the PFSS problem via expert-driven imitation learning, which accelerates convergence faster and more efficiently. **2)** Our learning model is based on the graph structure, which achieves a better representation capability. Our graph-based imitation learning model is much lighter in parameter weights, more stable in convergence, and more accurate in performance, compared to the state-of-the-art reinforcement learning methods. **3)** We explore and extend the job number to 1000 in solving PFSS problems. Extensive experiments on both benchmark datasets and generated datasets show the competitive performance of our proposed model, especially the excellent generalizability.

2   Related Work

Traditional Optimization Algorithms for PFSS. Traditional optimization algorithms for solving the PFSS include mathematical models and heuristics. The PFSS can be formulated as mixed-integer programming. Many different mathematical models have been proposed for the PFSS and solved by the exact methods such as branch-and-bound and brand-and-cut methods (Ríos-Mercado and Bard 1998). Over the last few decades, many heuristics have been applied to the PFSS problems. Random search (Zabinsky et al. 2009) can settle ill-structured global optimization problems with continuous and discrete variables. Iterated local search (Lourenço, Martin, and Stützle 2019) is based on a local search strategy using a single solution along the iterative process. The iterated greedy (Ruiz, Pan, and Naderi 2019) consists of improved initialization, iterative construction, and destruction procedures, along with a local search. Sharma et al. (Sharma and Sharma 2021) apply a tie-breaking rule to the Nawaz-Enscore-Ham (NEH) heuristic. For the large-scale PFSS problems, the heuristics could still hardly balance the solution accuracy and runtime efficiency: random search and iterated local search obtain poor makespans, while iterated greedy and NEH algorithms are time-consuming.

Learning-Based Methods for PFSS. With the development of deep learning (DL) and deep reinforcement learning (DRL) (Bengio, Lodi, and Prouvost 2020; Li and Wu 2022), some work has been starting to solve the scheduling problems from the RL perspectives. Ren et al. used value-based methods to solve the PFSS, such as Sarsa and Q-learning (Ren, Ye, and Yang 2021). However, this method focuses on the scale of less than 100 jobs and performs poorly in large-scale problems. R. Pan et al. (Pan, Dong, and Han 2020) applied PN (Vinyals, Fortunato, and Jaitly 2015) in solving the PFSS and explored the actor-critic to train the PFSS models, which achieved the state-of-the-art performances for solving the PFSS. Z. Pan et al. (Pan et al. 2021) also used actor-critic to train the model. The difference was that (Pan et al. 2021) still used recurrent neural network (RNN) as an encoder whereas (Pan, Dong, and Han 2020) used LSTM as an encoder which incorporated the gating mechanism. However, those RL-based methods take a long time for the network training until the convergence, and the solution accuracy still needs more improvements. Our experiments mainly compared with (Pan, Dong, and Han 2020). For more related works (such as learning-based methods for combinatorial optimization, and imitation learning), please refer to *Appendix A*.

3   Problem Descriptions

As shown in Figure 1(a), the goal of the PFSS is to output the optimal permutation of the jobs, which minimizes the makespan. We denote the number of jobs as n and the number of machines as m ($m \geq 2$). Let i be the machine index, j be the job index, $i \in I=\{1, 2, ..., m\}$, $j \in J=\{1, 2, ..., n\}$. Then the inputs and outputs for the PFSS are given as:

- *Inputs*: matrix of processing times $\boldsymbol{X}_{m \times n}$. Each element x_{ij} denotes the processing time of job j on machine i, $x_{ij} \geq 0$, $x_{ij} \in \boldsymbol{X}_{m \times n}$;

- *Outputs*: the optimal permutation $\boldsymbol{\tau}^*=[\tau_0^*, \tau_1^*, ..., \tau_{n-1}^*]$, where the corresponding makespan is minimized.

Figure 1(b) utilizes the Gantt charts to reveal the scheduling process of different job permutations. Givens the same inputs $\boldsymbol{X}_{m \times n}$, different job permutation leads to different makespan. Herein, the PFSS process can be characterized by the following properties:

- It is assumed that the start time is zero.

- One machine corresponds to only one operation. One machine can only process one job at a time.

- Each job needs to be processed on every machine once, with the pre-defined operational order.
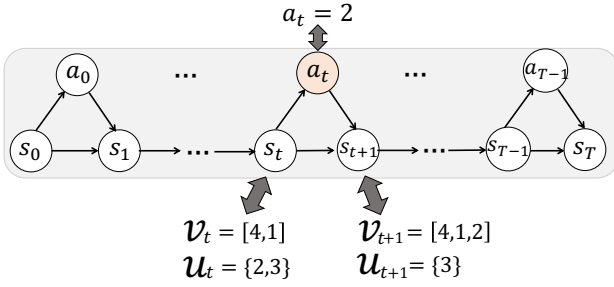
$a_t = 2$

$\mathcal{V}_t = [4,1]$ $\mathcal{V}_{t+1} = [4,1,2]$
$\mathcal{U}_t = \{2,3\}$ $\mathcal{U}_{t+1} = \{3\}$

Figure 2: An example of the PFSS as a MDP: the state s_t transits to state s_{t+1} through the action a_t, which is obtained via the policy $\pi_\theta(\mathbf{a}|s_t)$. Each state is a concatenation of scheduled job list $\mathcal{V}$ and unscheduled job set $\mathcal{U}$. The figure is scheduling $T=n=4$ jobs with final output $\boldsymbol{\tau} = [4, 1, 2, 3]$.

