

Article

Applying Decision Transformers to Enhance Neural Local Search on the Job Shop Scheduling Problem

Constantin Waubert de Puiseau ^{*}, Fabian Wolz, Merlin Montag , Jannik Peters , Hasan Tercan 
and Tobias Meisen 

Institute for Technologies and Management of Digital Transformation, Lise-Meitner-Strasse 27,
42119 Wuppertal, Germany; merlin.montag@uni-wuppertal.de (M.M.); jpeters@uni-wuppertal.de (J.P.);
tercan@uni-wuppertal.de (H.T.); meisen@uni-wuppertal.de (T.M.)

^{*} Correspondence: waubert@uni-wuppertal.de

Abstract: *Background:* The job shop scheduling problem (JSSP) and its solution algorithms have been of enduring interest in both academia and industry for decades. In recent years, machine learning (ML) has been playing an increasingly important role in advancing existing solutions and building new heuristic solutions for the JSSP, aiming to find better solutions in shorter computation times. *Methods:* In this study, we built on top of a state-of-the-art deep reinforcement learning (DRL) agent, called Neural Local Search (NLS), which can efficiently and effectively control a large local neighborhood search on the JSSP. In particular, we developed a method for training the decision transformer (DT) algorithm on search trajectories taken by a trained NLS agent to further improve upon the learned decision-making sequences. *Results:* Our experiments showed that the DT successfully learns local search strategies that are different and, in many cases, more effective than those of the NLS agent itself. In terms of the tradeoff between solution quality and acceptable computational time needed for the search, the DT is particularly superior in application scenarios where longer computational times are acceptable. In this case, it makes up for the longer inference times required per search step, which are caused by the larger neural network architecture, through better quality decisions per step. *Conclusions:* Therefore, the DT achieves state-of-the-art results for solving the JSSP with ML-enhanced local search.

Keywords: neural combinatorial optimization; job shop scheduling; decision transformers; deep reinforcement learning; neural networks; operations research; heuristics



Academic Editor: Demos T. Tsahalidis

Received: 31 January 2025

Revised: 17 February 2025

Accepted: 21 February 2025

Published: 1 March 2025

Citation: Waubert de Puiseau, C.; Wolz, F.; Montag, M.; Peters, J.; Tercan, H.; Meisen, T. Applying Decision Transformers to Enhance Neural Local Search on the Job Shop Scheduling Problem. *AI* **2025**, *6*, 48. <https://doi.org/10.3390/ai6030048>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Effective and efficient scheduling presents an ongoing challenge that is critical for success in many sectors, from manufacturing [1,2] to cloud computation [3]. Scheduling, broadly, deals with the allocation of resources to tasks over time with the goal of optimizing a given objective [4]. The job shop scheduling problem (JSSP) (Section 2.1) is an abstracted combinatorial scheduling problem that underlies many real-world problems and has been extensively studied in the literature. To this day, new solution methods for scheduling problems are developed and tested on the JSSP due to its interesting properties and the availability of popular public datasets that allow for rigorous benchmarking [5–7]. Much of the research aims at finding solutions with smaller makespans with less computational effort.

In recent years, machine learning has emerged as a promising technique for new solution methods for scheduling problems by enhancing or replacing heuristic decisions in existing dispatching rules [8–12], metaheuristics and search heuristics [13–16], and optimal solvers [17] to varying degrees. A notable advancement in this area is the integration of

deep reinforcement learning (DRL) with local search algorithms. For instance, Falkner et al. [18] proposed an approach wherein a DRL agent is trained to learn three critical components of a local search algorithm: solution acceptance, neighborhood operators, and perturbation decisions. This ML-enhanced search significantly outperforms state-of-the-art search methods on the JSSP because it can learn to adjust its search strategy according to the specific JSSP instance it is solving. Meanwhile, transformer architectures [19] are nowadays deployed at scale and achieve breakthrough results in various domains, most publicly known in natural language processing [20], by capturing and processing important features in sequential data. Aiming at harnessing the strength of sequential data processing of transformers, decision transformers (DTs) have recently been introduced for learning strategies in sequential decision-making processes from simulation data [21].

In this work, we integrated the DT with the NLS approach by learning from data generated during the local search process of trained NLS models. Using the DT in combination with NLS comes with two potential benefits over the original NLS algorithm: first, by considering a history of past actions, the DT can learn how previous decisions have influenced the solution of the current problem instance and adjust its search strategy by reacting to inherent properties of the instance that become visible through interaction. Second, the DT is trained to align its actions with a manually set reward prior. This can lead to performance improvements over teacher algorithms—in our case, the already highly competitive NLS models. The main contributions of this paper are as follows:

- The introduction of a decision transformer approach to boost the performance of a learned local search heuristic on the JSSP.
- An experimental analysis of both the performance and the learned behavior of the model with respect to the teacher models.
- A resulting ML-method that, depending on the computational time constraints, leads to state-of-the-art results for solving the JSSP through ML-based local search.

The remainder of this work is structured as follows: first, the JSSP and general solution approaches are introduced in Section 2. Here, we also provide a brief introduction to DRL, DTs, and the neural neighborhood search algorithm that jointly build the basis of our work. Next, related work is discussed in Section 3 before our methods are detailed in Section 4. We present the achieved results in Section 5, and a detailed comparison between the teacher and DT in terms of practical implications and learned behavior is provided in Section 6. Section 7 gives a conclusion and an outlook for future work.

2. Preliminaries

2.1. Job Shop Scheduling Problems

The classical, abstract version of the JSSP considered in this paper deals with the allocation of a set of i jobs J_i , each comprising j operations O_j with processing times P_{ij} , to m machines M_m over time, with the objective of minimizing the maximum completion time C_{max} —that is, the duration from the initiation of the first operation to the completion of the last one, commonly referred to as the makespan. Operations within each job underlie precedence and no-overlap constraints, such that any job visits each machine exactly once in a predefined order. Accordingly, in this setup, the number of operations j and the number of machines m are equal. In addition, each machine may only process one job at a time. In terms of notation, we refer to problem instances with j jobs and m machines as instances of size $j \times m$. For example, a 20×15 problem instance consists of 20 jobs, each with 15 operations on 15 machines in total. The described JSSP problem is NP-hard, making enumerative algorithms for finding optimal solutions to large instance sizes impractical in real-world applications.

2.2. Solution Methods for Job Shop Scheduling Problems

Solutions algorithms for the JSSP have been under active development and research for decades in the operations research domain. These algorithms are designed to achieve a balance among minimizing the objective (e.g., makespan), ensuring rapid execution times, and requiring minimal implementation effort. Provenly optimal solutions can be found by means of constraint programming solvers such as the CP-SAT solver by Google OR-Tools [22]. However, such solvers become impractical for large JSSP instances due to the exponential growth of the solution space that must be explored.

A common category of solution methods in practice is priority dispatching rules (PDRs), in which dispatching and ordering decisions are governed by job prioritization rules such as “shortest processing time first”. A representative collection of such rules has been studied by Haupt et al. [23]. Although PDRs are straightforward to understand and implement in industrial settings, they often fail to produce competitive makespans when compared to other solution methods. Another solution category comprises a variety of (meta-) heuristics, such as the shifting bottleneck heuristic [24], genetic algorithms [25], and heuristic search algorithms [26,27]. Depending on the problem size and considered constraints, tailored metaheuristics have long been the de-facto state of the art. With the advent of more powerful machine learning algorithms, particularly deep learning techniques, the aforementioned solution methods have increasingly been either enhanced or partially replaced by data-driven approaches in recent years. Further details and significant developments related to this work are discussed in Section 3, “Related Work”.

