Deep learning-driven scheduling algorithm for a single machine problem minimizing the total tardiness

Michal Bouška[a,b], Přemysl Šůcha[a,*], Antonín Novák[a,b], Zdeněk Hanzálek[a]

[a]*Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague, Jugoslávských partyzánů 1580/3, Prague, Czech republic*
[b]*Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Control Engineering, Karlovo náměstí 13, Prague, Czech republic*

Abstract

In this paper, we investigate the use of the deep learning method for solving a well-known NP-hard single machine scheduling problem with the objective of minimizing the total tardiness. We propose a deep neural network that acts as a polynomial-time estimator of the criterion value used in a single-pass scheduling algorithm based on Lawler's decomposition and symmetric decomposition proposed by Della Croce et al. Essentially, the neural network guides the algorithm by estimating the best splitting of the problem into subproblems. The paper also describes a new method for generating the training data set, which speeds up the training dataset generation and reduces the average optimality gap of solutions. The experimental results show that our machine learning-driven approach can efficiently generalize information from the training phase to significantly larger instances. Even though the instances used in the training phase have from 75 to 100 jobs, the average optimality gap on instances with up to 800 jobs is 0.26%, which is almost five times less than the gap of the state-of-the-art heuristic.

Keywords: Scheduling, Machine Learning, Single Machine, Total Tardiness, Deep Neural Networks.

[*]Corresponding author

Email addresses: `bouskmi2@cvut.cz` (Michal Bouška), `suchap@cvut.cz` (Přemysl Šůcha), `antonin.novak@cvut.cz` (Antonín Novák), `zdenek.hanzalek@cvut.cz` (Zdeněk Hanzálek)

1. Introduction

The classical approaches for solving combinatorial problems have several undesirable properties. First of all, there is a lack of systematic methods that improve the performance of algorithms on unseen instances by gathering the experience from the instances solved in the past. Therefore, all the information obtained during the past runs of an algorithm is neglected when a new instance is encountered. A good example is the branch-and-bound-and-remember method [34, 40], where the algorithm remembers the information derived during an instance solving, but the information is forgotten as soon as the instance is solved. Second, the development of efficient heuristic rules requires a substantial amount of time devoted to its design and testing. This process is tedious and requires a skilled human professional to fine-tune the heuristic's parameters. A typical example of this feature is genetic algorithms having many parameters for selection, cross-over, mutation, and other operators.

The apparent response to the above challenges is utilizing the existing data. However, the main obstacle to the successful application of machine learning to enhance algorithms for combinatorial problems remains. It can be formulated as the following fundamental question—*is it possible to extract any useful information from the solved instances and use it efficiently to accelerate solving of an unseen instance?*

This paper addresses a classical NP-hard single machine total tardiness scheduling problem ($1||\sum T_j$), i.e., the problem given by a set of jobs that need to be scheduled on a single machine such that total violation of due-dates is minimized. Specifically, we investigate the use of deep learning [27], to guide the solution space exploration of $1||\sum T_j$ instances. The presented approach extracts specific information from already solved instances, i.e., parameters of the instance and the optimal value of the objective function. This information is used as a training data set. Furthermore, the paper describes a deep neural network that is trained using the training data set. Then the network can predict the optimal value of the objective function for other $1||\sum T_j$ instances. Unlike some existing works addressing the use of machine learning (ML) to solve combinatorial problems (for example, [49]), our approach does not rely solely on machine learning, but we combine it with the approaches known from operations research (OR) domain. The described scheduling algorithm shows the way the deep neural network can be combined with classical decomposition schemes [30, 14] to achieve a fast and efficient solution space exploration. The experiments show that our heuristic algorithm outperforms the existing approaches on the standard benchmark data set. Apart from that, we address the question of how to generate the training data set for the deep neural network. A straightforward approach would require solving hundreds of thousands of NP-hard problems which could take many days. We show that for problem $1||\sum T_j$, there is a much more elegant way that requires only a fraction of that time.

The contributions of this paper can be summarized as follows. We (i) propose an innovative heuristic algorithm integrating the ML and OR approaches; (ii) improve the process of generating the training data, which leads to faster training and smaller error of our method; (iii) provide an analysis of deep neural network hyperparameters' impact on the solution's quality, and; (iv) show that the proposed approach outperforms the state-of-the-art algorithms on the standard benchmark instances.

The rest of the paper is structured as follows. In Section 2, we present a review of the literature covering $1||\sum T_j$ and the use of ML for solving combinatorial problems. The studied problem is formally introduced in the subsequent section. Section 4 describes our approach integrating the ML into a decomposition-based approach and analyzes its time complexity. We present results for standard benchmark instances in Section 5. The conclusion is drawn in Section 6, and lists of notations and abbreviations are provided in the Appendix.

2. Related Work

The first part of the related work focuses on the current approaches to solve $1||\sum T_j$. This part extends the survey by Koulamas [26]. In the second part, we concentrate on existing works exploiting ML for solving combinatorial problems.

2.1. Single Machine Total Tardiness Problems

In 1977 it was shown by Lawler [30] that the weighted single machine total tardiness problem is NP-hard. However, it took more than a decade to prove that the unweighted variant of this problem is weakly NP-hard [17]. Lawler [30] proposed a pseudo-polynomial (in the sum of processing times) algorithm for solving $1||\sum T_j$. The algorithm is based on a decomposition of the problem into subproblems. The decomposition firstly sorts the jobs in earliest due date (EDD) order. Subsequently, it selects the job with the maximum processing time and tries to assign it to the current position and to all the following positions in the EDD sequence. For each position, two subproblems are generated. The first subproblem contains all the jobs preceding the job with the maximum processing time. The second subproblem contains all the jobs following the job with the maximum processing time. Besides that, Lawler introduced rules for filtering the candidate positions of the job with the maximum processing time. This algorithm can solve instances with up to one hundred jobs. Della Croce *et al.* [14] proposed the shortest processing time (SPT) decomposition that selects the job with the minimal due date and tries to assign it to every position preceding its original position in the SPT order. Similarly to Lawler's decomposition, two subproblems are generated where the first subproblem contains all the jobs preceding the job with the minimal due date, and the second subproblem contains all the jobs following the job with the minimal due date. Della Croce *et al.* combined both EDD and SPT decompositions together. Their algorithm is able to solve instances with up to 150 jobs. Szwarc *et al.* [43] integrated the double decomposition from [14] and a Split rule [45]. Their algorithm solved instances with up to 300 jobs. The same authors further improved the algorithm using paradoxes associated with the problem [44]. This algorithm was the state-of-the-art method for a long time, with the ability to solve instances with up to 500 jobs.

Recent papers by Shang *et al.* [39] and Garraffa *et al.* [18] proposed a branch-and-merge algorithm that avoids the solution of equivalent sub-instances in the branching tree. The algorithm uses so-called memorization, i.e., a technique that memorizes the solution of solved sub-problems so that when that sub-problem is reencountered, its solution is retrieved directly from memory instead of solving it again. The authors shown that the algorithm run time converges to $O^*(2^n)$, i.e., the run time is limited by 2^n while polynomial factors are omitted. The same authors have shown that memorization during the solution space exploration is also efficient for other problems, e.g., $1|r_j|\sum C_j$ and $1|\tilde{d}_j|\sum w_j C_j$ [40]. Nowadays, the algorithm published by Shang, T'Kindt and Della Croce [40] is the fastest known exact algorithm for $1||\sum T_j$, able to solve instances with up to 1200 jobs. In this paper, we denote this algorithm as Total Tardiness Branch-and-Merge Algorithm (TTBM).

Exact algorithms, such as the ones mentioned above, have very large computation times, while the optimal solution is rarely needed in practice [50]. Hence, heuristic algorithms are often more practical. The existing heuristics algorithms can be categorized into the following three major groups.

The first group of heuristics consists of list scheduling algorithms that create a job order and schedule the jobs according to this order. There are various methods for creating a job order. The easiest one is to sort jobs by the Earliest Due Date rule (EDD). A more efficient algorithm, called NBR, was proposed in paper [21]. It is a local search constructive heuristic that starts with job set J sorted by EDD and constructs the schedule from the end by swapping two jobs by a hand-designed rule. Panwalkar *et al.* [36] proposed an alternative constructive local search heuristic called PSK. Russel and Holsenback [38] compared PSK and NBR heuristics and concluded that neither heuristic is inferior to another one. However, NBR finds a better solution in more cases.

Heuristics in the second group are based on Lawler's decomposition rule [30]. In this case, the heuristic evaluates each node of the search tree, and the most promising node is expanded. This heuristic approach is evaluated in [37] with an EDD heuristic as a guide for the search.

The third group of heuristics contains metaheuristics. Papers [37, 3, 6] present the simulated annealing algorithm for $1||\sum T_j$. The same problem is solved in [16, 41] by a genetic algorithm while the authors of [5, 11] assumed ant colony optimization. All the reported results in the previous studies are for instances with up to 100 jobs. However, these instances are solvable by the current state-of-the-art exact algorithm in a fraction of a second.

2.2. *Use of Machine Learning in Algorithms for Combinatorial Optimization Problem*

The integration of ML into algorithms for solving combinatorial optimization problems has several difficulties. First, the instances of scheduling problems naturally appear in different sizes, e.g., with a variable number of jobs. In opposite to this, the majority of ML models are often designed with a fixed size of the input feature vector and the output vector. This issue can be addressed by recurrent networks and, more recently, by encoder-decoder type of architectures. Vinyals [49] applied an architecture called Pointer Network that, given a set of graph nodes, outputs a solution as a permutation of these nodes. The authors applied the Pointer Network to Traveling Salesman Problem (TSP) with up to 20 nodes; however, this approach for TSP is still not competitive with the best classical solvers such as Concorde [4] that can find optimal solutions to instances with $80,000$ nodes. Moreover, the Pointer Network output needs to be corrected by the beam-search procedure, which underlines the weaknesses of this end-to-end approach. Pointer Network has achieved an optimality gap of around 1% for instances with 20 nodes after performing the beam search.

The second difficulty with using ML models for solving combinatorial problems lies in the acquisition of training data. Obtaining a single label for a training instance usually requires solving a problem of the same complexity as the original problem itself, while ML usually requires millions of training samples. This issue can be addressed by the reinforcement learning paradigm. Deudon *et al.* [15] used encoder-decoder architecture trained with REINFORCE algorithm to solve 2D Euclidean TSP with up to 100 nodes. It is shown that (i) repetitive sampling from the network is needed, (ii) applying a well-known 2-opt heuristic on the results still improves the solution of the network, and (iii) both the quality and run times are worse than classical exact solvers. A similar approach, used to solve TSP, is described in [25] which, if it is treated as a greedy heuristic, beats simple heuristics such as Christofides algorithm on small instances. To be competitive with a relevant baseline algorithm such as Lin-Kernighan heuristics [31], they perform repeated sampling from the model and output the best solution. Moreover, they do not directly compare their approach with state-of-the-art classical algorithms while admitting that general-purpose Integer Programming solver Gurobi solves their largest instances optimally within 1.5 s.

Reinforcement learning was also used to solve other combinatorial problems. For example, Khalil *et al.* [24] presented an approach for learning greedy algorithms over graph structures. The authors show that their S2V-DQN model can obtain competitive results on MAX-CUT and Minimum Vertex Cover problems. For TSP, S2V-DQN performs about the same as 2-opt heuristics. Unfortunately, the authors do not compare running times with Concorde solver. Interesting results for graph coloring were introduced by Huang *et al.* [22]. Huang *et al.* proposed a *reinforcement learning* (RL) heuristic with a neural network able to outperform the state-of-the-art heuristic by 1-2%, when trained on the same type of graph as the one used during the evaluation. Abe *et al.* [1] presented an RL approach for Minimum Vertex Cover and MAX-CUT. For Minimum Vertex Cover problem, they have up to 10% better solutions than the 2-approximation algorithm. For MAX-CUT problem, they are not able to outperform the heuristic of Laguna [28]. More details can be found in the survey by Mazyavkina *et al.* [33] addressing the use of RL approaches for solving combinatorial problems.

Integration of ML with scheduling problems has received little attention so far. Earlier attempts of integrating neural networks with job-shop scheduling were published in [53] and [23]. However, their computational results are inferior to the traditional algorithms, or it is not possible to assess their quality. Alternative use of ML in the scheduling domain is focused on the evaluation of criterion functions. For example, the authors in [48] addressed a nurse rostering problem and proposed a classifier, implemented as a neural network, able to determine whether a particular change in a solution leads to a better solution or not. This classifier is then used in a local search algorithm to filter out solutions having a low chance of improving the criterion function. Nevertheless, the approach is sensitive to changes in the problem size, i.e., the length of the planning horizon. If the size of the problem is changed, a new neural network must be trained. Another method, which does not directly predict a solution to the given instance, is proposed in [47]. In this case, an online ML technique is integrated into an exact algorithm where it acts as a heuristic. Specifically, the authors use regression for predicting the upper bound of a pricing problem in a Branch-and-Price algorithm. Correct prediction leads to faster computation of the pricing problem, while incorrect prediction does not affect the optimality of the algorithm. This method is not sensitive to the change of the problem size; however, it is designed specifically for the Branch-and-Price approach and cannot be generalized to other approaches. The authors of paper [29] use the neural network as hyper-heuristic switching between several known heuristics for the job-shop scheduling problem. Nevertheless, it is hard to assess the benefit of the method since a comparison with existing approaches is not provided. Recently, Zhang *et al.* [52] solved the same scheduling problem using end-to-end deep reinforcement learning. The authors trained Graph Neural Network to generate a priority dispatching rule. The results show that their approach provides better results compared to simple priority rules like Shortest Processing Time, Most Work Remaining, etc. Shu Luo [32] proposed a Deep Q-Network (DQN) trained by reinforcement learning for the online dynamic flexible job shop scheduling problem. The neural network is trained to select one out of six dispatching rules. The selected rule is then applied in each iteration of their algorithm. A similar work of [2] applies reinforcement learning to an additive manufacturing machine scheduling problem. The authors describe a reinforcement learning iterated local search meta-heuristic that switches different operators of the local search.