- All the jobs share the same operational order.
- Once started, the operation can not be interrupted.

According to (Ríos-Mercado and Bard 1998), the PFSS can be mathematically formulated as a mixed-integer programming model, which is given in the *Appendix B*.

4   Proposed Method

In this Section, we formulate the PFSS as a Markov decision process (MDP) (Howard 1960) in Section 4.1, and demonstrate our encoder-decoder policy network in Section 4.2. The policy network training via the IL is shown in Section 4.3. The comparison between the LSTM encoder and our GGCN encoder is exhibited in Figure 3. The training overview of our proposed IL model is given in Figure 4.

4.1. Markov Decision Process for PFSS

The PFSS problem is about scheduling for a permutation of n jobs. If scheduling one job is a one-step decision, then the sequential decisions of totally n steps made for scheduling one n-job PFSS problem can be seen as MDP, as shown in Figure 2. Considering the PFSS-solving process as the environment and the scheduling method as the agent, we set the state, action, and policy as:

- *State s_t*: the current state of the scheduling, represented by $[\mathcal{V}_t, \mathcal{U}_t]$, meaning the concatenation of the already scheduled job list and the remaining unscheduled job set. $t \in \{0, 1, 2, ..., n\}$.
- *Action a_t*: a job index, chosen to process next from the unscheduled job set. The action is obtained by the policy network with masking, which enforces that the selected job as action a_t is always from the unscheduled set $\mathcal{U}_t$. The masking mechanism can help to avoid selecting the scheduled job again. $t \in \{0, 1, ..., n-1\}$.
- *Policy $\pi_\theta(\mathbf{a}|s_t)$*: the policy π determines how the actions proceed, where θ represents the weights of the network. It outputs a probability distribution over all jobs, denoted by $\boldsymbol{p} = \pi_\theta(\mathbf{a}|s_t)$. $\mathbf{a}$ is the set of actions.

Scheduling a n-job PFSS problem can be seen as a n-step sequential decision process. The initial state s_0 is that all jobs are unscheduled, and the final state s_n is that all job are scheduled to obtain the permutation of n

jobs, denoted by $\boldsymbol{\tau}=[\tau_0, \tau_1, ..., \tau_{n-1}] \in \mathcal{T}$, where $\tau_t=a_t$, $t\in\{0, 1, 2, ..., n-1\}$.

4.2. Policy Network

We follow the encoder-decoder structure of previous work for the PFSS problems (Pan, Dong, and Han 2020). We incorporate the graph structure GGCN (Bresson and Laurent 2018) as an encoder and follow previous work to utilize the attention mechanism as the decoder. Particularly, the motivations and the technical details for the encoder and decoder are given in the following from **1)** to **4)**. We explain how our model pre-trained on small-sized jobs can be generalized to larger-sized jobs in **5)**.

1) Motivations of Using Graph Encoder. The goal of our PFSS task is to find the optimal job permutation which minimizes the makespan. The input is $\boldsymbol{X}_{m \times n}$, where j-th column vector $\boldsymbol{x}_j \in \mathbb{R}^m$ denotes the feature for job j, $j=\{1, .., n\}$. If all jobs have the same features, the problem could be trivially solved, because any permutation leads to the same makespan; Otherwise, different permutations correspond to different makespans. Thus we believe that the difference between job features could impact on the permutation results. Based on this observation, we incorporate the graph structure, regarding one job feature as one node and using the edge to represent the difference between two jobs. As mentioned in Eq.(1), we initialize the edge by the embedding of the Euclidean distance between two adjacent jobs. In our experiments, we evaluate how job difference influences the permutation results in Section 5.2.(5) and Figure 5(c). Actually, the results proved our assumption. Compared to the previous recurrence-based LSTM encoder, our graph-based GGCN encoder explicitly formulates the embeddings of job and job differences using the node and edge modules, respectively.

2) Graph Encoder. In the n-job m-machine PFSS problem, the input is a matrix $\boldsymbol{X}_{m \times n}$ of all processing times, where $x_{ij} \in \boldsymbol{X}_{m \times n}$ represents the processing time of job j on machine i. We denote $\boldsymbol{x}_j^T=[x_{1j}, x_{2j}, ..., x_{mj}]$ as the input features of job j. As shown in Figure 3, the encoder starts with a fully connected graph, and derives a sparse graph using sparsification techniques. We follow Dai et. al. (Dai et al. 2017) to use a fixed graph diameter and connect each node in the graph to its $n \times 20\%$-nearest neighbors by default. Anisotropic GNNs such as GGCN (Bresson and Laurent 2018) have shown competitive performances across several challenging domains (Joshi et al. 2021). In this paper, we choose GGCN as our encoder. Let $\boldsymbol{h}_j^\ell$ and $\boldsymbol{e}_{jk}^\ell$ denote the job and edge feature at layer ℓ associated with job j and edge jk, $\ell \in \{0, 1, ..., \mathcal{L}-1\}$,