2.3. Deep Reinforcement Learning

DRL is a machine learning paradigm, in which an agent learns a parameterized policy, approximated by a deep neural network through interaction with an environment. In each timestep t , the agent receives a representation S_t of the state of the environment, takes an action A_t which alters the state of the environment, and receives a feedback signal R_t , called reward. The training process aims at maximizing the cumulated reward across an episode, $\sum_{t_0}^{t_{final}} r_t$, i.e., from the first interaction at t_0 to the final interaction t_{final} . A common way to learn a policy is using the Q-Learning algorithm. Q-values, or state–action–values, are assigned to each state–action pair and represent the total cumulative reward that can be achieved by taking action A_t in state S_t and subsequently following the learned policy. Once Q-values have been learned from experience, the optimal policy is derived by iteratively estimating the Q-values for all possible actions in each state of the episode and selecting the action that corresponds to the highest predicted Q-value. DRL can be applied to any sequential decision-making process that adheres to the Markov property, meaning that the state of the environment at any given time is independent of prior states [28].

2.4. Decision Transformers

The DT is an architecture for offline reinforcement learning, in which Markov decision processes are abstracted as sequence-modeling problems [21]. The DT has already been successfully applied in various domains such as active object detection in robotics [29], recommender systems [30], and chip placement optimization [31]. In traditional RL, models output (or evaluate) actions A given a representation of the current problem state S and receive a reward r in response. Hence, these sequences may be represented as lists of collected state–action–reward tuples, $[\{S_0, A_0, r_0\}, \{S_1, A_1, r_1\}, \dots, \{S_{final}, A_{final}, r_{final}\}]$. In contrast, DTs are trained to output actions by considering as input the sequence of the last K states, actions, and the corresponding return-to-go values R . In this context, K is referred to as the context length. The return-to-go at a given iteration step s , denoted as R_s , is defined as the sum of the remaining rewards: $R_s = r_s + r_{s+1} + \dots + r_{S-1} + r_S$. At each

time step s , the DT uses the sequence of past states, actions, and returns-to-go, along with the current state S_s and return-to-go R_s . This is formally expressed as follows:

$$A_s = DT(R_{s-k}, S_{s-k}, A_{s-k}, \dots, R_s, S_s) \quad (1)$$

In other words, the DT learns to generate action sequences that aim to achieve the manually set target R (i.e., the prior) by minimizing the difference between R and the cumulated sum of rewards, instead of minimizing the cumulated sum of rewards directly. The underlying concept is depicted in Figure 1.

The mapping function of the DT is approximated with a transformer architecture, originally designed for processing sequential data in text format [19]. In the case of DTs, instead of embedding words into tokens, we embed returns-to-go, states, and actions into a fixed vector space. For more details on the DT, readers are referred to the work by Chen et al. [21]. We give details of our own implementation in Section 4.

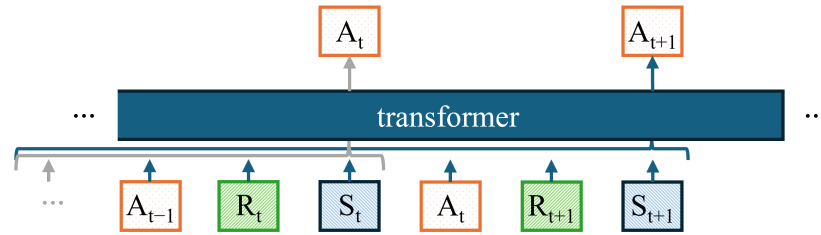


Figure 1. Depiction of the Decision Transformer and its inputs, adapted from [21].

2.5. Neural Local Search

NLS is a recent approach to control a local neighborhood search heuristic on the JSSP with a DRL agent [18]. The underlying local search (LS) heuristic first constructs an initial solution and then takes iterative steps to improve upon it. In each iteration, the heuristic

- Accepts or declines the solution of the last iteration,
- Chooses a new neighborhood operation that defines the next neighborhood, i.e., the set of solutions to integrate in the search,
- Or chooses a perturbation operator to jump to a new area in the search space in which to continue the search.

Falkner et al. [18] translated the heuristic decisions into actions that may be taken by a DRL agent, experimenting with three different action spaces that are defined by the following three sets of actions:

1. Acceptance decisions: the decision of whether the last LS step is accepted or not:

$$A_A := \{0, 1\}$$

2. Acceptance–Neighborhood decision: a tuple representing the above acceptance decision and the choice between four different neighborhood operations in the set Φ :

$$A_{AN} := \{0, 1\} \times \Phi$$

3. Acceptance–Neighborhood–Perturbation decision: a tuple representing the acceptance and neighborhood decisions plus a perturbation decision from the perturbation operator set Ψ :

$$A_{ANP} := \{0, 1\} \times \{\Phi \cup \Psi\}$$

The neighborhood operators comprise the Critical Time (CT) operator, which swaps adjacent nodes in critical blocks [32]; the Critical End Time (CET) [33] operator, which swaps

nodes at the start or end of a critical block; the Extended Critical End Time (ECET) [34] operator, which swaps nodes at both the start and end of a critical block; and the Critical End Improvement (CEI) [35] operator, which shifts a node within a critical block to a new position within the same block.

All actions are taken based on states that are derived from a learned representation consisting of the current problem instance to be solved and its current solution. Specifically, the authors employ an encoder–decoder architecture depicted in Figure 2. The encoder generates a representation of the problem instance and its current solution using a graph neural network, which is then enriched with the following explicit state features: the current makespan, the best makespan found so far, the last acceptance decision, the last applied operator, the current time step s , the number of consecutive steps without improvement, and the number of perturbations applied so far. Utilizing this enriched representation, the decoder computes Q-values that guide the selection of the next action. These output Q-values are learned with Double Deep Q-Learning [36] aiming to maximize the total improvement in makespan between the initial and final solution across the episode.

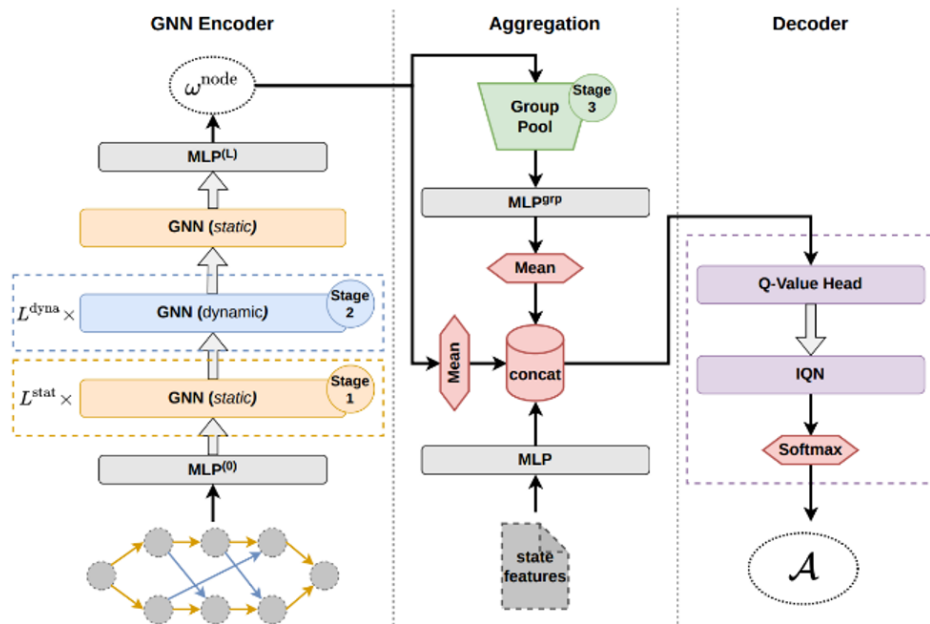


Figure 2. Neural network architecture of the NLS approach, reprinted from [18].

