Deep Reinforcement Learning Based Genetic Framework for Flexible Job-Shop Scheduling under Practical Constraints

Kjell van Straaten, Robbert Reijnen, Zaharah Bukhsh, Yaoxin Wu, and Yingqian Zhang

Department of Industrial Engineering & Innovation Sciences, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
kjell.van.straaten@wefabricate.com
{r.v.j.reijnen, z.bukhsh, y.wu2, yqzhang}@tue.nl

Abstract. In this paper, we propose a DRL-based genetic framework (DRL-GF) for solving flexible job shop scheduling problems (FJSPs) under various practical constraints in real-world applications. First, we model the genetic algorithm (GA) process as a Markov decision process (MDP). Then, we use a double-layer encoding scheme to represent the population of schedules for an FJSP instance, where we develop a set of problem-agnostic features to describe the state of the GA solution process. We train a multilayer perceptron (MLP) using a proximal policy optimization (PPO) algorithm to determine the mutation probability, crossover probability, and mutation rate simultaneously. We evaluate the proposed DRL-GF on the standard FJSP instances and the FJSP with sequence-dependent setup time (SDST). Moreover, we test our method on real-world FJSP instances with additional practical constraints. Extensive results demonstrate that DRL-GF outperforms conventional heuristics and end-to-end DRL methods in each scenario, requiring minimal problem-specific customization. In addition, we show that even if we train DRL-GF using the classical FJSP instances, the learned policy can be used directly to solve the heavily constrained FJSP-SRMN , greatly outperforming the benchmarked methods.

Keywords: Deep reinforcement learning · Flexible job-shop scheduling problem · Genetic algorithm

1 Introduction

Flexible job-shop scheduling problems (FJSPs) focus on the efficient allocation of sequences of operations across multiple machines. This problem is inherently more complex than job-shop scheduling problems, which are already NP-hard. This complexity arises from the eligibility of multiple machines for each operation, leading to a combinatorially increasing number of potential allocations. These allocations are often optimized based on specific criteria such as the total completion time (i.e., the makespan) and the total lateness. FJSPs have broad

applications in industrial manufacturing, maintenance planning, and cloud computing (Gao et al., 2019).

Conventional approaches to FJSPs include mathematical programming, approximation algorithms, and heuristics. Although the former two are characterized by (near)optimal guarantee, they often suffer from high computational costs or are limited to only a few approximable cases (Fleischer et al., 2006). In contrast, heuristics are widely used to solve FJSPs for efficient solutions (Liang et al., 2021). However, tuning heuristics to solve a specific problem is far from automatic and requires laborious manual work to attain favorable performance.

Deep reinforcement learning (DRL) has recently been extensively explored to solve JSP and its variants (Zhang et al., 2020; Yang and Xu, 2021; Park et al., 2021; Liu et al., 2020). Most DRL methods attempt to learn priority dispatching rules (PDRs), a type of constructive heuristic, to prioritize certain operations or jobs over others for scheduling. The learned PDRs are shown to be more advantageous than conventional ones, e.g., First-Come-First-Served rule and Shortest Processing Time rule (Sels et al., 2012). As a natural extension, DRL has been applied to learn PDRs for JSPs (Song et al., 2022; Lei et al., 2022; Zhang et al., 2023b; Luo et al., 2021).

However, existing DRL methods for FJSPs are still inferior in terms of the following aspects: 1) They only learn simple constructive heuristics (i.e., PDRs), which cannot handle complex constraints; 2) They suffer the inferior flexibility to be deployed with different scenarios; 3) They suffer from poor generalization and cannot perform persistently good when the test instances are different from the trained ones. In fact, FJSPs are hard to solve in real life with complex yet practical constraints, such as Sequence-Dependent Setup Times (SDST).

To address these limitations, there has been a growing interest in the integration of machine learning (ML) methods with evolutionary algorithms (EAs)(Zhang et al., 2023a; Reijnen et al., 2024; Song et al., 2024; Zhou et al., 2024; Sharma et al., 2019; Reijnen et al., 2023b). These learning-assisted evolutionary algorithms aim to exploit knowledge from search dynamics to overcome common EA challenges such as slow convergence and poor generalization(Song et al., 2024). More recently, reinforcement learning (RL), and especially DRL, has shown promise in this context due to its ability to learn from sequential interactions. Within this framework, parameter configurations are treated as actions, and the EA's performance guides the learning process via reward signals. Several studies have demonstrated the effectiveness of DRL in this role.

Motivated by these insights, this paper proposes a DRL-based Genetic Framework (DRL-GF) for solving FJSPs under practical constraints. DRL-GF leverages DRL to assist Genetic Algorithms (GAs) in automatically determining parameters (e.g., crossover and mutation rates) in iterations of the search process. Compared to the existing (few) approaches that tune the parameters of GAs with DRL for FJSP, as shown in (Chen et al., 2020), we model the algorithmic procedure of GAs as a Markov decision process (MDP) in a generic manner, allowing its application to a wide range of FJSP variants (including practical constraints). A double-layer encoding scheme is designed to represent the pop-