$$\begin{cases} \boldsymbol{h}_j^0 = W^h \boldsymbol{x}_j, \ \boldsymbol{h}_k^0 = W^h \boldsymbol{x}_k, \ \boldsymbol{e}_{jk}^0 = W^e \cdot \|\boldsymbol{x}_j - \boldsymbol{x}_k\|_2 \\ \boldsymbol{h}_j^{\ell+1} = \boldsymbol{h}_j^\ell + \mathrm{ReLU}\Big(\mathrm{Nm}(B^\ell \boldsymbol{h}_j^\ell + \mathrm{Ag}_{k \in \boldsymbol{N}_j}(\sigma(\boldsymbol{e}_{jk}^\ell) \odot C^\ell \boldsymbol{h}_k^\ell))\Big) \\ \boldsymbol{e}_{jk}^{\ell+1} = \boldsymbol{e}_{jk}^\ell + \mathrm{ReLU}\big(\mathrm{Nm}(D^\ell \boldsymbol{e}_{jk}^\ell + E^\ell \boldsymbol{h}_j^\ell + F^\ell \boldsymbol{h}_k^\ell)\big), \end{cases}$$
(1)

where $\boldsymbol{x}_j, \boldsymbol{x}_k \in \mathbb{R}^m$ are the input job features, $\boldsymbol{h}_j^0, \boldsymbol{h}_k^0 \in \mathbb{R}^d$ are their linear projections via $W^h \in \mathbb{R}^{d \times m}$. $\boldsymbol{e}_{jk}^0 \in \mathbb{R}^d$ is the embedding of the Euclidean distance of j and k via
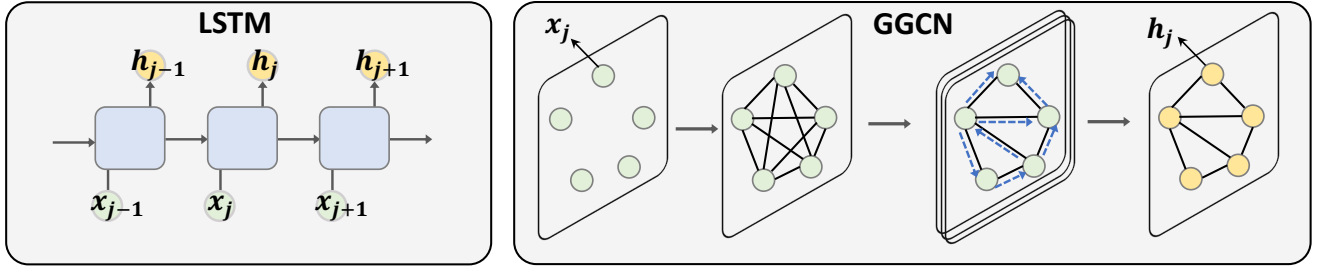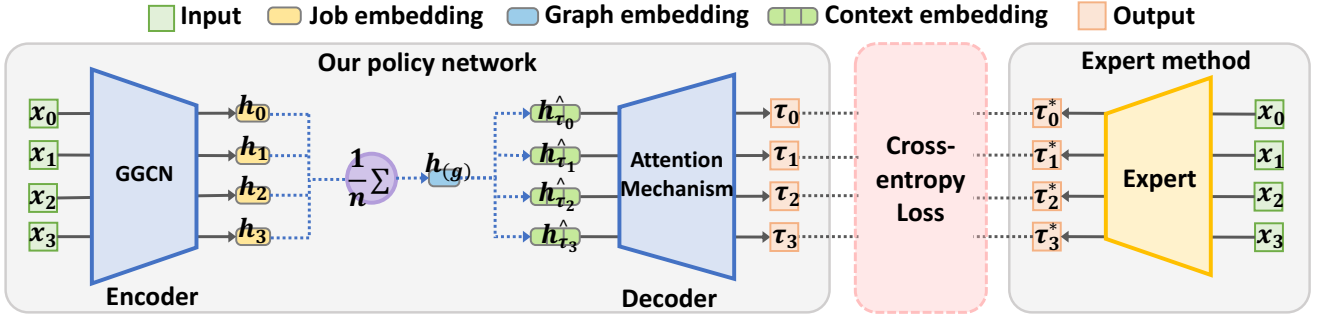
Figure 3: The comparison between the two encoders. Left: recurrence-based encoder. Right: graph-based encoder.

Figure 4: The training overview of our proposed IL method for solving n-job m-machine PFSS.

$W^e \in \mathbb{R}^d$. And $B^\ell, C^\ell, D^\ell, E^\ell, F^\ell \in \mathbb{R}^{d \times d}$ are the parameters to learn. Nm$(\cdot)$ denotes the method of normalization: LayerNorm (Ba, Kiros, and Hinton 2016) or BatchNorm (Ioffe and Szegedy 2015). Ag$(\cdot)$ means the aggregation functions: SUM, MEAN or MAX. σ is the sigmoid function, and $\odot$ is the Hadamard product. We make the aggregation function anisotropic using the edge gates $\sigma(e_{jk})$ via dense attention mechanism which scales the neighborhood features $h_k, \forall k \in N_j = \{0, ..., n-1\} \backslash \{j\}$, N_j is the neighbor set of j.

3) Motivations of Using Attention Decoder. The PFSS problems actually have feasibility constraints. When decoding, previously scheduled jobs cannot be selected next. Thus, the earlier scheduled jobs have more impact on the overall makespan than the latter jobs. It is not technically enough if the decoder just uses a simple masking procedure as Eq.(4). In essence, the attention mechanism allows to focus more on a certain part. Thus we apply the attention, aiming to focus more on the front jobs, to improve the solution quality regarding the overall makespan.