In the field of Constraint Satisfaction Problem (CSP) solving, Xu *et al.* [51] presented a neural network estimating the satisfiability of the CSP. The authors assume that the neural network can be integrated into an algorithm for solving CSP. However, Xu *et al.* tested their approach only on instances with up to 128 binary variables. Cappart *et al.* [10] trained neural network able to estimate values of variables during of CSP solving. The estimation of the value of a variable with a neural network leads to an earlier finding of the part of the state space with an optimal solution and thus to faster convergence. The approach is able to solve the instances of Travelling Salesman Problem with Time Windows with up to 100 nodes. Although the proposed method shows an interesting idea, it should be noted that in the literature, there are classical approaches that solve instances with up to 200 nodes.

Nair *et al.* [35] introduced an approach to speed up a Mixed Integer Linear Programming solver with a neural network. They present two methods to speed up the solution — Neural Diving and Neural Branching. Neural Diving focuses on the improvement of the incumbent bound. It generates a partial solution of the instance (i.e., predicts a value only for a subset of variables), which is then fixed. The values of the remaining variables are solved by the solver. Neural Branching is used to select a branching variable in the *branch-and-bound* method. It aims to approximate a computationally expensive branching strategy with just a fraction of the computation time. By the combination of these two methods, Nair *et al.* achieves 1.5 times smaller optimality gap on MIPLIB benchmark set. Another different way to speedup MILP solvers is introduced by Tang *et al.* [46]. Tang *et al.* trained the RL agent, which learns to generate cutting planes and is shown to outperform the human-designed heuristic used in Gurobi MILP solver. Their approach achieves 2 to 3 times faster convergence on large instances for packing, production planning, and MAX-CUT problems.

The contemporary operations research literature has started to focus on machine learning approaches more intensively. A recent survey by Bengio *et al.* [7] identifies four main problems of the use of ML in combinatorial optimization, i.e., modeling, feasibility, scaling, and data generation. (i) Bengio *et al.* argue that unlike, e.g., computer vision, there are no neural network models in the literature that would be suitable for combinatorial problems. (ii) Apart from that, neural networks can be used only as a heuristic. Therefore, their current use is limited for exact approaches as well as for problems where it is difficult to find a feasible solution. (iii, iv) The last two problems correlate with the first two paragraphs in this section. The authors conclude that the existing approaches are at an early stage of development, but they open new opportunities for research addressing combinatorial optimization algorithms.

This paper is founded on the idea that the frequent limitation of the existing techniques applying ML to combinatorial problems is the use of end-to-end approaches. Their weakness is that they disregard fundamental properties of the combinatorial problems that have been studied in the literature for decades. The view studied in this paper is different, and the proposed solution efficiently combines both the ML and properties of the problem.

3. Problem Statement

In this paper, we study a single machine scheduling problem defined by a set of jobs $J = \{1,\dots,n\}$. The machine can process at most one job at a time, and all the jobs are available for processing at time zero. The execution of the jobs cannot be interrupted. Each job $j \in J$ is characterized by processing time $p_j \in \mathbb{Z}_{>0}$ and due date $d_j \in \mathbb{Z}_{\geq 0}$. A solution to this problem is a *schedule* given by a one-to-one correspondence $\pi : \{1,\dots,n\} \mapsto \{1,\dots,n\}$ mapping a position in the schedule to a job, i.e., $\pi_k \in J$ is the job at position k in schedule π. For a scheduled job, the problem defines its tardiness as an indicator measuring how much the job violates the due date. Tardiness of job π_k in schedule π is defined as $\mathcal{T}_{\pi_k}(J) = \max\left(0, \sum_{k' \in J: k' \leq k} p_{\pi_{k'}} - d_{\pi_k}\right)$. Then, the total tardiness of schedule π is defined as $T(J,\pi) = \sum_{k=1}^{n} \mathcal{T}_{\pi_k}(J)$. The goal of the scheduling problem is to find an optimal schedule π^* which minimizes the total tardiness $T^*(J) = \min_{\pi \in \Pi} T(J,\pi)$ where Π is the set of all jobs' permutations. To ease the readability of the paper, the list of notation and symbols used is provided in Appendix.

This combinatorial problem is proven to be weakly NP-hard [17]. Graham's notation [19] denotes it as $1||\sum T_j$ where 1 indicates that it is a single machine scheduling problem and $\sum T_j$ refers to the objective function, i.e., $\min_{\pi \in \Pi} T(J,\pi)$.

4. Proposed Decomposition Heuristic Algorithm

In this section, we introduce *Heuristic Optimizer using Regression-based Decomposition Algorithm* (HORDA) for $1||\sum T_j$. The algorithm's name comes from the fact that it uses decomposition controlled by a regressor. The regressor is realized using a deep neural network (neural network for short in the rest of the paper) approximating the relation between features of instance and $T^*(J)$.

The description of the algorithm is structured as follows. First of all, we summarize the problem decompositions used in the HORDA algorithm. Second, we describe HORDA and show how it effectively combines the well-known properties of $1||\sum T_j$ and an ML model. Next, we proceed by discussing the architecture of the regressor and its integration into $1||\sum T_j$ decompositions, and we describe training data acquisition, including the training of the neural network. Finally, we analyze the time complexity of HORDA algorithm.

In the rest of the paper, we use two definitions to describe the ordering of the job set J:

1. EDD: if $1 \leq j < j' \leq n$ then either (i) $d_j < d_{j'}$ or (ii) $d_j = d_{j'} \wedge p_j \leq p_{j'}$,
2. SPT: if $1 \leq j < j' \leq n$ then either (i) $p_j < p_{j'}$ or (ii) $p_j = p_{j'} \wedge d_j \leq d_{j'}$.

EDD (Earliest Due Date) is a sequence of jobs, sorted in non-decreasing order of due dates and SPT (Shortest Processing Time) is a sequence of jobs sorted in non-decreasing time of processing times.

4.1. Problem Decompositions

Before we describe HORDA, it is necessary to outline two decomposition approaches for $1||\sum T_j$ that are used in our algorithm. The first decomposition is Lawler's decomposition [30] which utilizes EDD order of jobs; therefore, the related notation is denoted by superscript EDD. The other decomposition, proposed by Della Croce *et al.* [14], is analogous but based on SPT order of jobs; thus, the related notation is denoted as SPT.

Both decompositions exploit the fact that any optimal schedule of $1||\sum T_j$ can be represented by a permutation of jobs π since the machine is never idle in an optimal schedule. Each decomposition $\circ \in \{\text{EDD}, \text{SPT}\}$ defines the splitting job $l^{\circ}(J) \in J$, i.e., $l^{EDD}(J)$ for Lawler's decomposition and $l^{SPT}(J)$ for the decomposition proposed by Della Croce *et al.* For a position k of job $l^{\circ}(J)$ in the schedule, the decomposition splits J into two subsets. The first subset $P^{\circ}(J,k)$ represents jobs preceding $l^{\circ}(J)$ in the schedule, and the second subset $F^{\circ}(J,k)$ represents jobs following job $l^{\circ}(J)$ in the schedule under decomposition $\circ$. The precise definition of the $P^{\circ}(J,k)$ and $F^{\circ}(J,k)$ is linked with a particular decomposition, as it is explained below.

The first decomposition, further denoted as EDD decomposition, is based on a theorem proposed by Lawler [30].

Theorem 4.1. *(Lawler, 1977) Suppose jobs J are ordered in EDD order and the splitting job is $l^{EDD}(J) = \arg\max_{i\in J} p_i$. Then, there is some integer k, $l^{EDD}(J) \le k \le n$, such that there exists an optimal sequence π^* in which the splitting job $l^{EDD}(J)$ is preceded by all jobs j such that $j \le k$, and followed by all jobs j such that $j > k$.*

Lawler's decomposition splits jobs into two subsets $P^{EDD}(J,k)$ and $F^{EDD}(J,k)$, which for job set J and position k contains jobs $\{1,\ldots,k\} \setminus \{l^{EDD}(J)\}$ and $\{k+1,\ldots,n\}$, respectively. Thus, for each eligible position $k \in \{l^{EDD}(J),\ldots,n\}$, the problem is decomposed into two subproblems defined by $P^{EDD}(J,k)$ and $F^{EDD}(J,k)$ such that job $l^{EDD}(J)$ is neither in P^{EDD} nor in F^{EDD}. When we denote the set of positions $\{l^{EDD}(J),\ldots,n\}$ as $K^{EDD}(J)$, then the optimal total tardiness $T^*(J)$ of instance J can be computed as

$$T^*(J) = \min_{k\in K^{EDD}(J)} Q(J,k), \tag{1}$$

where

$$Q(J,k) = T^*\left(P^{EDD}(J,k)\right) + \max\left\{0, \sum_{j\in P^{EDD}(J,k)} p_j + p_k - d_k\right\} + T^*\left(F^{EDD}(J,k)\right). \tag{2}$$

The optimal solution to the instance is found by recursively selecting position k with the minimal criterion $Q(J,k)$.

The second decomposition, denoted as SPT decomposition, was proposed by Della Croce *et al.* [14] and is described by the following theorem.

Theorem 4.2. *(Della Croce, 1998) Suppose jobs J are in SPT order and job $l^{SPT}(J) = \arg\min_{i\in J} d_i$. Then there exists an integer k, $1 \le k \le l^{SPT}(J)$, such that there exists and optimal sequence π^* in which the jobs preceding $l^{SPT}(J)$ are uniquely determined as follows: take jobs $\{1,\ldots,l^{SPT}(J)-1\}$ in SPT order and sort these jobs by the EDD order and select the first $k-1$ jobs. All other jobs follow the $l^{SPT}(J)$ job.*

Theorem 4.2 describes a similar decomposition to EDD decomposition but uses different position set $K^{SPT}(J) = \{1, \ldots, l^{SPT}(J)\}$. Furthermore, the set of preceding jobs is denoted as $P^{SPT}(J,k)$, while the set of following jobs is $F^{SPT}(J,k)$. Nevertheless, the basic idea of the decomposition is the same as the one formulated in Equation (1) for the EDD decomposition.

The efficiency of both decomposition approaches is significantly influenced by the number of the relevant positions for the splitting job $l^{\circ}(J)$, $\circ \in \{\text{EDD}, \text{SPT}\}$, i.e., cardinalities of $K^{EDD}(J)$ and $K^{SPT}(J)$. The size of the position set $K^{\circ}(J)$ for $\circ \in \{\text{EDD}, \text{SPT}\}$ can be reduced by filtering rules described in [30, 45, 43]. These rules can exclude some positions that provably cannot lead to an optimal solution. The efficiency of the decompositions can be improved by the following rules: (i) remove the position from $K^{\circ}(J)$ if the completion time of job $l^{\circ}(J)$ at position k is larger than the due date of the following job [43], (ii) remove the position from $K^{\circ}(J)$ if the completion time of job $l^{\circ}(J)$ at position k is smaller than the due date plus the processing time of the previous job [43]. The other rules are based on a similar idea. A detailed explanation, including the proof, can be found in [43]. In the rest of the paper, we denote the filtered set of positions by $\overline{K^{EDD}}(J) \subseteq K^{EDD}(J)$, $\overline{K^{SPT}}(J) \subseteq K^{SPT}(J)$, i.e., $\overline{K^{\circ}}(J) \subseteq K^{\circ}(J)$ for $\circ \in \{\text{EDD}, \text{SPT}\}$.

4.2. Scheduling Algorithm

Even though algorithms, e.g., [18], that use decompositions described in the previous section, are very efficient, their time complexity grows very quickly with the number of jobs. Therefore, we propose a heuristic algorithm, denoted as HORDA, which approximates the search of the solution space by *a priori* trained regressor.

HORDA is a greedy heuristic that combines the efficiency of EDD and SPT decompositions and ML. HORDA recursively applies one of the decompositions while the position k of the splitting job $l^{\circ}(J)$ is determined using the regressor. It aims to select the best position k^* of the splitting job $l^{\circ}(J)$ in $\overline{K^{EDD}}(J)$ or $\overline{K^{SPT}}(J)$ without solving the subproblems. Therefore, the regressor estimates values of $T^*(P^{\circ}(J,k))$ and $T^*(F^{\circ}(J,k))$ in Equation (2) for every relevant position $k \in \overline{K^{\circ}}(J)$. With that, the algorithm selects position k^* that minimizes the estimate of the objective function.

