TU WIEN Informatics

Minimierung der Produktionsdauer in Fabriken durch maschinelles Lernen

A Learning Beam Search for the No-Wait Flow Shop Scheduling Problem with Release Times

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jonas Mayerhofer, BSc
Matrikelnummer 01633065

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Projektass. Marc Huber, MSc

Wien, 20. April 2022

Jonas Mayerhofer	Günther Raidl

TU WIEN Informatics

Minimizing Makespan in Flow Shops with a Reinforcement Learning like Approach

A Learning Beam Search for the No-Wait Flow Shop Scheduling Problem with Release Times

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jonas Mayerhofer, BSc
Registration Number 01633065

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Projektass. Marc Huber, MSc

Vienna, 20th April, 2022

Jonas Mayerhofer	Günther Raidl

Erklärung zur Verfassung der Arbeit

Jonas Mayerhofer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. April 2022

Jonas Mayerhofer

Danksagung

Ich möchte mich bei allen Personen bedanken, welche mir beim Erstellen dieser Arbeit geholfen haben, die mir mit Rat und Tat zur Seite gestanden sind und welche mich über die Jahre hinweg in meiner universitären Laufbahn unterstützt haben.

Zuerst gilt mein Dank Herrn Prof. Günther Raidl, welcher meine Masterarbeit betreut und begutachtet hat. Danke für die hilfreichen Anmerkungen, die konstruktive Kritik und die investierte Zeit. Ebenso möchte ich mich bei Marc Huber, MSc bedanken. Danke für deine Mitwirkung bei der Betreuung dieser Arbeit und unsere Diskussionen.

Ich danke auch meiner Familie, meinen Studienkollegen und Freunden, die mich während der letzten Jahre begleitet und unterstützt haben. Danke, dass ihr mir ermöglicht und geholfen habt da zu sein, wo ich jetzt bin. Abschließend danke ich auch meiner Freundin. Danke für deine Geduld während des Schreibens an dieser Arbeit und deine Unterstützung durch deine LaTeX Kenntnisse, welche mir mehrmals Stunden mühsamen Suchens erspart haben.

Kurzfassung

Maschinenbelgungsplanungsprobleme (FSP) existieren in unterschiedlichen Varianten. Eine dieser Varianten ist das no-wait FSP with release times (NWFSP-RT). Es besteht aus einer Menge an Aufgaben und einer Menge an Maschinen. Beim NWFSP-RT müssen alle Aufgaben in derselben vordefinierten Reihenfolge auf allen Maschinen abgearbeitet werden. Zusätzlich dürfen Aufgaben auf der ersten Maschine erst nach einer gegebenen Release-Zeit beginnen. Anwendungen finden sich in der Stahl- und Lebensmittelproduktion zu finden. Hier können Wartezeiten zu einer Verschlechterung der Qualität führen.

Wir verwenden eine Beam Search (BS), d.h. eine Breitensuche mit beschränkter Breite, zum Lösen des NWFSP-RT. Auf jeder Suchebene behält BS nur die besten Knoten. Um zu entscheiden, welche Knoten behalten werden, verwendet BS eine heuristische Funktion (GF). In dieser Arbeit verwenden wir das Learning Beamsearch (LBS) Framework von Huber und Raidl [HR21] um GFs zu lernen. Bei der LBS werden in jeder Iteration Trainingsdaten unter Verwendung der aktuell gelernten GF erzeugt und diese anschließend verwendet um das neuronale Netzwerk (NN) zu trainieren.

Wir präsentieren zwei neuartige graphbasierte NN Typen inklusive Feature-Vektoren. Die NN Typen aggregieren Daten aller Aufgaben einer Instanz und der nähesten benachbarten Aufgaben jeder Aufgabe. Unter den Features ist eine neuartige Methode zur Berechnung unterer Schranken (ITLB) für das NWFSP-RT enthalten. Die beschriebenen Algorithmen und NN Typen wurden von uns implementiert und auf Vergleichsinstanzen, sowie zufälligen Instanzen evaluiert. Im Zuge der Evaluierung wurden auch statistische Signifikanztests durchgeführt. Die Ergebnisse zeigen, dass BS in Kombination mit den zwei neuartigen NN Typen in neun und zehn von 16 Konfigurationen signifikant bessere Ergebnisse erzielt als BS in Kombination mit ITLB verglichen auf Testinstanzen gleicher Größe, auf welcher die NN Typen trainiert wurden. Des Weitern generalisiert mindestens eine Konfiguration jedes der vier NN Typen gut über die Anzahl an Aufgaben, verglichen mit den besten bekannten Ergebnissen aus [Pou+20]. Während unserer Tests liefern die NN Typen für einzelne Instanzen bessere Ergebnisse als die besten bekannten Ergebnisse trotz der Einschränkung, dass die Tests mit einer kleineren maximalen Breite und ohne lokaler Suche durchgeführt wurden. Insgesamt war einer unserer Ansätze, evaluiert mit einer kleineren Breite als in [Pou+20], auf 11 von 46 getesteten Instanzklassen im Durchschnitt besser als die besten bekannten Ergebnisse und auf 43 von 46 Instanzklassen besser als BS ohne lokale Suche von [Pou+20].

Abstract

The flow shop problem (FSP) is a scheduling problem with many variants. One variant is the no-wait FSP with release times (NWFSP-RT). It consists of a set of jobs and a set of machines. It further imposes the constraints that jobs must pass all machines in a predefined order, the jobs are not allowed to wait on a machine until being processed, and jobs may only start processing on the first machine when their release time is exceeded. The goal is to find a schedule optimizing the desired objective, i.e., the makespan. The NWFSP-RT has applications in steel or food production where the product is not allowed to wait before being further processed to avoid degradation.

