Neural Decision Diagrams for Job Shop Scheduling

Eghonghon-aye Eigbe[1] , **Christoph Schmidl**[2] , **Bart De Schutter**[1] , **Nils Jansen**[3] , **Mitra Nasri**[4] , **Neil Yorke-Smith**[1]
[1]Delft University of Technology, [2]Radboud University, [3]Ruhr-University, Bochum, [4]Eindhoven University of Technology

Abstract

We develop a novel approach to job shop scheduling, by combining (deep) reinforcement learning (RL) with decision diagrams (DDs) such that portions of the diagram are created by a trained RL agent. This enables a combination of the strengths of both methodologies: the decision diagram can – based on good historical solutions – build better solutions faster, while the RL agent is able to improve the solution provided by its policy and possibly be provably optimal. We exploit the similarity of problem representation between RL and DDs to design a seamless handover between parts of the solution process. Empirical results show that our learned DD shows improvements over a standard DD approach to job shop scheduling.

1 Introduction

Combinatorial optimisation problems are ubiquitous in the real world and have received a lot of attention over the years from different research domains [19]. While traditional optimisation methods have been able to produce impressive solutions to many problems, recent advances in machine learning have propelled researchers to look harder at combining optimisation and machine learning methods.

The motivation for this is two-fold. First, as difficult as combinatorial optimisation problems are to solve, many real-life applications solve the same or similar problems over and over. Thus, learning to solve an entire class of problems holds much promise for industry. Secondly, many solution methods still depend on hand-crafted heuristics. Moreover, even within exact solvers, some solution steps are still performed heuristically. This also gives an additional opportunity to learn better heuristics or to automate the design of such heuristics.

The use of machine learning can be in learning to solve the problem directly as in [25, 20], learning to perform a particular step in the solution process of another algorithm as in [7], learning appropriate hyper-parameters and settings for an algorithm as in [17, 16], or using learning methods to make predictions that a solving algorithm takes as input [10]. Each of these approaches has its pros and cons depending on the problem specifics. For instance, where there is no uncertainty considered in the problem, we typically do not need to make predictions to serve as input. For more details, we point the reader to the survey by Zhang *et al.* [26].

We choose to go the route of integrating machine learning in the solution process of another solver, namely a decision diagram(DD). This kind of amalgamation of methods gives us better chances to gain from both methodologies directly. We do not learn a particular solving step as this still keeps the learning and solving portions of the algorithm separate. We instead, train an RL agent to solve the whole problem. With both the agent and the solver having the capability to solve the problem, they are able to correct for each other's shortcomings when combined.

In summary, our work closes multiple gaps, namely: (i) while Zhang *et al.* [25], Song *et al.* [20] already use reinforcement learning to solve job shop scheduling problems, we embed our agent in a solver such that we can still improve the solution provided by an agent and possibly be optimal, (ii) while Chalumeau *et al.* [7] already embed a reinforcement learning agent in an exact CP solver we give the agent much more freedom via our decision diagram structure so that we can get competitive solutions and finally, (iii) although Cappart *et al.* [5] also use reinforcement learning in a decision diagram, we focus on finding solutions as opposed to finding bounds.

The rest of the paper is organised as follows: Section 2 introduces preliminary background information, Section 3 discusses related work, and Section 4 provides the details of our solution approach. We perform empirical evaluations in Section 5 and Section 6 concludes the paper. Our code is available online[1].

2 Preliminaries

This section introduced some preliminary concepts, our terminology and notations on job-shop scheduling problems, decision diagrams, and reinforcement learning.

2.1 Job Shop Scheduling Problems

A job shop scheduling problem (JSSP) consists of a set of jobs $J = \{J_1, \ldots, J_n\}$ to be scheduled on a set of machines $M = \{\mu_1, \ldots, \mu_m\}$. Every job $J_i \in J$ has a set of operations $O = \{o_{i0}, \ldots, o_{ik}\}$ where operation o_{ij} is the jth operation

[1]https://github.com/Eghonghonaye/NDDLSTM

of the ith job and has processing time $P(o_{ij})$. Each operation is assigned to a machine.