The algorithm is outlined in Algorithm 1. It recursively applies one of the decompositions described in the previous section while splitting input jobs set J to its subsets $P^{\circ}(J,k^*)$ and $F^{\circ}(J,k^*)$. In the first step (lines 2 and 5), the algorithm handles job sets with five or fewer jobs. It turned out to be more efficient to run an exact algorithm instead of performing the inference from the regressor on a small set of jobs. Subsequently, the algorithm determines the splitting job and set of positions k for both decompositions, i.e., $l^{EDD}(J)$, $\overline{K^{EDD}}(J)$ and $l^{SPT}(J)$, $\overline{K^{SPT}}(J)$ for EDD and SPT decomposition, respectively (lines 6 and 7). To reduce the run time, HORDA uses either EDD or SPT decomposition depending on the cardinalities of their position sets (lines 8–13). The position set with smaller cardinality is selected and the selected position set and the related splitting job are stored to $\overline{K^{\circ}}(J)$ and $l^{\circ}(J)$, respectively. After the selection of the positions set, the algorithm greedily selects position k^* (line 14) having the minimal estimation of the optimal objective function, i. e., the total tardiness. Thus the position is determined as $k^* = \arg\min_{k \in \overline{K^{\circ}}(J)} \hat{Q}(J,k)$, where $\hat{Q}(J,k)$ is the estimate of the

Algorithm 1: Heuristic Optimizer using Regression-based Decomposition Algorithm (HORDA)

```
Input: a set of jobs $J$
Output: schedule $\pi$
1  Function HORDA ($J$):
       /* Small instances are solved using an exact approach.                    */
2      if $|J| \leq 5$ then
3          $\pi \leftarrow$ TTBM ($J$)
4          return $\pi$
5      end
       /* Determines the splitting job and the position set (for both
          decompositions).                                                       */
6      $l^{EDD}(J), \overline{K^{EDD}}(J) \leftarrow$ genEDDPos($J$)
7      $l^{SPT}(J), \overline{K^{SPT}}(J) \leftarrow$ genSPTPos($J$)
       /* Selection of position set with smaller cardinality.                    */
8      if $|\overline{K^{EDD}}(J)| \leq |\overline{K^{SPT}}(J)|$ then
9          $\overline{K^{\circ}}(J) \leftarrow \overline{K^{EDD}}(J)$; $l^{\circ}(J) \leftarrow l^{EDD}(J)$
10     end
11     else
12         $\overline{K^{\circ}}(J) \leftarrow \overline{K^{SPT}}(J)$; $l^{\circ}(J) \leftarrow l^{SPT}(J)$
13     end
       /* Determine the position of the splitting job using the regressor.       */
14     $k^* \leftarrow \arg\min_{k \in \overline{K^{\circ}}(J)} (\widehat{T}(P^{\circ}(J,k)) + \max(0, p_k - d_k + \sum_{j \in P^{\circ}(J,k)} p_j) + \widehat{T}(F^{\circ}(J,k)))$
       /* Recursively call of HORDA for both subproblems.                        */
15     $\pi_P \leftarrow$ HORDA ($P^{\circ}(J,k^*)$)
16     $\pi_F \leftarrow$ HORDA ($F^{\circ}(J,k^*)$)
       /* join sequences into one                                                */
17     $\pi \leftarrow (\pi_P, l^{\circ}(J), \pi_F)$
18     return $\pi$
```

objective function for position k computed as

$$\hat{Q}(J,k) = \widehat{T}(P^{\circ}(J,k)) + \max\left\{0, \sum_{j \in P^{\circ}(J,k)} p_j + p_k - d_k\right\} + \widehat{T}(F^{\circ}(J,k)) \,. \tag{3}$$

Estimates $\widehat{T}(P^{\circ}(J,k))$ and $\widehat{T}(F^{\circ}(J,k))$ are computed by the regressor described in the following section. Subsequently, the algorithm recursively solves job sets $P^{\circ}(J,k^*)$ and $F^{\circ}(J,k^*)$. Resulting partial sequences are stored as vectors π_P and π_F, respectively (lines 15 and 16). Finally, the algorithm merges $(\pi_P, l^{\circ}(J), \pi_F)$ into one sequence π, which is returned as the resulting schedule (line 18).

Please notice that if the estimates $\widehat{T}(P^{\circ}(J,k))$ and $\widehat{T}(F^{\circ}(J,k))$ in Equation (3) would be perfect (i.e., $\widehat{T}(P^{\circ}(J,k)) = T^*(P^{\circ}(J,k))$ and $\widehat{T}(F^{\circ}(J,k)) = T^*(F^{\circ}(J,k))$ for every $k \in \overline{K^{\circ}}(J)$), then HORDA would turn into an exact method. This claim directly follows from theorems 4.1 and 4.2. Since Algorithm 1 enumerates all positions k that may lead to an optimal solution, then accurate estimates $\widehat{T}(P^\circ(J,k))$ and $\widehat{T}(F^\circ(J,k))$ guarantee finding k leading to π^* according to theorems 4.1 and 4.2. The first iteration of Algorithm 1 with perfect estimates finds the optimal value of the objective function, while its recursive application finds π^*. Nevertheless, assuming that P$\neq$NP, it is impossible to guarantee that estimates are always correct. On the other hand, the better the precision of the regressor, the more likely that HORDA will find the optimal solution.

We also experimented with other improvements, known in the scheduling domain, like the Split rule proposed in [45]. The rule extends the filtering rules and allows to determine the optimal block sequence w.r.t. position k. However, the tests have shown that in our setting, it slows down HORDA and does not improve the quality of the solutions. Thus we do not use it in Algorithm 1.

4.3. Regressor

HORDA algorithm utilizes the regressor to estimate $\hat{T}(J')$ where J' is either $P^\circ(J,k) \subset J$ or $F^\circ(J,k) \subset J$. It comes as no surprise that the quality of the estimation significantly affects the ability of HORDA to find an optimal or near-optimal solution. The key advantage of HORDA is that it is not sensitive to the absolute error of the estimation since the algorithm compares multiple solutions obtained by the same regressor for different $k \in \overline{K^\circ}(J)$. Therefore, the proposed regressor uses a neural network since those have been shown to be successful for problems being sensitive to relative error [48].

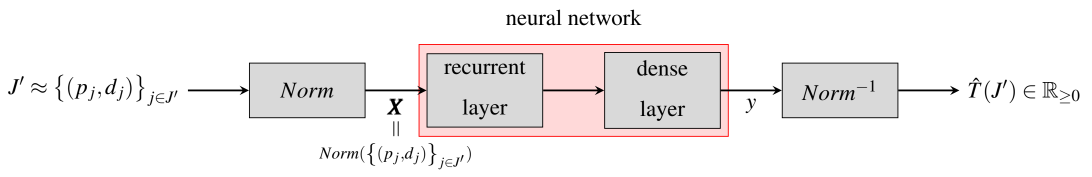

neural network
$J' \approx \{(p_j,d_j)\}_{j\in J'}$
Norm
$\boldsymbol{X}$
$\|$
$Norm(\{(p_j,d_j)\}_{j\in J'})$
recurrent layer
dense layer
y
$Norm^{-1}$
$\hat{T}(J') \in \mathbb{R}_{\geq 0}$

Figure 1: Regressor architecture.

The architecture of our regressor is illustrated in Figure 1. Its input is job set J' characterized by processing times and due dates of jobs. Before the input is passed to the neural network, it is normalized to a matrix of features $\boldsymbol{X}$ by the block denoted *Norm*. The dimension of $\boldsymbol{X}$ is $2 \times |J'|$ where the first dimension corresponds to the two parameters of jobs (p_j,d_j) entering the regressor. The output of the neural network, y, is normalized as well; thus, it needs to be denormalized by $Norm^{-1}$ in order to get estimate $\hat{T}(J')$.

The detailed description of the regressor is split into two subsections. First, in Section 4.3.1 we describe the normalization and denormalization of data, while the neural network is introduced in Section 4.3.2.

4.3.1. Normalization of the Input Data

The normalization of the input data is a preprocessing step that improves the accuracy of the neural network, improves its numerical stability, and reduces training time. In our case, the normalization takes the job set J' and normalizes it to a form suitable to the neural network. This preprocessing consists of (i) supplying the input to the network in a canonical form, (ii) normalization of the input features to $[0,1]$, and (iii) the normalization of the criterion.

The normalization used in the regressor sorts the jobs in a defined order and scales the processing times, due dates, and the value of the objective function by suitable constants. It is important to note that the normalization of inputs for the neural network occurs in two different ways. The first one concerns network training, when we need $\boldsymbol{X}$ and y, i.e., normalized input and output. The other way occurs during the exploitation of the regressor, i.e., *inference* from the neural network. Then the input is normalized to $\boldsymbol{X}$ but the output of the network y needs to be converted back to $\hat{T}(J')$ by block $Norm^{-1}$. The text below describes two normalization procedures that improved the precision of the regressor the most.

Both normalizations sort J' in EDD order. The first normalization, denoted SUMPROC, scales down all $p_j, d_j : \forall j \in J'$ as well as the objective function $\hat{T}(J')$ by factor $\max\left\{\sum_{j\in J'} p_j, \max_{j\in J'} d_j\right\}$. This normalizes the processing times and due dates to the interval $[0,1]$; nevertheless, the value of the objective function obtained for the rescaled parameters can be greater than 1. Despite its simplicity, we have observed a noticeable impact of this normalization on the accuracy of the regressor.

The second normalization, denoted as GAPEDDINV, redefines the output of the neural network. The neural network is not trained to predict the target criterion value but rather the difference between the optimal objective value and the objective value achieved by EDD job ordering. This is the key innovation, as the neural network has an easier job modeling residue between the optimal and EDD solutions rather than modeling the objective function from scratch. Except for a different quantity to predict, GAPEDDINV uses the same normalization constant for the processing times and due dates as SUMPROC; however, it redefines how the predicted value $\hat{T}(J')$ is computed.

During the training phase of the network, the GAPEDDINV normalization constructs the EDD sequence of jobs. Then it takes the optimal solution π^* and computes the optimality gap of the EDD sequence π^{EDD} as $gap_{EDD} = \frac{T(J',\pi^{EDD})-T^*(J')}{T(J',\pi^{EDD})}$. Subsequently, to have the output of the neural network distributed in interval $[0,1]$, the normalized criterion is computed as $y = \frac{1}{1+gap_{EDD}}$. The inverse transformation, needed for the inference from the neural network during the HORDA run, proceeds the other way around. First, it computes $gap_{EDD} = \frac{1}{y} - 1$ from the normalized output y. Then it computes the total tardiness of the π^{EDD} sequence, which can be constructed in polynomial time. Finally, it uses the definition of the EDD optimality gap to derive $\hat{T}(J')$.

4.3.2. Neural Network

The main reasons why we have used a deep neural network architecture in this paper are the following. One of the main limitations of the application of classical machine learning models (including standard neural network) is that they expect a fixed-sized input, meaning that the dimension of the input data is always the same number, which needs to be chosen before the training and cannot be changed afterward. However, the scheduling problem which we solve assumes an arbitrary number of jobs J, and it is not apparent how to compress the input instance into a fixed-sized input. Note that this is radically different from, e.g., computer vision problems, where the input image can be naturally scaled down to match the target dimensions.

Another reason is that deep neural network architectures have sufficient capability to discover their own features of the input data. For classical ML models, it is needed to design custom descriptors of the data (i.e., features). In the case of our scheduling problem with total tardiness minimization, such features might be statistical indicators of processing times and dues (e.g., means and variances). However, since we face an NP-hard problem (i.e., even the prediction of the optimal objective is a hard problem), it is very unlikely that such simple descriptors would work well. More complex interactions between the individual job parameters need to be considered for precise predictions. Indeed, when we tested in our experiments simpler networks, we could see that the prediction accuracy is affected by the capacity (i.e., model complexity) of the prediction models. For example, in Section 5.3 we evaluate the effect of the complexity of the network on the quality of solutions. There, we can see that reducing the size of the hidden state of the neural network has already a negative impact on the performance of HORDA. However, even the features developed by the network with a limited complexity are still much more complex than the handcrafted features like the statistical ones mentioned above. Thus, we let the network discover its own features from raw data.

The solution to both these challenges is offered by recurrent neural networks [42]. A recurrent neural network is a type of network that can use previous outputs as inputs in successive iterations. It maintains a hidden state which is updated by the current input and the past value of the hidden state. The whole input to the recurrent neural network is represented by a sequence that is processed in an iterative fashion. The final output of the network is a function of the last hidden state. In our case, the input sequence is the instance of the scheduling problem presented in a predefined canonical order. In each step of the computation, the features (i.e., processing time and due date) of a single job are fed to the network to update its hidden state. Finally, when the sequence is processed, the hidden state of the network reflects the whole problem instance which can be used to make predictions, e.g., about its optimal objective value.

The advantage of this model is the possibility of processing the input of any length (i.e., a variable number of jobs) and that the historical information is used during the whole computation over the sequence. This scheme has been shown to be effective for processing sequences with variable lengths, e.g., in *natural language processing* (NLP).

The neural network used inside the regressor consists of two layers (see the red box in Figure 1). The first layer is a recurrent neural network, which receives normalized job set $\boldsymbol{X}$ as the input. This layer is realized using the *Long Short-Term Memory* (LSTM) architecture [20]. The output of the recurrent layer, the hidden state, is passed into a *dense layer* without an activation function, and it produces estimation y. The main parameters of the recurrent layer are the size of the hidden state, also denoted as *capacity* in the literature. The recurrent layer's capacity affects the amount of information which is the recurrent layer able to approximate.

In our experiments, we compare two different types of recurrent layers, LSTM and *Gated Recurrent Unit* (GRU) [12]. In general, GRU is better suited for smaller training data sets. LSTM is more general and has a more complex structure compared to GRU. For example, GRU has only one reset gate, which substitutes the function of update and the forget gate in LSTM. The experimental results documented in Section 5 show that LSTM provides better results in our case.

Training of the neural network is complicated by its integration into the decompositions. It means that the training and validation errors (used in the training phase of the neural network) are not computed in the same way as the testing error (measured during benchmarking of the entire HORDA). The training and validation errors of the neural network are measured in terms of the mean square error of predicted objective values on the training and validation samples. On the other hand, the testing error is measured as the optimality gap of HORDA, which exploits the neural network. Computation of training and validation errors reflecting the optimality gap obtained by the whole HORDA would be highly time demanding for the training phase. Therefore, our approach relies on the relation between the prediction of objective values and their use in Equation (2) controlling decisions of HORDA. Thus the training is faster, and HORDA still provides accurate solutions.