Beam search (BS), a limited width breadth-first search technique, has shown to be an effective heuristic in finding proper solutions to optimization problems within a limited time. Only the best nodes are kept and further branched on at every layer when the number of nodes exceeds a specific limit. To decide which nodes to keep, BS uses a guidance function (GF). We build upon the learning beam search (LBS) framework proposed by Huber and Raidl [HR21] to learn GFs. The LBS framework uses an iterative approach. In every iteration, training data is generated with a BS guided by the currently learned GF, and the neural network (NN) is trained to approximate the training data.

We propose two novel NN types, inspired by graph neural networks, that aggregate data over all individual jobs in a problem instance and their nearest neighbors. Further, we present feature sets for the NN types, including a novel lower bound, called ITLB, for the NWFSP-RT. We implement the algorithms, evaluate them over benchmark sets and random test instances, and perform statistical tests. The results show that a BS guided by two of our NN types produces significantly better results in nine and ten out of 16 configurations, respectively, than a BS guided by ITLB alone when run on similar instance sizes as the NNs were trained. The evaluation of the generalization abilities of the NNs shows that for each of the four NN types, at least one configuration generalizes well over the number of jobs compared with the best-known results. Our approaches frequently improve the state-of-the-art on even though running with a smaller beam width and without local search compared to the BS from Pourhejazy et al. [Pou+20] that represented the state-of-the-art so far. Overall, one of our approaches, evaluated with smaller beam width than in [Pou+20], was able to outperform the state-of-the-art on 11 out of 46 tested instance classes and to outperform the BS from [Pou+20] without local search on 43 out of 46 tested instance classes on average.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**

2 Considered Problem **5**
2.1 Scheduling Problem 5
2.2 The No-Wait Flow Shop Scheduling Problem with Release Times . . . 6

3 Related Work **11**
3.1 The Flow Shop Scheduling Problem and its Variants 11
3.2 The Beam Search Framework 14
3.3 Machine Learning for Optimization Problems 18
3.4 The Learning Beam Search Framework 21

4 Solving the NWFSP-RT with Beam Search **23**
4.1 Beam Search for the NWFSP-RT 23
4.2 Lower Bounds for a Beam Search Node 25

5 Learning Beam Search for the NWFSP-RT **33**
5.1 General Details 33
5.2 Training Instance Generation 33
5.3 Neural Networks 34
5.4 Feature Sets 37
5.5 Observations 41

6 Results **43**
6.1 Test Setup and Benchmark Instances 43
6.2 Comparison on Same Instance Sizes as Trained 44
6.3 Generalization over m 45
6.4 Generalization over n and m 53

6.5 Approximation Errors over Layers 54
6.6 Comparison with State-of-the-Art 58

7 Conclusion **73**

8 Future Work and Open Questions **75**

List of Algorithms **77**

Glossary **79**

Acronyms **81**

Bibliography **83**

CHAPTER 1

Introduction

Flow shops exist everywhere in today's world. Just think of a wooden furniture factory. To produce the furniture, first, the wood needs to be cut. After this, it needs to be ground and probably also painted before it is finally assembled. In a factory, not only one product is produced. Many kinds and variants of wooden furniture are created on-demand when ordered by the customer. Every product takes a different amount of time in each of the mentioned production steps. This yields idle times when the machines wait for the next product to be ready for processing. Nevertheless, most factory owners would like to increase the efficiency of the factory and lower the production costs. Therefore, their goal is to finish processing all products as early as possible. This problem maps directly to the Flow Shop Schedueling Problem (FSP). The FSPs' goal is to schedule a set of jobs on a set of machines. All machines must be passed in the same predefined order by each job. Furthermore, the jobs should be ordered in a way that minimizes some objective, like, e.g., the makespan. A variant of the FSP is the no-wait FSP with release times (NWFSP-RT), where jobs, i.e., products, are not allowed to wait before being processed on the next machine and only become available after a job-specific release time. An example for the NWFSP-RT would be a steel production factory, which produces different types of steel. First, the ore must be melted. Then it is processed in many intermediate steps until it is finally cast into a beam. During those steps, the melted product is not allowed to wait until the next machine is ready to process it, as this would mean that it cools down and degrades until it can not be processed further anymore. Now additionally assume that components needed during production are only available after a particular time, i.e., the release time, as they still need to be delivered to the factory.

The FSP as well as the NWFSP-RT are NP-hard for more than two machines [GJS76]. A non-deterministic Turing machine can solve problems in NP in polynomial time. Equivalently, a deterministic Turing machine can verify this class of problems in polynomial time.

NP-hard implies that all other problems which are in NP can be reduced to NP-hard problems in polynomial time [KV18, Def. 15.40].

A heuristic approach to solve the NWFSP-RT is beam search (BS). Beam search is an incomplete breadth-first search technique where only a certain number of best nodes on every level are kept and branched further on while the others are discarded. A guidance function, also called a heuristic function, is used to decide which nodes on a level should be kept. In our setting, the guidance function should approximate the cost additionally introduced when completing the solution. Its result is added to the current objective value of the node's partial solution, and the best nodes according to this calculation are selected. However, finding proper guidance functions is a highly problem-specific task requiring expertise and testing.

To overcome the issue of finding proper guidance functions, Huber and Raidl [HR21] propose the learning beam search (LBS) framework for learning proper guidance functions for optimization problems in a reinforcement learning like setting. The framework iteratively generates random training samples. It labels them by using a BS run with the current learned guidance function and trains the used machine learning model on the generated training samples as well as training samples from previous iterations. In [HR21], it is shown that it is possible to use LBS to learn a proper guidance function for the Longest Common Subsequence Problem (LCSP). The LCSP asks to find the longest subsequence that occurs in all strings of the input set.