To this end, Falkner et al. [18] proposed a dense clipped reward that returns the makespan difference before and after the local search step induced by the action as in Equation (2). Note that notation-wise, we use the search step s for the local search step instead of the time step t common in the DRL notation.

$$r_s = \max(m_{s-1} - m_s, 0) \quad (2)$$

3. Related Work

3.1. Learned Construction Heuristics

A plethora of deep learning-based solution methods for the JSSP have been proposed in recent years [11,37–42], many of which serve as learned construction heuristics. Construction heuristics iteratively generate a solution by starting with an empty schedule and, in each iteration, determining the next unscheduled operation of a job to be added to the schedule. Learned construction heuristics have shown very promising results for the JSSP, showcasing that deep learning models are capable of capturing the structure of the JSSP effectively. In this study, we refrain from applying the DT to construction heuristics for

two reasons: firstly, although they perform well on the vanilla JSSP, the transfer to scheduling scenarios with additional constraints remains an open problem, where search heuristics are still dominant [14,41]. Secondly, one of the advantages of the DT is that it can effectively take past search actions on the same instance into consideration. This strength cannot be leveraged for construction heuristics, in which all the relevant information of past actions is summarized in the partially solved schedule, which cannot be altered retrospectively.

3.2. Learned Heuristic Search

In contrast to construction heuristics, in learned heuristic search, certain decision rules of search algorithms are replaced by neural network inferences. Typically, the search algorithms aim at improving an initial solution. Though the reported results on the JSSP are slightly worse than those of learned construction heuristics, learned heuristic search algorithms are more easily adaptable to problems with more real-world constraints and objectives because their action spaces remain constant [14]. This property makes them worth investigating, considering that most real-world use-cases are subject to multiple constraints. We, therefore, base our work on a learned local search heuristic.

A well-known representative of this category is local rewriting, in which agents iteratively decide on the local region and rewriting strategy for that region [16]. Another noteworthy approach was demonstrated by Reijnen et al., who trained an agent to control parameters of an evolutionary algorithm on a multi-objective flexible JSSP [14]. In this study, we build upon the most recent state-of-the-art approach by Falkner et al. [18], in which an agent controls acceptance, neighborhood, and perturbation decisions, as described earlier in Section 2.

3.3. Imitation Learning on the JSSP

In imitation learning scenarios, “student”-models learn to imitate the behavior in the sequential decision-making of a “teacher” in a supervised manner by following labels generated by the teacher instead of rewards. In the JSSP, teacher algorithms can range from dispatching rules to exact algorithms. The goal of this method is to imitate the behavior of the teacher and then either surpass it (e.g., a dispatching rule) or achieve faster computational inference times (e.g., compared to an exact algorithm). For example, Ingimundardottir and Runarsson [43] trained a linear machine learning model on data generated with dispatching rules. Rinciog et al. [44], for a related scheduling problem, pre-trained a neural network on the earliest-due-date dispatching rule before fine-tuning it with DRL. Although these approaches showed some success of imitation learning, the results are far from competitive in terms of the absolute results compared to other algorithms due to the limitation of their teacher algorithms. In contrast, Lee and Kim [45] trained a graph neural network on optimal solutions but also did not reach a competitive performance. Hypothesizing that optimal solutions provide noisy data from which it is difficult to learn, Corsini et al. [38] and Pirnay et al. [39] have instead very recently suggested to train a student model on its own most successful solutions generated during sampling. All imitation learning approaches mentioned here are either construction heuristics or online dispatchers. To the best of our knowledge, our study, by training a DT, is the first to address imitation learning for a learned heuristic search method.

4. Methods

In the following text, we describe how the DT and NLS are conceptually combined into one trainable algorithm for the JSSP and which specific design choices were taken. The descriptions of our method are structured along the three necessary steps “Dataset Generation”, “DT-Training”, and “DT-Testing”, which are conceptually visualized in Figure 3.

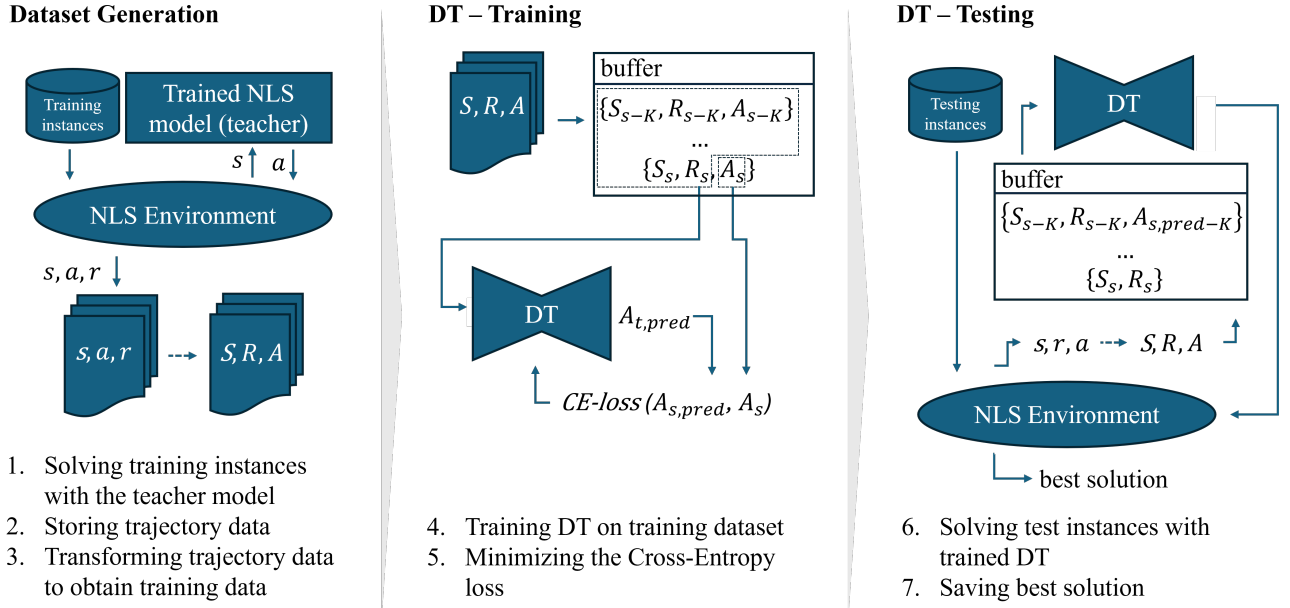


Figure 3. Schematic depiction of the three steps for the DT development.

4.1. Dataset Generation

Training Instances To train the DTs in a supervised manner, we must create suitable labeled datasets. To this end, we randomly generated 1000 problem instances for each problem size in the Taillard benchmark dataset according to the same reported specifications [5]. We then collected the state–action–reward tuples from local search trajectories on these instances with 100 and 200 search iterations generated by the interaction of the teacher NLS models with the NLS environment. The values 100 and 200 are in the range of iteration steps investigated in the original publication [18]. The collected information formed preliminary raw labeled datasets.

Teacher Models The teacher models are pretrained NLS checkpoint models, available online in the original repository [46] for each problem size. We performed this such that the DTs can learn from the successful learned solution strategies of the teachers. Note that the input to the DT, unlike that for the teacher models, is not only the state but also the return-to-go. We, therefore, expect the DT to be able to surpass the teacher models' performance during inference by giving a suitable (challenging) return-to-go prior. A small value would indicate that the DT should act to achieve a small makespan, while a large value should have the opposite effect.