4.4. Training Data Set Generation

The training data set for a neural network usually consists of thousands of millions of training samples. However, producing a training data set with this size can be extremely time demanding in case of NP-hard problems. Thus, it is extremely important to devise efficient methods that can produce it in a reasonable time. An equally important property of the training data set is its composition, i.e., how well it covers different parts of the parameter space. Essentially, the aim is to ensure that the distribution of training samples corresponds to the distribution of the test samples that arise from the decomposition during the run of HORDA. As it will be shown in the next section, the composition of samples in the training data set is indeed critical to the quality of solutions produced by HORDA. In this section, we propose two approaches to the generation of the training data set, and we describe their properties.

To generate $1||\sum T_j$ instances, we utilize the standard benchmark generator described by Potts *et al.* [37]. They generate processing times from a uniform distribution $[1, p_{max}]$ where p_{max} is the maximum processing time. The distribution of due dates is given by two parameters *rdd* (range of due dates) and *tf* (tardiness factor), describing the relations between the sum of processing times and the due dates of jobs. Specifically, each due date is drawn from a uniform distribution on interval $[(1 - tf - rdd/2)\sum_{j \in J} p_j, (1 - tf + rdd/2)\sum_{j \in J} p_j]$. This procedure is used in the literature to generate a benchmark set for measuring the quality of algorithms for $1||\sum T_j$ and various related problems. However, we need to design an efficient procedure that creates the training data set with the following properties: (i) the training data set needs to contain training labels (i.e., optimal objective values) for all the samples, (ii) the distribution of training instances should reflect what will be encountered during the inference, and (iii) the procedure should be able to produce millions of training samples in a reasonable time. In the following lines, we describe two choices of such training data set generators with their properties.

The most straightforward method is *Generate & Solve*. It produces the training data set as follows. First, it generates a random instance by the generator described by Potts *et al.* [37], and then it is solved by TTBM algorithm, which acquires the training label (optimal objective value) for that instance. The pair consisting of the instance and its solution is then treated as a single training sample. The entire training data set is obtained by generating random instances having a uniform distribution of n, *rdd*, and *tf*.

A different method to generate the training data set is named *Subproblem generator*. Its idea is to exploit all subproblems that are being solved by a decomposition-based algorithm (see Section 4.1) during the solution of a single problem instance. For the given input instance J and eligible position k of the splitting

Table 1: Time consumed for generation of a data set with $1.5 \cdot 10^6$ training samples.

(a) *Generate & Solve*

n	time [s]
5-100	16470
5-125	17743
5-150	20006
5-175	23766
5-200	29757

(b) *Subproblem generator*

n	time [s]
75-100	594
100-125	803
125-150	832
150-175	1040
175-200	1020

job $l^{\circ}(J)$ both EDD, SPT decompositions generate two subproblems, i.e. $P^{\circ}(J,k)$ and $F^{\circ}(J,k)$. Since the decomposition is applied recursively, these subproblems generate other subproblems. Thus, from a single run of a decomposition-based algorithm, multiple training instances together with their optimal objective values emerge.

Subproblem generator proceeds as follows. First, it generates a problem instance by the generator described above. The instances are generated with constant *rdd* and *tf* (specifically $rdd = 0.2$ and $tf = 0.6$). The reason for this setting is that the instances with those parameters are the most computationally demanding compared to other (rdd, tf) [39]. The second reason why it is possible to assume only a single (rdd, tf) pair is that the decompositions generate problems $P^{\circ}(J,k)$ and $F^{\circ}(J,k)$ with a different (rdd, tf) than the ones of J. Therefore, the generated data set covers different (rdd, tf) pairs as well.

After an instance is generated, it is solved by a combination of EDD and SPT decomposition where the algorithm always selects the decomposition having the smallest $\overline{K^{\circ}}(J)$, $\circ \in \{\text{EDD}, \text{SPT}\}$. In each step, the instance is recursively decomposed into a set of subproblems $P^{\circ}(J,k)$ and $F^{\circ}(J,k)$ for different eligible positions k of the splitting job. Subsequently, all subproblems are solved by the recursive application of the decomposition. In the end, the optimal solutions of all the subproblems are known; thus, all pairs of subproblems and their solutions are put into the training data set. Therefore, one can get multiple training samples from a single problem instance solved to the optimality. In addition, newly generated subproblems differ in the number of jobs and their characteristics in terms of *rdd* and *tf* parameters, which enriches the composition of the training data set.

Now, let us discuss the properties of the above generators. The *Generate & Solve* method has two main disadvantages. The first one is the time complexity of the generation. Every problem instance results in only a single sample of the training data set. On the other hand, the *Subproblem generator* naturally generates many training samples from a single input instance, which significantly reduces the time needed for the training data set synthesis. Table 1 shows the times needed for generating data sets with $1.5 \cdot 10^6$ training samples for different intervals of instance sizes. Comparing the two methods, it can be seen that *Subproblem generator* can generate the training data set more than twenty times faster.

The second disadvantage of *Generate & Solve* method is the spectrum of generated instances. While *Generate & Solve* method produces a uniform distribution of instances with respect to n, *rdd*, and *tf*, *Subproblem* *generator* generates a spectrum of instances focused on the needs of HORDA. Indeed, the distribution of instances in the training data set produced by *Subproblem generator* is similar to the distribution of subproblems whose objective value needs to be estimated during the run of HORDA (which is far from being uniform in terms of n, rdd, and tf).

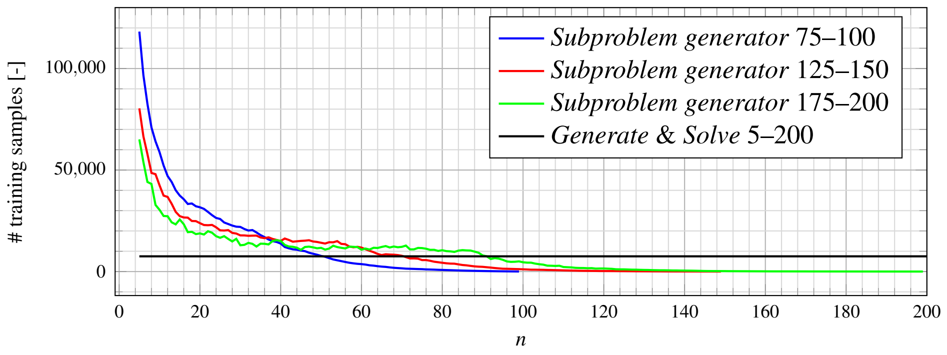


Figure 2: Distribution of training sample size for *Subproblem generator* method with different range of instances and *Generate & Solve*.

The composition of instances in the training data set needs to be analyzed from two perspectives. This first one is the number of instances with a particular n value. The distribution of the training samples' sizes is shown in Figure 2 on a data set with $1.5 \cdot 10^6$ training samples. The figure compares *Generate & Solve* for $n \in [5, 200]$ and *Subproblem generator* for three different ranges on n, i.e., $[75, 100]$, $[125, 150]$ and $[175, 200]$. It shows that while the number of instances for different n is constant for *Generate & Solve*, *Subproblem generator* generates significantly more instances with lower n. It indicates that decompositions inside HORDA need more examples with lower n; thus, these instances are more important for the training of the neural network.

The second aspect that needs to be studied is the distribution of rdd and tf parameters of instances generated by both methods. In this case, the situation is similar to the case with n. While instances with defined (rdd, tf) parameters are uniformly distributed in the training data set produced by *Generate & Solve* method, the distribution of (rdd, tf) is much more uneven in the case of *Subproblem generator* which is demonstrated in Figure 3. It illustrates the frequency of samples with a particular value of (rdd, tf) arising from subproblems generated from 20 instances having 150 jobs with $rdd = 0.2$ and $tf = 0.6$. The samples are divided into particular categories according to the number of jobs n.

For $n = 100$, one can see that the instances are close to the initial setting of $rdd = 0.2$ and $tf = 0.6$. With the decreasing value of n (which corresponds to smaller subproblems), the distribution of rdd and tf shifts significantly. The mass of the samples for the whole data set drifts towards $rdd = 0.4$ and $tf = 0.5$ and the covariance of (rdd, tf) increases as well. These observations underline that the composition of instances that occur during a single run of HORDA is significantly different from a uniform distribution produced by *Generate & Solve*. At the same time, it shows clear advantages of data sets produced by *Subproblem generator* since their distribution is closer to what HORDA encounters. For more details, see the comparison of these two approaches in Table 2, Figure 4, and Figure 5 in Section 5.2.

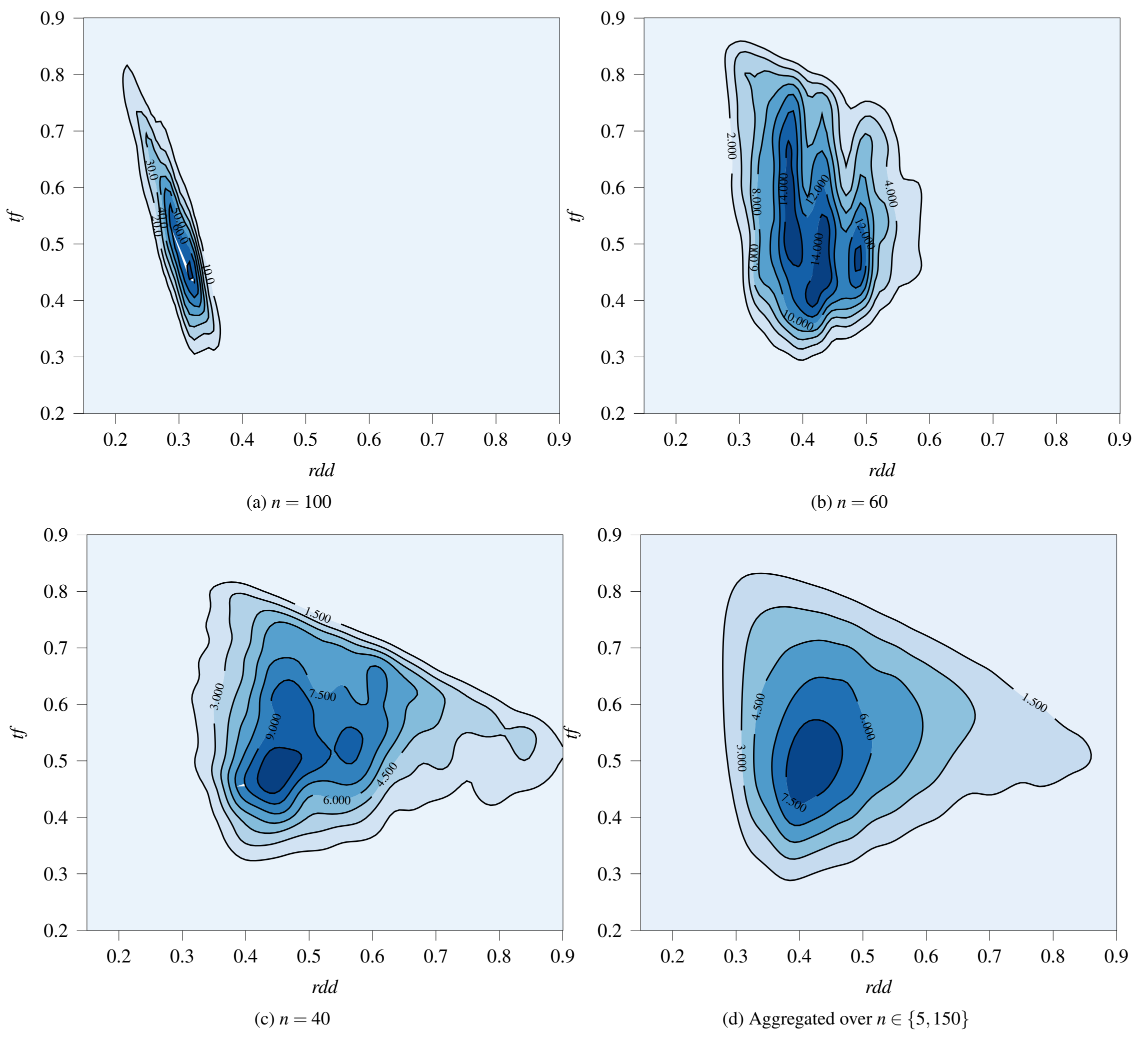


Figure 3: Distribution of *rdd* and *tf* over *n* in the data set generated by *Subproblem generator*.

4.5. *Time Complexity of the Scheduling Algorithm*

In this section, we present an analysis of the worst-case run time of HORDA. The most time-consuming part of HORDA is the estimation of the optimal position for the splitting job k^*. To do so, the algorithm tests at most n positions. For each position the algorithm uses the regressor to compute estimates $P^\circ(J,k)$ and $F^\circ(J,k)$ having $O(n)$ time complexity. Thus, the estimation of the optimal position takes $O(n^2)$. In the worst case, when the decomposition removes only one job from J in every recursion, HORDA makes $O(n)$ selections of position k^*. Therefore, the worst-case time complexity of HORDA is $O(n^3)$. Nevertheless, as the experiments in Section 5.2 show, the algorithm is very fast. The average CPU time on instances with 800 jobs is below 15 s.

5. Experimental Results

In this section, we present the experimental results related to HORDA focused on its relation to the neural network used in the regressor. The section is organized as follows. First, we describe the hardware and software used for the evaluation and the method of generating testing instances. Next, we present the comparison of our approach with the state-of-the-art heuristic, the state-of-the-art exact algorithm, and results from our previous work [9]. The following sections describe a detailed benchmarking of the algorithm. Section 4.4 presents the experiments with the generation of the training data set. The influence of neural network hyperparameters on HORDA is investigated in Section 5.3, and Section 5.4 describes sensitivity of HORDA to its parameters.

