applied sciences

MDPI

Article

Minimizing the Late Work of the Flow Shop Scheduling Problem with a Deep Reinforcement Learning Based Approach

Zhuoran Dong, Tao Ren *, Jiacheng Weng, Fang Qi and Xinyue Wang

Department of Software, Northeastern University, Shenyang 110819, China; 17640254655@163.com (Z.D.); xshxdwjj@163.com (J.W.); 1971147@stu.neu.edu.cn (F.Q.); chinaxyw@163.com (X.W.)
* Correspondence: chinarentao@163.com

Abstract: In the field of industrial manufacturing, assembly line production is the most common production process that can be modeled as a permutation flow shop scheduling problem (PFSP). Minimizing the late work criteria (tasks remaining after due dates arrive) of production planning can effectively reduce production costs and allow for faster product delivery. In this article, a novel learning-based approach is proposed to minimize the late work of the PFSP using deep reinforcement learning (DRL) and graph isomorphism network (GIN), which is an innovative combination of the field of combinatorial optimization and deep learning. The PFSPs are the well-known permutation flow shop problem and each job comes with a release date constraint. In this work, the PFSP is defined as a Markov decision process (MDP) that can be solved by reinforcement learning (RL). A complete graph is introduced for describing the PFSP instance. The proposed policy network combines the graph representation of PFSP and the sequence information of jobs to predict the distribution of candidate jobs. The policy network will be invoked multiple times until a complete sequence is obtained. In order to further improve the quality of the solution obtained by reinforcement learning, an improved iterative greedy (IG) algorithm is proposed to search the solution locally. The experimental results show that the proposed RL and the combined method of RL+IG can obtain better solutions than other excellent heuristic and meta-heuristic algorithms in a short time.

Keywords: combinatorial optimization; graph neural network; deep reinforcement learning; flow shop scheduling; late work; pointer network; iterated greedy

check for updates

Citation: Dong, Z.; Ren, T.; Weng, J.; Qi, F.; Wang, X. Minimizing the Late Work of the Flow Shop Scheduling Problem with a Deep Reinforcement Learning Based Approach. *Appl. Sci.* **2022**, *12*, 2366. https://doi.org/10.3390/app12052366

Academic Editor: Vincent A. Cicirello

Received: 10 January 2022
Accepted: 22 February 2022
Published: 24 February 2022

1. Introduction

The flow shop scheduling problem (FSSP) plays an important role in manufacturing systems. Optimizing multiple criteria of the FSSP can help reduce manufacturing costs and improve the manufacturing efficiency of enterprises. In recent years, many variants of the FSSP have emerged and many methods have been proposed to optimize some of its criteria. The permutation flow shop scheduling problem (PFSP) is a classical form of the FSSP, which was first introduced and formulated by Johnson [1]. The problem with more than three machines is shown to be NP-hard [2]. The goal of this problem is to schedule operations on the machines to optimize one or more performance criteria, such as minimizing the makespan, mean tardiness, total late work, and total flow time of all jobs. A rational scheduling algorithm can not only improve the efficiency and performance of the system but also reduce the cost of machines. In the past decades, various exact or heuristic algorithms for solving scheduling problems have been suggested [3]. Researchers have generalized the PFSP problem into multiple variants to simulate real production scenarios, such as no-wait flow shop, blocking flow shop, no-idle flow shop, and energy-efficient flow shop.

To solve problems of different scales, many effective exact methods, heuristics, and meta-heuristic algorithms have been proposed to optimize various criteria for the PFSP. Exact methods based on enumeration with an integer programming formulation are usually

employed to find the optimal solution for PFSP. When dealing with large-scale problems, the solution space gets bigger and the use of exact algorithms will always lead to the combinatorial explosion so that the computation time is usually unacceptable. The most commonly applied approaches for large-scale problems are heuristic approaches such as NEH [4] and CDS [5], which are capable of real-time decision making. However, hand-crafted heuristic algorithms only consider limited information which leads to the unstable performance of the algorithm. Meta-heuristic algorithms such as the genetic algorithm (GA) [6], tabu search algorithm (TS) [7] and the particle swarm optimization algorithm (PSO) [8,9] are a class of algorithmic frameworks that are problem-independent. The performance of these algorithms depends on adequate parameter tuning and initial solutions. Moreover, these algorithms have a slow convergence speed on problems that with high computational complexity. With the development of deep learning, reinforcement learning, and some high-performance search algorithms, methods applying these techniques have achieved better performance compared to traditional algorithms on some complex combinatorial optimization problems.

The late work appears to be important in production planning from the perspective of the customer as well as the manager. Customers are often concerned about the completion of the order, because those tasks that are not completed by the due dates need to be processed additionally. Managers are also interested in minimizing the late work criteria, as delays may cause financial losses. The late work phrase was first introduced in literature [10] and is defined as the number of tardy job units. The symbol Y is first used to represent it. Blazewicz et al. describe the difference between late work and other performance criteria such as makespan, tardiness, lateness, and proves that problems with a late work objective function are at least as difficult as problems with a maximum delay criterion [11]. Since then, many exact and heuristic algorithms have been proposed for single machine [12], parallel machine [13], and dedicated machine problems [14] with late work objective functions. The problem of minimizing late work has been demonstrated in many fields such as chip manufacturing [15], computer integrated manufacturing (CIM) [14], supply chain management [16], and software development processes [17]. The flow shop problem with the objective of minimizing late work was first proposed by [18] and solved using a genetic algorithm. Gerstl et al. studied the properties of the problem and extended it to the proportionate shop problem [19], but no high-performance optimization methods for late work have been proposed in recent years.

The FSSP is a branch of combinatorial optimization problems. Many studies applying reinforcement learning methods to solve combinatorial optimization problems have appeared in recent years. Many combinatorial optimization problems can be transformed into multi-stage decision-making problems, which means a sequence of decisions need to be performed to maximize/minimize the objective function. Therefore, some researchers have proposed agents suitable for these problems based on RL [20–22]. These methods can reach a level beyond existing algorithms. The deep reinforcement learning (DRL) methods in these problems can basically be divided into two categories, one that constructs the solution in an end-to-end way, and the other that improves on existing feasible solutions.

In construction-based methods, many studies [20,21,23] are based on the pointer network (PtrNet) [24]. The PtrNet solves the problems of variable size output dictionaries based on a sequence-to-sequence model. In literature [20], the PtrNet was trained using the Actor-Critic algorithm to obtain the distribution over all nodes to solve problems such as a knapsack problem. In literature [25], a simplified version of PtrNet is presented that is capable of handling routing problems in both static and dynamic environments. At each time step, the embedding of static elements as input to the RNN decoder, the output of the RNN, and the embedding of dynamic elements as output of the attention mechanism, the decision is made from the distribution on available destinations.

Different from construction heuristics, some studies [26–28] focused on improvements to existing solutions. Chen et al. designed a model called NeuRewriter and applied it to several domains [26]. First, a region is selected by a region selection policy, then a

rewrite rule is obtained by a region rewrite policy, a local solution is used instead of the original solution to obtain an improved solution, and the process is repeated until convergence. Another work [28] used only one policy network to select a solution within a neighborhood structured by pairwise local operators, surpassing [26] in both solution quality and generalization capability on the routing problems.

Most of the approaches used graph-independent sequence-to-sequence mapping and did not make full use of the graph structure of graph-based problems. In order to make full use of graph structure, graph embedding and the graph neural network (GNN) have been introduced to solve graph-based combinatorial optimization problems [29,30], which can take into account nodes, edges, and their accompanying labels, attributes, text, and other information in a network, enabling better use of the network structure for modeling and reasoning.

The traditional methods mentioned above for solving the PFSP are difficult for achieving a trade-off between computation time and solution quality. In shop scheduling problems and even combinatorial optimization problems, the optimization objective is generally to minimize some criteria such as total cost, total completion time, distance traveled, and delayed work. The smaller the value of these criteria, the better the quality of the current solution. Construction-based reinforcement learning methods are able to obtain better solutions in a short time, but the quality of the solutions generally cannot exceed that of the meta-heuristic methods, while improvement-based methods require artificial extraction of features and are difficult to train. This article innovatively proposes an end-to-end reinforcement learning method and an improved iterative greedy method to minimize the late work of PFSP. The contribution of this paper mainly includes the following three aspects.

(1) The proposed approach generates high-quality solutions using an end-to-end architecture based on reinforcement learning. The models can be trained without expert knowledge and labeled data, and the trained models can automatically extract features from the problem.
(2) The PFSP is innovatively regarded as a complete graph. Two multi-layer GIN are used to encode the constraint features and processing time features in the PFSP. The GINs are able to efficiently aggregate the nodes' own features and other neighbors' features to obtain a contextual representation of each node.
(3) An improved iterative greedy method is proposed. The RL model is able to obtain high-quality initial solutions in a short time and the IG method is used to improve the initial solutions. Experimental results show that the RL + IG method surpasses many excellent heuristic and meta-heuristic algorithms.

The rest of this article is organized as follows. Section 2 first describes the PFSP of minimizing the late work objective and models it as a sequential decision process, then describes the deep reinforcement learning architecture used and the training method of the model to generate an initial solution to the problem. Section 3 proposes a hybrid iterative greedy algorithm to further improve the generated initial solution. Section 4 illustrates the experimental setup of this article and shows the results of comparing the proposed algorithm with other methods. Section 5 concludes the article and presents several directions for future work.