ulation of schedules (or solutions) for an FJSP instance, with a corresponding parallel decoding and evaluating scheme. In addition to the parallel process of the population, we propose a set of problem-agnostic features to reflect the solving state of GAs. Then, we train a multilayer perceptron (MLP) by proximal policy optimization (PPO) algorithm to jointly determine the mutation probability, the crossover probability, and the mutation rate. The proposed hybrid use of DRL and GA complement each other to be more effective in constraint handling, flexibility, and robustness. Compared to simple constructive heuristic PDRs, the GAs, a type of mature metaheuristics, are more effective in gaining high-quality solutions, relying on the iterative search with advanced schemes to escape local minima. In addition, DRL-GF further enhances GAs by learning more intelligent policies to adaptively determine the key parameters in the search process, which are less myopic and consider the algorithmic long-term performance in the policy optimization by DRL. We evaluate the proposed DRL-GF on the basic FJSP, the complex FJSP with SDST constraint, and a real-world FJSP with more practical constraints, including release dates, maintenance jobs and night times. The results show that DRL-GF generally outperforms conventional heuristics and end-to-end DRL methods.

2 FJSP with Practical Constraints

The Flexible Job Shop Scheduling Problem (FJSP) builds upon the traditional Job Shop Scheduling Problem (JSP) by introducing greater flexibility in machine assignments. It consists of n independent jobs, represented as $J = \{J_1, J_2, \ldots, J_n\}$, and m independent machines, denoted by $M = \{M_1, M_2, \ldots, M_m\}$, often referred to as an $n \times m$ FJSP problem. Each job J_i has a number of operations $O_{i,j}$, where $O_{i,j}$ is the j-th operation of the i-th job, and where operation $O_{i,j+1}$ may only be started after $O_{i,j}$ is completed. The processing time of $O_{i,j}$ on machine M_k is denoted as $t_{i,j,k}$, and a machine m can process one operation at a time. $C_{i,j}$ is the completion time of $O_{i,j}$, and O_k the set of operations scheduled to be processed on machine M_k. The objective in FJSP is often to minimize the makespan, denoted as $C_{\max}$ that represents the total time required to complete all jobs, i.e., $C_{\max} = \max\{C_{i,j} : i, j = 1, \ldots, n\}$.

Real-world FJSPs are characterized by more practical and complex constraints built on top of the standard FJSP. In this paper, we focus on two variants of FJSP with practical constraints:

FJSP-SDST: FJSP with sequence-dependent setup time (SDST) describes the practical FJSP, where the setup time for a machine to process a job is influenced by the previously processed job on this machine. After completing any operation from job J_i and before beginning any operation from job J_j, a setup time l_{ij} is required if both operations are on the same machine. Formally, if an operation from job J_i ends at time T on machine M_k, then an operation from job J_j can start at time $T + l_{ij}$ on the same machine M_k. The setup time l_{ij} is known in advance for every pair of jobs J_i and J_j.

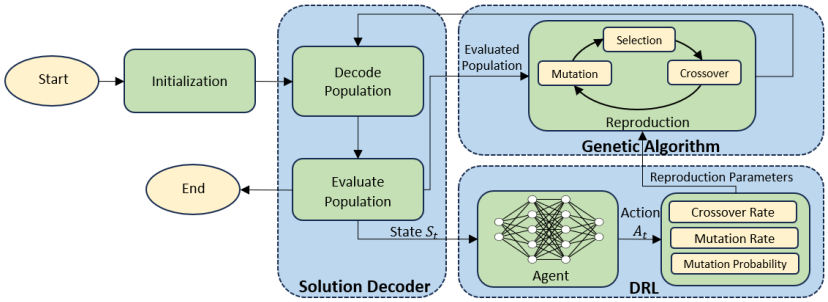

Fig. 1. An Overview of Deep Reinforcement Learning based Genetic Framework

FJSP-SRMN: FJSP with Sequence-dependent setup time, Release dates, Maintenance jobs and Night times (SRMN) can describe more complex industrial manufacturing procedures in real-world scenarios, e.g., the milling process in a production company with real data used in our experiments (see Section 4.3). *Release dates* specify when a job becomes available for processing: each job J_i has an associated release date r_i, representing the earliest time the job can start. The first operation $O_{i,1}$ of job J_i cannot begin before time r_i. *Maintenance jobs* refer to time periods when machines are unavailable due to planned maintenance. Let PM_k be the maintenance duration for machine M_k; if it starts at time T, then M_k is not available during the interval $[T, T + PM_k]$. Finally, *night times* represent periods during which no operations can be performed, typically due to work-time restrictions. Let NT be the duration of such a period; no operation can start during NT, and if an operation is running when NT begins, it may either be paused and resumed afterward or allowed to continue, depending on specific problem requirements.

3 DRL-based Genetic Framework

The proposed DRL-based genetic framework DRL-GF is structured into three primary modules as shown in Figure 1: 1) the solution decoder, which is responsible for interpreting the encoded solutions into schedules and its evaluation; 2) the GA, tasked with the evolution of the solution population (i.e., schedules); and 3) the DRL module, which dynamically adjusts the key parameters of the GA to enhance overall performance. The DRL-GF process begins with initializing solutions (encoded as chromosome representations). These solutions are then decoded and evaluated, with the resulting population analyzed by the DRL agent, which accordingly adjusts the GA parameters in response to the ongoing state of the search process. After the parameters are set, one iteration of the GA search (e.g., crossover, mutation, etc.) is exerted on the encoded solutions, and the new offspring population is sent back to the decoder for decoding and evaluation in the next step. This iterative process continues until the termination criterion is met. The final population and the best-found solutions are returned.