Job shop scheduling is a well-known combinatorial optimisation problem [1] that is widely used in industry. Different versions of the JSSP exist depending on the particular industrial setup. It is known that most versions of the JSSP are NP-hard [9]. The solution to a JSSP is a schedule Ω, i.e., a sequence of operations. We focus on the objective of makespan minimisation because minimising the makespan correlates with many other objectives, such a minimising tardiness and flow time. As such, we believe the design choices made in our method can easily be tweaked to fit a different objective.

2.2 Decision Diagrams

Decision diagrams are a graphical structure capable of compactly representing the solution space of combinatorial optimisation problems [4]. A decision diagram (DD) is a directed acyclic graph $G = (V, E)$ in which every node $v \in V$ represents a state s_v and every edge $e \in E$ is a transition between states based on some variable or value assignment. In the realm of scheduling, a state is a partial schedule, while an edge e is a transition based on a scheduling decision.

A decision diagram contains two special states, namely the root state s_r and the terminal state s_t. At the root state s_r, no decisions have been made, and at the terminal state s_t, all decisions have been made, i.e., all operations have been scheduled. Thus, every path from $s_\mathrm{r} \to s_\mathrm{t}$ represents a complete schedule.

Decision diagrams can be modelled fully recursively and as a Markov Decision Process (MDP) [4]. As we combine our solver with a reinforcement learning (RL) agent, we design our solution such that both the DD and the RL setup use the same MDP formulation of the problem. This formulation will be discussed later in Section 4.2.

Decision diagrams for combinatorial optimisation problems are known to grow exponentially, especially when they encode all possible solutions to a problem, i.e., when they are *exact*. Bounded size approximations of decision diagrams have been developed as a means to manage exponential growth [8]. Specifically, *restricted* decision diagrams encode a subset of possible solutions, while *relaxed* decision diagrams encode a superset of possible solutions by allowing some infeasible paths to exist in the diagram [4].

2.3 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that sets up the learning task as a sequential decision-making process. The basic principle of RL is that an agent interacts with an environment by taking actions within that environment. The environment is Markovian such that every state-action pair provides sufficient information to transition to a new state. After every action, the agent gets feedback – in the form of rewards or penalties – and based on this feedback, the agent learns to take better actions within the environment [23].

Typically, the RL agent is controlled by some policy π, and the goal during training is to learn an optimal policy. This can be done by learning a policy directly or learning value functions in a state such that the optimal policy can be later retrieved from the learned value functions.

RL is attractive for scheduling problems because many of them have to do with deciding on sequences, and this ties in neatly with a sequential decision-making agent. We point the reader to [11] for a more comprehensive understanding of RL.

3 Related Work

The concept of embedding RL agents in algorithms for combinatorial optimisation problems has been studied in recent literature. For instance, [16] uses RL to control parameter settings for evolutionary algorithms, while [17] uses RL to select operators for adaptive large neighbourhood search. However, in both of these methods, the agent is neither directly trained to solve the problem nor fully embedded in the solution, but instead learns a heuristic used within the algorithm.

A challenge in using machine learning methods to solve combinatorial optimisation problems – either directly or as a part of a larger solver – is that we often require our models to be size-agnostic and permutation-invariant. Although there are methods to mitigate the worsening effect of job permutations on the makespan in the context of RL [18], they are mostly just applicable to policies represented as a feed-forward neural network. For this reason, many methods have turned towards *graph neural networks* (GNNs) as they are size- and permutation-invariant by design. These features enable a GNN to create fixed-size state embeddings out of graph structures that act as a state representation for an RL agent [15]. For more details on advances in this line, the recent survey by Cappart *et al.* [6] provides an overview of how GNNs have been employed in solving combinatorial optimisation problems. Although GNNs can capture the relationships and interactions between nodes in the graph, they may lack the ability to fully capture temporal dynamics and sequences in a job shop scheduling problem. Given that a job shop has to adhere to precedence constraints that dictate operation sequences for jobs, autoregressive models may be a better fit than GNNs. Autoregressive models, such as recurrent LSTMs [12], have already been applied successfully to the JSSP [13] and can encode the operations of each job. A set2set [22] network then uses these encodings to create a new equivariant representation. This leads to similar features of a GNN, mentioned before, with the ability to capture temporal dynamics.