5.1. Experimental setup

Experiments were run on a single CPU core of the Xeon Gold 6140 processor with a memory limit set to 8 GB of RAM. HORDA and NBR algorithms were implemented in Python 3.7, and the neural network is implemented using TensorFlow 2 trained on a GPU Nvidia GTX 1080 Ti. Source code of HORDA is available at GitHub repository (https://github.com/CTU-IIG/horda); Source code of TTBM algorithm was provided by authors of [18] and is implemented in C++.

Instances used in this paper were generated in the manner suggested by Potts and Van Wassenhove [37]. They generate processing times of jobs uniformly on the interval from 1 to 100. Since we want to study the impact of processing time on the quality of our algorithm, we define maximal processing time p_{max} and generate the processing time of jobs uniformly on the interval from 1 to the p_{max}. The analysis presented in [39] implies that the hardest instances occur for $rdd = 0.2$ and $tf = 0.6$; therefore, our experiments focus on them. We use two data sets: the first with $p_{max} = 100$, and the second, with $p_{max} = 5000$, to mimic the duration of jobs in seconds used, e.g., nowadays in Advanced Planning and Scheduling systems. In all experiments, we have used separate data sets for training and evaluation; thus, all the methods are tested on instances that were not seen during the training.

For all graphs presented in this section, the points in the figure represent the mean over all instances of the same size. The colored surface represents the standard deviation of the measurement, and the line represents the running mean of the last five values. The evaluation data sets consist of 20 randomly generated instances for each n divisible by 5. The results are represented in a form of the optimality gap defined as $\frac{T(J,\pi)-T^*(J)}{T(J,\pi)} \cdot 100$, where π is a heuristic solution and $T^*(J)$ is the optimal objective value. Next, note that the training of neural networks is a random process (e.g., data set shuffling, initial weights, numerical instability), and the results vary over different runs. Thus, for each settings of neural network hyperparameters, we trained all neural networks five times with the same parameters and used the best one to carry out the experiments.

5.2. Comparison with state-of-the-art approaches

In this section, we present a comparison of our results with state-of-the-art approaches in terms of the optimality gap and run time. Our results are compared with NBR [21] state-of-the-art heuristic, its combination inside the decomposition HORDA (NBR) [13], TTBM [18] as the state-of-the-art exact algorithm, and our previous ML based approach denoted as HORDA (NN2020) [9].

For a fair comparison of all methods, we limited the run time of TTBM to 10 s, and 15 s. Time limit 15 s corresponds to the maximum time needed by HORDA to solve the largest instances. Notice that this gives an advantage to TTBM on small instances, but it makes the comparison simpler. Thus the comparison is made in favor of TTBM. We report these as TTBM(10 s), and TTBM(15 s) respectively.

Results of the presented approach are also compared with our previous approach HORDA (NN2020), presented in [9]. It utilizes HORDA heuristic with the LSTM based neural network (with capacity 256 and SUMPROC criterion normalization) trained on a data set generated by *Generate & Solve* (see Section 4.4).

HORDA (BEST) method, proposed in this paper, uses the regressor, as it is presented in Figure 1. The recurrent layer is LSTM with a capacity of 256. The neural network uses GAPEDDINV normalization, and it is trained on a data set generated with *Subproblem generator*. The data set was generated from input instances having $n \in [75, 100]$ while for each n there were 20 instances used by *Subproblem generator* to generate the training data set. In total, the data set consists of $1.6 \cdot 10^6$ instances.

First, let us discuss the time needed to train the neural network used in HORDA (BEST). The generation of the training data set takes about 600 s, and it is discussed in detail in Section 4.4. The training time of the neural network is about 3 hours. It comes as no surprise that the neural network training time is large compared to the time needed to solve a single instance by TTBM. The approach presented here assumes that the time needed for the neural network training is a part of the algorithm development, but the online execution of the algorithm must be fast. Indeed, the training time is spent just once, whereas the inference from the neural network after the training is much faster.

Table 2 compares the run times and optimality gaps of TTBM, TTBM(10 s), TTBM(15 s), NBR, HORDA (MDD), HORDA (NBR), HORDA (NN2020), and HORDA (BEST) on instances generated with $p_{max} = 5000$. The bold numbers in the table represent the best result on the interval of n in terms of the optimality gap. The time required for computing the optimal solution by TTBM is shown in the second column of the table. For the largest instances with $p_{max} = 5000$, the computation takes up to $15 \cdot 10^3$ s. Due to a software issue, we were not able to solve larger instances by TTBM. TTBM with limited time (TTBM(10 s), TTBM(15 s), i.e., third and fourth column, respectively) has small gaps on small instances, but from $n = 250$ it perform worse than HORDA (BEST). The average gap on the biggest instance is 3.92% for TTBM(10 s), and 3.75% for TTBM(15 s). NBR heuristic has the optimality gap almost independent of the size of instances achieving an average gap 2.14% over all instances (see the fifth column in the table). Nevertheless, this result is significantly worst compared to the gap obtained by HORDA (BEST). On the other hand, the average run time of NBR heuristic for the biggest instances is around 0.85 s compared to HORDA (BEST) achieving 13.95 s.

As noted by Della Croce *et al.* [13], constructive heuristics such as MDD [8] and NBR can be embedded into the decomposition utilized by HORDA as well. Thus, HORDA (MDD) and HORDA (NBR) columns present the integration of the respective constructive heuristic into HORDA, where it acts as a regressor replacing the neural network. The embedded MDD heuristic has an average gap 1.89% (over all instances), and the quality is slightly better than NBR. The integration of NBR into HORDA (NBR) improves the gap from 2.14% to 1.25% (over all instances). Nevertheless, HORDA (BEST) has a significantly lower run time and almost five times lower gap. The table illustrates that within the 15 seconds time limit for instances having more than 250 jobs, the best results were achieved by HORDA (BEST).

Table 3 compares the results of TTBM, HORDA (BEST) and a genetic algorithm presented by Suer *et al.* [41]. The Suer *et al.* defines its own evaluation protocol to generate test instances. Therefore, we evaluate TTBM and HORDA (BEST) on instances generated in the same manner as suggested in [41] and show the results

Table 2: Comparison with state-of-the-art approaches on instances with $p_{max} = 5000$.

n	TTBM (∞ s)	TTBM (10 s)	TTBM (15 s)	NBR	HORDA (MDD)	HORDA (NBR)		HORDA (NN2020)	HORDA (BEST)	
	time [s]	gap [%]	gap [%]	gap [%]	gap [%]	gap [%]	time [s]	gap [%]	gap [%]	time [s]
5 – 45	0.02 ±0.00	0.00 ±0.00	**0.00** ±0.00	0.95 ±2.92	0.38 ±0.87	0.09 ±0.28	0.01 ±0.01	0.56 ±0.86	0.22 ±0.51	0.03 ±0.10
50 – 95	0.03 ±0.03	0.00 ±0.00	**0.00** ±0.00	1.22 ±0.91	0.97 ±1.38	0.36 ±0.48	0.09 ±0.05	0.64 ±0.42	0.22 ±0.29	0.18 ±0.08
100 – 145	0.34 ±0.55	0.00 ±0.00	**0.00** ±0.00	1.59 ±0.70	1.11 ±1.04	0.71 ±0.48	0.31 ±0.13	0.46 ±0.31	0.39 ±0.40	0.46 ±0.16
150 – 195	2.30 ±2.34	0.01 ±0.07	**0.00** ±0.03	1.86 ±0.64	1.39 ±1.20	0.92 ±0.45	0.66 ±0.22	0.47 ±0.33	0.50 ±0.45	0.93 ±0.22
200 – 245	11.09 ±10.69	0.17 ±0.30	**0.07** ±0.18	1.99 ±0.65	1.54 ±1.12	1.05 ±0.48	1.25 ±0.41	0.59 ±0.29	0.45 ±0.29	1.56 ±0.31
250 – 295	38.37 ±27.57	0.82 ±0.68	0.56 ±0.57	2.08 ±0.52	1.60 ±1.13	1.19 ±0.44	2.07 ±0.66	0.55 ±0.27	**0.37** ±0.24	2.30 ±0.39
300 – 345	93.19 ±59.73	1.32 ±0.80	1.02 ±0.72	2.29 ±0.54	1.75 ±1.19	1.33 ±0.45	2.96 ±0.88	0.57 ±0.35	**0.31** ±0.17	3.07 ±0.43
350 – 395	209.70 ±113.55	1.99 ±0.87	1.68 ±0.81	2.35 ±0.48	2.06 ±1.34	1.42 ±0.45	4.41 ±1.25	1.16 ±0.60	**0.27** ±0.16	4.03 ±0.51
400 – 445	464.90 ±246.80	2.62 ±0.87	2.33 ±0.83	2.36 ±0.45	2.31 ±1.34	1.51 ±0.41	6.00 ±1.75	1.72 ±0.62	**0.24** ±0.15	5.01 ±0.63
450 – 495	814.21 ±398.34	2.83 ±0.83	2.55 ±0.78	2.43 ±0.45	2.33 ±1.15	1.56 ±0.41	7.70 ±2.15	1.32 ±0.60	**0.22** ±0.13	6.10 ±0.79
500 – 545	1528.93 ±620.02	3.23 ±0.80	2.98 ±0.80	2.43 ±0.43	2.34 ±1.24	1.54 ±0.38	10.28 ±2.88	1.28 ±0.51	**0.20** ±0.14	7.19 ±0.80
550 – 595	2578.16 ±1128.52	3.49 ±0.71	3.27 ±0.70	2.52 ±0.41	2.34 ±1.12	1.66 ±0.44	12.73 ±3.00	1.20 ±0.48	**0.18** ±0.12	8.50 ±0.89
600 – 645	4436.89 ±2341.48	3.70 ±0.70	3.46 ±0.70	2.54 ±0.37	2.37 ±1.15	1.62 ±0.33	16.13 ±3.90	1.20 ±0.41	**0.16** ±0.08	9.82 ±0.97
650 – 695	7311.81 ±3689.10	3.86 ±0.79	3.65 ±0.78	2.54 ±0.40	2.53 ±1.23	1.66 ±0.37	19.94 ±4.92	1.05 ±0.50	**0.15** ±0.09	11.18 ±1.08
700 – 745	11390.34 ±4360.18	3.90 ±0.75	3.71 ±0.73	2.56 ±0.41	2.57 ±1.09	1.74 ±0.38	22.96 ±6.18	1.01 ±0.37	**0.16** ±0.09	12.71 ±1.20
750 – 795	14135.70 ±4814.88	3.92 ±0.64	3.75 ±0.64	2.59 ±0.31	2.64 ±0.91	1.65 ±0.37	28.79 ±6.96	0.89 ±0.41	**0.11** ±0.08	13.95 ±1.29
avg	2688.50 ±1113.36	1.99 ±0.55	1.81 ±0.52	2.14 ±0.66	1.89 ±1.16	1.25 ±0.41	8.52 ±2.21	0.92 ±0.46	**0.26** ±0.21	5.44 ±0.61

Table 3: Comparison with a Genetic Algorithm [41].

	TTBM	GA [41]		HORDA (BEST)	
n	time [s]	gap [%]	time [s]	gap [%]	time [s]
10	0.011 ±0.001	0	2	0 ±0	0 ±0
20	0.012 ±0.002	0	51	0 ±0	0.06 ±0.004
30	0.012 ±0.002	0	354	0.101 ±0.101	0.013 ±0.005
50	0.013 ±0.002	2.12	536	0.006 ±0.006	0.018 ±0.007
100	0.013 ±0.002	6.32	1083	0.002 ±0.002	0.044 ±0.014

in the separate table. The run time of the genetic algorithm is scaled by the power ratio between the processor used by Suer *et al.* and us, i.e., 0.81. The run time of TTBM is 0.013 s for instances with 100 jobs, while the genetic algorithm has about 1083 s (see column GA in the table). Our method obtains the solutions with an average gap 0.002% within the 0.044 s for instances with 100 jobs, which is superior to the results reported by Suer *et al.*

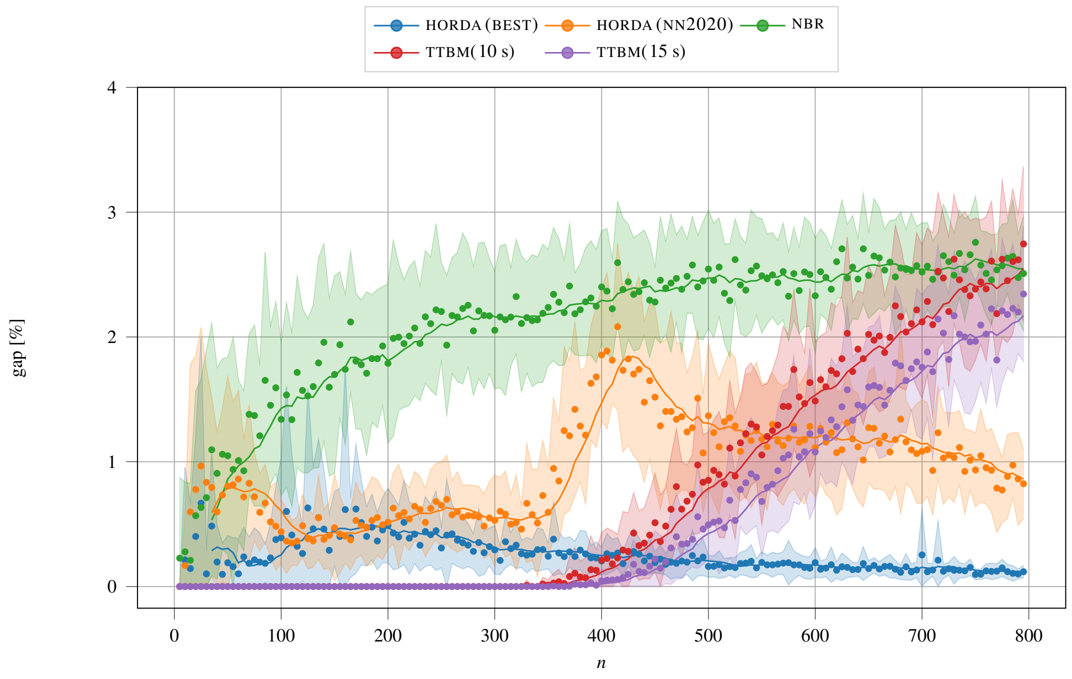


Figure 4: Optimality gap for instances with $p_{max} = 100$.