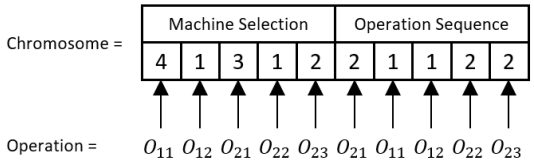

Fig. 2. Chromosome Representation for FJSP variants

3.1 Schedule Encoding and Decoding

Encoding. We encode a schedule in a chromosome representation to utilize the evolutionary search component of DRL-GF. We use the double-layer encoding scheme adapted from (Liang et al., 2021). This encoding scheme comprises a machine selection component and an operation sequence component. The machine selection component stores information about the assignment of operations to machines. It is encoded in an array of integers, where every operation has an assigned fixed position. The value of the array at a position corresponds to the index of the machines selected for the execution of the operation. The operation sequence component represents the sequence of scheduling the operations on allocated machines. Each index i of job J_i appears $O_{i,j}$ times in the operation sequence component. The length of each component is thus equal, and the total chromosome length is twice the number of operations to be scheduled.

Figure 2 provides an example of the encoded chromosome representation, where the FJSP instance consists of two input jobs J_1 and J_2. J_1 has two operations $O_{1,1}$ and $O_{1,2}$, whereas J_2 has three operations $O_{2,1}$, $O_{2,2}$ and $O_{2,3}$. The value of 4 in the first position of the machine selection component means that operation $O_{1,1}$ should be scheduled on its fourth machine alternative. The operation sequence component shows that this operation will be scheduled second, following the scheduling of operation $O_{2,1}$.

Decoding. We design the decoder to interpret schedules from the encoded representations, allowing us to evaluate the quality of initialized or evolved solutions. This process involves allocating specific start and end times to each operation, determined by the allocated machine and prior scheduled operations. The decoding scheme is initialized with the operation sequence and machine allocation from the encoded solutions, together with the job instance information (I), consisting of all required information for scheduling the operations, such as required resources, sequence-dependent setup times, and release dates. To ensure validity for the specific problem constraints, the decoder consists of three subroutines: *CheckBackFill*, *CalcSDST* and *NightPush*. The *CheckBackFill* subroutine examines each pair of subsequent operations, A and B, which have already been scheduled, and assesses whether it is feasible to schedule the current operation C at the end of operation A, or before the start of operation B, altering the required setups before and after the operation in question, and maintaining the satisfaction of precedence constraints. *CalcSDST* subroutine evaluates the current status of the machine and the operation in question. It uses this information to determine the necessary setup procedures the machine must undergo to start

the operation. The *NightPush* subroutine is designed to set the starting time of a schedule to the beginning of the next day. When a solution (i.e., schedule) is decoded, and timestamps are allocated to all individual operations, the schedule can be replayed to determine the makespan and cost of operations, which is conducted through simulation as the environment in our DRL-GF.

3.2 GA for FJSP and its Variants

Within DRL-GF, the GA is used to evolve the population to find high-quality solutions. The process begins by initializing a population of a specified size. Consequently, the population of individuals is recombined by a *crossover* function and modified by a *mutation* function. This population is evaluated by the decoder, where the objective of each schedule is calculated and used as the fitness indicator for each individual. Through a *selection* function, the most promising individuals are chosen over a predetermined number of generations.

Initialization. The initialization of the population is performed according to (Zhang et al., 2011), combining *global*, *local* and *random selection*, according to a 60%, 30%, and 10% ratio. Global selection aims to balance the load across all machines. More specifically, an array of size $|M|$ is initialized. Random jobs (J) are then selected, and their operations (O_{ij}) are scheduled on the machine currently having the lowest total load. After scheduling an operation, the duration is added to the total load of the selected machine. Local load balancing is similar but optimizes load on a per-job basis rather than optimizing machine load across all jobs. Finally, in random selection, both the machine allocation string and operation sequence are randomly initialized.

Crossover. The crossover process is different for the machine allocation and operation sequence string. Specifically, the operation sequence string is crossed over using precedence preserving order-based crossover (POX) (Lee et al., 1998). According to POX, two sub-job sets J_{s1} and J_{s2} are randomly generated from all jobs, and two-parent individuals p_1 and p_2 are randomly chosen. Genes in p_1 (p_2) that belong to J_{s1} (J_{s2}) are copied into the child individual c_1 (c_2) and remain in their original positions (as in the parent individuals). Any genes already in sub-job J_{s1} (J_{s2}) are removed from p_2 (p_1). Then, the vacant positions in c_1 or c_2 are filled with the genes of p_2 (p_1) according to their original sequence (as in the parent individuals). This process ensures the preservation of the relative scheduling sequence position of operations from a randomly selected set of jobs while rescheduling operations from other jobs based on the sequence of the alternate crossovered individual solution.

Mutation. For the machine allocation string, we select the machine for which the operation-machine pair has the lowest operating time in case of mutation. In the case of sequence-dependent setup times, the required setup between the current operation of the machine and eligible operations is added to the operating time before the operation-machine pair is picked. For the operation sequence string, we generate a new index $0 \leq i \leq |OS|$ and swap the gene under consideration with the gene in the generated index.