4) Attention Decoder. We follow the previous RL work (Pan, Dong, and Han 2020) to use the attention mechanism (Vaswani et al. 2017) in our decoder. Decoding can be seen as a n-step sequential decision-making process. For the n-job PFSS problem, at time-step $t \in \{0, 1, ..., n-1\}$, the state of the scheduling process can be denoted as s_t, and according to the policy π_θ, the decoder will output a selected job to schedule next, denoted as action a_t.

There are $\mathcal{L}$ layers in the encoder, and the graph embedding is $h_{(g)} = \frac{1}{n} \sum_{k=0}^{n-1} h_k^{\mathcal{L}}$, where $h_k^{\mathcal{L}}$ is the job embedding. In Figure 4, we omit the superscript $\mathcal{L}$ for readability. At time-step t, the decoder constructs a context embedding $\hat{h}_{(c)}$ using the encoder and the partial output of the decoder,

$$\hat{h}_{(c)} = \hat{h}_{\tau_t} = \begin{cases} [h_{(g)}, h_{\tau_0}^{\mathcal{L}}, h_{\tau_{t-1}}^{\mathcal{L}}] & t = \{1, ..., n-1\} \\ [h_{(g)}, v, v] & t = 0, \end{cases} \quad (2)$$

where the v is the placeholders for the first time decoding, $h_{(g)}$ is the graph embedding, $h_{\tau_0}^{\mathcal{L}}$ is the first job embedding, $h_{\tau_{t-1}}^{\mathcal{L}}$ is the previous job embedding, and $[\cdot, \cdot, \cdot]$ is the horizontal concatenation operator. The context embedding $\hat{h}_{(c)}$ can be interpreted as the $(3 \cdot d)$-dimensional concatenated job embedding, denoted by $\hat{h}_{(c)} = \hat{h}_{\tau_t}$. Next, one more layer of standard MHA refines the context embedding $\hat{h}_{(c)}$ to get $h_{(c)}$,

$$h_{(c)} = \text{MHA}\left(Q = \hat{h}_{(c)}, K = \{h_0^{\mathcal{L}}, .., h_{n-1}^{\mathcal{L}}\}, V = \{h_0^{\mathcal{L}}, .., h_{n-1}^{\mathcal{L}}\}\right). \quad (3)$$

The MHA uses $M = 8$ heads, with input Q, K, V. The logits $u_{(c)j}$ for the edge $e_{(c)j}$ are obtained via a final single-head attention,

$$u_{(c)j} = \begin{cases} r \cdot \tanh\left(\frac{(W^Q h_{(c)})^T \cdot (W^K h_j^{\mathcal{L}})}{\sqrt{d}}\right) & \text{if } j \in \mathcal{U}_t \\ -\infty & \text{otherwise,} \end{cases} \quad (4)$$

where the tanh function is used to maintain the value of logits within $[\text{-}r, r]$ ($r = 10$) (Bello et al. 2017), W^Q and W^K are the parameters to learn, $\mathcal{U}_t$ is the current unscheduled job set. The masking enforces that the scheduled jobs will not be selected again by setting the logits of those scheduled jobs to $-\infty$. We calculate the final output probability vector p using a softmax, each element p_j can be obtained by,

Testing Job Size		PFSS-20			PFSS-50			PFSS-100		
Type	Method	Makespan↓	Gap↓	Time↓	Makespan↓	Gap↓	Time↓	Makespan↓	Gap↓	Time↓
Heuristics	Random search	34.1 •	16.8%	**1.2s**	72.5 •	16.0%	**2.8s**	132.4 •	12.8%	**5.4s**
	Iterated local search	30.0	2.7%	31.0s	65.2 •	4.3%	69.7s	121.7 •	3.7%	131.2s
	Iterated greedy	29.3	0.3%	215.9s	63.5	1.6%	1303.9s	119.2	1.5%	5207.9s
	NEH [Expert]	**29.2**	**0.0%**	12.9s	**62.5**	**0.0%**	200.1s	**117.4**	**0.0%**	1544.6s
RL	Actor-critic	31.6 •	8.2%	**3.8s**	66.8 •	6.9%	**4.5s**	123.8 •	5.4%	**6.3s**
IL	Behavioral cloning [Ours]	**30.2**	**3.4%**	4.0s	**63.7**	**1.9%**	5.1s	**118.3**	**0.8%**	7.2s
Testing Job Size		PFSS-200			PFSS-500			PFSS-1000		
Type	Method	Makespan↓	Gap↓	Time↓	Makespan↓	Gap↓	Time↓	Makespan↓	Gap↓	Time↓
Heuristics	Random search	243.8 •	9.0%	**1.1s**	570.5 •	6.2%	**2.7s**	1103.4 •	5.3%	**54.2s**
	Iterated local search	230.2 •	3.0%	25.1s	549.4 •	2.5%	65.4s	1073.1 •	2.4%	140.0s
	Iterated greedy	229.3 •	2.5%	0.6h	542.1	0.9%	3.7h	1060.8	1.2%	14.6h
	NEH [Expert]	**223.6**	**0.0%**	0.3h	**537.2**	**0.0%**	5.5h	**1048.2**	**0.0%**	43.1h
RL	Actor-critic	234.6 •	4.9%	**17.2s**	574.2 •	6.9%	**41.4s**	1138.9 •	8.7%	**81.8s**
IL	Behavioral cloning [Ours]	**224.6**	**0.4%**	20.0s	**540.7**	**0.6%**	45.3s	**1055.5**	**0.7%**	89.9s