With RL, especially where policy gradient methods are used, it is common to greedily sample from the trained policy by picking the highest probability action in every state. However, we know that for combinatorial optimisation problems, this is not always the best strategy [2, 14]. Bello *et al.* [2] show that allowing the trained agent to perform some kind of search in its solution space yields better performance and Iklassov *et al.* [13] show that sampling, i.e., repeating the inference process by directly sampling from the policy distribution and then picking the best sample, performs well for job-shop scheduling. However, sampling is not necessarily aware of the position of a solution in the solution space and does not directly try to improve samples. Embedding the agent in a solver like we do, allows the agent to be guided to sample solutions from more promising parts of the solution space.

Another interesting related work is [5], where, similar to our work, RL is combined with decision diagrams. Cappart *et* *al.* [5] train an RL agent to decide the variable ordering in a decision diagram. The connection between decision diagrams, dynamic programming, and RL – each of these three solution methods can be reasoned about as MDPs – is exploited in their algorithm. Apart from the difference in the learning task as discussed above, our work focuses on finding solutions while [5] focuses on finding bounds.

4 Proposed Solution Method

Our method works by progressively building solutions within a decision diagram where building a solution is treated as a sequential decision-making process. The first step in the process is to build one initial solution – also a path in the DD – and continually improve this solution by branching out of different nodes. Every time a branching decision is made, a trained sequential decision-making agent is triggered to build the rest of the path leading to a new solution. Figure 1 shows the solution process. We make branching decisions based on several known rules in the literature, such as best- and depth-first expansion.

4.1 Decision Diagram Construction

A decision diagram $G = (V, E)$ consists of nodes V and edges E. Nodes V represent states of the job shop and edges E represent the scheduling decisions that transition the shop from one state to another. In this paper, a scheduling decision is simply an operation to be scheduled.

A state s_v is directly represented as a feature vector with three entries (A, Φ, EST) where $A = \{A_m | \mu_m \in M\}$ represents the earliest availability of all machines, $\Phi = \{o_{ij} | J_i \in J\}$ represents the next ready operation for each job, and $EST = \{J_i^{\text{est}} | J_i \in J\}$ is the earliest start time for any unscheduled operation of a job. We define a function $H(\Omega)$ that returns the (partial) makespan of a schedule.

Transition. We transition from one state to the next by choosing an operation to be scheduled next. Every time, we transition from state s to s' by scheduling an operation o_{ab} on machine μ_k, we create a new state s' such that $A'_k = \max(A_k, J_a^{\text{est}}) + P(o_a)$, $o'_{ab} = o_{a(b+1)}$, and $J_a^{'\text{est}} = A'_k$. Whenever there are no more unscheduled operations in a state, we transition to the terminal state.

Algorithm 1 describes the steps of constructing the DD. In Lines 1-3 we initialise an empty schedule and an empty DD. In Line 5, we select a node to expand according to a node selection heuristic. We then hand over to the agent to continually choose actions in Lines 6-10. This portion is illustrated in Step 1 in Figure 1. If the agent ends up in a state that is a solution, infeasible[2] or dominated, we go back to the DD in Line 5 to select another point to continue expansion from. We repeat this until a stopping criterion is met where different stopping criteria can be specified such as runtime or memory limitations. This repeated solution building is shown in Steps 2 and 3 of Figure 1.

[2]While the actions are masked such that only feasible options are explored, some problems have constraints whose violations can only be assessed after a decision has been made e.g., problems with relative deadlines.