Algorithm 1: DRL-based Genetic Framework

Input: Instance information (I); DRL policy (π); Population size (n_pop); Number of generations (n_gen)
Initialize: $pop \leftarrow []$; $hof \leftarrow []$
`// Generate initial population`
for $x \in range(n\_pop)$ **do**
  $rnd \leftarrow random(0, 1)$ ▷ Random value between 0 and 1
  if $rnd < 0.6$ **then**
    $Indv \leftarrow GlobalSelection()$ ▷ Select using global strategy
  else if $rnd < 0.9$ **then**
    $Indv \leftarrow LocalSelection()$ ▷ Select using local strategy
  else
    $Indv \leftarrow RandomSelection()$ ▷ Select randomly
  $pop[x] \leftarrow Indv$
`// Evolve population through generations`
for $gen \in range(n\_gen)$ **do**
  $pop \leftarrow evaluate(pop)$ ▷ Evaluate fitness of population
  $S \leftarrow getState(pop)$ ▷ Configure state representation
  $A \leftarrow \pi(S)$ ▷ Configure parameters using DRL policy
  $ofsp \leftarrow selection(pop)$ ▷ Select individuals for reproduction
  $ofsp \leftarrow crossover(ofsp, A)$ ▷ Apply crossover
  $new\_pop \leftarrow mutation(ofsp, A)$ ▷ Apply mutation
  $R \leftarrow reward(new\_pop, pop)$ ▷ Compute reward (training phase only)
  $pop \leftarrow new\_pop$ ▷ Replace old population

Selection. We use a tournament selection method with a tournament size set to 3, based on the makepan of solutions, following Zhang et al. (2011). After selection, a subset of solutions obtained is used for the iterative crossover and mutation process, as explained above.

3.3 DRL for Parameter Control

A DRL component is used within DRL-GF for adaptive parameter control. To do so, we define a Markov decision process (MDP) for the GA.

State space (S_t): The state is featured by the normalized mean fitness of the current population, the normalized best fitness of the current population, normalized standard deviation of fitness of the current population, normalized remaining budget, and normalized stagnation count. The fitness (f) values are normalized by scaling the fitness values on the range of worst (f_{max}) and best-seen fitness (f_{min}) values so far, i.e., $\hat{f} = (f - f_{min})/(f_{max} - f_{min})$. Following this formula, a value of 0 would indicate optimal fitness and 1 would indicate worst fitness. Moreover, we normalize the budget (b), i.e., the number of generations left, and stagnation count (s_c) by dividing them by the total number of generations (n_gen) as $\hat{b} = b/n\_gen$ and $\hat{s}_c = s_c/n\_gen$, respectively. The normalized standard deviation, and all other features in the state space are clipped to 1 to reduce the impact of extreme observations.

Action space (A_t): The action determines 1) the mutation probability (i.e., the probability of each individual mutating); 2) the crossover probability, and 3) the mutation rate (i.e., the probability each gene mutating). The range of the probabilities and the rate are all between 0 and 1, which allows for radical changes from one generation to the other. Based on the selected probabilities, the GA executes the set of genetic operations in the current iteration.

Reward (R_t): The reward is defined as $R_t = T_{c_{t-1}} - T_{c_t}$. This reward is sent to the agent as feedback, together with the next state space (S_{t+1}) based on the freshly created population (P_t).

Transition ($\mathcal{P}$): The next state S_{t+1} is derived from S_t by the GA operations (i.e., crossover, mutation, selection), which are configured with the parameters determined by the action A_t.

We provide the pseudocode of the DRL-GF procedure in Algorithm 1.

3.4 Policy Optimization

We use the proximal policy optimization (PPO) algorithm (Schulman et al., 2017) to train the proposed DRL model. PPO utilizes a probability ratio between two policies to maximize the improvement of the current policy without the risk of performance collapse. The objective function of PPO is defined as $L^{CLIP}(\theta) = \min\left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$, where $r_t(\theta)$ is the probability ratio of taking actions under the new policy to take the same actions under the old policy and $\hat{A}_t$ is the estimated advantage of taking action a_t at time step t, representing the expected reward difference between the current policy and the target. The term $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)$ represents the clipped surrogate objective which clips the probability ratio between $1-\epsilon$ and $1+\epsilon$. The policy is updated in an actor-critic setting, with N actor networks and a critic network all sharing the same network architecture. The actors run the old policy $\pi_{\theta_{old}}$ in an environment for T timesteps. Within the DRL-GF, multiple genetic algorithms are executed before the policy θ is updated. After the T timesteps, $\mathcal{L}^{CLIP}_{\theta_k}(\theta)$ is optimized for K epochs given a mini-batch size $M \leq N \times T$, resulting in a new policy.

Both policy and value networks consist of two linear layers. The first layer has 5 input features (i.e., the defined state space) and 64 output features. Using a Tanh activation function, the second layer has 64 input and output dimensions. Before the output layer, the 64 hidden features represent the embedding of the current state of the optimization algorithm. To estimate the value and action under a state, we employ an additional output layer with 64-dimensional input and 1-dimensional output to build a value network and the output layer with 3-dimensional output to build the actor network. The value network outputs the expected value of a state, which is used for PPO training. The actor network outputs the crossover rate, mutation rate, and mutation probability in the GA.