The dependency of the optimality gap on the number of jobs for $p_{max} = 100$ and $p_{max} = 5000$ is shown in Figure 4 and Figure 5, respectively. The figures show the optimality gap of NBR, TTBM(10 s), TTBM(15 s), HORDA (NN2020), and HORDA (BEST) on n depending on n from 5 to 800. By comparing these two graphs, one can see that results of TTBM are the best on the smaller instances. On the other hand, the TTBM run time is heavily dependent on p_{max}; while other methods do not depend on p_{max}, which provides an advantage to them.

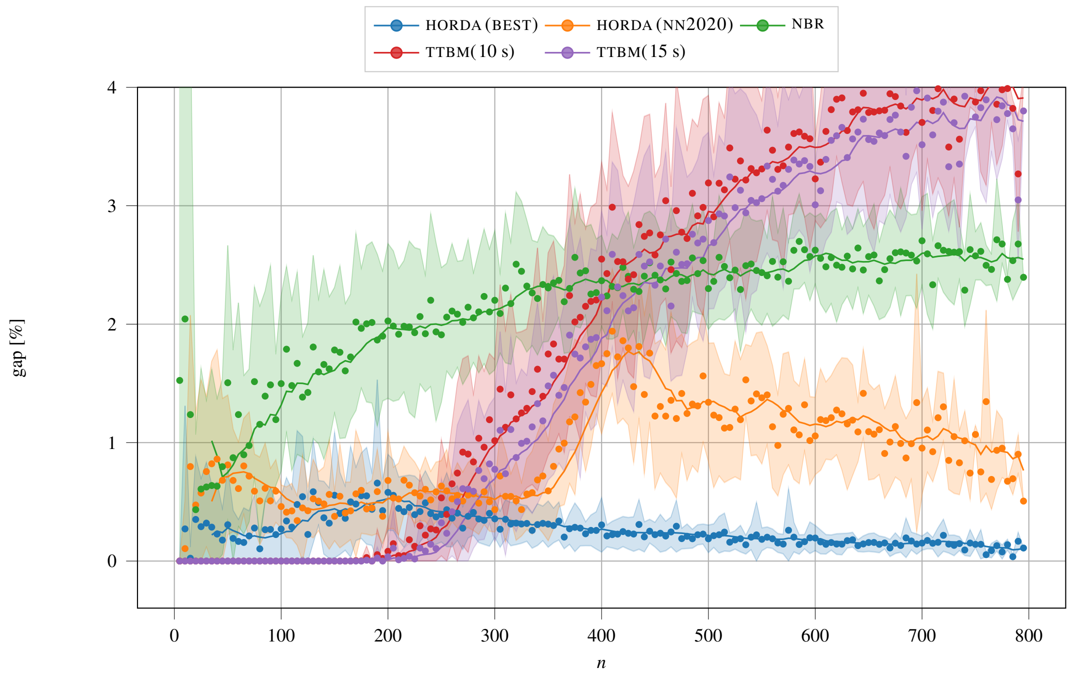


Figure 5: Optimality gap for instances with $p_{max} = 5000$.

Our previous method HORDA (NN2020) has very good results for instances with no more than 350 jobs, and its gap is less than 1%. The method presented in this paper, i.e., HORDA (BEST) has the optimality gap for all sizes of instances under 0.5%, i.e., it is superior to all other methods for instances with about more than 450 jobs for $p_{max} = 100$, and 250 jobs for $p_{max} = 5000$. Neither the run time nor the gap of HORDA (BEST) depends on the p_{max}. Indeed, thanks to the normalization of the input data and the generalization capabilities of the neural network, it is possible to train it on the data set with $p_{max} = 100$ and apply it on instances generated with $p_{max} = 5000$. Thus, even thought HORDA (BEST) is trained on instances generated for $p_{max} = 100$, the results in terms of optimality gap are almost the same for the $p_{max} = 5000$ as for $p_{max} = 100$. This provides us some confidence that HORDA (BEST) is able to generalize for instances with different p_{max}. Moreover, an advantage of our approach is that the training data set can be generated for $p_{max} = 100$, which is much faster than the generation of the training data set with $p_{max} = 5000$.

5.3. Neural network hyperparameters

In this section, we analyze the impact of the neural network on the performance of HORDA and on the quality of its solutions. Specifically, we study the impact of the LSTM capacity, GRU as an alternative to LSTM, the number of instances in the training data set, and the size of instances (i.e., n) used for training. The experiments listed below assume HORDA (BEST) as the baseline scenario. This means that every experiment varies one hyperparameter of the neural network while the others are set as in HORDA (BEST).

At first, we present the results of LSTM with different capacities. The size of the LSTM layer impacts the run time of HORDA and the ability of the neural network to fit on the training data and their generalization. The optimality gaps of HORDA with the neural networks having LSTM capacity 32, 64, 128, 256 are shown in Figure 6. For those capacities, the number of trainable parameters inside the neural network is equal to

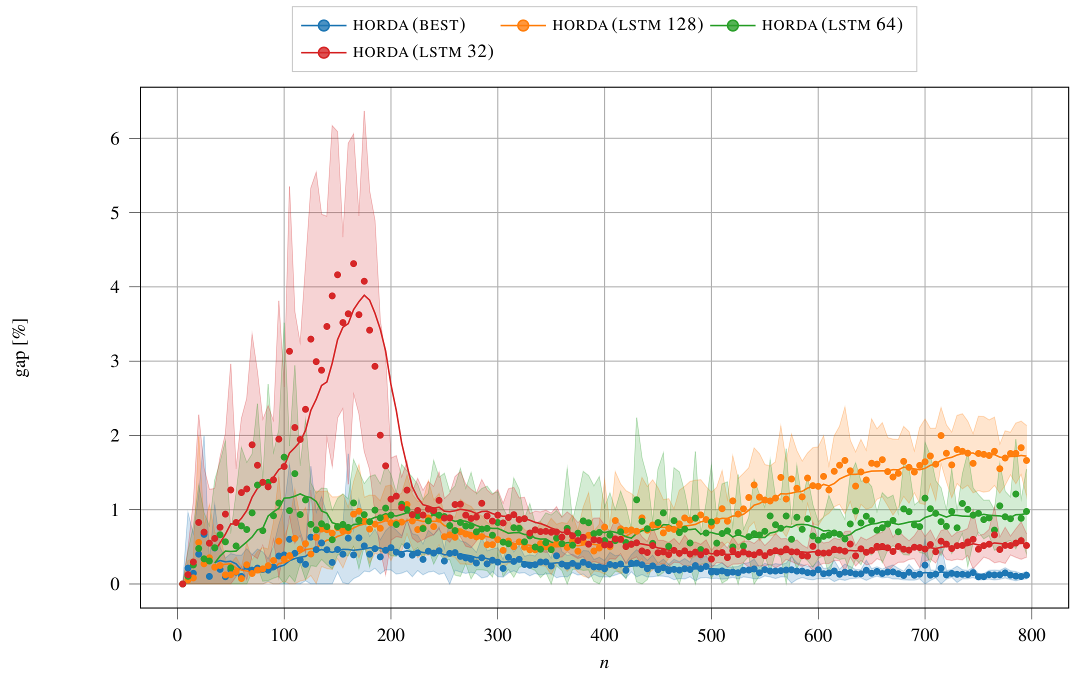


Figure 6: Optimality gap of HORDA with different capacity of LSTM as a regressor.

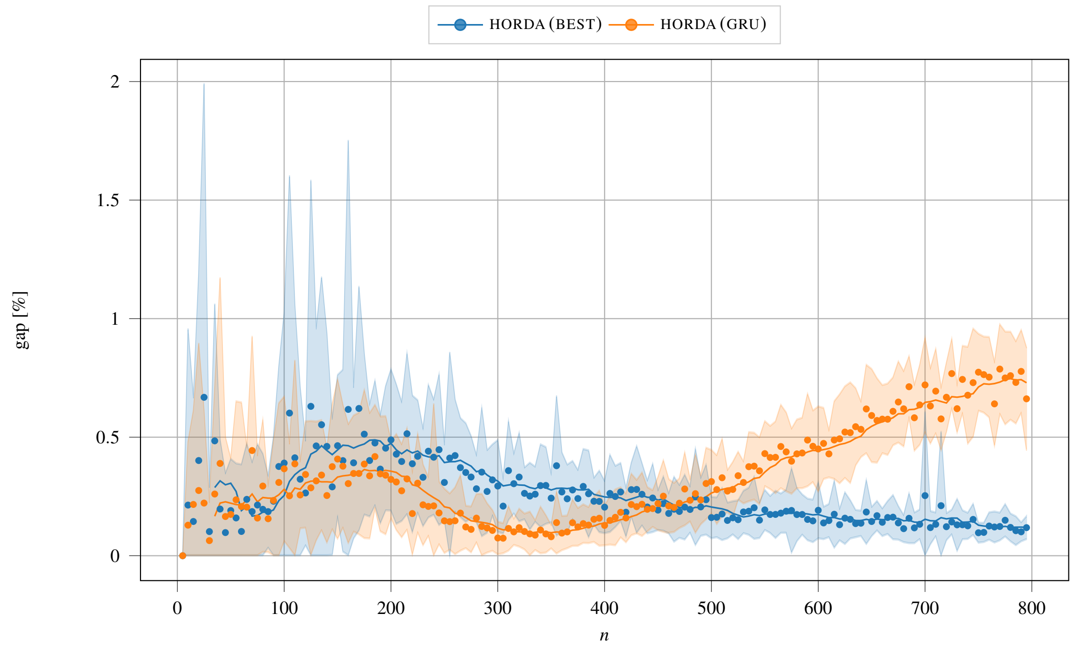


Figure 7: Optimality gap of HORDA with different neural networks.

4500, $1.7 \cdot 10^4$, $6.7 \cdot 10^4$, $2.6 \cdot 10^5$, respectively. Let us recall that the best model has a capacity equal to 256, and results show that it performs the best. The average ability to fit the training data is decreasing with the decreasing capacity of the LSTM layer, and thus the optimality gap of HORDA grows. On the other hand, larger capacities than 256 would require enormous training data set due to a huge number of training parameters.

An alternative to the LSTM layer used in the neural network is the GRU layer. GRU is a more restricted architecture (than LSTM), which can lead to a weaker ability of generalization, whereas the LSTM architecture can profit from a large number of training samples. The comparison of the best model and neural network with

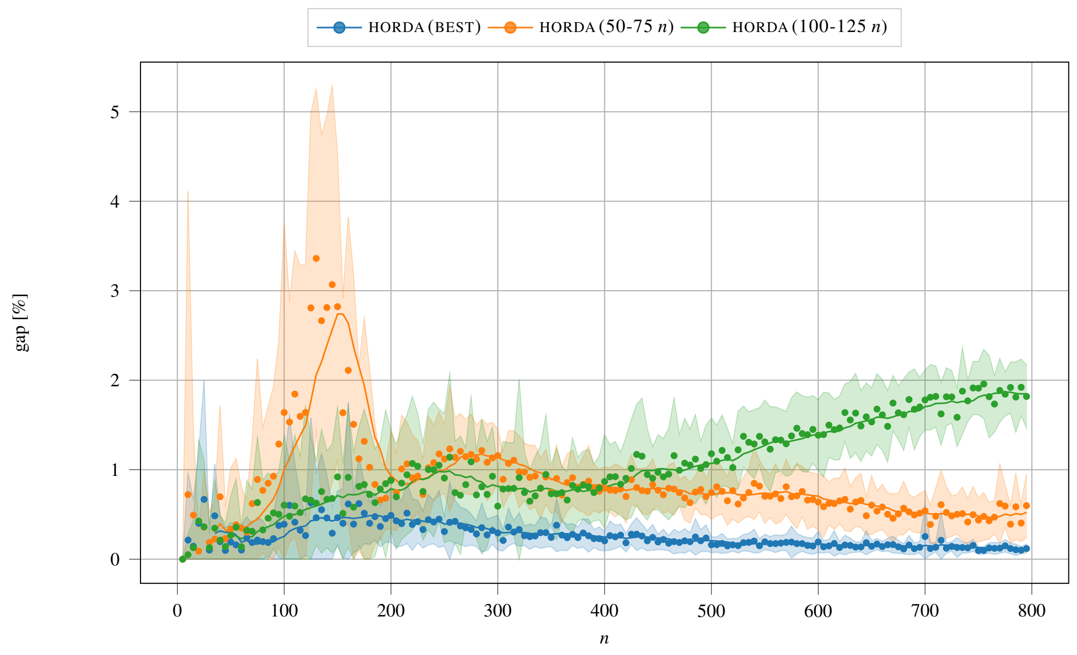


Figure 8: Optimality gap of HORDA with a different number of jobs in training instances.