The LBS framework is inspired by the work of Silver et al. in 2017 [Sil+17]. The authors aimed to master the game of Go and reach super-human performance by letting the algorithm play games against itself [Sil+17]. In LBS the algorithm does not play against itself. However, it uses its acquired knowledge from the last iteration to compute better, w.l.o.g. smaller, labels than before and thus pushing itself to continue improving the learned guidance function.

In this work we aim to adapt LBS such that it can be applied to the NWFSP-RT. This is described in Chapter 5. In Chapter 4, we introduce a novel lower bound for the NWFSP-RT which can be computed in $O(n^2m)$ and call it improved Taillard lower bound (ITLB). It is inspired by the lower bound presented in [Tai93] and the asymmetric Traveling Salesperson Problem (ATSP) lower bound that takes the sum of the minimum incoming or outgoing arcs of every city [Chr72]. This lower bound is used as a feature in our machine learning models as well as a reference guidance function. A set of features for a single BS node is derived from the instance and the state of the search. This set of features is used in combination with a multilayer perceptron (MLP). Further, we present two novel machine learning models: aggregated jobs neural network (AJNN_{add}) and nearest neighbors neural network ($\text{NN}_{\text{nearest}}$). The models are based on the idea of graph neural networks (GNNs). Both AJNN_{add} and $\text{NN}_{\text{nearest}}$ aim to use information about the structure of the instance and not only a global view of it. Thus, information about the structure is already incorporated into the feature set on a job-specific basis, considering information about close neighboring jobs. The distance measure used is defined later. In the AJNN_{add} a MLP is applied to each job's specific features, combined

with some global features, and the results are aggregated using a sum. The $\text{NN}_{\text{nearest}}$ uses a more elaborate structure that then the AJNN_{add}. Similarly, it applies to each job's specific features some learnable weights. Additionally, it contains links that pass aggregate information about the nearest neighbors of a job. The structure of the $\text{NN}_{\text{nearest}}$ also contains skip-connections and finally aggregates the results using the sum of all job-specific results. One can think of the AJNN_{add} and the $\text{NN}_{\text{nearest}}$ in a way that they sum up the "contribution" of each job to the makespan increase.

We evaluate several configurations of the presented models on the same instance sizes they were trained on and perform significance tests. The results show that our novel neural network (NN) models trained with the LBS framework perform significantly better than a BS guided by ITLB for many configurations. Further, we evaluate how well the trained models generalize when evaluating instances with a different number of machines and/or jobs as they were trained. The evaluation shows that generalizing over the number of jobs works well, whereas generalizing over the number of machines does not. Finally, we compare the best configurations with the state-of-the-art results in [Pou+20]. Overall, one of our approaches, evaluated with smaller beam width than in [Pou+20], was able to outperform the state-of-the-art on 11 out of 46 tested instance classes and to outperform the BS from [Pou+20] without local search on 43 out of 46 tested instance classes on average.

CHAPTER 2

Considered Problem

This chapter describes and defines the necessary preliminaries and notations used in the following chapters. Section 2.1 defines the general concept of optimization for scheduling problems. A detailed description of the NWFSP-RT and its notation as well as an example follows in Section 2.2.

2.1 Scheduling Problem

In this work, we define an Scheduling Problem (SP) as the task of scheduling a set of jobs J on a set of resources M to optimize (i.e., minimizing or maximizing) an objective function f in a vast but finite search space S. Function f maps every solution $v \in S$ to a numerical value $f\colon S \to \mathbb{R}$. How a solution is represented in detail needs to be defined for the concrete scheduling problem at hand. In S we search for an optimal solution $v* \in S$ with $z^* := f(v^*)$ for which $\forall v \in S\colon f(v^*) \circ f(v)$ (with $\circ$ one of $\{\geq, \leq\}$) holds. We assume w.l.o.g. for the whole work that a minimization problem is given and $\circ \equiv \leq$ and therefore:

$$z^* = \min_{v \in S} f(v) \tag{2.1}$$

$$v^* \in \arg\min_{v \in S} f(v) \tag{2.2}$$

Note that an SP may have multiple optimal solutions v^*. Throughout the work if it is stated that solution u is better than solution v, this refers to $f(u) < f(v)$.

Further, a partial solution v to an SP is a solution where not all jobs $j \in J$ are scheduled on all resources M. If f is applied to a partial solution, it returns the current objective value of the partial solution. We define the set $F(v)$ to contain all complete solutions which can be constructed out of the partial solution v. Finally, we define f' to take a set of (partial) solutions $W \subseteq S$ and output the best objective value when applying f to all $v \in W$.

We describe states of an (partial) solution in general for an SP. A state $o(v)$ of the solution v is a mapping from a (partial) solution v to a value defined for the specific problem at hand. The representation of states is independent of $f(v)$. Thus, it is possible that two solutions u, v with $u \neq v$ map to the same state $o(u) = o(v)$ and still have different objective values $f(u) \neq f(v)$. The state $o(v)$ needs to define all relevant parts of the problem such that $o(v)$ can be used as starting point for constructing a solution. If $w = v||u$ is the concatenation of solution v and a solution u for the instance defined by $o(v)$, and $||$ being the concatenation of vectors, then it must hold that $f(w) = f(v) + f(u)$.

Finally, we define the concept of dominance between solutions, which is later used to filter out partial solutions which definitely do not yield better solutions than others.