4 Experiments

We compare the performance of DRL-GF with various baselines. We first include the commonly used dispatching rules, i.e., shortest processing time (SPT), most work remaining (MWKR), most operations remaining (MOR), first-in-first-out (FIFO). Furthermore, we include random scheduling (RANDOM) and greedy scheduling (GREEDY), in which the operation-machine pair increasing the total makespan the least is selected. In addition, we include the advanced learning-based methods, including the SLGA of Chen et al. (2020) and the end-to-end DRL (E2E-DRL) approach of Song et al. (2022). We use the trained E2E-DRL model in two different ways during testing, i.e., obtaining actions by sampling (DRL-S) or in a greedy manner (DRL-G). The genetic algorithm (EGA) of Zhang et al. (2011) that serves as the base of the DRL-GF is also compared.

Performance indicator. The performance indicators include the optimality gap, ranking score, win count, and computation time. In specific, the optimality gap is calculated as $G_{opt} = \frac{\hat{C}_{\max}}{LB-1} * 100\%$, where $\hat{C}_{\max}$ and LB are the average makespan and the average lower bound for an instance set. Lower bounds are obtained as follows: For FJSP, we use values reported in the original benchmarking papers (Brandimarte, 1993; Hurink et al., 1994); For FJSP-SDST, we use best solutions found by the CP-SAT solver (i.e., the optimal solutions for instance 1-19, and the best-found solution for instance 20 in 72h); For Wfdata, it is the best makespan found across all algorithms for each instance.

We also introduce the ranking score and win account in the experiments on FJSP-SDST and FJSP-SRMN. An algorithm wins an instance if it has (or ties with) the best makespan for that instance. Multiple algorithms might “win" an instance. The win count is used to identify which algorithm performs best on all test instances, as the average optimality gap is influenced significantly by performance outliers in test instances. The ranking score of a method is defined as the average value of its rankings on all tested instances.

Training setup. All models are trained on an AIME A4000 - Multi GPU HPC rack server with 96GB of GPU memory, 132GB of RAM, and 48 cores. Training the DRL-GF until convergence took between 1.5 (10x05) and 5.5 hours (20x10), depending on the instance sizes. For the experiment on FJSP, we trained the models on the instances of Song et al. (2022) (*sodata*). For FJSP-SDST, we train the models using the same *sodata* instances but have added the sequence-dependent setup times, which are uniformly generated between 1 and 15. For FJSP-SRMN, we train our models on custom-generated data (*wfdata*).

Benchmark Datasets. The classical FJSP datasets used for testing are the Brandimarte dataset in (Brandimarte, 1993) (denoted by *mkdata*) and the Hurink datasets in (Hurink et al., 1994) that are composed of *edata*, *rdata* and *vdata* dataset. To investigate FJSP-SDST, we use an extension on the Fattahi dataset (Saidi-Mehrabad and Fattahi 2007) provided by LocalSolver [1], which is denoted by *ftdata*. To benchmark FJSP-SRMN, we generate a set of custom instances (denoted by *wfdata*). These custom instances are generated using a fixed set of

[1] https://www.localsolver.com/docs/last/exampletour/flexiblejobshop.html#data

Method	Training Size	mkdata		edata		rdata		vdata	
		$\hat{C}_{\max}$	*Gopt*	$\hat{C}_{\max}$	*Gopt*	$\hat{C}_{\max}$	*Gopt*	$\hat{C}_{\max}$	*Gopt*
OPT		163.3	-	1005	-	923	-	807.9	-
SPT		283.2	73.48%	1305.35	29.89%	1184.80	28.36%	-	-
MOR		202.3	23.89%	1211.2	20.52%	1064.0	15.28%	-	-
MWKR		200.17	22.58%	1179.9	17.40%	1046.2	13.35%	-	-
FIFO		206.1	26.20%	1255.46	24.92%	1082.1	17.24%	-	-
RANDOM		637.5	290.39%	1235.1	21.98%	1212.9	31.41%	1041.7	28.94%
GREEDY		484.9	196.94%	1290.5	28.41%	1150.4	24.64%	894.5	10.72%
EGA		192.2	17.09%	1144.6	13.89%	1105.3	19.75%	953.4	18.01%
SLGA		181.3	11.02%	-	-	-	-	-	-
DRL-G	10x05	200.1	22.54%	1193.1	18.72%	1049.7	13.73%	856.1	5.97%
	15x10	200.3	22.66%	1197.5	19.15%	1054.3	14.23%	858.0	6.20%
	20x05	220.1	34.78%	1269.5	26.32%	1125.1	21.90%	897.4	11.08%
	20x10	199.3	22.05%	1192.5	18.66%	1046.2	13.35%	841.3	4.13%
	Mixed	198.0	21.25%	1218.0	21.19%	1056.3	14.44%	845.0	4.59%
DRL-S	10x05	194.6	19.16%	1139.5	13.38%	1009.0	9.32%	**827.6**	**2.44%**
	15x10	193.1	18.25%	1144.9	13.92%	1008.2	9.23%	828.8	2.59%
	20x05	208.1	27.37%	1176.0	17.01%	1049.1	13.66%	857.8	6.18%
	20x10	195.2	19.53%	1152.85	14.71%	1015.5	10.02%	830.9	2.85%
	Mixed	196.8	20.51%	1107.5	10.20%	**990.0**	**7.26%**	831.1	2.87%
DRL-GF	10x05	180.7	10.66%	1103.5	9.80%	1047.8	13.52%	894.5	10.72%
	15x10	184.5	12.98%	1110.3	10.48%	1061.0	14.95%	914.7	13.22%
	20x05	180.1	10.29%	**1097.7**	**9.22%**	1040.3	12.71%	895.4	10.83%
	20x10	180.5	10.53%	1099.5	9.40%	1034.9	12.12%	888.13	9.93%
	Mixed	**178.6**	**9.37%**	1111.8	10.63%	1069.3	15.85%	955.8	18.31%