2. Generate Initial Solutions to PFSP Using a Deep Reinforcement Learning Method

2.1. The Formulations of PFSP

The PFSP consists of m machines $M = \{M_1, M_2, \ldots, M_m\}$ and n jobs $J = \{J_1, J_2, \ldots, J_n\}$. Let n jobs are to be scheduled on m machines in the same technological order. The schedule needs to satisfy the following assumptions. The jobs must be processed in the same order on each machine, each job needs to be processed on all machines, a machine can only process one job at a time and a job can be processed only after the release date. Other specific assumptions for the problem can be found in [31] and [32]. The symbols and definitions used to formulate the PFSP as an RL problem are described below. The processing time of the operation on each machine of the job is used as the feature vector of the current job,

$x_i = \{r_i, d_i, o_i^s, o_i^1, .., o_i^m\}$, where x_i is the feature vector of the i-th job, o_i^m is the processing time of the operation of the i-th job on the m-th machine, $o_i^s = \sum_{k=1}^{m} o_i^k$, r_i, and d_i are the release date and due date of job i, respectively. The late work can be calculated using the following formulas:

$$C\left(\pi_{(1)}, i\right) = r_{\pi_{(1)}} + \sum_{p=1}^{i} o_{\pi_{(1)}}^{p}, i = 1, 2, \ldots, m \tag{1}$$

$$C\left(\pi_{(k)}, 0\right) = r_{\pi_{(k)}}, k = 1, 2, \ldots, n \tag{2}$$

$$C\left(\pi_{(k)}, i\right) = max\left\{C\left(\pi_{(k)}, i-1\right), C\left(\pi_{(k-1)}, i\right)\right\} + o_{\pi_{(k)}}^{i}, k = 2, 3, \ldots, n; i = 1, 2, \ldots, m \tag{3}$$

$$Y_{\pi_{(k)}} = \sum_{p=1}^{m} \min\left\{\max\left\{C\left(\pi_{(k)}, m\right) - d_{\pi_{(k)}}, 0\right\}, o_{\pi_{(k)}}^{p}\right\} \tag{4}$$

$$Object\ function\ : Y = \min\left\{\sum_{k=1}^{n} Y_{\pi_{(k)}}\right\} \tag{5}$$

where $\pi = \left\{\pi_{(1)}, \ldots, \pi_{(n)}\right\}$ is the scheduling order of jobs, $C\left(\pi_{(k)}, i\right)$ represents the completion time of job $J_{\pi_{(k)}}$ on machine M_i, $C\left(\pi_{(k)}, 0\right)$ is the completion time of $J_{\pi_{(k)}}$ on the virtual machine 0, which also represents the release time of the job. Equation (1) constrains the order in which the first job is processed on adjacent machines. Equation (2) indicates that the job can be processed after the release date has arrived. Equation (3) shows the recursive calculation of the completion time. Equation (4) illustrates how the late work is calculated for each operation. The late work means the tasks that have not yet been processed after the dues date has been arrived. The objective function of this problem is $Y = \min\left\{\sum_{k=1}^{n} Y_{\pi_{(k)}}\right\}$, which means to minimize the total late work of each operation of all jobs. According to the tri-field representation, the problem can be expressed as $F|r_i|Y$ [18]. To illustrate the PFSP and the late work objective function more clearly, Table 1 gives an example of the problem. Figure 1 represents the Gantt chart when the scheduling sequence is 1-2-3, where the shaded part represents the late work after the arrival of the due dates, $Y_1 = 0$, $Y_2 = 5$, $Y_3 = 2$. A total of three operations are delayed, the total late work objective function value is 7.

Table 1. An example of a PFSP problem with the late work objective function.

Job No (i)	r_i	d_i	o_i^s	o_i^1	o_i^2	o_i^3
1	0	14	12	3	4	5
2	4	12	12	2	6	4
3	4	20	12	4	3	5

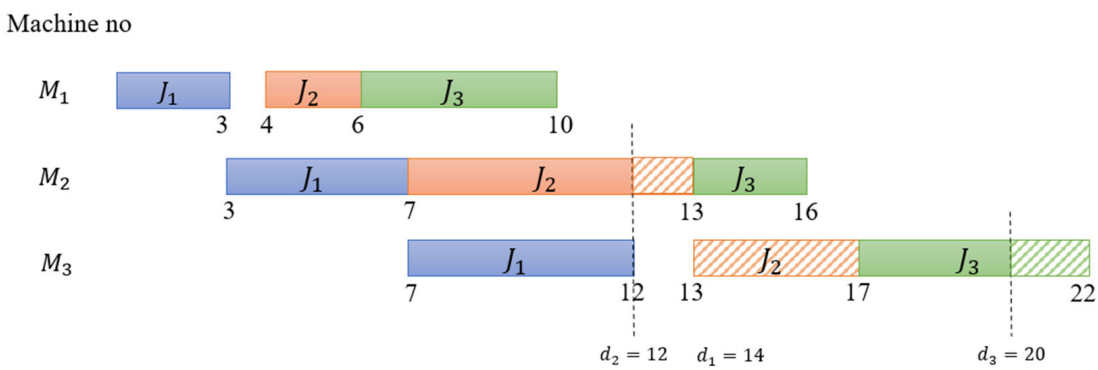

Figure 1. The Gantt chart with a job permutation of 1-2-3.

Since the scheduling order of jobs on each machine must be the same in the PFSP, the scheduling results can be abbreviated to a sequence of jobs. A job whose order has been

determined is called a scheduled job, and the job that has not been determined is called unscheduled job. Similar to the traveling salesman problem (TSP), nodes (jobs) that are not scheduled are continually expanded until all nodes (jobs) are expanded. In other words, given a set of jobs represented as a sequence of n jobs in a m dimensional space $s = \{x_i\}_{i=1}^{n}$, the goal is to find an optimal permutation π on all jobs that minimizes the late work. The chain decision method is used to calculate the late work of the job scheduling sequence determined by a permutation π as:

$$L(\pi \mid s) = \sum_{i=1}^{n} (Y_{\pi(i)} - Y_{\pi(i-1)}) \tag{6}$$

According to the chain rule, the probability of a job scheduling sequence can be factorized into the following form:

$$p(\pi \mid s) = \prod_{i=1}^{n} p(\pi(i) \mid \pi(< i), s) \tag{7}$$

where $\pi(i)$ indicates the job selected at the i-th time step to be scheduled, $\pi(0)$ is null since it is the initial state, $1 \leq i \leq n$. In summary, the flow shop problem is modeled as a continuous decision-making problem, and each step of decision-making outputs the probability distribution of candidate jobs. The probability distribution is generally obtained from the output of a policy network. In this study, a neural network is employed as a policy model to parameterize $p(\pi \mid s)$.

2.2. Policy Network Architecture

The PFSP can be viewed as a sequence-to-sequence problem [33], where the input is a random sequence of all jobs and the output is a well-designed sequence that makes the late work as small as possible, which is similar to the machine translation task, where the input and output are word vectors. The encoder encodes the input sequence into an intermediate vector, then the intermediate vector is decoded multiple times by the decoder to generate the output sequence. Since the encoding and decoding of a sequence need to consider the effect of the order of elements in the sequence, the long short-term memory (LSTM) networks with long and short-term memory are generally used as encoders and decoders. The structure of the encoder-decoder model is shown in Figure 2. In the decoding process, the output of a certain time step may depend on some specific parts of the input sequence rather than the whole input sequence, the attention mechanisms are introduced to learn to find the most valuable parts of the input sequence, which in turn improves the effectiveness of the model. The improved method aggregates the output hidden vector of the decoder of the current time step with the encoder's encoding result of each job using the attention mechanism. The output of the next time step is obtained by weighted aggregation of the encoded vectors of each job.

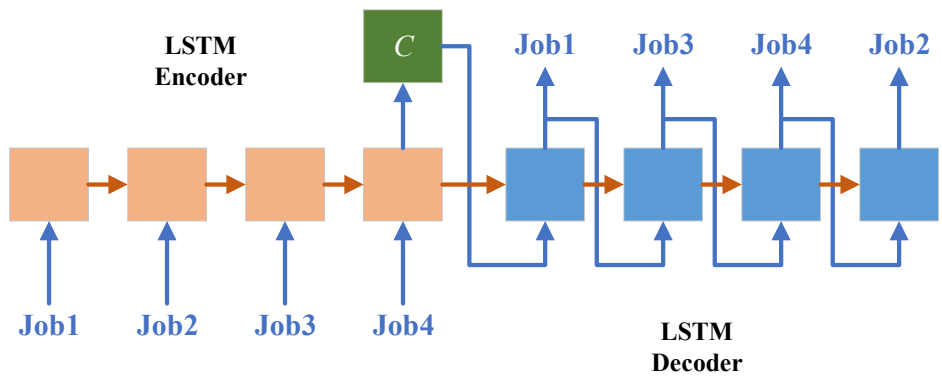

Figure 2. The sequence-to-sequence network structure.

The encoder-decoder network based on the attention mechanism has performed well on the sequence-to-sequence problem. However, directly applying it to solve the PFSP problem suffers from the following two issues.

(4) The output of the PFSP is heavily dependent on the input, the output of each step is one of the inputs, unlike the machine translation problem where the output is a completely different vector from the input.
(5) In the PFSP, it is inaccurate to assume that the model input is a sequence of jobs. In fact, the input sequence should have no effect on the output of the model regardless of the order of the jobs in the input. However, the encoder is an LSTM structure, the input jobs are entered and encoded one by one, taking into account unnecessary positional relationships.

Therefore, this study designed a policy network based on the idea of PtrNet, it turns the weights after the aggregation of the attention mechanism into a probability distribution directly through the softmax layer, the probability distribution points to the elements in the input sequence, which solves the first problem. The network structure of the PtrNet is shown in Figure 3. The network structure is redesigned by combining the graph neural network to solve the second problem mentioned above. The improved network structure is shown in Figure 4, which consists of two encoders (a job encoder and a graph encoder), and a decoder with the attention mechanism. The two encoders are responsible for encoding the input sequence information, converting the input into an intermediate vector form. The intermediate vector is then decoded by the decoder, the output is a probability distribution over the input elements.