Within the original NLS models that are suitable as teachers, specific action sets were most successful for different problem sizes. For example, action set A_A showed the best performance for 15×15 and A_{ANP} for 30×20 problems (cf. the original results of Falkner et al. [18]). Aiming for the best teacher model performances for the dataset creation, we used the best respective reported action sets for each problem size to generate training data. There are two exceptions to this rule: problem sizes 15×15 and 20×20 . In these two cases, A_A performs best, but preliminary experiments showed that the NLS models had learned to take the same action (“accept”) exclusively, which resulted in DTs that simply overfitted on the training data and returned the exact same results as their teacher models. The same phenomenon of overfitting to taking one action was observed in the 15×15 A_{AN} teacher model. For this reason, we resorted to the action sets in Table 1 for the teacher models per problem size throughout this study.

NLS Environments The NLS environments are closely adapted from [18] and implement the corresponding action dynamics and evaluations. As previously mentioned,

we performed local search for 100 and for 200 interactions with these environments and recorded the state–action–reward tuples to create raw labeled datasets. As in the original NLS publication [18], the initial solutions needed for NLS were obtained from the FDD/MWKR composite dispatching rule introduced by Sels et al. [47], combining information on the flow due date and remaining work of each job. The local searches result in two raw datasets per problem size, with 100,000 and 200,000 data points, respectively:

$$\text{dataset_raw} = \left[\underbrace{\{s_0, a_0, r_0\}}_{\text{one raw datapoint}}^0, \dots, \{s_s, a_s, r_s\}^i \right] \quad (3)$$

where $s \in [100, 200]$ for the number of iterations and $i \in [0, \dots, 1000]$ for all training instances.

The returns-to-go required for DT training are obtained by traversing the training data backwards and recursively adding the rewards within a local search sequence, such that the returns-to-go $R(s)$ per step become $R_s = \sum_{t=s}^i r_t$ and the final datasets have the following format:

$$\text{dataset_raw} = \left[\underbrace{\{s_0, a_0, R_0\}}_{\text{one final datapoint}}^0, \dots, \{s_s, a_s, R_s\}^i \right] \quad (4)$$

Table 1. Action set used in this paper per respective problem size. Used action sets are marked with “X”.

	15 × 15	20 × 15	20 × 20	30 × 15	30 × 20	50 × 15	50 × 20	100 × 20
A _A	-	-	-	-	-	-	-	-
A _{AN}	-	X	X	X	-	-	-	-
A _{ANP}	X	-	-		X	X	X	X

4.2. DT-Training

Training Procedure The DT model was trained for 500 epochs to minimize the categorical cross entropy loss between the predicted action $a_{s,pred}$ and the label action a_s (cf. Figure 3). Preliminary experiments with varying hyperparameters on the 15 × 15 JSSP indicated that a context length K between 50 and 200 was most effective. To balance effectiveness and model size, $K = 50$ was chosen. The Adam optimizer [48] was used in combination with cosine learning-rate-decay to smoothly reduce the overall learning rate, with periodical increases back to the initial value of 6×10^{-2} . Unlike the NLS model, the DT generates an action based on the last K state–action–reward tuples. Given that the dataset contains only single tuples, a buffer is used that returns the last K tuples depending on the current iteration step. For the iteration steps that are smaller than the context length K , the remaining tokens are zero-padded and masked within the self-attention mechanism. We tested our model every 33 epochs on 30 JSSP instances of the respective size, generated with a different random seed than the training and evaluation instances, and saved the weights whenever the achieved test makespan improved.

Model Architecture and Hyperparameters Our architecture combines components of the original NLS model and our DT implementation. We used the learned latent space representation generated by the aggregator component in the NLS architecture as state embeddings. The original pre-trained checkpoint model weights, which were also used during data generation, were taken from [46]. Note that both the trained model weights and the dataset bias the DT towards the teacher model strategy. This is a deliberate

choice to speed up training and converge towards a known successful strategy before improving upon it. Our DT transformer implementation is based on the minGPT model by Karpathy [49], from which most hyperparameters were adopted (see Table 2). The DT transformer replaces the decoder part of the original models to generate the action distribution in the output layer, as shown in Figure 4. State embeddings, actions, and rewards are all projected to the same embedding dimension of the tokens through linear layers. The resulting sequence of tokens is then fed into the first multi-head attention layer of the transformer. In contrast to the original implementation for textual data, our positional embedding is not created in relation to the block size but considers the maximum trajectory length of the problem, i.e., the number of iteration steps.

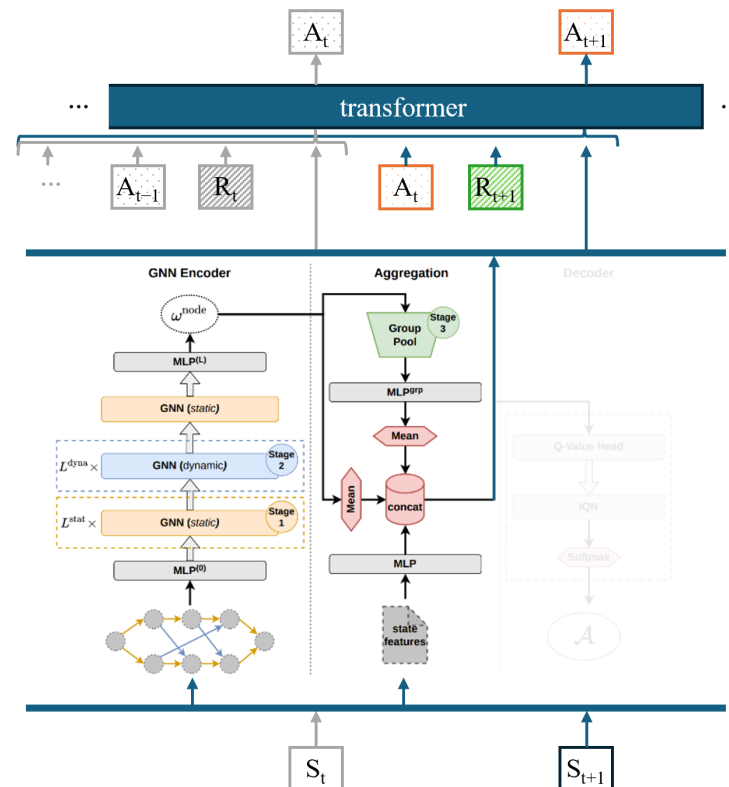


Figure 4. Schematic view of the integration of the learned state space representation.

Table 2. Hyperparameters of the minGPT model of the DT.

Hyperparameter/Design	Value
Number of layers	6
Number of attention heads	8
Embedding dimension	128
Batch size	512
Context length K	50
Nonlinearity	GeLU
Dropout	0.1
Adam betas	(0.9, 0.95)
Grad norm clip	1.0
Weight decay	0.1
Learning rate decay	Cosine decay

4.3. DT-Testing

Augmented Return-to-Go As described earlier, the original NLS environment returns the clipped relative makespan improvement before and after the action is executed as a reward signal, starting at step zero with an initially created schedule.

This presents a challenge during inference with the DT that requires a return-to-go value as input. Ideally, in step zero, the return-to-go R_0 would be the difference between the makespan of the initial schedule m_{init} (obtained from the FDD/MWKR dispatching rule) and the makespan of the optimal solution $m_{optimal}$: $R_0 = m_{init} - m_{optimal}$. However, $m_{optimal}$ is usually unknown prior to solving it. Therefore, instead, we calculate a lower bound makespan m_{lb} , defined as the maximum sum of processing times p of operations O_m that require the same machine:

$$m_{lb} = \max_{\text{machine} \in M} \sum_{o \in O_m} p(o) \quad (5)$$

Note that $m_{lb} \leq m_{optimal}$ in all cases since the m_{lb} solution practically ignores precedence constraints within jobs. We then define our initial return-to-go as $R_0 = m_{init} - m_{lb}$ and reduce it in each step by the given reward. Since this R_0 is an optimistic approximation, it forces the DT to take the best possible actions.