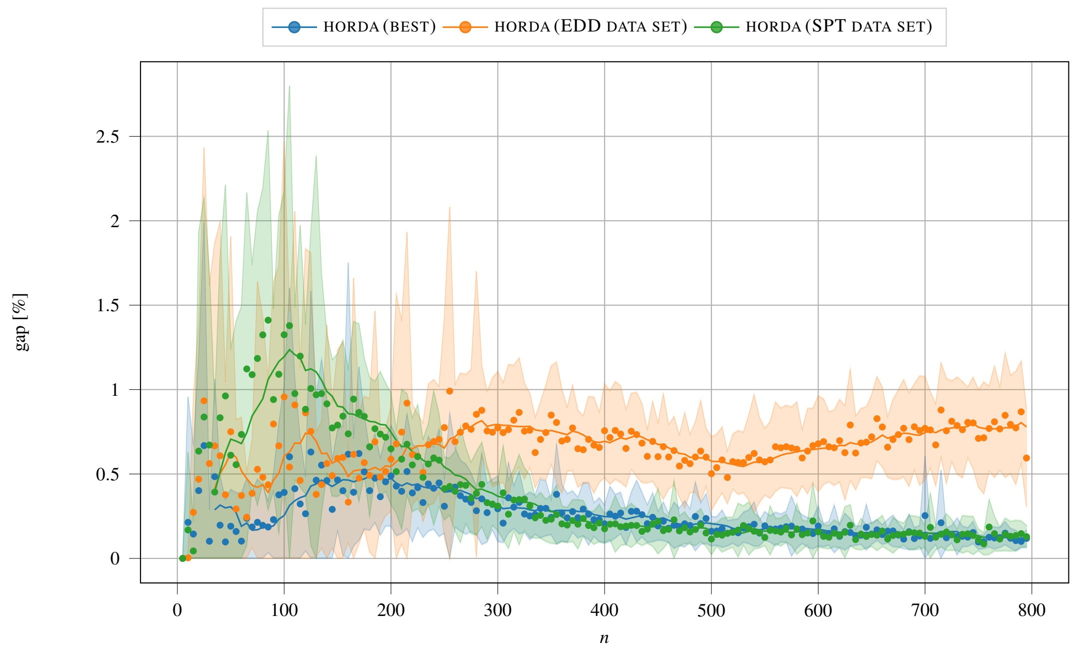


Figure 9: Optimality gap of HORDA with different decomposition used to generating training data set.

the GRU layer is shown in Figure 7. HORDA (GRU), i.e., the algorithm with the neural network containing the GRU layer, provides solutions with similar quality in terms of the optimality gap for instances with up to 450 jobs. For the instances with more than 450 jobs, the optimality gap grows up to 1%. Therefore, for our purposes is better to use LSTM, since it is computationally inexpensive to generate large training data sets with the *Subproblem generator* method.

The following experiment, illustrated in Figure 8, evaluates the impact of the number of jobs n of instances used to generate the training data set. The experiment assumes *Subproblem generator*, where the training used input instances with 50 - 75, 75 - 100, and 100 - 125 jobs. For each size of input instances, we generate 20

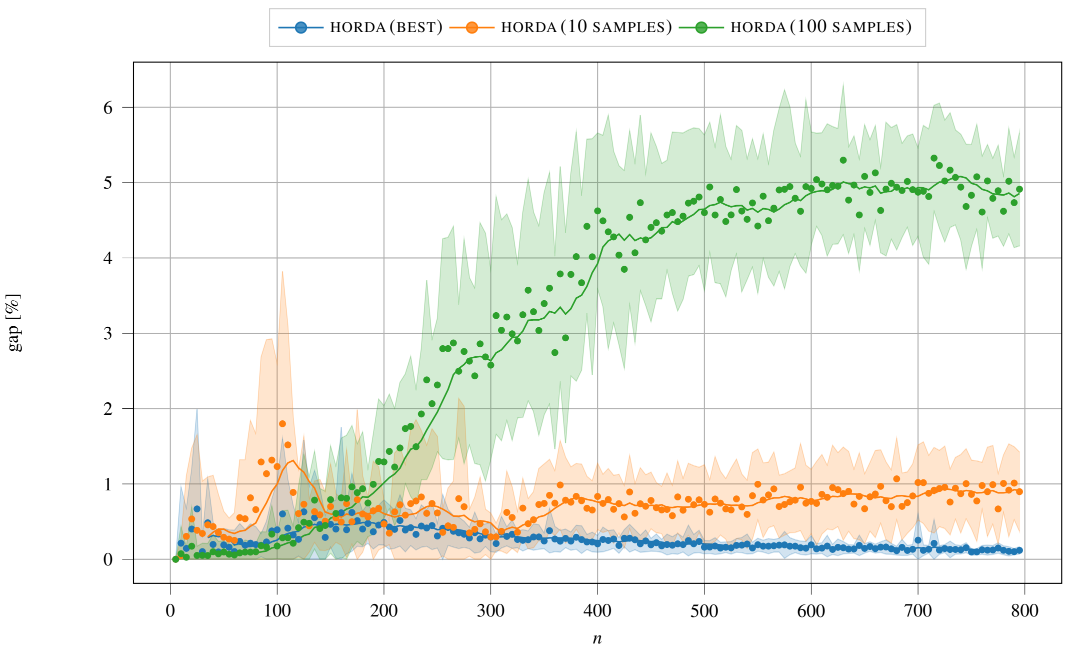


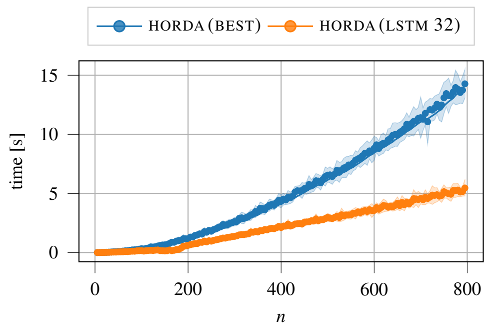
Figure 10: Optimality gap of HORDA with a different number of training instances for different n.

instances and generate the training samples that include subproblems as it is described in Section 4.4. The best model is trained with input instances of size 75 - 100, i.e., HORDA (BEST). HORDA with a neural network trained on the data set generated from instances with 100 - 125 jobs has a slowly growing optimality gap. For the training on the data set with smaller input instances 50 - 75, we observe the peak on the optimality gap at 150. A similar peak on the gap curve was observed even in other experiments; however, it is interesting how the change of the training instances' size affects the position of the peak and its width. Nevertheless, HORDA (BEST) is superior to either HORDA (50-75 n), and HORDA (100-125 n).

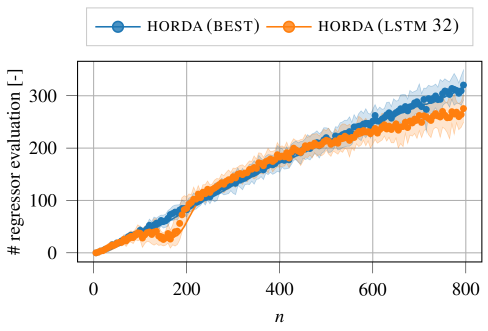
The training data set can be created by *Subproblem generator* using EDD decomposition, SPT decomposition, or their combination denoted *shorter* where the algorithm always selects the decomposition having the smallest $\overline{K^{\circ}}(J)$. Those three possibilities are denoted as HORDA (EDD DATA SET), HORDA (SPT DATA SET), and HORDA (BEST), respectively. The optimality gap of HORDA using neural networks trained on data set generated with different decompositions is shown in Figure 9. HORDA (BEST) has the smallest optimality gap, as it utilizes the same decomposition for the generation of the training data set and the evaluation. This underlines the importance of training a neural network on the data with the same distribution as in the evaluation phase achieved by *Subproblem generator* generator.

A common practice of improving the performance of the neural network is enlarging the training data set; therefore, the following experiment is focused on the effect of its size. The neural network was trained on sub-instances generated by *Subproblem generator* where for each $n \in [75, 100]$ we generated 10, 20, 100 input instances. In other words, the data sets consist $7.5 \cdot 10^5$, $1.6 \cdot 10^6$, and $7.7 \cdot 10^6$ training samples, respectively. The corresponding experiments shown in Figure 10 are denoted as HORDA (10 SAMPLES), HORDA (BEST), and HORDA (100 SAMPLES). HORDA (10 SAMPLES) has the optimality gap under 1%. By increasing the training data set's size, we get HORDA (BEST) and the worst gap decreases under 0.5%. HORDA (100 SAMPLES) has the smallest optimality gap for the instances with up to 125 jobs. However, for larger instances, the optimality gap grows rapidly. It is important to observe that HORDA (100 SAMPLES) has the optimality gap smaller than HORDA (BEST) on the range of instances from the training data set, i.e., instances with 75 – 100 jobs. Thus, even though it seems counter-intuitive at first, increasing the training data set the size above a certain critical level can lead to overfitting, resulting in worse prediction performance of the model on the instances that do not lay in a range of the training instances.

5.4. Parameters of the Scheduling Algorithm

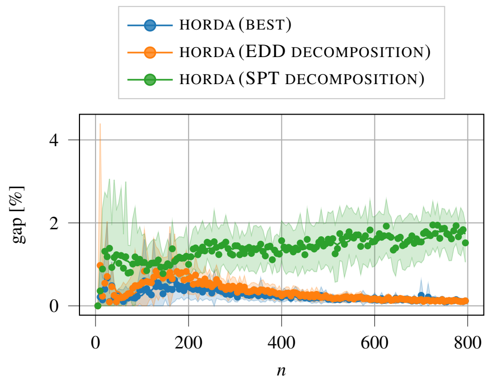


(a) Run time of HORDA with different sizes of LSTM as the regressor.

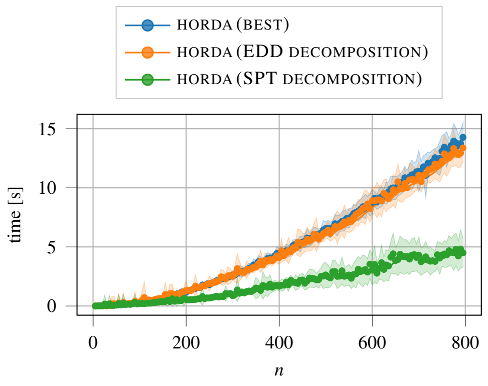


(b) The number of regressor evaluations in HORDA with different sizes of LSTM as a regressor.

Figure 11: Impact of the LSTM capacity on the run time and number of regressor calls in HORDA.



(a) Optimality gap of HORDA with different (EDD, SPT, *shorter*) decompositions used during evaluations.



(b) Run time of HORDA with different (EDD, SPT, *shorter*) decomposition used in evaluations.

Figure 12: Impact of the decomposition used in HORDA to the optimality gap and run time.

In this section, we study the impact of the HORDA parameters on its performance. At first, we analyze how the time of the inference from the neural network affects the run time of HORDA. Then, we compare results for different decomposition rules used in HORDA.

In the first experiment, we present the impact of the neural network inference time on the run time of HORDA. For the illustration, we use two neural networks with different inference times. The inference time of our neural network is primarily affected by the capacity of the LSTM layer. The faster neural network has a capacity equal to 32. The slower one has a capacity 256, which corresponds to HORDA (BEST) scenario. The other parameters are set according to HORDA (BEST) scenario. The run times of HORDA assuming capacity 32 and 256 (denoted HORDA (LSTM 32) and HORDA (BEST), respectively) are shown in Figure 11a.

HORDA (LSTM 32) is about three times faster on the instance with 800 jobs than HORDA (BEST), and this difference increases with the size of the instance. Apart from that, HORDA (LSTM 32) run time slightly drops for instances around 150 jobs. This is caused by a lower number of neural network evaluations, illustrated in Figure 11b, where the drop near 150 jobs is obvious even more. The lower number of evaluations is connected with a phenomenon we observed in Figure 6 that compares the optimality gap for different capacities of LSTM. In that figure, one can see a sudden deterioration of the solution quality for instances around $n = 150$. The most probable explanation of the correlation between the quality of results and the number of regressor evaluations is that HORDA with a poor regressor makes some decisions that lead to subproblems where filtering rules dramatically reduce the candidate set $K^{\circ}(J)$. Due to this, we see fewer regressor evaluations in Figure 11b. However, these choices are suboptimal and lead to particularly degenerative solutions with poorer quality.

The last experiment presents the impact of the decomposition used in HORDA on the quality of the solutions and run times. Specifically, it compares the cases when position sets $K^{\circ}(J)$ are generated either by EDD, SPT, or *shorter* decomposition. Figure 12a and Figure 12b show the solution quality and run time for different decompositions used in HORDA. HORDA (BEST) utilizes *shorter* decomposition, HORDA (SPT DECOMPOSITION) utilizes the SPT decomposition, and HORDA (EDD DECOMPOSITION) utilizes the EDD decomposition. The quality of HORDA (SPT DECOMPOSITION) solutions is inferior to other methods; the results of HORDA (BEST) and HORDA (EDD DECOMPOSITION) are similar. This is caused by the fact that during the evaluation of HORDA with the *shorter* decomposition, the EDD decomposition is selected more frequently than the SPT decomposition. Since the neural network is trained on the data set generated by *shorter* decomposition, the data set contains fewer samples related to SPT decomposition. This property can lead to relatively small differences in results between the EDD and *shorter* decomposition and is significantly different from the SPT decomposition. HORDA (BEST) provides a better solution than HORDA (EDD DECOMPOSITION), mainly for the instances with less than 300 jobs; for the bigger instances, the difference is negligible.

6. Conclusion

To the best of our knowledge, this is one of the first scheduling algorithms where deep learning is successfully used to guide solution-space exploration. Our approach lies in the synergy between the state-of-the-art operations research method and our neural network. This is opposite to the classical approach in ML, e.g., of Vinyals et al. [49] with an end-to-end approach for Traveling Salesman Problem. For the single machine scheduling problem minimizing total tardiness, we show how a neural network can extend standard decomposition techniques. Besides, we provide an efficient way to generate the training data set, which is a very time costly operation for combinatorial problems. The experimental results show that our approach provides near-optimal solutions very quickly and is also able to generalize the acquired knowledge to larger instances without significantly affecting the quality of the solutions. Our approach has an average gap 0.26% for instances with up to 800 jobs and outperforms state-of-the-art constructive heuristic NBR with gap 2.14%, as well as the decomposition-based heuristic having gap 1.25%. Moreover, with limited time to 15 s the state-of-the-art exact algorithms [18] have an average gap 1.81% and is also dominated by our approach in this scenario.