2.2.1. Job Encoder

For the job encoder, two linear transformations are used to embed each job processing time vector $x_i = \left\{x_i^c, x_i^p\right\}$, where $x_i^c = \left\{r_i, d_i, o_i^s\right\}$ and $x_i^p = \left\{o_i^1, .., o_i^m\right\}$. The x_i^c vector includes constraint attributes for each job such as release date, due date, and total processing time, which can optimize the schedule from a macroscopic point of view; and the x_i^p consists of the processing time of each operation of the job, which helps to fine-tune the scheduling sequence. Therefore, two linear transformations are used to embed x_i^c and x_i^p, then the two embedded vectors $\widetilde{x}_i^c$ and $\widetilde{x}_i^p$ are concatenated as a higher dimensional vector $\widetilde{x}_i \in \mathbb{R}^d$, which can be of dimension 128, 256, etc. The weights of the two linear combinations are shared among all jobs. The feature vector dimension of the job is extended to combine each operation in the job, allowing the network to learn combinatorial relationships between multiple operations of a job. Since x_i^c and x_i^p are of different orders of magnitude, they are normalized separately to avoid unbalanced feature values. The job encoder is an LSTM, the vector $\widetilde{x}_i$ of the job x_i selected at the current time step will be encoded by the job encoder. The hidden state of the LSTM output needs to be fed into the decoder and then into the job encoder at the next time step. More specifically, the job encoder encodes the currently known scheduling sequence (not finished) and outputs a hidden vector $\widetilde{x}_i^h$ for the current time step. $\widetilde{x}_i^h$ is input to the decoder to find out which job should be selected next. Then, the vector $\widetilde{x}_j$ of the selected job j is input to the encoder along with $\widetilde{x}_i^h$ to obtain $\widetilde{x}_j^h$ of the next time step. This is a cyclic process until the complete sequence is obtained.

2.2.2. Graph Encoder

A complete graph $G = (V, E)$ is introduced to describe the PFSP, as shown in Figure 5. $V = \{x_1, ..., x_n\}$ is the set of nodes and E is the set of edges. All elements of the adjacency matrix are equal to one because any two jobs are related to each other, which means that any job can point to all other jobs.

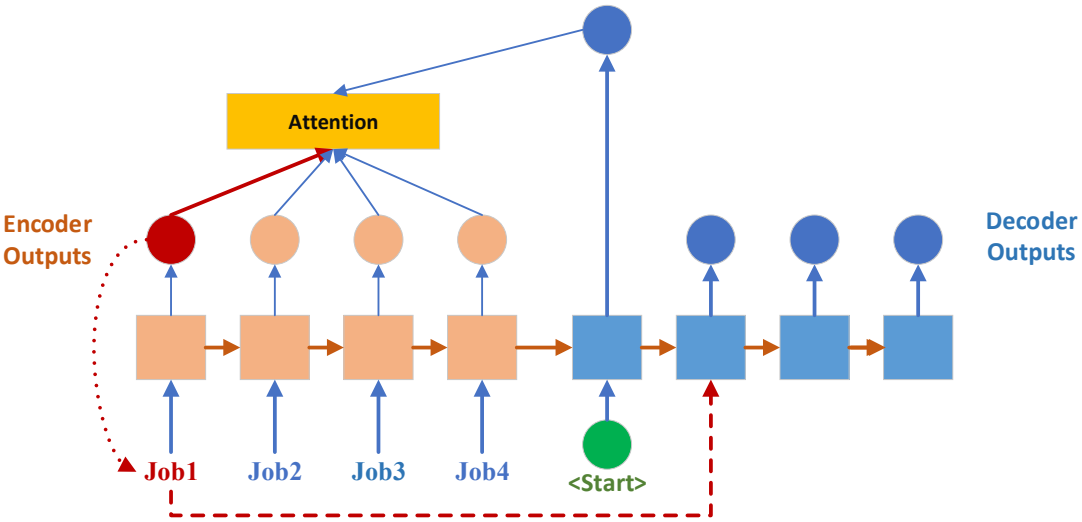

Figure 3. The PtrNet structure (with attention mechanism).

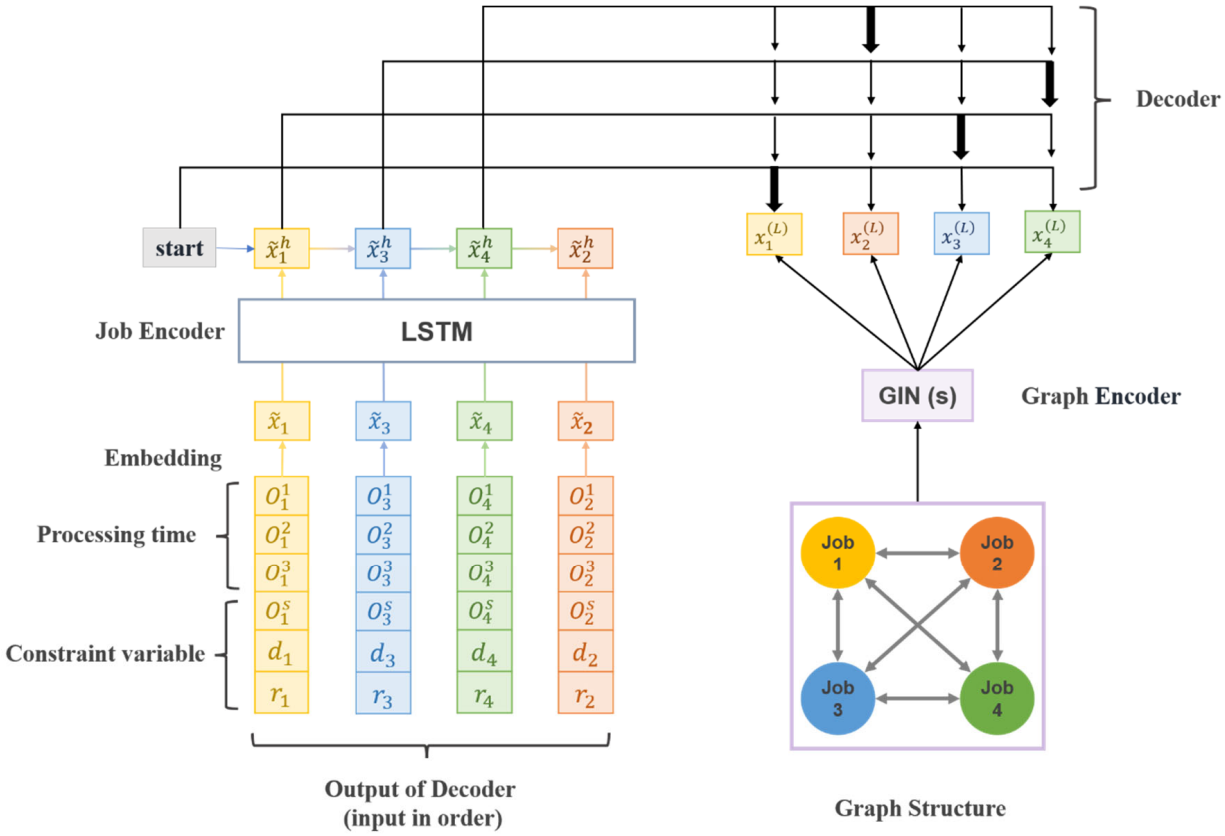

Figure 4. The proposed policy network structure (scheduling sequence: 1, 3, 4, 2).

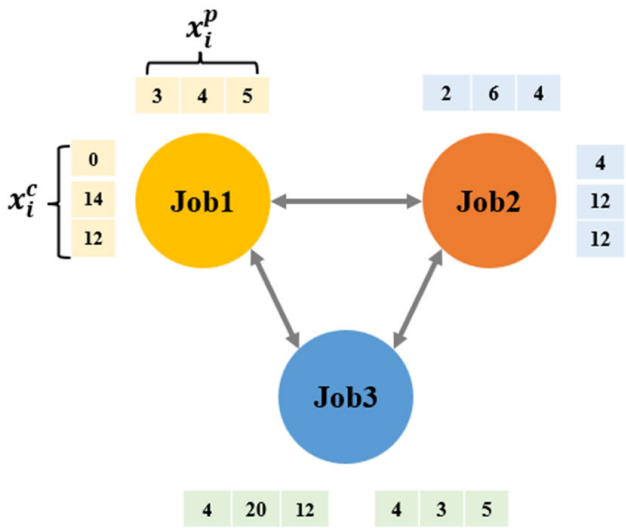

Figure 5. A complete graph representation of the PFSP problem.

Originally proposed by [34], GNNs extend the existing neural network models to process data from graphs or topological structures, and have achieved good performance in graph node classification, regression problems, and other fields. In the proposed network structure, job context information is obtained by encoding all job nodes through a graph isomorphic network (GIN) [35,36]. GIN is utilized to learn the context information between one job and other jobs to obtain a high-dimensional embedded representation of each job. The graph encoder learns how messages are passed between jobs. That is, the graph encoder updates the representation (feature vector) of each job using information from the entire problem. Each layer of the improved GIN network is expressed as:

$$\begin{cases} x_i^{(l)} = \gamma^{(l)} \cdot \omega^{(l)} \cdot x_i^{(l-1)} + \left(1 - \gamma^{(l)}\right) \varphi^{(l)} \left(\frac{1}{|Adj(i)|} \left\{ x_j^{(l-1)} \right\}_{j \in Adj(i) \cup \{i\}} \right), \forall l \in \{1, \ldots, L\} \\ x_i^{(0)} = x_i \end{cases} \tag{8}$$

where $x_i^{(l)} \in \mathbb{R}^{d_l}$ is the graph-encoded vector of job i at the l-th layer with $l \in \{1, \ldots, L\}$, the eigenvalues of the weight matrix $\omega^{(l)} \in \mathbb{R}^{d_{l-1} \times d_l}$ are regularized using a trainable variable $\gamma^{(l)}$, $Adj(i)$ denotes the set of all neighbouring nodes of node i, the information between the two layers is aggregated using the function $\varphi^{(l)} : \mathbb{R}^{d_{l-1}} \to \mathbb{R}^{d_l}$ [37]. The second equation in Equation (8) represents the first layer of the GIN network, where each node inputs a feature vector (basic information about each job). The first equation indicates that in other layers of the network, each node updates its own features by aggregating them with those of its surrounding neighboring nodes, and the specific aggregation function and weight matrix are trainable. The aggregation function is implemented as a neural network, aggregating information from nodes in the lower layers of the GIN to the next layer. The deeper the layer of the network, the larger the range of neighbor nodes that will be aggregated. The mechanism for embedding aggregation of a node vector in GIN can be represented simply as Figure 6. The figure approximately shows the aggregation and update process of the GIN network when $L = 2$. The information of the one-order and two-order neighbors of $job1$ will be aggregated. Each update of a node requires a combination of information about the node itself and information about its neighbors, and the aggregation function and the weights of the summation operation are parameters that can be trained.