5. Numerical Results

5.1. Results on Taillard Benchmark

Table 3 shows the results of NLS models with different action sets and the DTs on the Taillard benchmark dataset [5], which consists of ten instances for each of the reported instance sizes. In the table, DT₁₀₀ represents those DTs that were trained on the datasets in which only 100 search steps were performed by NLS. Nevertheless, all presented results were generated using 200 local search steps of the DT. The best results per column are printed in bold. Note that we re-printed the results by Falkner et al. [18] on those action sets that were not used as teacher models in gray.

Table 3. Results (makespans and optimality gaps) on Taillard benchmark instances. The best results per column are bold. Results of NLS agents that were not used as teachers are printed in gray.

		15 × 15	20 × 15	20 × 20	30 × 15	30 × 20	50 × 15	50 × 20	100 × 20	Average
NLS _A	gap	7.74%	12.16%	11.54%	14.33%	19.42%	18.00%	11.22%	5.89%	12.54%
	makespan	1324.0	1530.7	1804.0	2043.1	2267.0	3273.0	3163.0	5682.0	2635.85
NLS _{AN}	opt. gap	8.72%	11.39%	11.67%	14.31%	19.57%	18.29%	11.15%	5.84%	12.62%
	makespan	1336.0	1520.2	1806.0	2042.7	2270.0	3281.0	3161.0	5679.0	2636.99
NLS _{ANP}	opt. gap	10.42%	15.63%	13.83%	13.82%	19.10%	10.62%	10.83%	5.73%	12.50%
	makespan	1357.0	1578.1	1841.0	2033.9	2261.0	3068.5	3152.0	5673.0	2620.56
DT	opt. gap	8.63%	11.15%	11.73%	14.12%	19.21%	10.55%	10.59%	5.88%	11.48%
	makespan	1335.0	1517.0	1807.0	2039.3	2263	3066.4	3145.0	5681.0	2606.71
DT ₁₀₀	opt. gap	7.66%	12.05%	11.42%	13.62%	19.26%	10.74%	10.83%	5.56%	11.39%
	makespan	1323.0	1529.3	1802.0	2030.4	2264.0	3071.7	3152.0	5664.0	2604.55
optimal	makespan	1228.9	1364.8	1617.3	1787.0	1898.4	2773.8	2843.9	5365.7	2359.98

It is evident that the DTs can outperform the NLS teacher models in almost all problem sizes and in some cases by significant margins. On average, using DT₁₀₀ leads to 1.11%p, 1.23%p, and 1.15%p better optimality gaps than NLS_A, NLS_{AN}, and NLS_{ANP}, respectively. Note that for the JSSP, such improvements are non-trivial. While DT and DT₁₀₀ perform similarly well on average, each taken alone does not consistently outperform the NLS models. In practice, this implies that both variants should be tried to achieve the best final result for a problem size. Also, none of the DT models were able to outperform the NLS

teacher model on 30×20 , which means that there is no empirical guarantee that the DT always outperforms the student.

5.2. Results on Own Test Instances

To validate our observations on a larger set of instances, the results on 100 randomly generated JSSP instances per problem size in the Taillard benchmark dataset are reported in Table 4. The best results per column are printed in bold. Indeed, the results on this ten-times-larger test dataset confirm a similar trend that the DTs outperform NLS on average. However, compared to the results on the benchmark instances, they do so by a smaller margin and not as consistently. This indicates that the performance of each DT and DT₁₀₀ models varies between different problem instances.

Table 4. Results(makespans) on 100 randomly generated instances. The best results per column are bold.

	15 × 15	20 × 15	20 × 20	30 × 15	30 × 20	50 × 15	50 × 20	100 × 20	Average
NLS _{AN} /NLS _{ANP}	1319.7	1499.4	1736.8	1972.2	2178.2	2974.9	3137.0	5694.7	2564.1
DT	1310.0	1496.4	1739.5	1972.1	2181.0	2974.7	3138.1	5689.6	2562.7
DT ₁₀₀	1301.3	1499.2	1737.0	1972.6	2181.1	2975.3	3136.9	5675.2	2559.8

6. Student–Teacher Comparison

In this section, we examine the learning behavior of the DT models in detail. Recall that they differ from their respective NLS teachers in two noteworthy ways. Firstly, DT models can base decision on a context of up to 50 previously taken actions, states, and returns-to-go. This can enable them to leverage knowledge about the influence of past decisions of their local search on the same problem instance. In fact, in preliminary experiments, we observed performance improvements only when contexts lengths of 50 steps or larger were used. On the downside, to take the context into account, a much larger neural network architecture was used, which led to increased inference times compared to the original NLS architectures. The second major difference to DRL models aiming at minimizing the cumulative rewards is that DT models have an artificial return-to-go prior, which the models are trained to match and which can be used to push them towards shorter makespans than those observed during training. These differences lead to the following three questions we aim to answer in the following analysis to deepen our understanding of the proposed method:

1. What are the practical implications of using the comparatively larger and slower DT models during inference with respect to performance?
2. Is there a correlation between the relatively better performance of DT models in comparison with their teacher models and a greater deviation in learned behavior?
3. Is there an optimal return-to-go to achieve the best performance? The results in Tables 3 and 4 indicate a better performance of the model when the same number of local search steps is performed. However, inferences of the DT take longer to compute than those of the teacher models on the same hardware.

The implication of longer inference times is depicted in Figure 5a on the left and shows the mean makespan in relation to the number of search iterations of the teacher (NLS_{ANP}) and the DT model on our 15 × 15 JSSP instances. The gap between the models increases with a larger number of iterations, i.e., the DT model continues to improve the solution faster with more steps. Figure 5b on the right shows the same achieved mean makespans in relation to the computational time required for the complete search on our hardware (Intel Core i9-9980HK CPU @ 2.40 GHz with 64 GB RAM and NVIDIA Quadro T2000 GPU).

Since the teacher model performs at par with the DT for small numbers of iterations but is faster, it outperforms the DT in search times of up to about 7 s.

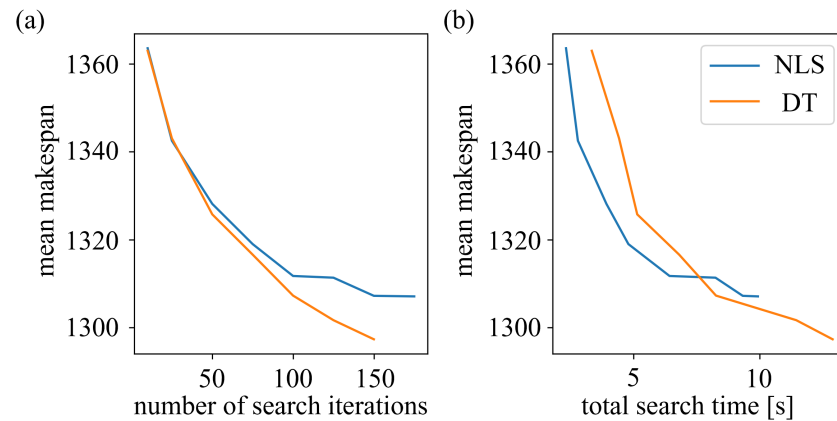


Figure 5. Comparison of NLS and DT models with respect to the number of iterations (a) and the total search time in seconds (b).

However, since the teacher model converges to a worse average makespan, the DT achieves better makespans for all search times larger than seven seconds. Therefore, practically speaking, using the DT is beneficial if search times greater than seven seconds are acceptable. It is important to note that the point at which this tradeoff occurs can vary significantly depending on the final performance of the DT and the size of the problem instances. In some cases, such as the 20×20 JSSP, the DT may not outperform the teacher model.