Algorithm 1: Decision Diagram Construction

```
   Input: Job Shop problem $S$, RL Policy $\pi$
   Result: Schedule $\Omega$
1  Create root and terminal nodes $r$,$t$
2  Create empty schedule $\Omega$
3  $L \leftarrow \{r\}$ /* Set of leaf nodes */
4  while not stopping criterion do
5      Select $v$ from $L$ /* Perform node
           selection */
6      while $v$ not terminal $\vee$ infeasible $\vee$ dominated do
7          $a \leftarrow \pi(s_v)$ /* Select action based
               on RL Policy */
8          Create new node $v'$ from $(v, a)$
9          Evaluate $v'$ /* Update start times,
               check dominance and
               feasibility */
10         $v \leftarrow v'$
11     if $v$ is terminal $\wedge H(\text{Path}(r, v)) < H(\Omega)$ then
12         $\Omega \leftarrow \text{Path}(r, v)$ /* Update $\Omega$ with
               path from root to $v$ */
13 return $\Omega$
```

4.2 Reinforcement Learning Agent

We train an RL agent to solve the problem independently, i.e., without the support of an external solver. The intuition is that an agent can learn a good sequencing policy, which produces solutions that can be quickly improved using the DD. Setting up such an agent requires a few definitions, which are stated below.

State. States carry the same representation and meaning between the DD and the RL agent, e.g., s_0, is both the root state of the DD and the initial state of the RL agent. This makes for a smooth interaction between the agent and solver.

Action. We define an action as an operation selection. This also carries the same meaning and representation as an edge in the DD. Therefore, the action space is discrete and contains indices of operations as integer values.

Reward. We keep the reward simple. For every transition from state s to s', an agent gets a reward that is the difference in makespans of the partial solutions where a makespan of a partial solution is the maximum machine availability i.e., $r = \max(A) - \max(A')$.

Policy. We use the same policy setup as [13], which takes as input the instance description x and the *internal state* during the problem resolution s_τ and outputs a policy $\pi(x, s_\tau)$, which is a distribution over the next pending operations at the current decision step τ. The underlying policy architecture is a combination of an LSTM [12] that encodes the operations on each job recurrently, a set2set [22] network to achieve equivariance for job combinations, and actor-critic networks to finally compute distribution over actions. The LSTM can effectively capture temporal sequences between operations, while the set2set network can prevent makespan degradation that may be caused by job permutations [18]. The policy is trained using the Reinforce algorithm [24]. We make use of

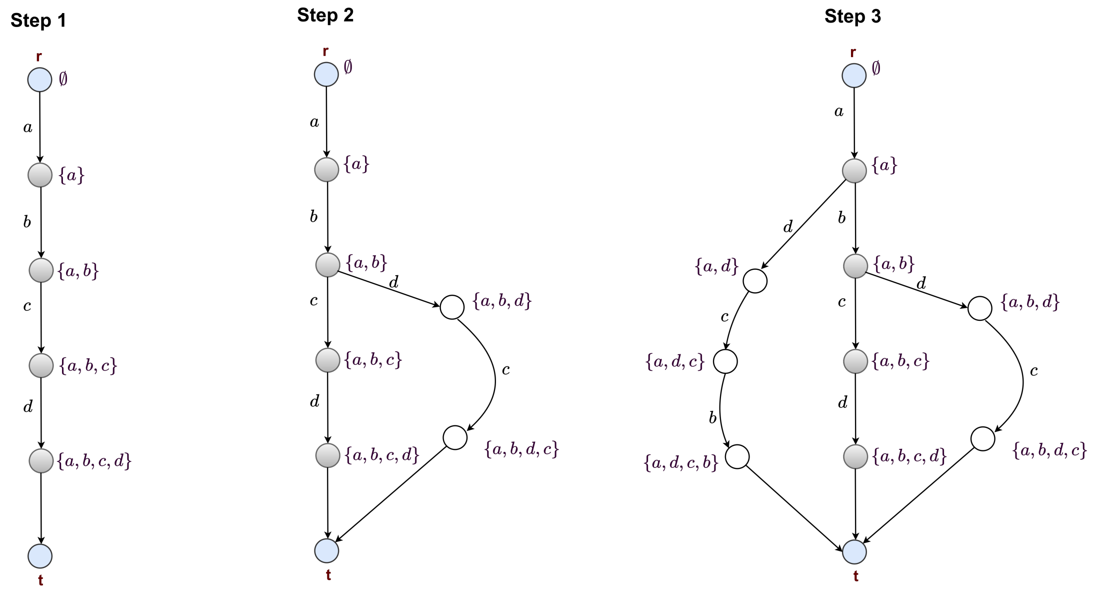

Figure 1: DD solving process

the policy implementation provided by [13], which is also available online[3].