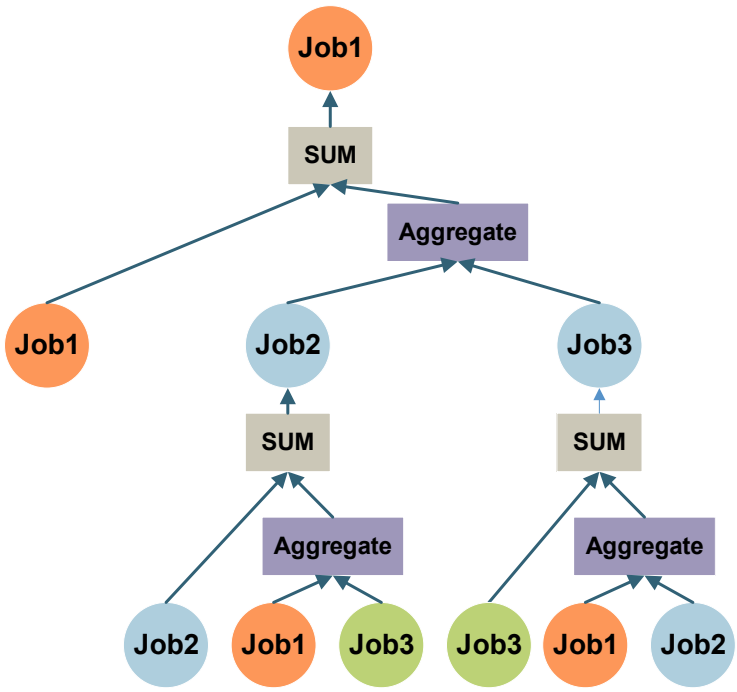

Figure 6. The information aggregation of GIN.

Since the PFSP graph structure is defined as a complete graph, each layer in the GIN can in turn be represented as:

$$X^{(l)} = \gamma^{(l)} \cdot \omega^{(l)} \cdot X^{(l-1)} + \left(1 - \gamma^{(l)}\right) \cdot \varphi^{(l)}\left(\frac{X^{(l-1)}}{|Adj(i)|}\right) \tag{9}$$

where $X^{(l)} \in \mathbb{R}^{n \times d_l}$, and $\varphi^{(l)} : \mathbb{R}^{n \times d_{l-1}} \to \mathbb{R}^{n \times d_l}$, n is the total number of jobs. In the actual structure of the network, a fully connected network is used to simulate the role of aggregate function, X will be replaced by $\widetilde{X}^c$ and $\widetilde{X}^p$ embedded after the linear transformations mentioned in the job encoder, $\widetilde{X}^c$ and $\widetilde{X}^p$ will be fed to two separately trained graph encoders. The graph encoders actually used are expressed as:

$$\widetilde{X}^{c/p,(l)} = \gamma \cdot \omega \cdot \widetilde{X}^{c/p,(l)} + (1 - \gamma) ReLU\left(\widetilde{X}^{c/p,(l)} W + b\right) \tag{10}$$

$$\widetilde{x}_i^{(L)} = concate\left(\widetilde{x}_i^{c,(L)}, \widetilde{x}_i^{p,(L)}\right) \tag{11}$$

where $ReLU$ is the activation function, $W \in \mathbb{R}^{d_{l-1} \times d_l}$ and $b \in \mathbb{R}^{n \times d_l}$. Equations (9) and (10) are simplifications of Equation (8). Equation (11) indicates that two GIN networks of the same structure are used to embed $\widetilde{X}^c$ and $\widetilde{X}^p$ (introduced in Section 2.2.1), respectively, and then the outputs of the two networks are stitched together by corresponding jobs.

2.2.3. Decoder

The decoder consists of an attention computation layer and a softmax layer. The pointer vector u computed by the attention mechanism is passed through the softmax layer to generate a probability distribution over the candidate jobs. Through continuous learning, the attention mechanism can learn to the extent that each element in the input sequence needs to be paid attention to. Similar to the pointer network [24], the attention mechanism is defined as Equation (12), its process is shown in Figure 7.

$$u_i = \begin{cases} v^T \cdot \tanh\left(W_{ref} \cdot r_i + W_q \cdot q\right) \; if \; i \neq \pi(k), \; \forall k < i \\ -\infty \; otherwise \end{cases} \tag{12}$$

where u_i is the i-th entry of the vector u, W_{ref} and W_q are trainable matrices, $v \in \mathbb{R}^d$ is an attention vector, q is the query vector, and r_i is a reference vector in the reference vector set. The output of the attention mechanism is obtained by calculating the similarity between q and each element in the set of reference vectors. In this article, $q = \tilde{x}^h$ and $r_i = \tilde{x}_i^{(l)}$, $\tilde{x}^h$ is the hidden variable of the job encoder, $\tilde{x}_i^{(l)}$ is the contextual information of a job from the graph encoder, and the set of reference vectors contains the contextual information of each job. The logits of jobs that already appeared in the scheduled sequence are set to $-\infty$, as shown in Equation (12), to ensure that the model only points to jobs that have not yet been scheduled to generate an available scheduling sequence. The policy distribution on all candidate jobs can be expressed as Equation (13).

$$\begin{aligned} p_\theta(a_i|s_i) &= p(\pi(i)|\pi(<i), s) \\ &= A\left(r_i, q; W_{ref}, W_q, v\right) \stackrel{\text{def}}{=} softmax(C * tanh(u)) \end{aligned} \tag{13}$$

where C is a hyperparameter that controls the range of the logits and hence the entropy of $p_\theta(a_i|s_i)$, i is the current time step. The next job to be scheduled is $a_i = x_{\pi(i)}$, predicted by sampling or choosing greedily from the policy $p_\theta(a_i|s_i)$. a_i and s_i can be thought of as an action and a state in RL introduced in the next section.

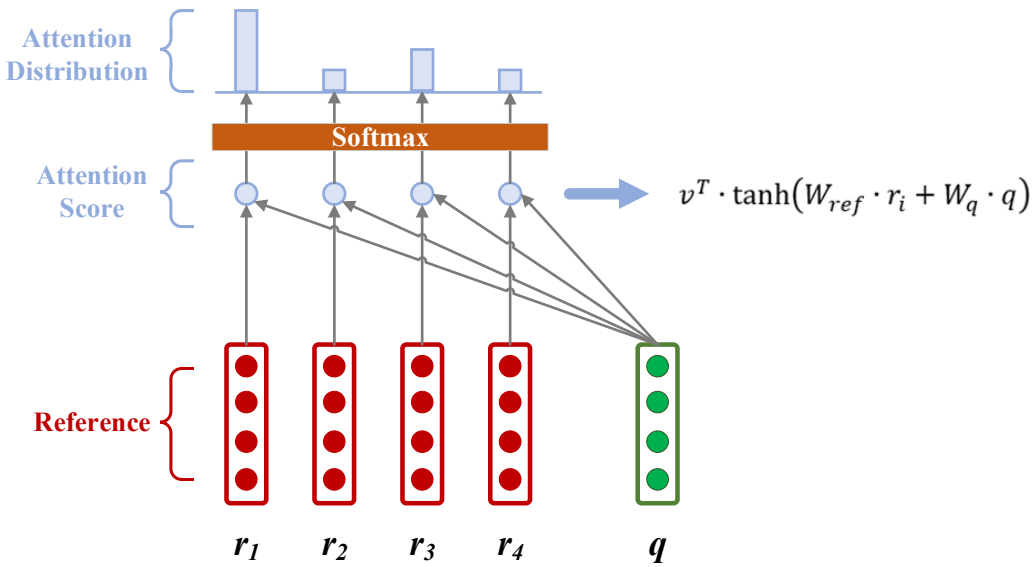

Figure 7. The calculation process of the attention mechanism.

2.2.4. Decode Strategies

Greedy: After the model is trained, the policy network outputs the probability distribution of candidate jobs in each decision step. The most common way is to select the job with the highest probability in each decision step, and then input the job into the network to obtain the probability distribution of candidate jobs in the next step.

Sampling: Since the model is trained on a large number of randomized problems, the single solution generated by greedy search is not necessarily suitable for the test set. Therefore, in the probability distribution of each output, candidate jobs are selected using sampling to produce more diverse solutions. In this study, *sampleSize* is set to 1280, the one with the smallest objective function value among the 1280 solutions generated is taken as the final solution.

2.3. Reinforcement Learning for PFSP

In RL, the agent learns through an iterative trial process by interacting with the environment and observing the resulting reward signals. The RL problem is modeled as a Markov decision process (MDP), which provides a mathematical framework for modeling

sequential decisions under uncertainty [38]. It consists of four main elements, Agent, State, Action, and Reward, and the goal is to obtain the most cumulative rewards. The process of RL is shown in Figure 8.

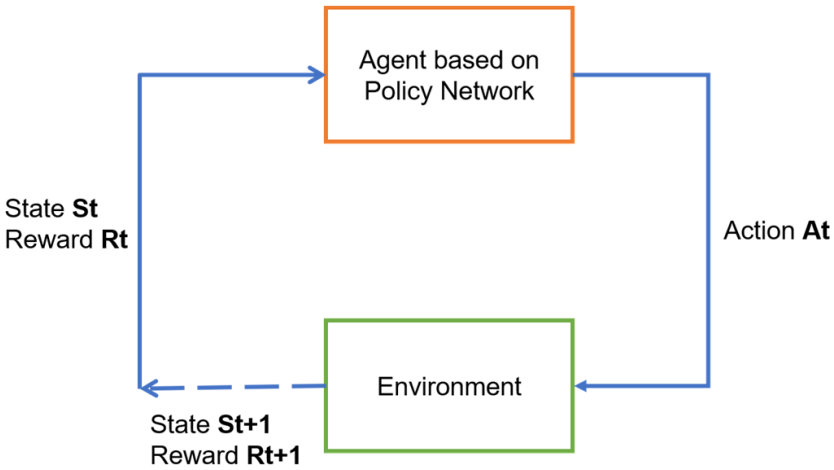

Figure 8. RL learning environment interaction.