Definition 1. *(Dominance) A (partial) solution v dominates another (partial) solution u iff $f'(F(v)) \leq f'(F(u))$, and $v \neq u$.*

2.2 The No-Wait Flow Shop Scheduling Problem with Release Times

The NWFSP-RT consists of n jobs contained in the set $J = \{1, \ldots, n\}$ and m machines contained in the set $M = \{1, \ldots, m\}$. Every job must pass all m machines in order from 1 to m. For each job j and every machine i the processing time is denoted by $p_{i,j} \geq 0$. The time when a job j is available on the first machine is denoted by r_j. It is prohibited to start with the processing of j before time r_j is exceeded. Jobs are not allowed to wait before being further processed after the processing on machine one is finished. Thus, if waiting is required, the job needs to be delayed as a whole and start its processing on machine one later, such that it can pass all machines without waiting. An early work that introduces the no-wait FSP (NWFSP) without release times is [Pie60]. A solution s to the NWFSP-RT is a permutation of all jobs in J, i.e., $s = (3, 2, 1, 5, 4)$ for an instance with $n = 5$. Its length is denoted by $|s|$. The job on position $k \in \{1, \ldots, |s|\}$ of solution s is denoted by s_k. To compute the makespan of a solution, we compute its completion time matrix $C \in \mathbb{R}^{n \times m}$, which can be done in time $O(nm)$. The indices of the C matrix start at one. An entry in the matrix denotes when the according job in the solution s is finished on the respective machine, i.e., $C_{k,i}(s)$ is the finishing time of job s_k on machine i. Note that the parenthesis and the denotation of the solution are omitted when clear from the context. If a solution v does not contain all jobs J, i.e. $|J| > |v|$, we call it a partial solution and denote the set of all unscheduled jobs by $U = J \setminus v$. The last scheduled job in a solution is denoted by $a(v)$ and if v is empty, $a(v) = 0$. The makespan of a (partial) solution v is denoted by $C_{\max}(v) = C_{n,m}(v)$, which is the time when the last job of s finishes on the last machine m. Thus, the function $f(\cdot)$ as defined for SP yields for solution v: $f(v) = C_{\max}(v)$. To simplify the computation of the C matrix we define that all accesses to $C_{0,i} = 0$ for $i \in M$, as all machines are empty when starting the first job. The computation is done row-wise from top to bottom. To compute row k of the matrix for the (partial) solution v, a forward sweep and a backward sweep are

used. We treat the leftmost column separately:

$$C_{k,1} := \max(C_{k,1}, r_{s_k}) + p_{s_k,1} \tag{2.3}$$

For the forward sweep we do for all $2 \leq i \leq m$:

$$C_{k,i} := \max(C_{k,i-1}, C_{k-1,i}) + p_{s_k,i} \tag{2.4}$$

As we did not propagate the idle times introduced by taking the maximum back, only $C_{k,m}$ is computed correctly. To finish the computation, a backward sweep is needed, iterating over $m - 1 \geq i \geq 1$:

$$C_{k,i} := C_{k,i+1} - p_{s_k,i+1} \tag{2.5}$$

Now the calculation of the completion times for the k^{th} row is finished and the $k + 1^{\text{th}}$ row can be computed, until all $|v|$ rows are computed.

We define the start time of a job j on a machine i for a solution s analogous to the C matrix as $T_{j,i}(s)$, and compute it by:

$$T_{k,i} = C_{k,i} - p_{s_k,i} \tag{2.6}$$

At any time during processing, before a machine has processed all jobs but is currently not processing any job, it incurs idle time. For a solution s the sum of all idle times on all machines is the idle time $I(s)$. The idle time for solution s can be computed by:

$$I = \sum_{k \in \{1..|s|\}} \sum_{i \in M} T_{k,i} - C_{k-1,i} \tag{2.7}$$

In the following, we will provide an example solution for an instance with $m = 4$ and $n = 5$. The p-Matrix is:

$$p = \begin{bmatrix} 16 & 31 & 54 & 54 \\ 44 & 7 & 52 & 66 \\ 26 & 19 & 65 & 34 \\ 74 & 83 & 94 & 76 \\ 19 & 41 & 31 & 50 \end{bmatrix},$$

and thus, e.g., for job $j = 5$ and machine $i = 2$ $p_{5,2} = 41$. The release times are:

$$r = \begin{bmatrix} 60 & 180 & 33 & 17 & 95 \end{bmatrix}.$$

Thus, one optimal solution to this problem is $s = (3, 5, 1, 4, 2)$ with $C_{max} = 548$. The resulting matrix $C(s)$ is:

$$C(s) = \begin{bmatrix} 59 & 78 & 143 & 177 \\ 114 & 155 & 186 & 236 \\ 155 & 186 & 240 & 294 \\ 229 & 312 & 406 & 482 \\ 423 & 430 & 482 & 548 \end{bmatrix}.$$

Solutions of an NWFSP-RT instance can be visualized as Gantt charts. A Gantt chart for s is provided in Figure 2.1. There, jobs are displayed as bars with different colors, and the release times are included as dotted vertical lines. It can be seen that between jobs three and five, idle time exists on every machine. This is since job five needs to start later than the minimum delay time between these jobs due to its release time of $r_5 = 95$. The total delay on machine one is $d(s) = 244$ and is, due to the release times, greater than if the sum of the according $d_{j,k}$ entries, described in Section 2.2.1, are taken, which yields 202.