Action masking In our approach, we use action masking during training and testing. In every state, infeasible actions are masked out according to the constraints of the problem. After training, when the agent is used within the DD, we further mask out actions that have already been taken in the past to prevent duplication of paths in the diagram.

5 Results

Benchmark. We evaluate our method on the popular Taillard instances introduced in [21]. The benchmark consists of problem instances with the number of machines in the set {15,20}, the number of jobs in the set {15,20,30,50,100}, and the processing times uniformly distributed in the interval [1,99]. The benchmark contains 80 problem instances and we train our RL agent on 10 randomly selected instances and learn only one policy across all instance sizes. We use the full set of 80 instances in testing.

Baselines. We compare our method (referred to as DD-RL) with two classes of baselines. The first class is other RL methods to solve this problem. We compare our work with greedily sampling our trained agent (referred to as RL), the results of [25] (referred to as Zhang *et al.*) and the results of [13] (referred to as Iklassov *et al.*). The second class of baselines is the decision diagram alone using the same scheme as in Algorithm 1 but with the RL agent replaced by a simple depth first search. We refer to this as DD in the results.

We compare all methods to the best known solutions in the literature (referred to as UB) and compute optimality gaps as the distance in percentage from these solutions.

[3]https://github.com/Optimization-and-Machine-Learning-Lab/Job-Shop/tree/main_nips

Note that we only compare with the base model results in [13] and not the best results presented in the paper. The best results were achieved via curriculum learning [3] and always perform better across all instances. However we believe comparing with these results would not be fair as it takes significantly more training resources to achieve the curriculum learning results. In our future work, we plan to deploy more strategies that allow us to use a curriculum learning trained agent along with the DD on smaller resources.

Configurations. All experiments were run on a machine with an AMD Ryzen 9 7950X3D (5.7GHz), 64GB of RAM, and a single Nvidia Geforce RTX 4090. The operating system is Ubuntu 22.04 LTS. We train our model for 20 000 iterations using a batch size of 8 and a constant learning rate of 10^{-4}. All DD solutions are given a stopping criterion of 5 minutes run time.

Discussion. Table 1 summarises our results. Table 1 shows the achieved makespans and their optimality gaps with the best performing solution highlighted in bold. Overall, the DD-RL approach outperforms its competitors for all benchmark instances.

For both setups with 15 and 20 machines, DD-RL yields better makespans than other methods. We observe up to 5% improvement in optimality gap for DD-RL compared to Iklassov *et al.* For setups with 30 and 50 jobs, the DD-RL approach is still performing better than the rest, but we can observe that the approach by Iklassov *et al.* dips in performance and is worse than the pure RL approach. This could be attributed to some forgetting as we train our base model for 20 000 iterations while Iklassov *et al.* trains the same model for 45 000 iterations.

6 Conclusion

In this paper, we explore the combination of decision diagrams and reinforcement learning to solve job-shop scheduling prob-

M/Cs	Jobs		RL	DD	DD-RL	Zhang *et al.*	Iklassov *et al.*	UB
15	15	Obj.	1538.30	2016.4	**1404.60**	1547.4	1413.0	1228.90
		Gap	*25.24%*	*64.07%*	***14.35%***	*25.96%*	*14.98%*	
	20	Obj.	1773.50	2425.7	**1635.60**	1774.7	1692.1	1364.80
		Gap	*29.38%*	*77.78%*	***19.86%***	*30.03%*	*23.97%*	
	30	Obj.	2255.90	3311.1	**2171.70**	2378.8	2275.3	1790.20
		Gap	*26.16%*	*85.36%*	***21.43%***	*33.0%*	*27.19%*	
	50	Obj.	3292.30	4863.0	**3195.10**	3393.8	3346.7	2773.80
		Gap	*18.76%*	*75.38%*	***15.22%***	*22.38%*	*20.69%*	
20	20	Obj.	2068.30	3008.2	**1932.40**	2128.1	1958.4	1617.50
		Gap	*27.85%*	*86.02%*	***19.49%***	*31.61%*	*21.11%*	
	30	Obj.	2569.20	3873.7	**2447.90**	2603.9	2540.2	1948.50
		Gap	*31.89%*	*98.83%*	***25.66%***	*33.62%*	*30.4%*	
	50	Obj.	3439.30	5654.8	**3363.10**	3593.9	3518.0	2843.90
		Gap	*20.97%*	*99.02%*	***18.27%***	*26.51%*	*23.73%*	
	100	Obj.	5949.40	9582.0	**5900.70**	6097.6	6145.5	5365.70
		Gap	*10.87%*	*78.61%*	***9.96%***	*13.61%*	*14.52%*	