Let S and A denote state space and action space, respectively.

State: Each state $s_t \in S$ mainly includes two parts, which are the graph encoding of all jobs $\widetilde{X}^{(L)}$, and the LSTM encoding of the jobs that have been scheduled at time step t.

Action: The action $a_t \in A$ is defined as the next selected job, that is $a_t = x_{\pi(t)}$, the action will be performed in a job that has not been selected.

Policy: The policy is expressed as $p_\theta(a_t \mid s_t)$, which is a distribution over candidate jobs a_t. Given a set of scheduled jobs, the policy will return a probability distribution over the candidate jobs that have not been chosen and the next job to be scheduled is greedily selected or sampled according to the probability distribution. The policy is implemented with a neural network with an ensemble of trainable weights θ.

Reward: Since the gradient descent algorithm is used to train the network, the reward function is generally set to be the negative cost of taking action a_t from the state s_t. i.e., $r(s_t, a_t) = -\left(Latework_{\pi(i)} - Latework_{\pi(i-1)}\right)$. The DRL method can optimize various goals such as late work, makespan, total completion time, and delay time through the gradient descent method without changing the network structure. Thus, the expected reward is defined as follows, where Obj represents the objective function that needs to be optimized.

$$\begin{aligned} &\mathbb{E}_{(s_t,a_t)\sim p_\theta(s_t,a_t)}\left[\sum_{i=1}^{n} r(s_t,a_t)\right] \\ &= \mathbb{E}_{\pi\sim p_\theta(\Gamma)}\left[\sum_{i=1}^{n} -\left(Obj_{\pi(i)} - Obj_{\pi(i-1)}\right)\right] \\ &= -\mathbb{E}_{\pi\sim p_\theta(\Gamma)}[L(\pi,S)] \end{aligned} \tag{14}$$

where all possible permutations π over $s = \{x_i\}_{i=1}^{n}$ constitute the space Γ, and $p_\theta(\Gamma)$ is the distribution on Γ predicted by the policy network. When a complete solution is obtained, the cumulative reward obtained is $Y_\pi - Y_{\pi(0)}$ which is the value of the current solution's late work. The policy network must learn to minimize the expected object function value. Furthermore, the policy gradient algorithm [39] is employed to train the network to learn to maximize the reward obtained, as presented below.

2.4. Training Method

The object function of the policy is $J(\theta \middle| s) = -\mathbb{E}_{\pi\sim p_\theta(\Gamma)}[L(\pi,S)]$. Based on the REINFORCE algorithm [39], the gradient of the policy is expressed as:

$$\nabla_\theta J(\theta \mid s) = \mathbb{E}_{\pi\sim p_\theta(.|s)}[(L(\pi \mid s) - b(s))\nabla_\theta log p_\theta(\pi \mid s)] \tag{15}$$

where $b(s)$ denotes a baseline function that estimates the late work of the expected scheduling sequence to reduce the variance of the gradients. In actual training, based on Monte Carlo sampling [20], the gradient can also be approximated as:

$$\nabla_\theta J(\theta) = \frac{1}{B}\sum_{i=1}^{B}\left[\left(\sum_{i=1}^{n} r(s_{i,t}, a_{i,t}) - b_i\right) \times \left(\sum_{i=1}^{n} \nabla_\theta log p_\theta(a_{i,t}|s_{i,t})\right)\right] \tag{16}$$

where B is the batch size, r is the reward function, b_i is the baseline for a problem instance in the current batch (late work value obtained by the baseline), $\sum_{i=1}^{n} r(s_{i,t}, a_{i,t})$ is equal to the late work value of the solution, $p_\theta(a_{i,t}|s_{i,t})$ denotes the probability that each action in the action sequence is selected (output by policy network). On the basis of Equation (16), the parameters θ can be optimized by gradient descent using the update rule $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$.

The algorithm similar to self-critic [40] is used as the baseline. The main idea is to use the results when the model is tested as the estimated scheduling sequence. Furthermore, during the training phase, samples will be taken from the generated probability distribution. In order to increase the model's exploration ability and avoid falling into the local optimum. The sampling operation allows the model to trade-off between 'exploitation' and 'exploration'. In the testing phase, the action with the highest probability in the probability distribution at each step is selected until the complete scheduling sequence is obtained. It is also possible to sample the probability distribution to search for a better solution, but at the expense of some computational time. The self-critic baseline b_i is expressed as:

$$b_i = \sum_{i=1}^{n}(r(\widetilde{s}_{i,t}, \widetilde{a}_{i,t})) + \left[\frac{1}{B}\sum_{i=1}^{n}\sum_{j=1}^{B}(r(s_{i,t}, a_{i,t}) - r(\widetilde{s}_{i,t}, \widetilde{a}_{i,t}))\right] \tag{17}$$

where the action $\widetilde{a}_{i,t} \sim Greedy(p_\theta)$ is sampled greedily from the policy. The right-hand part of the plus sign of Equation (17) is the gap between the rewards obtained by sampling and greedy. If the sampling result is better than the baseline, the gradient of some of the better actions will increase. Eventually, the probability of a good action being selected will be increased, while the probability of a poor action being selected will be decreased. The final optimization process is shown in Algorithm 1. In order to improve the quality of the solutions obtained by RL, an improved hybrid iterative greedy method is proposed in the next section.

Algorithm 1 Policy Gradient Optimization

variable declaration: Training set T, training steps S, batch size B, learning rate α

```
Initialize network parameters θ
for s = 1 to S do
    x_i = Sample(T) for i ∈ {1, ..., B}
    a_{i,t} = Sample(p_θ(·|s_{i,t}))
    ã_{i,t} = Greedy(p_θ(·|s̃_{i,t}))
    Calculate J(θ), ∇_θ J(θ)
    θ ← θ + α∇_θ J(θ).
return p_θ
```

3. A Hybrid Iterated Greedy Method to Improve the Initial Solutions

The iterative greedy (IG) methods are mainly used to solve the flowshop problem of minimizing makespan, which was first proposed in [41] and is the most effective metaheuristic method so far. The algorithm first uses a heuristic method (such as NEH) to generate an initial solution Π, for which the initial solution is improved using a local search algorithm with the insertion neighborhood algorithm. Then, d jobs are removed from this sequence (destruction phase) and reinserted one after another to the position that minimizes the objective function value obtained after insertion (construction phase), looping this process to obtain a new sequence Π'. Finally, the sequence Π'' is obtained

using a local search method based on the insertion neighborhood of Π'', using a probability similar to simulated annealing to decide whether to accept Π'' and start the next iteration. The current well-known variant of the IG algorithm [42] applies a local search to both complete and partial solutions to speed up the search, applying the NEH algorithm with local search to produce an initial solution, the IG framework is shown in Algorithm 2.

Algorithm 2 The IG framework with local search

input: a PFSP problem instance P, the number of jobs removed d, simulated annealing parameter T.
output: The best solution found Π^*.
$\Pi = NEH(P)$; //**replaced by** $\boldsymbol{DRL(P)}$
$\Pi = LocalSearch(\Pi)$;
$\Pi^* = \Pi$;
while termination criteria not met **do**
    Randomly remove d jobs from Π (destruction);
    (Let Π^R be the remaining sequence and Π^D be the extracted jobs);
    $\Pi^{R'} = LocalSearch(\Pi^R)$;
    $\Pi' = Construction\left(\Pi^{R'}, \Pi^D\right)$;
    $\Pi'' = LocalSearch(\Pi')$;
    $\Pi = AcceptanceCriterion\left(\Pi'', T\right)$;
    if $(\Pi'') < f(\Pi^*)$ then
        $\Pi^* = \Pi''$;
  return Π^*

The proposed deep reinforcement learning architecture (DRL for short) can replace the combination of NEH and the insertion neighborhood local search as a high-performance initial solution generator. In the destruction and construction phase, an adaptive local search strategy is proposed, which consists of an insertion operator (LS1), a swap operator (LS2). The weights of the three operators are $weight_1$ and $weight_2$, respectively, and the corresponding probabilities are p_1, p_2. The weights of the two operators are initialized to 1, and the probabilities are initialized to 1/2. As the number of iterations grows, the probabilities of the operators that can improve the current solution more are increased. In order to improve the effectiveness of the local search, the tie-breaking mechanism is added. If the late work values obtained from multiple positions inserted or swapped by the operator are the same, then the one with the smallest idle time in the scheduling sequences obtained after the execution of the operators is taken as the final result. The $loopSize$ of the local search is set to half the number of jobs to improve the efficiency of the search. The late work optimization objective is very sensitive to the permutation of the jobs, especially if the problem with a release date constraint, small changes can lead to a dramatic deterioration of the late work value. Therefore, the inverse operator is not used in the improved IG algorithm, which would be destructive to the current solution. The p_1, p_2 of the next iteration are calculated by $weight_1$, $weight_2$. The update and calculation method of these parameters are shown in Equations (18)–(20).

$$p_i = \frac{weight_i}{weight_1 + weight_2},\ i = 1, 2 \tag{18}$$

$$\mu = \begin{cases} 1\ ,\ f_{new} < f_{best} \\ 0\ ,\ f_{new} \geq f_{best} \end{cases} \tag{19}$$

$$weight_i = weight_i + \theta * \mu_i \tag{20}$$

where f_{new} is the fitness after local search update, and f_{best} is the fitness of the best solution (the solution before a local search) currently found. In the same iteration, only one of

μ_1 and μ_2 is 1, and the other is 0. θ is the learning factor, which is generally set to 0.2. The pseudocode of the local search framework is shown in Algorithm 3. The operator pseudocode used in it is shown in Algorithm 4.