Table 1: Comparative study on the generated datasets. We report the *average* makespan, the *average* makespan gap, and the *sum* of runtime. All results are the average of *three* trials. The best performances are in bold among the heuristics and the learning methods, respectively. '•' indicates that the makespan decrease of our method over the baseline method is statistically significant (via Wilcoxon signed-rank test at 5% significance level.)

$$p_j = \frac{e^{u_{(c)j}}}{\sum_k e^{u_{(c)k}}}. \tag{5}$$

According to the output probability vector $\boldsymbol{p}$, we can select the job whose probability is the highest to process next, denoted by τ_t or a_t. Update the scheduled job list $\mathcal{V}$ by appending a_t, and update the unscheduled set $\mathcal{U}$ by eliminating a_t.

From Eq.(2) to (5), after taking all the n steps for the decoding process, we will obtain the final output permutation of all jobs, denoted by $\boldsymbol{\tau} = (\tau_0, \tau_1, ..., \tau_{n-1}) \in \mathcal{T}$.

5) Availability of Generalization. We would like to explain how our model can be pre-trained only on small job sizes and then generalized to any other job size. We can briefly interpret the encoding and decoding process by the projection: $\boldsymbol{H}_{d \times n} = W^{en}_{d \times m} \boldsymbol{X}_{m \times n}$ and $\boldsymbol{O}_{1 \times n} = W^{de}_{1 \times d} \boldsymbol{H}_{d \times n}$, where $\boldsymbol{X}, \boldsymbol{H}, \boldsymbol{O}, d$ refer to the input, hidden representation, output and hidden dimension. W^{en} and W^{de} denote the network weights for the encoder and decoder, respectively. As mentioned above, one step of decoding obtains one job and we need to go through n steps to obtain the permutation of n jobs. It is obvious that the dimensions of $W^{en}_{d \times m}$ and $W^{de}_{1 \times d}$ are only relevant to m and d, not to n. To that end, it enables the generalizability to different job sizes, though pre-trained on small job sizes. However, their machine number m for training and testing sets should always be consistent.

4.3. Expert-Driven Imitation Learning

Motivations. RL is a currently popular approach to finding good scheduling policies, such as (Pan, Dong, and Han 2020). However, RL methods train from scratch and possibly run into many issues: the randomly initialized policies may perform poorly, and the convergence could be slow in training. In this paper, instead of using RL, we choose to learn directly from an expert method, referred to as IL (Hussein et al. 2017).

During our model training using IL, we chose the state-of-the-art NEH heuristic (Sharma and Sharma 2021) as the expert method. The mathematical model as formulated in the *Appendix 1* solved by branch-and-bound method (Gmys et al. 2020) is another potential option as the expert method, due to its exact solutions. However, the computational time is much slower than expected, and the processing times of inputs require only to be integers. Considering these limitations, we would rather choose the NEH heuristic as the expert method. More experimental analysis is given in *Appendix C*.

We choose behavior cloning (Torabi, Warnell, and Stone 2018) as the training method of IL. We firstly run the expert on a set of training instances and record a dataset of expert state-action pairs $\mathcal{R} = \{(s^i_t, a^{*i}_t)^{n-1}_{t=0}\}^{N_1-1}_{i=0}$, where N_1 is the number of training instances, and each instance is a n-job m-machine PFSS problem. The set of actions by the expert constructs a permutation of job: $\boldsymbol{\tau}^* = \{\tau^*_0, \tau^*_1, ..., \tau^*_{n-1}\} = \{a^*_0, a^*_1, ..., a^*_{n-1}\}$. The policy network is learned by minimizing the cross-entropy loss,

$$\mathcal{J}(\theta) = -\frac{1}{N_1 \cdot n} \sum_{i=0}^{N_1-1} \sum_{t=0}^{n-1} \log \pi_\theta(a^{*i}_t | s^i_t). \tag{6}$$

5 Experiments

5.1. Setup

1) Datasets. To evaluate the efficacy of our method, we conduct the experiments on two randomly generated datasets and two benchmarks (Taillard (Taillard 1993) and the VRF (Vallada, Ruiz, and Framinan 2015)). Considering that in real life jobs are not always evenly and regularly distributed, we thus choose two challenging distributions for random generation, one is Gamma distribution with $k=1, \theta=2$, and the other is Normal distribution with $\mu=6, \sigma=6$. Due to space limitations, we only record the results on the Gamma

	Instance size for testing: (n,m)	(50,5)		(100,5)		(100,20)		(200,20)		(500,20)	
	Method	Gap ↓	Time↓	Gap↓	Time↓	Gap↓	Time↓	Gap↓	Time↓	Gap↓	Time↓
Taillard	NEH	0%	2.0s	0%	15.9s	0%	58.8s	0%	468.2s	0%	2.0h
	RL	14.6%	**3.3s**	13.2%	**3.5s**	**12.5%**	**3.6s**	12.6%	**3.8s**	9.9%	**5.1s**
	IL [Ours]	**12.8%**	4.5s	**12.3%**	3.6s	**12.5%**	4.8s	**10.7%**	4.6s	**9.4%**	5.2s
	Instance size for testing: (n,m)	(40,5)		(60,5)		(600,20)		(700,20)		(800,20)	
	Method	Gap ↓	Time↓	Gap↓	Time↓	Gap↓	Time↓	Gap↓	Time↓	Gap↓	Time↓
VRF	NEH	0%	0.9s	0%	3.4s	0%	3.4h	0%	5.8h	0%	8.3h
	RL	17.4%	**3.5s**	14.5%	**3.3s**	9.7%	**4.2s**	10.0%	**5.3s**	9.1%	**5.1s**
	IL [Ours]	**15.9%**	4.1s	**14.1%**	4.8s	**9.1%**	5.6s	**9.5%**	5.5s	**8.2%**	6.4s

Table 2: Comparative study on the benchmarks Taillard and VRF. We compare our IL method with NEH heuristics and the state-of-the-art RL method. The better performances between ours and the RL method are in bold.