As a side note on the computational feasibility of the DT extension to the NLS approach, the computational time remains within the same range as the original NLS method. While GPU and CPU memory consumption naturally depends on specific inference settings, such as batch size, these factors did not present any challenges on the consumer-grade laptop used for inference in this study. The dedicated GPU never exceeded 1.2 GB of memory usage, and the RAM usage remained below 4.5 GB—both well within the capabilities of typical industrial systems. Furthermore, the trained DT model weights required 5.471 KB of storage, compared to 4.050 KB for the original NLS model. Consequently, inference using the DT extension should pose no practical issues for real-world applications.

To compare the learned strategies between teacher and DT models, we compared the frequencies with which available actions were chosen when solving the 100 generated instances. We hypothesized that DT models achieving makespans similar to their teacher models may have learned to merely replicate the demonstrated behavior. The action frequencies of the 15×15 DT, representing the DT with the greatest relative improvement over its teacher model, are shown in Figure 6a. The 20×20 and 50×20 DT models, as representatives with similar performance to their teacher models, are shown in Figure 6b and Figure 6c, respectively. The available actions on the x-axis comprise the neighborhood operators CT, CET, ECET, and CEI, and the perturb operator, each in combination with a reject or accept action (odd and even numbers). As one may expect based on the performance difference, the action frequencies vary between the teacher and DT models on 15×15 instances. While the frequency of taking (CT, reject) action 0 is similar, the DT model has learned to never take ECET actions 4 and 5, and it heavily prioritizes (CET, accept) action 3 in favor of fewer (CEI, not reject) action 7 s. The action frequency distribution in 20×20 is, in comparison, much more similar between the teacher and the DT models. This, along with the similar makespans reported in Table 4, suggests a very similar behavior of the two models. Contrarily, 50×20 presents itself as a counterexample with similar makespans but a very different learned behavior expressed by the action frequencies in Figure 6c.

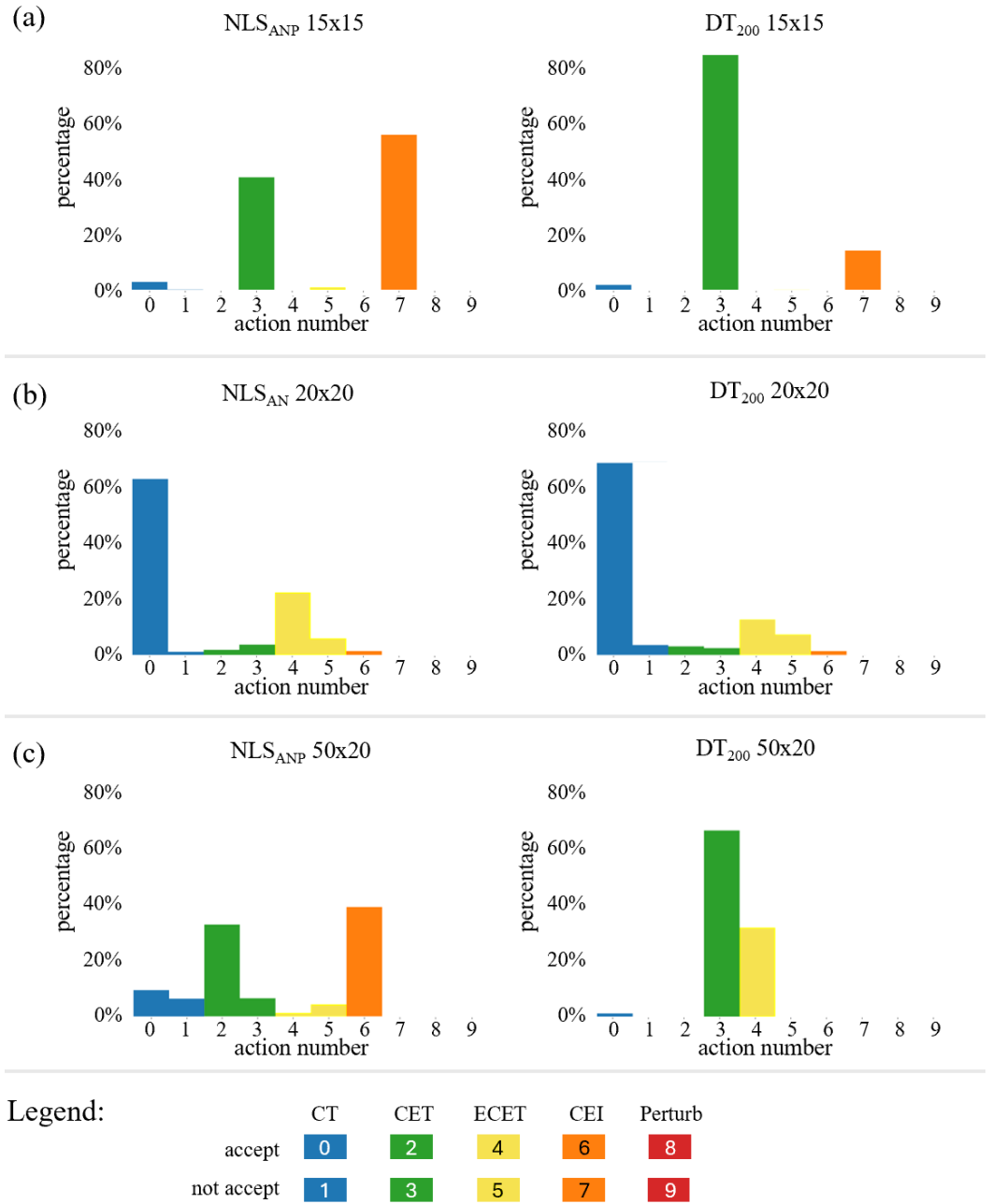


Figure 6. Action frequencies of NLS and DT models for problem sizes (a) 15×15 , (b) 20×20 , and (c) 50×20 .

This falsifies our hypothesis that similar performance results alone indicate that the DT models are merely copying the teacher's behavior. As an interesting observation, no perturbation operation actions 8 and 9 were taken by the neither teacher nor student models that include this action (NLS_{ANP} and DT 15×15 , NLS_{ANP} and DT 50×20). This is noteworthy because the ANP action sets, which include these operations, performed better than the AN action sets. This behavior is an artifact of the NLS training, not the DT training. However, it serves as an indication that the DT learns to ignore actions that have not been used by the teacher and is, therefore, limited in how much it can deviate from demonstrated strategies. To analyze the influence of the return-to-go prior, we tested our 15×15 DT_{100} on the 100 randomly generated instances while multiplying the initial lower bound makespan m_{lb} that was used as a return-to-go starting point during inference with return-to-go factors in the interval $[0.05, 1.75]$ in 0.05 increments. The interval was chosen such that the resulting return-to-go priors cover the achieved makespans of the DT_{100} .

model. The results are shown in Figure 7. We do not observe a trend in the mean makespan over different return-to-go factors. Additionally, the 95% confidence interval, which is depicted as the shaded area, is much larger than the difference between the minimal and maximal values. We, therefore, conclude that the return-to-go has no statistically significant influence on the resulting mean makespans. Interestingly, all obtained mean makespans are smaller than those of the teacher model. This means that the DT has learned a new superior strategy independently from the return-to-go.

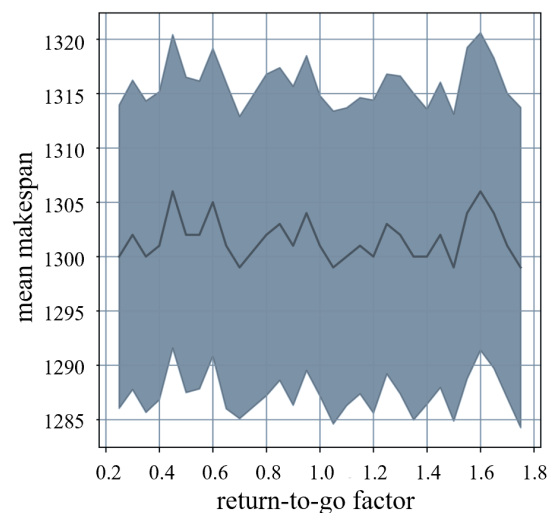


Figure 7. Mean makespans of the DT on 100 15×15 instances with varying returns-to-go.