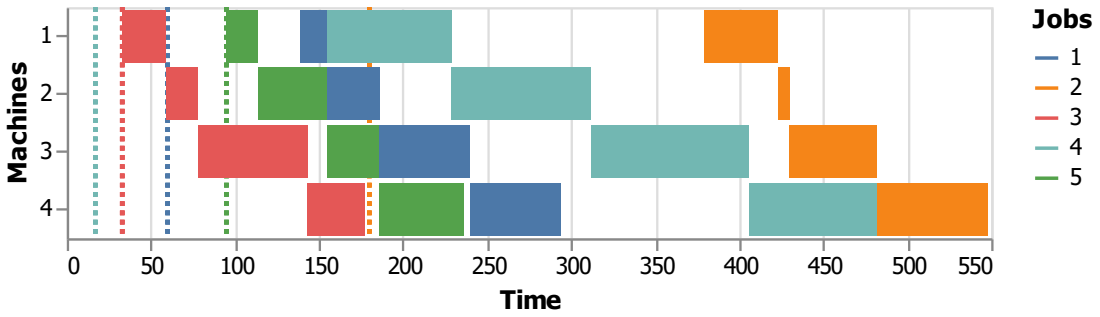

Figure 2.1: Gantt chart of solution $s = (3, 5, 1, 4, 2)$ of example sequence. Machines are on the vertical axis and jobs are displayed horizontally. Release times are included as vertical dotted lines in the color of the job they belong to.

2.2.1 Auxiliary Instance Data

During the search, when expanding partial solution v by appending some $j \in U(v)$, calculating the new row for j in the C matrix takes time $O(m)$. We define auxiliary instance data to reduce the time needed when appending one job. The proposed alternative way to compute $f(\cdot)$ is provided in Chapter 4.1.2.

For each job $j \in J$, let $p_j^{\text{tot}} = \sum_{i \in M} p_{i,j}$ be its total processing time over all machines. Moreover, let $q_{i,j} = \sum_{i'=1}^{i} p_{i',j}$ be the aggregated processing time of job j over the first $i' \leq i$ machines. Note that $q_{m,j} = p_j^{\text{tot}}$ holds, and the job's processing on a machine i starts by $q_{i,j} - p_{i,j}$ after the job's overall start on the first machine. The q-matrix is stored within the instance as auxiliary data.

When a job $j' \in J$ is scheduled directly after a job $j \in J$, $j \neq j'$, and we disregard release times, their starting times (on the first machine) always differ by the (minimum) *starting time differences*:

$$\delta^1_{j,j'} = \max_{i \in M}(q_{i,j} - q_{i,j'} + p_{i,j'}), \quad \forall j \in J, \forall j' \in J \setminus \{j\}. \tag{2.8}$$

Further, we define this difference in the starting times also on all other machines:

$$\delta^i_{j,j'} = \delta^1_{j,j'} + q_{i-1,j'} - q_{i-1,j}, \quad i \in \{2, \ldots, m\}, \forall j \in J, \forall j' \in J \setminus \{j\}. \tag{2.9}$$

We further define that when speaking of *delay on machine i* we refer to $d^i_{j,j'} = \delta^i_{j,j'} - p_{i,j}$ and when speaking of *delay* in general of the delay on machine one.

Due to the property of the minimum starting time differences it follows that:

Lemma 1. *For three different jobs $j, k, l \in J$ the triangle inequality $\delta^i_{j,k} \leq \delta^i_{j,l} + \delta^i_{l,k}$ holds for all $i \in M$.*

I.e., it is impossible to reduce the minimum starting time differences between any two jobs j, k by scheduling some other job l in between. This yields for delays $d^i_{j,k} \leq d^i_{j,l} + d^i_{l,k} + p_{l,i}$.

Moreover, we define the total idle time over all machines arising when scheduling job j' as successor of job j in-between these jobs as

$$\omega_{j,j'} = \sum_{i \in M} (\delta^i_{j,j'} - p_{i,j}), \tag{2.10}$$

where we again disregard release times.

Finally, we define how much job j' must end later on the last machine than job j such that j ends at an earlier or equal time on all machines:

$$\psi_{j,j'} = \max_{i \in M}(q_{i,j} - q_{i,j'}) + p^{\text{tot}}_{j'} - p^{\text{tot}}_{j}. \tag{2.11}$$

The above definition is needed to perform a dominance check in Section 4.1.3 efficiently.

2.2.2 Reduction of the NWFSP to the Asymmetric Traveling Salesperson Problem

Augmenting the jobs J with an artificial job 0 with $p_{i,0} = 0$, $i \in M$, we can reduce an NWFSP instance to an ATSP instance with the distance matrix δ^1. A solution to this ATSP instance in the form of a permutation of the jobs $J \cup \{0\}$ is then transformed back to the respective NWFSP instance by rotating the solution, so that job 0 comes last and finally removing this job. Note that, in general, the minimum starting time differences on any machine δ^i could be used for the reduction. A reduction that uses the d^1 matrix, i.e., δ^1 without the processing times on the first machine, and instead uses two artificial jobs is presented in [Wis72].

Considering this, we also define $\delta^1_{j,0}$, $\delta^1_{0,j}$, $\omega_{0,j}$, and $\omega_{j,0}$, $j \in J$, respectively, in order to deal with the beginning and the end of a schedule. The remaining auxiliary data like δ^i and d^i is calculated accordingly. To simplify future computations we define $r_0 = 0$.

CHAPTER 3

Related Work

In this chapter, we present related work relevant to this work. First, we present the relevant history up to the state-of-the-art for the FSP and its relevant variants in Section 3.1. Section 3.2 explains how the BS framework works and Section 34 describes the BS how it was used in [Pou+20] to solve the NWFSP-RT. Further, Section 3.3 presents relevant work for Machine Learning (ML) methods in the context of combinatorial optimization and the FSP. Finally, the LBS framework is described in Section 3.4.