Method	Getting Expert	Training	Testing	**Total**
NEH	/	/	$\mathcal{O}(N_3 \cdot mn^3)$	$\mathcal{O}(N_3 \cdot mn^3)$
RL	/	$\mathcal{O}(N_1 \cdot (m\hat{n} \cdot F_{RL} + m\hat{n} \cdot B_{RL}))$	$\mathcal{O}(N_3 \cdot mn \cdot F_{RL})$	$\mathcal{O}(N_1 \cdot m\hat{n}(F_{RL} + B_{RL}) + N_3 F_{RL} \cdot mn)$
IL [Ours]	$\mathcal{O}(N_1 \cdot m\hat{n}^3)$	$\mathcal{O}(N_1 \cdot (m\hat{n} \cdot F_{IL} + m\hat{n} \cdot B_{IL}))$	$\mathcal{O}(N_3 \cdot mn \cdot F_{IL})$	$\mathcal{O}(N_1 \cdot m\hat{n}(\hat{n}^2 + F_{IL} + B_{IL}) + N_3 F_{IL} \cdot mn)$

Table 3: Time complexity analysis on different methods. The total time complexities come from: getting expert time (only for our IL method), training time (only for learning-based methods), and testing time.

	Dataset:	VRF		Taillard		Generated Datasets					
	Testing size: (N_3, n, m)	(10,60,5)		(10,100,5)		(100,20,5)		(100,100,5)		(100,1000,5)	
Method	Expert↓	Training↓	Testing↓	Total↓	Testing↓	Total↓	Testing↓	Total↓	Testing↓	Total↓	Testing↓	Total↓
NEH	0	0	3.4s	**3.4s**	15.9s	**15.9s**	1.2s	**1.2s**	154.5s	**0.04h**	43.1s	43.1h
RL	0	1550.0s	**3.3s**	0.4h	**3.5s**	0.4h	3.0s	0.4h	**3.6s**	0.4h	**81.8s**	0.5h
IL [Ours]	165.1s	620.0s	4.8s	0.2h	3.6s	0.2h	3.3s	0.2h	3.7s	0.2h	89.9s	**0.3h**

Table 4: Time cost analysis on different methods. We evaluate different datasets, based on the same pre-trained model on 12800 training instances with $\hat{n}=20, m=5$. The total time consists of getting expert time, training time, and testing time.

distribution in the paper and put the detailed results on the Normal distribution in the *Appendix D*.

2) Implementation Details. Let N_1, N_2, N_3 be the number of training, validation, and testing instances, and $\hat{n}, n, m$ denote the training job size, the testing job size, and the machine size, respectively. m is set to 5 for all phases by default. We train the model on $N_1=12800$ instances with $\hat{n}=20$, validate on $N_2=1000$ instances with the same job size. We generalize the small pre-trained model to any larger job sizes. For the random datasets, we test on $N_3=1000$ instances with $n=20, 50, 100$, and test on $N_3=100$ instances with $n=200, 500, 1000$, respectively. For the two benchmarks, we use our pre-trained model based on Gamma distribution with $\hat{n}=20, m=5$ or $m=20$, to test different sets of instances, where each testing size has 10 instances.

For these learning methods, the batch size is set to 128, and the learning rate is initialized as $1e-4$ with 0.96 per epoch as decay. We train for 50 epochs. The experiments are conducted in Ubuntu 20.04 LTS 64-bit System with Intel(R) Xeon(R) Silver 4214 2.20GHz ×48 CPU and one NVIDIA GeForce RTX 2080Ti GPU. All methods are implemented by Python, and the two learning methods use Pytorch.

3) Baselines. We compare our method with some heuristics and learning-based methods. We consider four of the popular and mostly-used heuristics including random search (Zabinsky et al. 2009), iterated local search (Lourenço, Martin, and Stützle 2019), iterated greedy (Ruiz, Pan, and Naderi 2019), and NEH algorithm (Sharma and Sharma 2021). We also compare with the state-of-the-art RL-based methods, where actor-critic is used to solve the PFSS (Pan, Dong, and Han 2020).

4) Evaluations. We evaluate these methods from three perspectives: the *average* makespan, the *average* makespan gap towards the expert method (the NEH algorithm), and the *sum* of runtime. For all of the evaluation criteria, the lower the better. We run all the experiments for *three* different seeds and record the *average* results. We find that the results of our method are robust against different seeds.