Algorithm 3 The local search framework

```
take permutation Π′ or Π^R as input Π^input, set l = 0;
generate a random number r ∈ (0, 1);
while l < loopSize do:
    if r ≤ p_1 then
        Π^LS = chooseOperator(Π^input, op = insert);
    else r ≥ p_1 and r ≤ p_1 + p_2 then
        Π^LS = chooseOperator(Π^input, op = swap);
    if f(Π^LS) < f(Π*) then
        Π* = Π^LS
    if f(Π^LS) < f(Π^input) then
        Π* = Π^input
    update the weights and probabilities of each operator if necessary
    l = l + 1
    Π = Π^input
return Π
```

Algorithm 4 The process of two operators

```
randomly select a location a of the input Π^input, let b = 1, j = job number, bestPosFit = +∞
and op = insert or swap;
while b < j do
    if a ≠ b then
        if op == insert then
            insert job Π_a into the b-th position of the permutation (LS1);
        if op == swap then
            swap the π_a in position a and the π_b in position b (LS2);
        a candidate solution cand is obtained;
        if f(Π^cand) < bestPosFit then
            bestPosFit = f(Π^cand);
            bestcand = cand
        if f(Π^cand) = bestPosFit then
            if the tie-breaking conditions are met then
                bestPosFit = f(Π^cand);
                bestcand = cand
        b = b + 1;
    return bestcand, bestPosFit
```

This method uses the same acceptance criterion as the IG_{RS} method, which applies the idea of simulated annealing to decide whether to accept the candidate solution Π'' obtained from each iteration. If the candidate solution Π'' is better than or equal to the current best solution Π^*, the Π^* is directly replaced with Π''. If Π'' is worse, the decision to accept the candidate solution is made with a probability given by $e^{(f(\pi)-f(\pi'))/T}$ where T is calculated as Equation (21) and T_p is a hyperparameter that can be adjusted.

$$T = T_p \times \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}}{n \cdot m \cdot 10} \tag{21}$$

4. Experiments

4.1. Experiment Setup

The multi-layer GIN with $L = 3$ is used as the graph encoder if the problem size $n <= 50$. For larger-scale problems, $L = 5$. The $(m + 3) \times n$ problem matrix of PFSP is the main training data, where the processing time of an operation is generated randomly from a $U[0, 100)$ distribution. In order to enable most of the jobs to be completed before the due date arrives, r_i and d_i are generated from $U[0, 50 \times n]$ and $U[r_i + 2 \times m \times p_i, r_i + 0.4 \times n \times p_i]$, respectively. To ensure that the problem instance is reasonable, a job is randomly selected and its release date is set to 0. The proposed method is compared with the results of other methods over the Taillard [43] benchmark instances. The Taillard benchmark does not include the due date and release date constraints, so they are also generated from the above distribution. At the beginning of each step, the data of the training set is regenerated. The scale of the training set is *BatchSize*, and the data of the validation set are only generated once in the entire training process. The hyperparameter configuration of the method proposed is mainly shown in Table 2. All the codes of the experiment are written in Python, and the calculation of the objective function is accelerated by Cython. The experiments are all running in the environment of AMD 5600X CPU, RTX TITAN GPU, and 16G memory. It should be noted that the training and validation sets of the reinforcement model are randomly generated, the algorithms are compared on the Taillard [43] benchmarks, and the release dates and due dates are generated only once to ensure that the problem instances used to evaluate each method are the same.

Table 2. Hyperparameter configuration.

Parameter	Value
BatchSize	128
Epoch	100
Steps per epoch	3000
Learning rate	1×10^{-3}
Learning rate decay	0.975
Decay step	3000
Hidden size	128
Validation set size	1000
Optimizer	Adam

To improve the performance of the proposed hybrid iterative greedy method (IG_h in short), orthogonal experiments are conducted on two hyperparameters, d and T_p. The factor level is given in Table 3. The average improvement percent (AIP) is used to evaluate the performance of the algorithm under different combinations of parameters. $AIP = \frac{(Y_{ini} - Y_{res})}{m \times n \times 10} \times 100\%$, where Y_{ini} is the initial latework value of the algorithm, and Y_{res} is the latework value after the algorithm is executed. The results of the orthogonal tests are shown in Table 4. The effect of the parameters on the performance of the algorithm is plotted in Figure 9a,b. Finally, d of the IG_h algorithm is set to 4, and t_p is set to 0.7, due to the fact that the algorithm has the best performance at this parameter setting.

Table 3. Factor level of IG_h.

Level	d	T_p
1	2	0.3
2	4	0.5
3	6	0.7
4	8	0.9

Table 4. Results of the orthogonal test.

No.	d	T_p	AIP
1	1	1	30.23
2	1	2	30.42
3	1	3	31.34
4	1	4	27.65
5	2	1	33.87
6	2	2	37.54
7	2	3	40.18
8	2	4	39.11
9	3	1	35.54
10	3	2	37.29
11	3	3	38.93
12	3	4	33.74
13	4	1	29.76
14	4	2	33.66
15	4	3	35.85
16	4	4	29.54

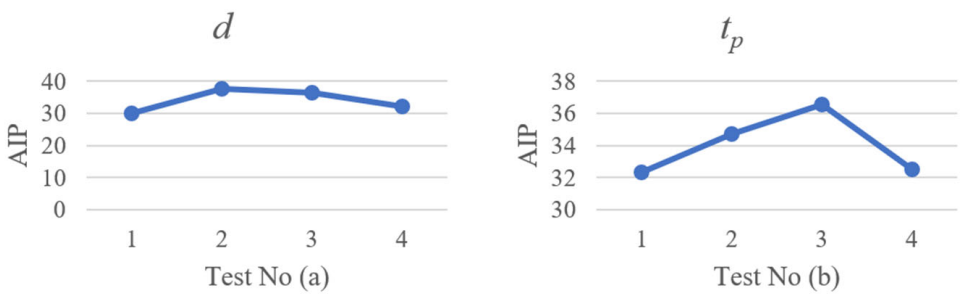

Figure 9. Effect of parameter settings on algorithm performance. ((**a**) is the orthogonal test result for d, (**b**) is the orthogonal test result for t_p).

4.2. Experiment Results

The proposed method consists of two parts, where RL is used to generate high-quality initial solutions and IG_h is used to optimize the solutions obtained by RL. The role of RL methods is similar to that of heuristics, so RL is compared with some classical heuristics such as NEH [4], earliest-due-date first (EDD) [19], and smallest late work insertion (SLW) [18]. A comparison of the effects of these methods is shown in Table 5. The table shows a description of the problem instances, the average relative difference percent (ARDP) values and average computation time (ACT) of each method. The *RDP* is calculated as Equation (22):

$$RDP = \frac{Obj_x - Obj_{min}}{m \times n \times 10} \times 100\% \tag{22}$$

where Obj_x is the objective function value of the solution obtained by method x, Obj_{min} represents the minimum objective function value obtained by all methods. The ARDP can indicate the difference between the objective function values obtained by several methods solving the same set of benchmarks, and a value of 0 means that the current method obtains the smallest late work among several methods. In short, the smaller the ARPD value, the better the performance of the algorithm and the higher the quality of the solutions obtained. As can be seen from Table 5, the RL algorithm using the sampling decoding strategy performs better than the other heuristics on problems of all sizes. Since EDD uses the simplest dispatch rule, the computation time of the EDD heuristic can be neglected, but the largest ARDP value 100.54 is obtained. The NEH algorithm is a widely used heuristic. The computation time of RL-Greedy is slightly increased compared to NEH, but the ARDP value is reduced by half (16.79 to 9.9). Box plot 10 visually shows the distribution of

achieved RDPs and skewness by displaying the data quartiles (or percentiles) and averages. The box plots generally include the five-number summary of a set of data: the minimum score, first quartile, median, third quartile, and maximum score [44]. As can be seen in Figure 10, the RL-based method is significantly better than other heuristics.

Table 5. Comparison of RL and multiple heuristic algorithms.

Problem Instances	RL-Sample		RL-Greedy		NEH		SLW		EDD
	ARDP	ACT(s)	ARDP	ACT(s)	ARDP	ACT(s)	ARDP	ACT(s)	ARDP
20 × 5	**0**	0.17	11.60	0.11	17.62	0.003	95.57	0.002	90.39
20 × 10	**0**	0.18	19.65	0.12	33.23	0.003	138.2	0.002	84.29
20 × 20	**0**	0.18	17.75	0.11	20.25	0.003	108.6	0.002	125.72
50 × 5	**0**	0.24	7.00	0.14	20.20	0.022	65.78	0.010	184.57
50 × 10	**0**	0.29	10.26	0.14	23.17	0.023	91.85	0.012	130.32
50 × 20	**0**	0.30	20.27	0.13	47.62	0.023	151.7	0.015	114.15
100 × 5	**0**	0.55	3.56	0.21	4.42	0.083	38.28	0.040	101.34
100 × 10	**0**	0.61	6.50	0.22	7.04	0.084	52.42	0.042	92.54
100 × 20	**0**	0.63	5.25	0.21	7.26	0.088	58.51	0.042	84.33
200 × 10	**0**	1.41	1.62	0.32	0.45	0.360	34.43	0.210	44.31
200 × 20	**0**	1.49	5.49	0.33	3.45	0.400	41.75	0.220	54.05
average	**0**	0.55	9.90	0.19	16.79	0.099	79.74	0.054	100.54

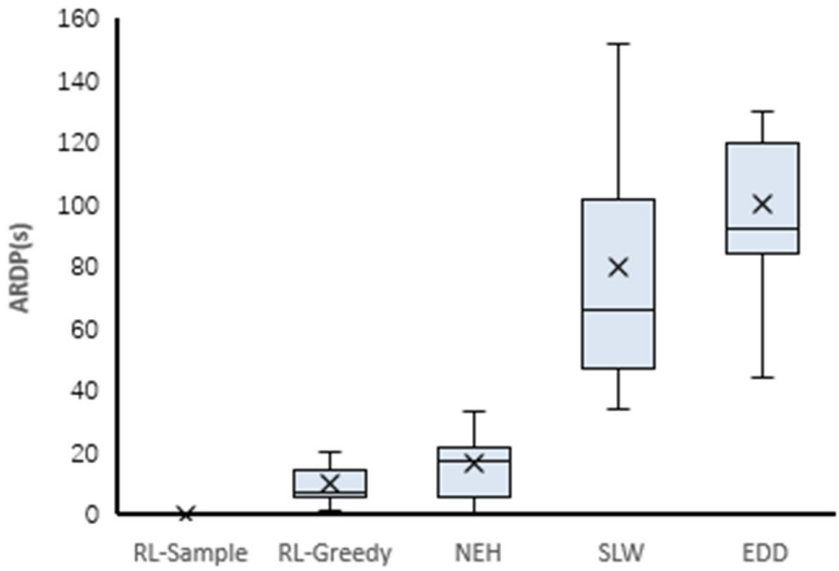

Figure 10. Box plot of RL and several heuristic algorithms.