7. Concluding Discussion and Outlook

In conclusion, we have shown that the DT can be leveraged to increase the performance of trained state-of-the-art DRL-based NLS models on the JSSP by learning to take more effective decisions during the search. However, the benefit of using the DT varies for different problem sizes and parameterizations of the approach. We did not observe any trend that allows us to predict how much better a DT will perform a priori. From a practical perspective, this means that the benefit of using the DT may not always be given when a short search time limit applies. Upon closer examination of why the DT models outperform the NLS models, we found that their main advantage lies in the broader context of previous search steps that the DT effectively leverages through the transformer architecture. The returns-to-go, on the other hand, seem to play only a minor role in reaching superior performance. These observations motivate future work, in which we aim to push the DT towards considering the return-to-go through a forced variation in the quality of solutions generated by the teacher NLS models. This could be achieved by learning from different teachers at the same time or by curriculum learning, which varies the difficulty the algorithm has in solving certain instances well, as performed in [50]. Moreover, since the return-to-go prior has no significant impact on performance, one of the theoretical advantages of the DT compared to other offline reinforcement learning methods is not leveraged. Therefore, other recent algorithms of this class present promising candidates for future research [51,52].

Author Contributions: Conceptualization, C.W.d.P., J.P., H.T. and T.M.; methodology, C.W.d.P. and F.W.; software, C.W.d.P. and F.W.; validation, F.W. and M.M.; resources, H.T. and T.M.; writing—original draft preparation, C.W.d.P., F.W. and M.M.; writing—review and editing, J.P., H.T. and T.M.; visualization, C.W.d.P., F.W. and J.P.; funding acquisition, H.T. and T.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data and code are available via our GitHub repository (<https://github.com/tmdt-buw/Decision-Transformer-4-NLS-JSSP>, accessed on 17 February 2025).

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

Abbreviations

The following abbreviations are used in this manuscript:

CEI	Critical End Improvement
CET	Critical End Time
CT	Critical Time
DRL	Deep Reinforcement Learning
DT	Decision Transformer
ECET	Extended Critical End Time
JSSP	Job Shop Scheduling Problem
LS	Local Search
ML	Machine Learning
NLS	Neural Local Search
PDR	Priority Dispatching Rules

References

- Samsonov, V.; Kemmerling, M.; Paegert, M.; Lütticke, D.; Sauermann, F.; Gützlaff, A.; Schuh, G.; Meisen, T. Manufacturing Control in Job Shop Environments with Reinforcement Learning. In Proceedings of the ICAART 2021, Online, 4–6 February 2021; Rocha, A.P., Steels, L., den van Herik, J., Eds.; SCITEPRESS - Science and Technology Publications Lda: Sétubal, Portugal, 2021; pp. 589–597. [\[CrossRef\]](#)
- Park, I.B.; Park, J. Scalable Scheduling of Semiconductor Packaging Facilities Using Deep Reinforcement Learning. *IEEE Trans. Cybern.* **2023**, *53*, 3518–3531. [\[CrossRef\]](#)
- Kocot, B.; Czarnul, P.; Proficz, J. Energy-Aware Scheduling for High-Performance Computing Systems: A Survey. *Energies* **2023**, *16*, 890. [\[CrossRef\]](#)
- Pinedo, M. *Scheduling: Theory, Algorithms, and Systems*, 5th ed.; Springer International Publishing: Berlin/Heidelberg, Germany, 2016. [\[CrossRef\]](#)
- Taillard, E. Benchmarks for basic scheduling problems. *Eur. J. Oper. Res.* **1993**, *64*, 278–285. [\[CrossRef\]](#)
- van Hoorn, J.J. The Current state of bounds on benchmark instances of the job-shop scheduling problem. *J. Sched.* **2018**, *21*, 127–128. [\[CrossRef\]](#)
- Demirkol, E.; Mehta, S.; Uzsoy, R. Benchmarks for shop scheduling problems. *Eur. J. Oper. Res.* **1998**, *109*, 137–141. [\[CrossRef\]](#)
- van Ekeris, T.; Meyes, R.; Meisen, T. Discovering Heuristics And Metaheuristics For Job Shop Scheduling From Scratch Via Deep Reinforcement Learning. In Proceedings of the Conference on Production Systems and Logistics: CPSL 2021, Online, 10–11 August 2021; pp. 709–718. [\[CrossRef\]](#)
- Stricker, N.; Kuhnle, A.; Sturm, R.; Friess, S. Reinforcement learning for adaptive order dispatching in the semiconductor industry. *CIRP Ann.* **2018**, *67*, 511–514. [\[CrossRef\]](#)
- Tassel, P.; Gebser, M.; Schekotihin, K. A Reinforcement Learning Environment For Job-Shop Scheduling. *arXiv* **2021**, arXiv:2104.03760.
- Tassel, P.; Kovács, B.; Gebser, M.; Schekotihin, K.; Kohlenbrein, W.; Schrott-Kostwein, P. Reinforcement Learning of Dispatching Strategies for Large-Scale Industrial Scheduling. In Proceedings of the International Conference on Automated Planning and Scheduling, Singapore, 13–24 June 2022; Volume 32, pp. 638–646. [\[CrossRef\]](#)
- Waubert de Puiseau, C.; Peters, J.; Dörpelkus, C.; Tercan, H.; Meisen, T. schably: A Python framework for deep reinforcement learning based scheduling experiments. *SoftwareX* **2023**, *22*, 101383. [\[CrossRef\]](#)
- Cheng, L.; Tang, Q.; Zhang, L.; Zhang, Z. Multi-objective Q-learning-based hyper-heuristic with Bi-criteria selection for energy-aware mixed shop scheduling. *Swarm Evol. Comput.* **2021**, *69*, 100985. [\[CrossRef\]](#)
- Reijnen, R.; Zhang, Y.; Bukhsh, Z.; Guzek, M. Learning to Adapt Genetic Algorithms for Multi-Objective Flexible Job Shop Scheduling Problems. In Proceedings of the Companion Conference on Genetic and Evolutionary Computation, New York, NY, USA, 15–19 July 2023. [\[CrossRef\]](#)