Table 1. Results on FJSP test dataset. **-** marks that objective values and gaps are not reported in original papers or the benchmark. The training size is not applicable to the non-learning based methods.

potential jobs. This set of potential jobs consists of 75 jobs, each consisting of 1-5 operations. The jobs can have a quantity between one and five. The processing time is given per operation per product, whereas the total processing time for an operation is then calculated by multiplying the processing time by the quantity. The duration of operations ranges anywhere from 0 to 5000. Because quantities are between 1 and 5, there exist 375 different jobs which can exist within an instance. The number of machines and jobs is set before instances are generated. The instance is considered as completely flexible, meaning that all operations can be scheduled on all machines. Furthermore, the operations have equal processing times on each machine. Sequence-dependent setup times are based on either 0 or 10000 time units. The instances can be found in the machine scheduling benchmark in (Reijnen et al., 2023a).

4.1 Experiment on Classical FJSP

In the first experiment, we benchmark the DRL-GF on classical FJSP instances. We proceed with the following experiments. Each method is tested on different datasets. The first is the Brandimarte dataset from (Brandimarte, 1993), i.e., *mkdata*, and the other three datasets are *edata*, *rdata* and *vdata* derived from the Hurink dataset (Hurink et al., 1994). In the Hurink dataset, we only use the *la* instances, 1-40 for *edata* and *rdata*, and 1-30 for *vdata*. These datasets are used to benchmark our method against existing learning-based methods, heuristic methods, and dispatching rules. The results are gathered in Table 1.

Method	Training Size	$\hat{C}_{\max}$	*Gopt*	Win Count	Average Rank
OPT		468.8	-	-	-
MWKR		787.4	67.96%	0	8.85
RANDOM		638.6	36.22%	10	3.75
GREEDY		667.5	42.38%	0	7.30
EGA		607.6	29.54%	4	5.10
DRL-G	10x05	643.2	37.20%	3	5.25
	15x10	634.9	35.43%		
	20x05	681.3	45.33%		
	20x10	648.4	38.31%		
DRL-S	10x05	577.0	18.81%	7	2.70
	15x10	572.8	22.18%		
	20x05	594.6	26.83%		
	20x10	576.3	22.93%		
DRL-S*	10x05	580.0	23.72%	7	3.10
	15x10	580.3	23.78%		
	20x05	589.9	25.83%		
	20x10	583.3	24.42%		
DRL-GF	10x05	**543.3**	15.89%	15	1.25
	15x10	568.4	21.25%		
	20x05	556.4	18.69%		
	20x10	549.4	17.19%		
DRL-GF*	10x05	547.7	16.83%	**16**	**1.20**
	15x10	547.1	16.70%		
	20x05	552.2	17.79%		
	20x10	545.5	16.36%		

Table 2. Results on FJSP-SDST test dataset (i.e., *ftdata*). OPT (optimal solution) is found using Gurobi. *models are trained on classical FJSP instances as described in the training setup.

For the *mkdata*, the E2E-DRL performance is inferior to DRL-GF and EGA. The DRL-S model (which consistently outperforms DRL-G) only reaches an optimality gap of 18.25% at best. Our DRL-GF models trained on different-sized instances generally reach an optimality gap of around 10%. Especially, the DRL-GF model trained on mixed sizes obtains the smallest optimality gap among all methods. For the *edata*, we observe almost the same effect as shown in *mkdata*, where the DRL-GF models generally perform better than other methods, and the DRL-GF model trained on 20×05 obtains the best performance. However, the DRL-S approach performs better for the *rdata* and *vdata*. The reasons behind the difference in performance advantages across the datasets need to be further investigated in the future. Meanwhile, it is worth noting that the DRL-GF generally outperforms traditional heuristics.

In summary, the E2E-DRL outperforms DRL-GF on *rdata* and *vdata* FJSP instances, while DRL-GF outperforms E2E-DRL on the *mkdata* and *edata*. In the following experiments, we will demonstrate that DRL-GF significantly surpasses E2E-DRL for complex FJSP variants with more practical constraints. Such extended experiments will indicate that DRL-GF is more advantageous than E2E-DRL over a broad spectrum of FJSP variants.