3.1 The Flow Shop Scheduling Problem and its Variants

The problem of scheduling a fixed set of n jobs J on a fixed set of m machines M, with a different execution time $p_{i,j} \geq 0$ for every job $j \in J$ on every machine $i \in M$, without preemption, and where each machine can only process one job at a time, is called the Job Shop Scheduling Problem (JSP). In the JSP, the order of machines a job is processed on is given by precedence constraints and may vary for each job. Therefore, the jobs can be ordered differently on every machine, and jobs can start on different machines. An early work analyzing the JSP and its anomalies is [Gra66]. There are different objectives for the JSP in the literature, such as minimizing the total tardiness or makespan. Nevertheless, in this work, for all scheduling problems, we aim to minimize the time needed until all jobs are finished on all machines, i.e., the makespan. Therefore, if not stated otherwise, we always refer to this objective.

If the same machine order for all jobs is fixed upfront, we get the Flow Shop Scheduling Problem (FSP). Still, the order on every machine can be different as jobs may overtake each other. The FSP was first introduced by [Joh54] in 1954, together with algorithms for solving the FSP with two machines and a special case with three machines. The Campbell-Dudek-Smith (CDS) algorithm [CDS70] extends the algorithm of [Joh54] to apply to m machines by clustering the machines into two virtual machines and solving the generated two-machine problem by repeatedly using the algorithm of [Joh54]. Furthermore, [DJ64]

summarizes all assumptions implicitly used in the FSP and suggests to solve the FSP with an exact approach. Later, in 1976 [GJS76] showed that the FSP is NP-complete when minimizing the makespan for $m \geq 3$. For $m = 2$ there exist efficient algorithms [Joh54]. In the later years, many other variants of the FSP were introduced, and many solution heuristics for FSP and its variants were suggested. In the survey [Gra+79] the variants were first put in a common notation. Later, in [NEH83] the Nawaz-Enscore-Ham (NEH) algorithm is proposed to greedily construct a solution for the FSP. It iteratively tries to insert a job at the best position in a partial solution, starting with the jobs with higher total processing time. In [Tai90] NEH is refined to run in time $O(n^2m)$. Taillard et al. generated a random benchmark set for the FSP [Tai93]. They used different heuristic methods to generate upper bounds together with a self-developed lower bound to select the random instances with the biggest gap between the two bounds. The first part of the lower bound consists of multiple components. It computes for every machine the minimum time any job needs from the start up to this specific machine and similarly the minimum time any job needs from this machine up to finishing on all machines. Further, for every machine, the operations of all jobs on this machine are summed up. To compute the first part of the lower bound, these three components are summed up for every machine, and the maximum value over all machines is taken. The idea behind this computation is that a machine has to process all jobs operations for it and that by taking the minimum time until the computation can start on this machine and until it ends, a lower bound (LB) is derived. For the second part, the total processing time for every job is calculated by summing up the times for all its operations on every machine and taking the maximum of those. This yields a lower bound as every job must be executed until it is complete. Now the maximum of both parts is computed, which again yields a lower bound for the FSP. Since the paper of Johnson [Joh54], the FSP was intensively studied and became one of the most extensively investigated topics in literature [LH05]. Nevertheless, the FSP remained hard to solve, as up to the mid of the 1990s the best Branch and Bound (B&B) algorithms had problems solving instances with 15 jobs and 4 machines [And+97][p. 393]. Due to its NP-hard nature, the FSP was also tackled with many heuristic approaches like Genetic Algorithm (GA), Tabu Search (TS), B&B, and others.

A variant of the FSP is the Permutation Flow Shop Schedueling Problem (PFSP), which is similar but enforces the same processing order on every machine, and this order is subject to optimization. Starting with $m \geq 4$ there might exist non-permutation schedules (i.e. for the FSP, with a specific job order for every machine) which dominate permutation schedules for the PFSP [PSW91]. A comparison over seven lower bounds used by B&B approaches to solve the PFSP while minimizing the makespan is given in [LH05]. The summarized lower bounds reduce the PFSP to a one or two-machine problem, i.e. the constraint that a machine can only process one job at a time is relaxed for all machines except one or two. Release dates, time lags, and delivery dates are computed out of the other jobs' operations to yield good lower bounds. Most of the summarized lower bounds can be computed in polynomial time. Nevertheless, two of the proposed lower bounds can not be computed in polynomial time, as they use a B&B approach or an optimal

algorithm to compute a lower bound for the PFSP by solving a relaxed problem.

Another variant of the FSP, where jobs are not allowed to wait before being processed on the next machine, is called the no-wait FSP (NWFSP), or in the notation of [Gra+79]: $Fm|\text{no-wait}|C_{\max}$. This definition results in jobs not overtaking each other, i.e., the execution order of the jobs is the same on all machines as for the PFSP. Thus, every solution for the NWFSP is also a valid solution for the PFSP, with a different objective value though [HS96]. The NWFSP can be reduced to the ATSP [Pie60; RR72; Wis72] as already discussed in Section 2.2.2. The reduction, according to [Wis72], is to compute delays on the first machine between all job pairs and use them as distances in the ATSP together with two artificial jobs used as the first and the last job. As a reduction to the ATSP exists, all lower bounds designed for the ATSP might be used as well for the NWFSP. An example is the lower bound, where in the ATSP the sum of every city's cheapest in- and outgoing arc divided by two is taken, which is similar to the assignment problem described in [Chr72]. A lower bound for the NWFSP which can be calculated in time $O(\max(m, n^2 \log n)n)$ is provided in [KK07]. There the NWFSP is for every pair of machines h and i with $1 \leq h \leq i \leq m$, reduced to a two machine problem, which is optimally solved with the Gilmore and Gomory algorithm (GG) [GG64], see also [Vai03]. If $h = 1$ or $i = m$ then $O(\max(m, n \log n)n)$ and if $h = 1$ and $i = m$ then $O(\max(m, \log n)n)$ time is needed. Due to the solving with the GG algorithm, also a solution sequence is returned, which might provide a good initial solution for the m machine NWFSP to be further improved by metaheuristics. The authors of [KK07] present also a network representation for the NWFSP where the critical (longest) path of the network yields the minimum makespan of the instance.