We believe that the proposed methodology opens new possibilities for the design of efficient heuristics algorithms where the manual tuning of the heuristic is substituted by automatic ML. Therefore, future research should address other simple scheduling problems that cannot be efficiently decomposed, like $1||\sum T_j$. One possibility may be problem $1||\sum w_j T_j$ which is still simple and suitable for neural networks. Another research direction is the generation of the training data set and its efficiency for NP-hard problems. This paper has shown that there are better ways to generate the training instances; nevertheless, it is tailored to problem $1||\sum T_j$.

7. Acknowledgements

The authors want to thank Vincent T'Kindt from Université de Tours for providing the source code of TTBM algorithm.

This work was supported by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466), the Grant Agency of the Czech Republic under the Project GACR 22-31670S, and the EU and the Ministry of Industry and Trade of the Czech Republic under the Project OP PIK CZ.01.1.02/0.0/0.0/20_321/0024399.

Appendix

Table 4: List of used notations.

J	set of jobs (problem instance)
n	number of jobs
p_j	processing time of job j
d_j	due date of job j
π	permutation of jobs J
$T^*(J)$	optimal total tardiness of instance J
$\mathcal{T}_{\pi_k}(J)$	tardiness of job π_k in permutation π of J
$T(J,\pi)$	total tardiness of J under permutation π
$\circ$	problem decomposition (EDD or SPT)
$l^\circ(J)$	splitting job in decomposition $\circ$
k	possible positions of $l^\circ(J)$ in the schedule
$K^\circ(J)$	set of positions k of job $l^\circ(J)$ defined by decomposition $\circ$
$\overline{K^\circ}(J)$	filtered $K^\circ(J)$
$P^\circ(J,k)$	preceding subset of jobs for decomposition $\circ$, set J and position k
$F^\circ(J,k)$	following subset of jobs for decomposition $\circ$, set J and position k
J'	a subproblem (either $P^\circ(J,k) \subset J$ or $F^\circ(J,k) \subset J$)
$Q(J,k)$	optimal total tardiness of J with splitting job $l^\circ(J)$ at position k
$\hat{T}(J)$	estimated total tardiness of J
$\hat{Q}(J,k)$	estimated total tardiness of J with splitting job $l^\circ(J)$ at position k
k^*	position of splitting job $l^\circ(J)$ with minimal $\hat{Q}(J,k)$
$\boldsymbol{X}$	input vector of the neural network
y	output of the neural network
π^{EDD}	permutation of J in EDD order
gap_{EDD}	gap of the EDD schedule w.r.t. to the optimal solution
p_{max}	maximal processing time of a job

Table 5: List of used abbreviations.

LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
ML	Machine Learning
GA	Genetic Algorithm
HORDA	Heuristic Optimizer using Regression-based Decomposition Algorithm
TTBM	Total Tardiness Branch-and-Merge Algorithm
TSP	Traveling Salesman Problem
EDD	Earliest Due Date
SPT	Shortest Processing Time
OR	Operations Research
CSP	Constraint Satisfaction Problem
MDD	Modified Due Date Rule

References

[1] Abe, K., Xu, Z., Sato, I., and Sugiyama, M. Solving NP-Hard problems on graphs with extended AlphaGo Zero. *arXiv preprint arXiv:1905.11623* (2019).

[2] Alicastro, M., Ferone, D., Festa, P., Fugaro, S., and Pastore, T. A reinforcement learning iterated local search for makespan minimization in additive manufacturing machine scheduling problems. *Computers & Operations Research* (2021), 105272.

[3] Antony, S. R., and Koulamas, C. Simulated annealing applied to the total tardiness problem. *Control and Cybernetics 25* (1996), 121–130.

[4] Applegate, D., Bixby, R., Chvatal, V., and Cook, W. Concorde TSP solver, 2006.

[5] Bauer, A., Bullnheimer, B., Hartl, R. F., and Strauss, C. An ant colony optimization approach for the single machine total tardiness problem. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)* (1999), vol. 2, IEEE, pp. 1445–1450.

[6] Ben-Daya, M., and Al-Fawzan, M. A simulated annealing approach for the one-machine mean tardiness scheduling problem. *European Journal of Operational Research 93*, 1 (1996), 61–67.

[7] Bengio, Y., Lodi, A., and Prouvost, A. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research 290*, 2 (2021), 405–421.

[8] Bertrand, J. A dynamic priority rule for scheduling against due-dates. *J. Oper. Management3 (1)* (1982), 37–42.

[9] Bouska, M., Novak, A., Sucha, P., Modos, I., and Hanzalek., Z. Data-driven algorithm for scheduling with total tardiness. In *Proceedings of the 9th International Conference on Operations Research and Enterprise Systems (ICORES)* (2020), INSTICC, SciTePress, pp. 59–68.

[10] Cappart, Q., Moisan, T., Rousseau, L.-M., Premont-Schwarz, I., and Cire, A. Combining reinforcement learning and constraint programming for combinatorial optimization. *arXiv preprint arXiv:2006.01610* (2020).

[11] Cheng, T. E., Lazarev, A. A., and Gafarov, E. R. A hybrid algorithm for the single-machine total tardiness problem. *Computers & Operations Research 36*, 2 (2009), 308–315.

[12] Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Doha, Qatar, Oct. 2014), Association for Computational Linguistics, pp. 1724–1734.

[13] Della Croce, F., Grosso, A., and Paschos, V. T. Lower bounds on the approximation ratios of leading heuristics for the single-machine total tardiness problem. *Journal of Scheduling 7*, 1 (2004), 85–91.

[14] DELLA CROCE, F., TADEI, R., BARACCO, P., AND GROSSO, A. A new decomposition approach for the single machine total tardiness scheduling problem. *Journal of the Operational Research Society 49*, 10 (1998), 1101–1106.

[15] DEUDON, M., COURNUT, P., LACOSTE, A., ADULYASAK, Y., AND ROUSSEAU, L.-M. Learning heuristics for the TSP by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (2018), Springer, pp. 170–181.

[16] DIMOPOULOS, C., AND ZALZALA, A. A genetic programming heuristic for the one-machine total tardiness problem. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)* (1999), vol. 3, IEEE, pp. 2207–2214.

[17] DU, J., AND LEUNG, J. Y. T. Minimizing total tardiness on one machine is NP-Hard. *Mathematics of Operations Research 15*, 3 (Aug. 1990), 483–495.

[18] GARRAFFA, M., SHANG, L., DELLA CROCE, F., AND T'KINDT, V. An exact exponential branch-and-merge algorithm for the single machine total tardiness problem. *Theoretical Computer Science 745* (2018), 133–149.

[19] GRAHAM, R. L., LAWLER, E. L., LENSTRA, J. K., AND KAN, A. R. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of discrete mathematics 5* (1979), 287–326.

[20] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural Comput. 9*, 8 (Nov. 1997), 1735–1780.

[21] HOLSENBACK, J. E., AND RUSSELL, R. M. A heuristic algorithm for sequencing on one machine to minimize total tardiness. *Journal of the Operational Research Society 43*, 1 (1992), 53–62.

[22] HUANG, J., PATWARY, M. M. A., AND DIAMOS, G. F. Coloring big graphs with AlphaGo Zero. *CoRR abs/1902.10162* (2019).

[23] JAIN, A. S., AND MEERAN, S. Job-shop scheduling using neural networks. *International Journal of Production Research 36*, 5 (1998), 1249–1272.

[24] KHALIL, E., DAI, H., ZHANG, Y., DILKINA, B., AND SONG, L. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems* (2017), pp. 6348–6358.

[25] KOOL, W., VAN HOOF, H., AND WELLING, M. Attention, learn to solve routing problems! In *International Conference on Learning Representations* (New Orleans, USA, 2019).

[26] KOULAMAS, C. The single-machine total tardiness scheduling problem: Review and extensions. *European Journal of Operational Research 202*, 1 (2010), 1 – 7.

[27] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems 25* (2012).

[28] LAGUNA, M., DUARTE, A., AND MARTÍ, R. Hybridizing the cross-entropy method: An application to the max-cut problem. *Computers & Operations Research 36*, 2 (2009), 487–498.

[29] LARA-CARDENAS, E., SANCHEZ-DIAZ, X., AMAYA, I., AND ORTIZ-BAYLISS, J. C. Improving hyper-heuristic performance for job shop scheduling problems using neural networks. In *Mexican International Conference on Artificial Intelligence* (2019), Springer, pp. 150–161.

[30] LAWLER, E. L. A "pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. In *Studies in Integer Programming*, P. Hammer, E. Johnson, B. Korte, and G. Nemhauser, Eds., vol. 1 of *Annals of Discrete Mathematics*. Elsevier, 1977, pp. 331–342.

[31] LIN, S., AND KERNIGHAN, B. W. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research 21*, 2 (1973), 498–516.

[32] LUO, S. Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning. *Applied Soft Computing 91* (2020), 106208.

[33] MAZYAVKINA, N., SVIRIDOV, S., IVANOV, S., AND BURNAEV, E. Reinforcement learning for combinatorial optimization: A survey. *CoRR abs/2003.03600* (2020).

[34] MORRISON, D. R., SEWELL, E. C., AND JACOBSON, S. H. An application of the branch, bound, and remember algorithm to a new simple assembly line balancing dataset. *European Journal of Operational Research 236*, 2 (2014), 403–409.

[35] NAIR, V., BARTUNOV, S., GIMENO, F., VON GLEHN, I., LICHOCKI, P., LOBOV, I., O'DONOGHUE, B., SONNERAT, N., TJANDRAATMADJA, C., WANG, P., ET AL. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349* (2020).

[36] PANWALKAR, S., SMITH, M., AND KOULAMAS, C. A heuristic for the single machine tardiness problem. *European Journal of Operational Research 70*, 3 (1993), 304–310.

[37] POTTS, C., AND VAN WASSENHOVE, L. N. Single machine tardiness sequencing heuristics. *IIE Transactions 23*, 4 (1991), 346–354.

[38] RUSSELL, R., AND HOLSENBACK, J. Evaluation of greedy, myopic and less-greedy heuristics for the single machine, total tardiness problem. *Journal of the Operational Research Society 48*, 6 (1997), 640–646.

[39] SHANG, L., T'KINDT, V., AND DELLA CROCE, F. Exact solution of the single machine total tardiness problem: The power of memorization. In *7th International Conference on Industrial Engineering and System Management (IESM 2017)* (2017), pp. 268–272.

[40] SHANG, L., T'KINDT, V., AND DELLA CROCE, F. Branch & memorize exact algorithms for sequencing problems: Efficient embedding of memorization into search trees. *Computers & Operations Research 128* (2021), 105171.

[41] SÜER, G. A., YANG, X., ALHAWARI, O. I., SANTOS, J., AND VAZQUEZ, R. A genetic algorithm approach for minimizing total tardiness in single machine scheduling. *International Journal of Industrial Engineering and Management (IJIEM) 3*, 3 (2012), 163–171.

[42] SUNDERMEYER, M., SCHLÜTER, R., AND NEY, H. LSTM neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association* (2012), pp. 194–197.

[43] SZWARC, W., DELLA CROCE, F., AND GROSSO, A. Solution of the single machine total tardiness problem. *Journal of Scheduling 2*, 2 (1999), 55–71.

[44] SZWARC, W., GROSSO, A., AND CROCE, F. D. Algorithmic paradoxes of the single-machine total tardiness problem. *Journal of Scheduling 4*, 2 (2001), 93–104.

[45] SZWARC, W., AND MUKHOPADHYAY, S. K. Decomposition of the single machine total tardiness problem. *Operations Research Letters 19*, 5 (1996), 243–250.

[46] TANG, Y., AGRAWAL, S., AND FAENZA, Y. Reinforcement learning for integer programming: Learning to cut. In *International Conference on Machine Learning* (2020), PMLR, pp. 9367–9376.

[47] VÁCLAVÍK, R., NOVAK, A., ŠŮCHA, P., AND HANZÁLEK, Z. Accelerating the branch-and-price algorithm using machine learning. *European Journal of Operational Research 271*, 3 (2018), 1055–1069.

[48] VÁCLAVÍK, R., ŠŮCHA, P., AND HANZÁLEK, Z. Roster evaluation based on classifiers for the nurse rostering problem. *Journal of Heuristics 22*, 5 (Oct 2016), 667–697.

[49] VINYALS, O., FORTUNATO, M., AND JAITLY, N. Pointer networks. In *Advances in Neural Information Processing Systems* (2015), pp. 2692–2700.

[50] WASZNIOWSKI, L., KRÁKORA, J., AND HANZÁLEK, Z. Case study on distributed and fault tolerant system modeling based on timed automata. *Journal of Systems and Software 82*, 10 (2009), 1678–1694. SI: YAU.

[51] XU, H., KOENIG, S., AND KUMAR, T. S. Towards effective deep learning for constraint satisfaction problems. In *International Conference on Principles and Practice of Constraint Programming* (2018), Springer, pp. 588–597.

[52] ZHANG, C., SONG, W., CAO, Z., ZHANG, J., TAN, P. S., AND XU, C. Learning to dispatch for job shop scheduling via deep reinforcement learning. *arXiv preprint arXiv:2010.12367* (2020).

[53] ZHOU, D. N., CHERKASSKY, V., BALDWIN, T. R., AND OLSON, D. E. A neural network approach to job-shop scheduling. *IEEE Transactions on Neural Networks 2*, 1 (Jan 1991), 175–179.