4.2 Experiment on FJSP-SDST

In our second experiment, we benchmark trained models against traditional heuristics on FJSP-SDST instances, i.e., *ftdata*. The sizes of these instances

range from 2×2 to 12×8. Furthermore, we compare our models trained on FJSP-SDST instances against our models trained on classical FJSP instances without SDSTs, in order to identify whether retraining is necessary. As observed from Table 2, DRL-GF outperforms E2E-DRL significantly. DRL-GF manages to reach an optimality gap of 1.29%, whereas DRL-S only reaches 6.80% at best. The computation of E2E-DRL is faster than DRL-GF. The runtime difference is larger than in the experiment on classical FJSP, since the *ftdata* dataset has smaller instances (12×8 at most) than the *mkdata* (20×15 at most). The win count and average rank also manifest that DRL-GF outperforms E2E-DRL for FJSP-SDST. The win count of DRL-GF is 15 and 16 for the models trained on FJSP and FJSP-SDST, respectively, while DRL-S only reaches a win count of 7. The average rank of DRL-GF equals 1.20 and 1.25, which is the best across all methods. While DRL-S is inferior to our DRL-GF, we note that DRL-S still outperforms traditional heuristics. It is indicated by the optimality gap, where DRL-S reaches a value of 18.81%. The vanilla genetic algorithm follows with an optimality gap of 29.54%. Random scheduling does have a win count of 10, which is higher than the win count of DRL-S (7). However, the average rank (3.75) of random scheduling is worse than the rank of DRL-S (2.70). This is due to the fact that random scheduling is able to brute-force the first 10 instances which are relatively small (4 jobs, 5 machines at most). For larger instances, DRL-S achieves much better makespans and thus has a better average rank.

When comparing the newly trained algorithms versus the ones from the previous experiment on classical FJSP, we observe that retraining enables a better makespan to be reached. DRL-GF improves from an optimality gap of 16.36% (20x10) to 15.89% (10x05). The average rank, however, decreased from 1.20 to 1.25. Since this improvement is considered insignificant, we conclude that retraining doesn't add any extra benefits for DRL-GF. Instead, retraining does help improve DRL-S, since the best optimality gap decreases from 23.72% (10x05) to 18.81% (15x05), and the average rank decreases from 3.10 to 2.70. However, we note that the retrained DRL-S model remains inferior to the DRL-GF models trained on either classical FJSP or FJSP-SDST instances. It implies that DRL-GF generalizes better to FJSP instances with unseen characteristics.

4.3 Experiment on FJSP-SRMN

We proceed to benchmark our DRL-GF on FJSP-SRMN. Here, we follow a similar approach as in the first two experiments, where models are trained on instances of several different sizes (17×02, 42×02, 64×04 and 88×08), and tested on a diverse set of instance sizes. More specifically, instances where the number of jobs ranges between 5 and 100 and the number of machines ranges between 2 and 10 are considered. An instance of size 100x10 should match the industry scale. We use these instances in order to test generalizability and scalability.

Table 3 displays the results on FJSP-SRMN instances. Note that the makespan is significantly larger than the aforementioned results, because the makespan is given in seconds for these instances. As shown, the DRL-GF model trained on 17×02 instances performs best, with an optimality gap of 4.96%. The models

Method	Training Size	$\hat{C}_{\max}$	*Gopt*	Win Count	Average Rank
LB		132173	-	-	-
MWKR		266848	101.89%	1	6.85
RANDOM		195869	46.95%	3	5.29
GREEDY		164040	23.07%	24	3.16
EGA		161651	22.31%	13	3.14
DRL-G	17x02	177421	34.23%	1	3.96
	42x02	181551	37.44%		
	64x04	192354	45.53%		
	88x08	187690	42.00%		
DRL-S	17x02	175477	32.76%	5	3.51
	42x02	182804	38.31%		
	64x04	177147	34.03%		
	88x08	183488	38.82%		
	15x10*	188850	42.88%		
DRL-GF	17x02	**138733**	**4.96%**	**81**	**1.21**
	42x02	160726	21.60%		
	64x04	142333	7.69%		
	88x08	144898	9.63%		
	15x10*	143992	8.03%		

Table 3. Results on FJSP-SRMN test dataset (i.e., *wfdata*). LB (lower bound) is calculated by the best makespan found per instance across all algorithms. * indicates that models are trained on classical FJSP.

trained on 64×04 and 88×08 instances achieve 7.69% and 9.63%, respectively. The average rank of DRL-GF is 1.21, with a total win count of 81 (out of 100).

DRL-S tends to outperform DRL-G, given the fact that DRL-S samples actions, whereas DRL-G picks actions in a greedy fashion. When training on instances of sizes 17×02, 64×04 and 88×08, DRL-S improves performance by 1.5%, 11.5% and 3.2%, respectively. However, DRL-S underperforms greedy scheduling and EGA (while it has advantage for FJSP and FJSP-SDST), receiving a significantly lower rank. We also tried to retrain DRL-GF and E2E-DRL models, which can increase performance by around 3% and 10%, but are still inferior to DRL-GF. In combination with results for *mkdata* and *ftdata*, we summarize that the proposed DRL-GF outperforms the baselines, showing a stronger versatility to solve FJSPs under diverse practical constraints.

5 Conclusion

We propose a DRL-based genetic framework (DRL-GF) for solving FJSPs under various practical constraints. We assess the DRL-GF on the classical FJSP, the FJSP with SDST constraint, and a real-world FJSP with various practical constraints. Extensive results show that the proposed DRL-GF generally exceeds conventional heuristics and end-to-end DRL methods. Despite the significant superiority of DRL-GF on complex FJSP variants, one limitation of our work is that DRL-GF cannot consistently outperform E2E-DRL on all classical FJSP datasets, which we will explore underlying reasons for the performance variance.

Bibliography

Brandimarte, P., 1993. Routing and scheduling in a flexible job shop by tabu search. Annals of Operations Research 41, 157–183.