An extensive survey about the FSP and its variants is provided in [All16], where the NWFSP is discussed in chapter 4.1. The NWFSP was tackled by greedy algorithms, TS and GA [All16] in the last years.

When jobs are only available for processing on the first machine after a job-specific release time r_j, fixed upfront, the problem is called NWFSP with release times (NWFSP-RT), or in the notation of [Gra+79]: $(Fm|\text{no-wait}, r_j|C_{\max})$. This work focuses on the NWFSP-RT minimizing the makespan without preemption, i.e., processing can not be paused, and jobs must be immediately scheduled on the next machine after finishing on one machine. Setup times are included in the jobs processing time, and sequence-dependent setup times of machines are not used. The NWFSP-RT is a special case of the NWFSP. When adding release dates equal to 0, any NWFSP instance yields an NWFSP-RT instance.

Two lower bounds for the NWFSP-RT with sequence-dependent setup time are provided in [BDG99]. Note that when setting all setup times equal to zero, this problem is equivalent to the NWFSP-RT. The authors first provide a reduction to the ATSP with additional visiting time constraints (ATSP-RT) and a linear programming formulation to solve it. They then relax the formulation and iteratively compute a Lagrangian relaxation by adapting the input vector to the relaxation by a small factor towards the ascending direction until no further improvement happens. To calculate the second lower bound the ATSP-RT is taken again. First, the outgoing arcs of all jobs j are altered such that

their weights are equal to the weight of the minimum outgoing arc of job j to any other job except itself. Then all jobs are sorted by increasing release dates, and the objective value of the solution is calculated and returned. This lower bound can be computed in time $O(n \log n)$.

We remark that we assume the jobs to be heterogeneous in the sense that typically they all have different processing times on the machines. The case that jobs are from a small set of different types and jobs of the same type have the same processing times on the machines is, for example, considered in [LS00], where a dynamic variant of the problem is approached in which jobs of predefined types become available only over time. Moreover, job types are considered in [EM95], where a reduction to the TSP is described.

3.2 The Beam Search Framework

The BS framework was originally proposed in the context of speech recognition [Low76]. Basically, BS is an incomplete breath first search utilizing a heuristic function for selecting promising nodes. It is frequently used to generate solutions for combinatorial optimization problems in a short time.

Some variants of the FSP were tackled with BS. The authors of [Pou+20] noticed the research gap for the NWFSP-RT and tackled it with a BS combined with a LS. The FSP with constraints on shared resources was solved by a combination of A* and BS, where only a fraction of all successor nodes is used for the A* search [Pas00]. A hybridization of Ant Colony Optimization (ACO) and BS was used in [Blu05] to tackle the open shop job problem. In [FVVF18] a BS approach was used to generate initial solutions for the PFSP with the objective to minimize the total tardiness. It is guided by a sophisticated heuristic function that weights the total tardiness, earliness, and idle time depending on the number of already scheduled jobs.

In BS a heuristic function $h(\cdot)$ is used to determine which partial solutions are promising and should be kept. In the context of our work, we aim to approximate the cost for completing the partial solution with $h(\cdot)$. The pseudo-code of the basic BS framework is given in Algorithm 3.1. In general, the heuristic function $h(\cdot)$ yields for every possible node a numerical value. During the search, nodes are used to keep track of the current partial solutions. A node v always has a parent node parent(v) and an action $a(v)$ which, when applied to parent(v), results in node v. The beam B is a central element of the search, containing at most β nodes. The parameter β is a strategy parameter that determines how many partial solutions should be kept at most in every layer. Increasing β normally leads to an increase in solution quality but also a higher time and memory consumption. The search proceeds in layers and starts at the initial layer zero, whereas every layer L has its own beam B_L. The search starts from a root node r with parent(r) $= \perp$ and $a(v) = \perp$, representing an empty solution, which is added to the initial layer $B_0 = \{\text{r}\}$. At every layer L all solutions in $v \in B_{L-1}$ are branched over all possible actions $U(v)$ and added to the set V_{ext}. Further, V_{ext} is sorted by $h(\cdot)$ and the β-best nodes according to $f(\cdot) + h(\cdot)$ are added to B_L. The search ends when a terminal node t is reached. A