5.2. Results and Analysis

1) Comparative Study. Table 1 reveals the comparisons on the random datasets under the Gamma distribution. Table 2 shows the comparisons of the two benchmarks. According to Table 1, the NEH algorithm generally achieves the lowest makespan performance among the four heuristics. However, as job size increases, the time of NEH is rapidly increasing. Compared with the heuristics, learning methods are fast in computations when testing. The testing time of the RL
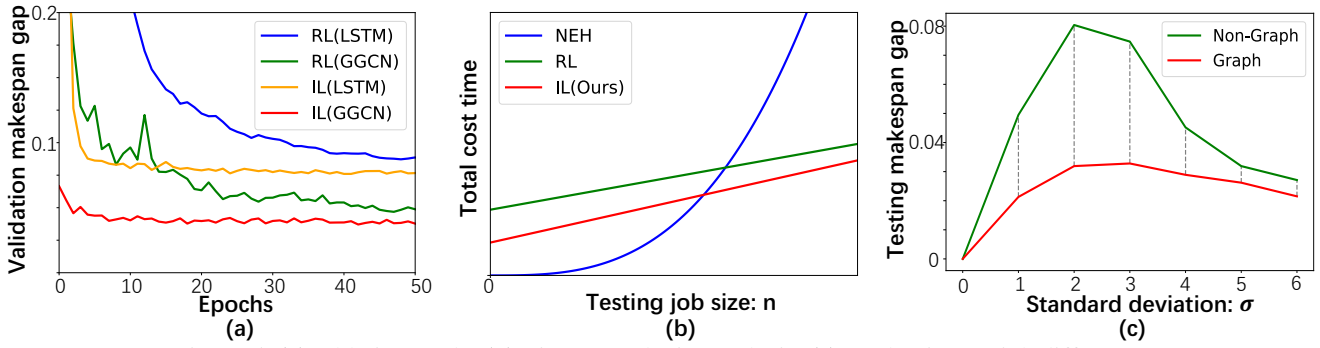
Figure 5: **(a)** Ablation study. **(b)** Time complexity analysis. **(c)** Evaluation on job difference.

Model Training Method	Training No. Para.	Time per epoch	Testing Makespan
RL(LSTM)	709k+267k	31.0s	31.6
RL(GGCN)	365k+267k	18.6s	30.6
IL (LSTM)	709k	13.6s	31.0
IL(GGCN) [Ours]	**365k**	**12.4s**	**30.2**

Table 5: Ablation study on different training models.

model is a bit faster than ours. The reason is that the RL model merely uses a single network when testing, though it requires bi-network during training, and our IL model has the complex graph structure GGCN. However, our proposed IL model outperforms the RL model in the makespan accuracy, where the makespan gap towards the NEH algorithms decreases by 5.5% on average from 6.8% (4.9%∼8.7%) to 1.3% (0.4%∼3.4%). According to Table 2, our IL model also shows excellent generalization capability to the unseen benchmark instances, outperforming the RL model in the makespan accuracy, though a little bit slower in time.

2) Ablation Study. We conduct the ablation study to show the efficacy of the GGCN encoder and IL method, as shown in Table 5 and Figure 5(a). Pan et al. (Pan, Dong, and Han 2020) used RL for training and LSTM for the encoder. Our method uses IL and GGCN. Both methods use the attention mechanism for the decoder. For different methods, we use the pre-trained model with $\hat{n}=20$ to validate and test on 1000 instances with $n=20$. The results show that our IL model is much lighter with fewer parameters and trains much faster than others. Notably, compared with (Pan, Dong, and Han 2020), our model's network parameters are only 37% of theirs. Figure 5(a) reveals the training processes, where our model leads to faster and more stable convergence.

3) Time Complexity Analysis. We give detailed time complexity analysis among our IL model and the state-of-the-art NEH heuristic (Sharma and Sharma 2021) and the state-of-the-art RL method (Pan, Dong, and Han 2020). The total complexity of our IL method consists of three parts: getting expert solutions, network training, and testing. Their complexities are summarized in Table 3 and Figure 5(b). We use F_{RL}, B_{RL} to denote the complexity of one forward pass and one backward pass of the RL network, while F_{IL}, B_{IL} is for our IL method. With respect to the testing job size n, NEH has *polynomial* cost while our model has merely *linear* cost.

From the mathematical perspective, with the testing job size n and the number of testing instances N_3 increasing, the efficiency of the learning method will be further highlighted.
4) Time Cost Analysis. We use the same pre-trained model on 12800 training instances with $\hat{n}=20, m=5$, then choose 5 different testing sets and separately analyze their total time costs, as shown in Table 4. For the NEH algorithm, the total time is just the testing time. For the RL method, the total time contains the training time and the testing time. For our IL method, the total time contains all three parts. From Table 4, we can see that for the sets with small job sizes such as (10,60,5), (10,100,5), and (100,20,5), the NEH is efficient. For the three generated sets, with the increasing of job size n, the total time of the NEH increases rapidly while the total time of learning-based methods go increasing steadily. Generally, our IL method has lower total time costs than the RL method, which corresponds to Figure 5(b).
5) Evaluations on Job Differences. As mentioned in Section 4.2.(1), we believe that the difference between job features could impact the final results. Then we follow the Normal distribution, set the mean $\mu=6$, set the standard deviation σ from 0 to 6 for the testing sets, and evaluate how the makespan gaps towards NEH present for non-graph structure (LSTM) and graph structure (GGCN), when both using the IL method. As shown in Figure 5(c), when $\sigma=0$, all jobs are the same and both models show the same performance; When σ increases, the advances of GGCN over LSTM firstly increase and then decrease. The graph structure generally outperforms the non-graph structure.
6) More Experiments Are in The Appendix. More experiments are put in the Appendix, including the evaluations on the Normal distribution (*Appendix D*), different machine numbers (*Appendix E*), the composition analysis of GGCN (*Appendix F*), and the evaluations on TSP (*Appendix G*).