Furthermore, the combination of RL-Sample and IG_h is compared with other metaheuristic algorithms such as hybrid cuckoo search algorithm (HCS) [45], discrete artificial bee colony algorithm (DABC) [46], and NEH + IG algorithmA [42]. The hyperparameters of other algorithms use the default settings in the corresponding article. The running times of the IG-based algorithm for solving problems of 20, 50, 100, and 200 scales are 2 s, 4 s, 8 s, and 16 s, respectively; the number of iterations for the other algorithms is fixed at 500. The experimental results of several metaheuristics are shown in Table 6, where BR indicates the best RDP value, AR indicates the average RDP value. The BR value can represent the optimization ability of the algorithm, and the AR represents the stability of the algorithm. Table 6 shows that the RL + IG_h algorithm achieves the smallest BR and AR values and runs in much less time than the HCS and DABC algorithms (especially for problems above the 50-job scale), RL + IG_h showed a 42% improvement in AR compared to NEH + IG, and the BR obtained by RL + IG_h outperforms the NEH + IG method on all size benchmarks. In order to compare the performance difference between RL + IG_h and other algorithms,

a Wilcoxon test (with a 95% confidence interval) was constructed based on the obtained BR and AR. Table 7 shows the *p*-values obtained by comparing the proposed method with other methods, if the *p*-value is less than 0.05 means that there is a statistically significant difference between the method and other methods. From Table 7 and Figure 11, it can be seen that the proposed method is significantly better than NEH + IG, HCS, and DABC.

Table 6. Comparison of RL + IG_h and multiple meta-heuristic algorithms.

Problem Instances	RL + IG_h		NEH + IG		HCS			DABC		
	BR	**AR**	**BR**	**AR**	**BR**	**AR**	**ACT(s)**	**BR**	**AR**	**ACT(s)**
20 × 5	**0**	**0.03**	**0.00**	1.22	0.29	2.03	3.42	0.29	2.14	1.30
20 × 10	**0**	**1.01**	4.38	5.72	5.97	8.13	4.90	5.91	8.09	2.75
20 × 20	**0**	**0.31**	1.49	1.88	1.67	2.22	6.53	1.80	2.29	3.80
50 × 5	**0**	**2.10**	2.54	5.80	2.99	7.05	7.73	3.20	7.36	7.10
50 × 10	**0**	**3.63**	2.55	5.77	2.89	6.33	9.94	3.61	6.47	8.20
50 × 20	**0**	**3.26**	2.01	5.87	3.83	9.74	13.10	3.91	9.72	10.10
100 × 5	**0**	**2.95**	0.35	3.98	1.45	4.41	21.56	1.69	4.45	19.54
100 × 10	**0**	**2.49**	0.29	2.58	0.94	3.15	26.40	1.26	3.59	22.32
100 × 20	**0**	**3.05**	1.79	3.41	3.88	5.53	33.52	4.38	5.98	26.45
200 × 10	**0**	**2.03**	0.38	2.05	2.85	8.14	68.47	4.90	9.16	53.23
200 × 20	**0**	**2.59**	0.54	2.71	8.03	13.50	75.33	9.37	14.05	59.54
average	**0**	**2.13**	1.48	3.73	3.16	6.38	24.62	3.66	6.66	19.48

Table 7. Results achieved by Wilcoxon test.

RL + IG_h vs	*p*-Values of BR	*p*-Values of AR
NEH + IG	5.06×10^3	3.35×10^3
HCS	3.35×10^3	3.35×10^3
DABC	3.35×10^3	3.35×10^3

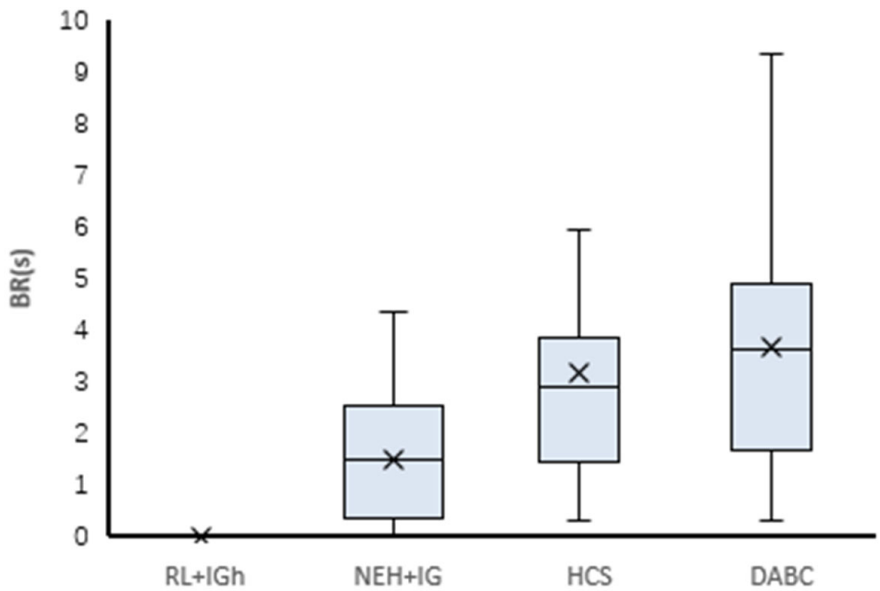

Figure 11. Box plot of RL + IG_h and other meta-heuristic algorithms.

The method is also compared with the original PtrNet [24,47] and there is no GIN graph encoding module in the structure. Before the job feature vector x_i is input into the network, the proposed method will expand the dimension through a simple linear transformation layer, and the PtrNet uses a simple graph embedding of node aggregation. The major difference between the two methods is that the encoder and decoder of the PtrNet

are both LSTM structures, the output of each step of the encoder is used as a reference vector r_i. Table 8 shows the objective function values of the two methods on some problem instances, the decoding strategy of both methods is greedy strategy. The performance of PtrNet deteriorates rapidly as the number of jobs increases, and if the number of artifacts is greater than 50, the objective function value of the obtained solution is worse than NEH. The PtrNet model can converge during training, but the model ends up in a local optimum, which reflects that GIN can learn the context information of the job; it greatly improves the model's performance, and the proposed RL method has a 6.7% performance improvement over PtrNet.

Table 8. Results achieved by Proposed RL, PtrNet, and NEH.

Problem Instances	Proposed RL	PtrNet	NEH
Ta10	**689**	693	704
Ta20	**4354**	4407	4449
Ta30	**8338**	8379	8424
Ta40	**1278**	1339	1362
Ta50	**5335**	5458	5460
Ta60	**9956**	10,385	10,244
Ta70	**2074**	2395	2296
Ta80	**6693**	7065	6722
Ta90	**17,089**	18,767	17,406
Ta100	**9931**	11,278	10,124
Ta110	**19,999**	21,779	20,093
Average late work	**7794.2**	8358.6	7934.9

The result of the attention mechanism is visualized as a heat map, as shown in Figure 12. Each color block in the figure represents the probability of selecting each job at time step i. The brighter the color, the greater the probability. Figure 12a shows the heat map of the probability distribution of attention output before the model training, and Figure 12b shows the heat map result after the training. The color blocks are messy before training, and the probability distribution is clear after training. According to the greedy decoding strategy, it can be easily derived from the figure; the final job scheduling sequence is 3-17-15-8-9-14-11-13-5-7-4-19-6-16-1-18-2-12-10-20.

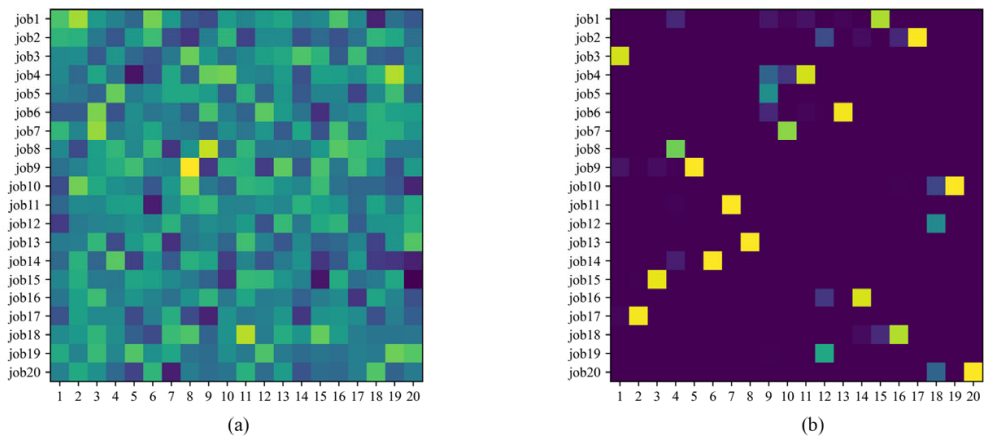

Figure 12. The heat map of the probability distribution of attention. ((**a**) denotes the matrix before training, (**b**) is the matrix after training).

5. Conclusions

In this study, the PFSP is innovatively modeled as a sequential decision process and a reinforcement learning (RL) method applying graph neural networks is proposed to minimize the late work of PFSP. In addition, a hybrid iterative greedy algorithm (IG_h) with

a tie-breaking mechanism is proposed to improve the solution obtained by the RL method. The experimental results show that the improved RL outperforms some classical heuristics and pointer network-based reinforcement learning methods, the proposed RL is able to obtain high-quality solutions in a short time. The performance of the combination of RL and IG_h also outperforms some excellent metaheuristics such as HCS, DABC, and NEH + IG. In summary, reinforcement learning is more competitive than traditional methods in solving problems such as flow shop scheduling.

Future work will focus on using reinforcement learning methods to solve scheduling problems with more complex constraints and dynamic scheduling problems, the performance and efficiency of reinforcement learning models on large-scale problems also need to be improved.