Chen, R., Yang, B., Li, S., Wang, S., 2020. A self-learning genetic algorithm based on reinforcement learning for flexible job-shop scheduling problem. Computers & industrial engineering 149, 106778.

Fleischer, L., Goemans, M.X., Mirrokni, V.S., Sviridenko, M., 2006. Tight approximation algorithms for maximum general assignment problems, in: SODA, Citeseer. pp. 611–620.

Gao, K., Cao, Z., Zhang, L., Chen, Z., Han, Y., Pan, Q., 2019. A review on swarm intelligence and evolutionary algorithms for solving flexible job shop scheduling problems. IEEE/CAA Journal of Automatica Sinica 6, 904–916.

Hurink, J., Jurisch, B., Thole, M., 1994. Tabu search for the job-shop scheduling problem with multi-purpose machines. OR Spectrum = OR Spektrum 15, 205–215. doi:https://doi.org/10.1007/BF01719451.

Lee, K.M., Yamakawa, T., Lee, K.M., 1998. A genetic algorithm for general machine scheduling problems, in: 1998 Second International Conference. Knowledge-Based Intelligent Electronic Systems. Proceedings KES'98 (Cat. No. 98EX111), IEEE. pp. 60–66.

Lei, K., Guo, P., Zhao, W., Wang, Y., Qian, L., Meng, X., Tang, L., 2022. A multi-action deep reinforcement learning framework for flexible job-shop scheduling problem. Expert Systems with Applications 205, 117796.

Liang, X., Chen, J., Gu, X., Huang, M., 2021. Improved adaptive non-dominated sorting genetic algorithm with elite strategy for solving multi-objective flexible job-shop scheduling problem. Ieee Access 9, 106352–106362.

Liu, C.L., Chang, C.C., Tseng, C.J., 2020. Actor-critic deep reinforcement learning for solving job shop scheduling problems. Ieee Access 8, 71752–71762.

Luo, S., Zhang, L., Fan, Y., 2021. Dynamic multi-objective scheduling for flexible job shop by deep reinforcement learning. Computers & Industrial Engineering 159, 107489.

Park, J., Chun, J., Kim, S.H., Kim, Y., Park, J., 2021. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. International Journal of Production Research 59, 3360–3377.

Reijnen, R., van Straaten, K., Bukhsh, Z., Zhang, Y., 2023a. Job shop scheduling benchmark: Environments and instances for learning and non-learning methods. arXiv preprint arXiv:2308.12794 .

Reijnen, R., Zhang, Y., Bukhsh, Z., Guzek, M., 2023b. Learning to adapt genetic algorithms for multi-objective flexible job shop scheduling problems, in: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, Association for Computing Machinery, New York, NY, USA. p. 315–318. URL: https://doi.org/10.1145/3583133.3590700, doi:https://doi.org/10.1145/3583133.3590700.

Reijnen, R., Zhang, Y., Lau, H.C., Bukhsh, Z., 2024. Online control of adaptive large neighborhood search using deep reinforcement learning, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 475–483.

Saidi-Mehrabad, M., Fattahi, P., 2007. Flexible job shop scheduling with tabu search algorithms. The international journal of Advanced Manufacturing technology 32, 563–570.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal policy optimization algorithms. `arXiv:1707.06347`.

Sels, V., Gheysen, N., Vanhoucke, M., 2012. A comparison of priority rules for the job shop scheduling problem under different flow time-and tardiness-related objective functions. International Journal of Production Research 50, 4255–4270.

Sharma, M., Komninos, A., López-Ibáñez, M., Kazakov, D., 2019. Deep reinforcement learning based parameter control in differential evolution, in: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 709–717.

Song, W., Chen, X., Li, Q., Cao, Z., 2022. Flexible job-shop scheduling via graph neural network and deep reinforcement learning. IEEE Transactions on Industrial Informatics 19, 1600–1610.

Song, Y., Wu, Y., Guo, Y., Yan, R., Suganthan, P.N., Zhang, Y., Pedrycz, W., Das, S., Mallipeddi, R., Ajani, O.S., et al., 2024. Reinforcement learning-assisted evolutionary algorithm: A survey and research opportunities. Swarm and Evolutionary Computation 86, 101517.

Yang, S., Xu, Z., 2021. Intelligent scheduling and reconfiguration via deep reinforcement learning in smart manufacturing. International Journal of Production Research , 1–18.

Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P.S., Chi, X., 2020. Learning to dispatch for job shop scheduling via deep reinforcement learning, in: Advances in Neural Information Processing Systems, pp. 1621–1632.

Zhang, F., Mei, Y., Nguyen, S., Zhang, M., 2023a. Survey on genetic programming and machine learning techniques for heuristic design in job shop scheduling. IEEE Transactions on Evolutionary Computation .

Zhang, G., Gao, L., Shi, Y., 2011. An effective genetic algorithm for the flexible job-shop scheduling problem. Expert Systems with Applications 38, 3563–3573.

Zhang, J.D., He, Z., Chan, W.H., Chow, C.Y., 2023b. Deepmag: Deep reinforcement learning with multi-agent graphs for flexible job shop scheduling. Knowledge-Based Systems 259, 110083.

Zhou, T., Zhang, W., Niu, B., He, P., Yue, G., 2024. Parameter control framework for multiobjective evolutionary computation based on deep reinforcement learning. International Journal of Intelligent Systems 2024, 6740701.