6 Conclusion

The PFSS is a significant CO problem with wide applications. In this paper, we introduce why and how to use the graph-based imitation learning method in solving PFSS, which leads to a lighter network, faster and more stable convergence, and lower makespan gaps, compared to the state-of-the-art RL method. The extensive experiments on generated and benchmark datasets clearly exhibit the competitiveness of our proposed method.

Acknowledgments

References

Alfaro-Fernández, P.; Ruiz, R.; Pagnozzi, F.; and Stützle, T. 2020. Automatic algorithm design for hybrid flowshop scheduling problems. *European Journal of Operational Research*, 282(3): 835–845.

Ba, J. L.; Kiros, J. R.; and Hinton, G. E. 2016. Layer normalization. *Advances in NIPS Deep Learning Symposium.*

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2017. Neural combinatorial optimization with reinforcement learning. *International Conference on Learning Representations.*

Bengio, Y.; Lodi, A.; and Prouvost, A. 2020. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research.*

Bresson, X.; and Laurent, T. 2018. An experimental study of neural networks for variable graphs. *International Conference on Learning Representations Workshop.*

Dai, H.; Khalil, E. B.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. *In Advances in neural information processing systems.*

Gmys, J.; Mezmaz, M.; Melab, N.; and Tuyttens, D. 2020. A computationally efficient Branch-and-Bound algorithm for the permutation flow-shop scheduling problem. *European Journal of Operational Research*, 284(3): 814–833.

Howard, R. A. 1960. Dynamic programming and markov processes. *MIT Press and Wiley.*

Hussein, A.; Gaber, M. M.; Elyan, E.; and Jayne, C. 2017. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2): 1–35.

Ioffe, S.; and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, 448–456. PMLR.

Joshi, C. K.; Cappart, Q.; Rousseau, L.-M.; Laurent, T.; and Bresson, X. 2021. Learning TSP requires rethinking generalization. *International Conference on Principles and Practice of Constraint Programming (CP).*

Lenstra, J. K.; and Kan, A. R. 1981. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2): 221–227.

Li, L.; Fu, X.; Zhen, H.-L.; Yuan, M.; Wang, J.; Lu, J.; Tong, X.; Zeng, J.; and Schnieders, D. 2022. Bilevel learning for large-scale flexible flow shop scheduling. *Computers & Industrial Engineering*, 168: 108140.

Li, L.; and Wu, B. 2022. Learning to accelerate approximate methods for solving integer programming via early fixing. *arXiv preprint arXiv:2207.02087.*

Lourenço, H. R.; Martin, O. C.; and Stützle, T. 2019. Iterated local search: Framework and applications. In *Handbook of metaheuristics*, 129–168. Springer.

Monteiro, J.; Gavenski, N.; Granada, R.; Meneguzzi, F.; and Barros, R. 2019. Augmented behavioral cloning from observation. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19).*

Pan, R.; Dong, X.; and Han, S. 2020. Solving permutation flowshop problem with deep reinforcement learning. In *2020 Prognostics and Health Management Conference (PHM-Besançon)*, 349–353. IEEE.

Pan, Z.; Wang, L.; Wang, J.; and Lu, J. 2021. Deep Reinforcement Learning Based Optimization Algorithm for Permutation Flow-Shop Scheduling. *IEEE Transactions on Emerging Topics in Computational Intelligence.*

Ren, J.; Ye, C.; and Yang, F. 2021. Solving flow-shop scheduling problem with a reinforcement learning algorithm that generalizes the value function with neural network. *Alexandria Engineering Journal*, 60(3): 2787–2800.

Ribas, I.; Leisten, R.; and Framiñan, J. M. 2010. Review and classification of hybrid flow shop scheduling problems from a production system and a solutions procedure perspective. *Computers & Operations Research*, 37(8): 1439–1454.

Ríos-Mercado, R. Z.; and Bard, J. F. 1998. Computational experience with a branch-and-cut algorithm for flowshop scheduling with setups. *Computers and Operations Research*, 25(5): 351–366.

Ruiz, R.; Pan, Q.-K.; and Naderi, B. 2019. Iterated Greedy methods for the distributed permutation flowshop scheduling problem. *Omega*, 83: 213–222.

Sharma, M.; and Sharma, S. 2021. An improved NEH heuristic to minimize makespan for flow shop scheduling problems. *Decision Science Letters*, 10(3): 311–322.

Soukhal, A.; Oulamara, A.; and Martineau, P. 2005. Complexity of flow shop scheduling problems with transportation constraints. *European Journal of Operational Research*, 161(1): 32–41.

Taillard, E. 1993. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2): 278–285.

Torabi, F.; Warnell, G.; and Stone, P. 2018. Behavioral cloning from observation. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18).*

Vallada, E.; Ruiz, R.; and Framinan, J. M. 2015. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3): 666–677.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. *Advances in neural information processing systems*.

Zabinsky, Z. B.; et al. 2009. Random search algorithms. *Department of Industrial and Systems Engineering, University of Washington, USA*.