Author Contributions: Conceptualization, T.R. and Z.D.; methodology, Z.D.; software, Z.D. and J.W.; validation, F.Q., X.W.; formal analysis, F.Q.; investigation, F.Q.; resources, T.R.; data curation, J.W.; writing—original draft preparation, Z.D.; writing—review and editing, Z.D. and T.R.; visualization, J.W.; supervision, T.R.; project administration, T.R.; funding acquisition, T.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Fundamental Research Funds for the Central Universities (N181706001, N2017009, N2017008, N182608003, N181703005), National Natural Science Foundation of China (61902057), Joint Fund of Science & Technology Department of Liaoning Province and State Key Laboratory of Robotics, China (2020-KF-12-11).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Johnson, S.M. Optimal two-and three-stage production schedules with setup times included. *Nav. Res. Logist. Q.* **1954**, *1*, 61–68. [CrossRef]
2. Garey, M.R.; Johnson, D.S.; Sethi, R. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.* **1976**, *1*, 117–129. [CrossRef]
3. Ruiz, R.; Maroto, C. A comprehensive review and evaluation of permutation flowshop heuristics. *Eur. J. Oper. Res.* **2005**, *165*, 479–494. [CrossRef]
4. Nawaz, M.; Enscore, E.E., Jr.; Ham, I. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega* **1983**, *11*, 91–95. [CrossRef]
5. Campbell, H.G.; Dudek, R.A.; Smith, M.L. A heuristic algorithm for the n job, m machine sequencing problem. *Manag. Sci.* **1970**, *16*, B-630–B-637. [CrossRef]
6. Tseng, L.-Y.; Lin, Y.-T. A hybrid genetic algorithm for no-wait flowshop scheduling problem. *Int. J. Prod. Econ.* **2010**, *128*, 144–152. [CrossRef]
7. Nowicki, E.; Smutnicki, C. A fast tabu search algorithm for the permutation flow-shop problem. *Eur. J. Oper. Res.* **1996**, *91*, 160–175. [CrossRef]
8. Pan, Q.-K.; Tasgetiren, M.F.; Liang, Y.-C. A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem. *Comput. Oper. Res.* **2008**, *35*, 2807–2839. [CrossRef]
9. Tasgetiren, M.F.; Sevkli, M.; Liang, Y.-C.; Gencyilmaz, G. Particle Swarm Optimization Algorithm for Permutation Flowshop Sequencing Problem. In Proceedings of the International Workshop on Ant Colony Optimization and Swarm Intelligence, Brussels, Belgium, 5–8 September 2004; pp. 382–389.
10. Potts, C.N.; van Wassenhove, L.N. Single machine scheduling to minimize total late work. *Oper. Res.* **1992**, *40*, 586–595. [CrossRef]
11. Błażewicz, J.; Pesch, E.; Sterna, M.; Werner, F. Total late work criteria for shop scheduling problems. In Proceedings of the Operations Research Proceedings 1999, Magdeburg, Germany, 1–3 September 1999; pp. 354–359.
12. Chen, R.; Yuan, J.; Ng, C.; Cheng, T. Single-machine scheduling with deadlines to minimize the total weighted late work. *Nav. Res. Logist.* **2019**, *66*, 582–595. [CrossRef]
13. Chen, X.; Sterna, M.; Han, X.; Blazewicz, J. Scheduling on parallel identical machines with late work criterion: Offline and online cases. *J. Sched.* **2016**, *19*, 729–736. [CrossRef]
14. Leung, J. Minimizing Total Weighted Error for Imprecise Computation Tasks and Related Problems. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*; CRC Press: Boca Raton, FL, USA, 2004; p. 34.

15. Ren, J.; Zhang, Y.; Sun, G. The NP-hardness of minimizing the total late work on an unbounded batch machine. *Asia-Pac. J. Oper. Res.* **2009**, *26*, 351–363. [CrossRef]
16. Ren, J.; Du, D.; Xu, D. The complexity of two supply chain scheduling problems. *Inf. Processing Lett.* **2013**, *113*, 609–612. [CrossRef]
17. Sterna, M. A survey of scheduling problems with late work criteria. Omega, 2011, 39(2): 120-129. *Omega* **2011**, *39*, 120–129. [CrossRef]
18. Pesch, E.; Sterna, M. Late work minimization in flow shops by a genetic algorithm. *Comput. Ind. Eng.* **2009**, *57*, 1202–1209. [CrossRef]
19. Gerstl, E.; Mor, B.; Mosheiov, G. Scheduling on a proportionate flowshop to minimise total late work. *Int. J. Prod. Res.* **2019**, *57*, 531–543. [CrossRef]
20. Bello, I.; Pham, H.; Le, Q.V.; Norouzi, M.; Bengio, S. Neural combinatorial optimization with reinforcement learning. *arXiv* **2016**, arXiv:1611.09940.
21. Hu, H.; Zhang, X.; Yan, X.; Wang, L.; Xu, Y. Solving a new 3d bin packing problem with deep reinforcement learning method. *arXiv* **2017**, arXiv:1708.05930.
22. Kool, W.; van Hoof, H.M.; Welling, M. Attention, Learn to Solve Routing Problems! In Proceedings of the International Conference on Learning Representations, Vancouver, BC, Canada, 30 April 2018.
23. Zhang, R.; Prokhorchuk, A.; Dauwels, J. Deep Reinforcement Learning for Traveling Salesman Problem with Time Windows and Rejections. In Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN), Glasgow, UK, 28 September 2020; pp. 1–8.
24. Vinyals, O.; Fortunato, M.; Jaitly, N. Pointer networks. *Adv. Neural Inf. Processing Syst.* **2015**, *28*, 1–9.
25. Nazari, M.; Oroojlooy, A.; Snyder, L.; Takáč, M. Reinforcement learning for solving the vehicle routing problem. *Adv. Neural Inf. Processing Syst.* **2018**, *31*, 1–11.
26. Chen, X.; Tian, Y. Learning to perform local rewriting for combinatorial optimization. *Adv. Neural Inf. Processing Syst.* **2019**, *32*, 6281–6292.
27. Lu, H.; Zhang, X.; Yang, S. A Learning-Based Iterative Method for Solving Vehicle Routing Problems. In Proceedings of the International Conference on Learning Representations, New Orleans, LA, USA, 6–9 May 2019.
28. Wu, Y.; Song, W.; Cao, Z.; Zhang, J.; Lim, A. Learning improvement heuristics for solving the travelling salesman problem. *arXiv* **2019**, arXiv:1912.05784.
29. Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; Song, L. Learning combinatorial optimization algorithms over graphs. *Adv. Neural Inf. Processing Syst.* **2017**, *30*, 1–11.
30. Lederman, G.; Rabe, M.N.; Seshia, S.A. Learning heuristics for automated reasoning through deep reinforcement learning. *arXiv* **2018**, arXiv:1807.08058.
31. Gupta, J.N.; Stafford, E.F., Jr. Flowshop scheduling research after five decades. *Eur. J. Oper. Res.* **2006**, *169*, 699–711. [CrossRef]
32. Baker, K.R.; Trietsch, D. *Principles of Sequencing and Scheduling*; John Wiley & Sons: New York, NY, USA, 2013.
33. Sutskever, I.; Vinyals, O.; Le, Q.V. Sequence to sequence learning with neural networks. *Adv. Neural Inf. Processing Syst.* **2014**, *27*, 1–9.
34. Scarselli, F.; Gori, M.; Tsoi, A.C.; Hagenbuchner, M.; Monfardini, G. The graph neural network model. *IEEE Trans. Neural Netw.* **2008**, *20*, 61–80. [CrossRef]
35. Ma, Q.; Ge, S.; He, D.; Thaker, D.; Drori, I. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *arXiv* **2019**, arXiv:1911.04936.
36. Xu, K.; Hu, W.; Leskovec, J.; Jegelka, S. How Powerful Are Graph Neural Networks? In Proceedings of the International Conference on Learning Representations, Vancouver, BC, Canada, 30 April 2018.
37. Kipf, T.N.; Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv* **2016**, arXiv:1609.02907.
38. Bennett, C.C.; Hauser, K. Artificial intelligence framework for simulating clinical decision-making: A Markov decision process approach. *Artif. Intell. Med.* **2013**, *57*, 9–19. [CrossRef]
39. Williams, R.J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.* **1992**, *8*, 229–256. [CrossRef]
40. Rennie, S.J.; Marcheret, E.; Mroueh, Y.; Ross, J.; Goel, V. Self-Critical Sequence Training for Image Captioning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017; pp. 7008–7024.
41. Ruiz, R.; Stützle, T. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *Eur. J. Oper. Res.* **2007**, *177*, 2033–2049. [CrossRef]
42. Dubois-Lacoste, J.; Pagnozzi, F.; Stützle, T. An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. *Comput. Oper. Res.* **2017**, *81*, 160–166. [CrossRef]
43. Taillard, E. Benchmarks for basic scheduling problems. *Eur. J. Oper. Res.* **1993**, *64*, 278–285. [CrossRef]
44. Ivković, N.; Jakobović, D.; Golub, M. Measuring performance of optimization algorithms in evolutionary computation. *Int. J. Mach. Learn. Comput.* **2016**, *6*, 167–171. [CrossRef]
45. Wang, H.; Wang, W.; Sun, H.; Cui, Z.; Rahnamayan, S.; Zeng, S. A new cuckoo search algorithm with hybrid strategies for flow shop scheduling problems. *Soft Comput.* **2017**, *21*, 4297–4307. [CrossRef]

46. Ince, Y.; Karabulut, K.; Tasgetiren, M.F.; Pan, Q.-K. A Discrete Artificial Bee Colony Algorithm for the Permutation Flowshop Scheduling Problem with Sequence-Dependent Setup Times. In Proceedings of the 2016 IEEE congress on Evolutionary computation (CEC), Vancouver, BC, Canada, 24–29 July 2016; pp. 3401–3408.
47. Wang, X.; Ren, T.; Bai, D.; Ezeh, C.; Zhang, H.; Dong, Z. Minimizing the sum of makespan on multi-agent single-machine scheduling with release dates. *Swarm Evol. Comput.* **2021**, *69*, 100996. [CrossRef]