Table 1: Results on Taillard Instances.

lems. We make use of an existing RL model in the literature and show via an empirical study that the combination of DD and RL into one solver performs better than either solution method on its own.

However, we deploy a relatively simple training for our agent and results from more robust agents in the literature still outperform those presented here. We hypothesize that the performance of RL agents trained with extensive resources and more complex schemes can be replicated by simpler training and better inference such as the integration with a solver presented in this paper. Should this be the case, RL could become more usable in industrial settings where extensive training resources may not be available. It will be fruitful for future research to explore and possibly validate this hypothesis.

Acknowledgments

This research has been partially funded by the NWO project 17931 (SAM-FMS) in the MasCot program and the ERC Starting Grant 101077178 (DEUCE).

References

[1] David Applegate and William Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on computing*, 3(2):149–156, 1991.

[2] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[4] David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[5] Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451, 2019.

[6] Quentin Cappart, Didier Chételat, Elias B Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

[7] Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 392–409. Springer, 2021.

[8] Andre A Cire and Willem-Jan Van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, 2013.

[9] Jr EG Coffman. Computer and job-shop scheduling theory, john wiley&sons. *Inc. New York*, 1976.

[10] Nikolaos Efthymiou and Neil Yorke-Smith. Predicting the optimal period for cyclic hoist scheduling problems. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 238–253. Springer, 2023.

[11] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] Zangir Iklassov, Dmitrii Medvedev, Ruben Solozabal Ochoa De Retana, and Martin Takac. On the study of curriculum learning for inferring dispatching policies on the job shop scheduling. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI*, pages 5350–5358, 2023.

[14] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[15] Junyoung Park, Jaehyeong Chun, Sang Hun Kim, Youngkook Kim, and Jinkyoo Park. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59(11):3360–3377, 2021.

[16] Robbert Reijnen, Yingqian Zhang, Zaharah Bukhsh, and Mateusz Guzek. Deep reinforcement learning for adaptive parameter control in differential evolution for multi-objective optimization. In *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 804–811. IEEE, 2022.

[17] Robbert Reijnen, Yingqian Zhang, Hoong Chuin Lau, and Zaharah Bukhsh. Operator selection in adaptive large neighborhood search using deep reinforcement learning. *arXiv preprint arXiv:2211.00759*, 2022.

[18] Christoph Schmidl, Thiago D. Simão, and Nils Jansen. A supervised learning approach to robust reinforcement learning for job shop scheduling. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Proceedings of the 16th International Conference on Agents and Artificial Intelligence, ICAART 2024, Volume 3, Rome, Italy, February 24-26, 2024*, pages 1324–1335. SCITEPRESS, 2024.

[19] Alexander Schrijver et al. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer, 2003.

[20] Wen Song, Xinyang Chen, Qiqiang Li, and Zhiguang Cao. Flexible job-shop scheduling via graph neural network and deep reinforcement learning. *IEEE Transactions on Industrial Informatics*, 19(2):1600–1610, 2022.

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

[22] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

[23] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[24] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[25] Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1621–1632, 2020.

[26] Cong Zhang, Yaoxin Wu, Yining Ma, Wen Song, Zhang Le, Zhiguang Cao, and Jie Zhang. A review on learning to solve combinatorial optimisation problems in manufacturing. *IET Collaborative Intelligent Manufacturing*, 5(1):e12072, 2023.