15. Han, Z.Y.; Pedrycz, W.; Zhao, J.; Wang, W. Hierarchical Granular Computing-Based Model and Its Reinforcement Structural Learning for Construction of Long-Term Prediction Intervals. *IEEE Trans. Cybern.* **2022**, *52*, 666–676. [\[CrossRef\]](#)
16. Chen, X.; Tian, Y. Learning to Perform Local Rewriting for Combinatorial Optimization. In Proceedings of the 33rd International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; pp. 6281–6292.
17. Tassel, P.; Gebser, M.; Schekotihin, K. An End-to-End Reinforcement Learning Approach for Job-Shop Scheduling Problems Based on Constraint Programming. In Proceedings of the 2nd Conference on Production Systems and Logistics, Querétaro, Mexico, 28 February–2 March 2023; pp. 614–622.
18. Falkner, J.K.; Thyssens, D.; Bdeir, A.; Schmidt-Thieme, L. Learning to Control Local Search for Combinatorial Optimization. In Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Grenoble, France, 19–23 September 2022; Volume 13717, pp. 361–376. [\[CrossRef\]](#)
19. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention Is All You Need. In Proceedings of the 31st International Conference on Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; pp. 6000–6010.
20. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language Models are Few-Shot Learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
21. Chen, L.; Lu, K.; Rajeswaran, A.; Lee, K.; Grover, A.; Laskin, M.; Abbeel, P.; Srinivas, A.; Mordatch, I. Decision Transformer: Reinforcement Learning via Sequence Modeling. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 15084–15097.
22. Perron, L.; Didier, F. CP-SAT. 2024. Available online: https://developers.google.com/optimization/cp/cp_solver/ (accessed on 17 February 2025).
23. Haupt, R. A survey of priority rule-based scheduling. *OR Spectr.* **1989**, *11*, 3–16. [\[CrossRef\]](#)
24. Adams, J.; Balas, E.; Zawack, D. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Manag. Sci.* **1988**, *34*, 391–401. [\[CrossRef\]](#)
25. Bhatt, N.; Chauhan, N.R. Genetic algorithm applications on Job Shop Scheduling Problem: A review. In Proceedings of the 2015 International Conference on Soft Computing Techniques and Implementations (ICSCTI), Faridabad, India, 8–10 October 2015; IEEE: Piscataway, NJ, USA, 2015. [\[CrossRef\]](#)
26. Lourenço, H.R.; Martin, O.C.; Stützle, T. Iterated Local Search: Framework and Applications. *Handb. Metaheuristics* **2019**, *272*, 129–168. [\[CrossRef\]](#)
27. Hansen, P.; Mladenović, N.; Brimberg, J.; Pérez, J.A.M. Variable Neighborhood Search. *Handb. Metaheuristics* **2019**, *272*, 57–97. [\[CrossRef\]](#)
28. Sutton, R.S.; Barto, A. *Reinforcement Learning: An Introduction*, 2nd ed.; Adaptive Computation and Machine Learning; The MIT Press: Cambridge, MA, USA; London, UK, 2018.
29. Ding, W.; Majcherczyk, N.; Deshpande, M.; Qi, X.; Zhao, D.; Madhivanan, R.; Sen, A. Learning to View: Decision Transformers for Active Object Detection. In Proceedings of the 2023 IEEE International Conference on Robotics and Automation (ICRA), London, UK, 29 May–2 June 2023; IEEE: Piscataway, NJ, USA, 2023. [\[CrossRef\]](#)
30. Wang, S.; Chen, X.; Jannach, D.; Yao, L. Causal Decision Transformer for Recommender Systems via Offline Reinforcement Learning. In Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval, New York, NY, USA, 23–27 July 2023. [\[CrossRef\]](#)
31. Lai, Y.; Liu, J.; Tang, Z.; Wang, B.; Hao, J.; Luo, P. ChipFormer: Transferable Chip Placement via Offline Decision Transformer. In Proceedings of the International Conference on Machine Learning, Honolulu, HI, USA, 23–29 July 2023; pp. 18346–18364.
32. van Laarhoven, P.J.M.; Aarts, E.H.L.; Lenstra, J.K. Job Shop Scheduling by Simulated Annealing. *Oper. Res.* **1992**, *40*, 113–125. [\[CrossRef\]](#)
33. Nowicki, E.; Smutnicki, C. A Fast Taboo Search Algorithm for the Job Shop Problem. *Manag. Sci.* **1996**, *42*, 797–813. [\[CrossRef\]](#)
34. Kuhpfahl, J.; Bierwirth, C. A study on local search neighborhoods for the job shop scheduling problem with total weighted tardiness objective. *Comput. Oper. Res.* **2016**, *66*, 44–57. [\[CrossRef\]](#)
35. Balas, E.; Vazacopoulos, A. Guided Local Search with Shifting Bottleneck for Job Shop Scheduling. *Manag. Sci.* **1998**, *44*, 262–275. [\[CrossRef\]](#)
36. van Hasselt, H.; Guez, A.; Silver, D. Deep Reinforcement Learning with Double Q-Learning. In Proceedings of the AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; Volume 30. [\[CrossRef\]](#)
37. Zhang, C.; Song, W.; Cao, Z.; Zhang, J.; Tan, P.S.; Chi, X. Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1621–1632.
38. Corsini, A.; Porrello, A.; Calderara, S.; Dell’Amico, M. Self-Labeling the Job Shop Scheduling Problem. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 105528–105551.
39. Pirnay, J.; Grimm, D.G. Self-Improvement for Neural Combinatorial Optimization: Sample without Replacement, but Improvement. *Trans. Mach. Learn. Res. arXiv* **2024**, arXiv:2403.15180.

40. Park, J.; Chun, J.; Kim, S.H.; Kim, Y.; Park, J. Learning to schedule job-shop problems: Representation and policy learning using graph neural network and reinforcement learning. *Int. J. Prod. Res.* **2021**, *59*, 3360–3377. [\[CrossRef\]](#)
41. Waubert de Puiseau, C.; Meyes, R.; Meisen, T. On reliability of reinforcement learning based production scheduling systems: A comparative survey. *J. Intell. Manuf.* **2022**, *33*, 911–927. [\[CrossRef\]](#)
42. Waubert de Puiseau, C.; Dörpelkus, C.; Peters, J.; Tercan, H.; Meisen, T. Beyond Training: Optimizing Reinforcement Learning Based Job Shop Scheduling Through Adaptive Action Sampling. *arXiv* **2024**, arXiv:2406.07325v1.
43. Ingimundardottir, H.; Runarsson, T.P. Discovering dispatching rules from data using imitation learning: A case study for the job-shop problem. *J. Sched.* **2018**, *21*, 413–428. [\[CrossRef\]](#)
44. Rinciog, A.; Mieth, C.; Scheikl, P.M.; Meyer, A. Sheet-Metal Production Scheduling Using AlphaGo Zero. In Proceedings of the Conference on Production Systems and Logistics: CPSL 2020, Stellenbosch, South Africa, 17–20 March 2020; pp. 342–352. [\[CrossRef\]](#)
45. Lee, J.H.; Kim, H.J. Imitation Learning for Real-Time Job Shop Scheduling Using Graph-Based Representation. In Proceedings of the 2022 Winter Simulation Conference (WSC), Singapore, 11–14 December 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 3285–3296. [\[CrossRef\]](#)
46. Falkner, J.K. NeuroLS. *GitHub*. 2024. Available online: <https://github.com/jokofa/NeuroLS> (accessed on 17 February 2025).
47. Sels, V.; Gheysen, N.; Vanhoucke, M. A comparison of priority rules for the job shop scheduling problem under different flow time- and tardiness-related objective functions. *Int. J. Prod. Res.* **2012**, *50*, 4255–4270. [\[CrossRef\]](#)
48. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. *arXiv* **2017**, arXiv:1412.6980v9.
49. Karpathy, A. minGPT. 2020. Available online: <https://github.com/karpathy/minGPT> (accessed on 17 February 2025).
50. Waubert de Puiseau, C.; Tercan, H.; Meisen, T. Curriculum Learning in Job Shop Scheduling using Reinforcement Learning. In Proceedings of the Conference on Production Systems and Logistics: CPSL 2023—1, Querétaro, Mexico, 28 February–3 March 2023; pp. 34–43. [\[CrossRef\]](#)
51. Hu, S.; Fan, Z.; Huang, C.; Shen, L.; Zhang, Y.; Wang, Y.; Tao, D. Q-value Regularized Transformer for Offline Reinforcement Learning. In Proceedings of the International Conference on Machine Learning, Vienna, Austria, 21–27 July 2024.
52. Kostrikov, I.; Nair, A.; Levine, S. Offline Reinforcement Learning with Implicit Q-Learning. In Proceedings of the 35th International Conference on Neural Information Processing Systems, Online, 6–14 December 2021; Volume 35.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.