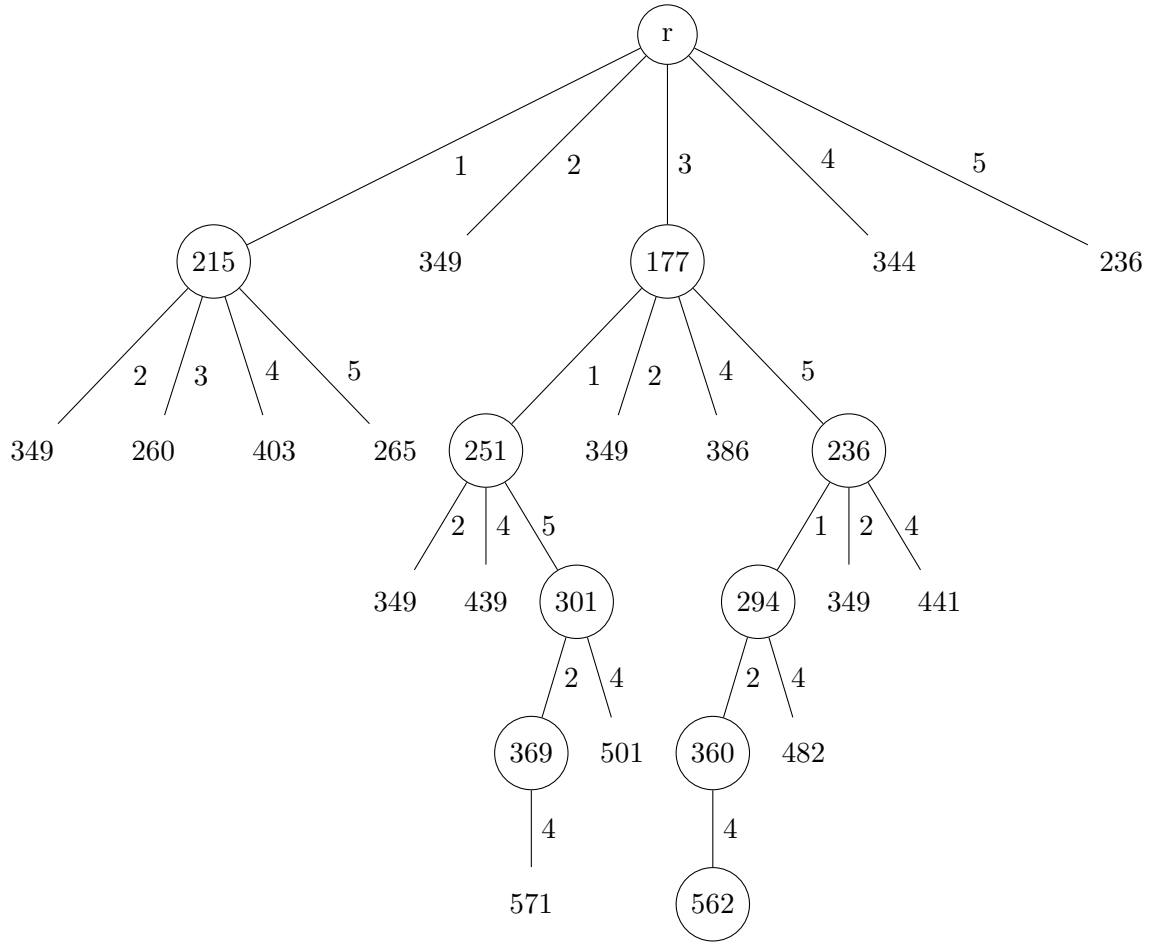

Figure 3.1: A beam search run with $\beta = 2$ for the example given in Section 2.2 with $n = 5$. The search starts at the root node r denoting the empty solution. It branches over all unscheduled jobs of every node in the beam. The arc labels denote the job which is added to the parents partial solution, i.e. scheduled next. The nodes values in the figure are the $f(\cdot)$ values of the partial solution represented by this node. We assume that $h(\cdot) = 0$ and thus the search greedily selects the two nodes with the best objective value in every layer. They are added to the beam and further branched on. The resulting solution is $v = (3, 5, 1, 2, 4)$ with an objective value $f(v) = 562$. Note that the best solution for this example has an objective value of 548.

criterion for determining terminal nodes is problem specific and needs to be defined at hand. In the context of this work, we define similarly to [SB99] that all nodes at layer $L_n = |U(\mathrm{r})| = n$ are terminal nodes, as all actions are taken and $U(v)$ is empty. Every path $\pi_{\mathrm{r}t}$ leading from r to t corresponds to a feasible solution. To obtain the concrete solution sequence, one needs to iterate from t over the parent links up to r and concatenate all actions accordingly. As we only store one parent pointer in every node, a path $\pi_{\mathrm{r}v}$ of node v represents exactly one (partial) solution, and therefore, we use the term *node* also to denote a (partial) solution. Further, we define all functions defined for partial solutions equally for nodes. Algorithm 3.1 contains two optional parts marked with //optional. These parts are only applicable if a dominance definition for the problem at hand exists, which can be controlled by the parameter dominance_enabled. Line 9 to 14 check if a certain state $o(u)$ was already reached by any other node in V_{ext} or if the node u is dominated by any already added node. It follows directly from Definition 1 that no partial solution w which might yield a better overall solution is removed from $W = V_{ext} \cup u$, i.e. $f'(F(W)) = f'(F(W \setminus w))$ for any partial solution w removed, or not added, in lines 9 to 17. Note for clarity that the case that u dominates some node $w_1 \in V_{ext}$ and is itself dominated by some node $w_2 \in V_{ext}$ can not happen because Definition 1 is transitive. Thus w_1 would have already been removed earlier.

Line 23 defines a dominance check, which only adds v if it is not dominated by any $w \in B_i$ and in this case removes all dominated nodes from B_i. It follows again from Definition 1 that $f'(F(B_i \cup v))$ does not get worse by removing dominated solutions. The dominance check is not used in our implementation of BS for the NWFSP-RT, as we only define that nodes can dominate each other if they are in the same state, and thus dominance is already checked in lines 9 to 17.

Assuming that the reached and dominance check are disabled and that creation of one node takes the problem-specific time t_{node}, the BS takes time $O(n\beta t_{\mathrm{node}})$. Further, we assume that checking the dominance between two nodes takes t_{dom} time. In theory with a naive implementation the reached check may take up to $O((\beta|U|)^2 t_{\mathrm{dom}}) = O((\beta n)^2 t_{\mathrm{dom}})$ time with $|U| \leq n$ and $|V_{ext}| \leq \beta|U|$ at most. When implemented with a hash map using the state as a key and a list of all nodes leading to this state as a value, finding duplicate states can be done in a much lower time in the average case, as only a few dominance checks need to be done per node. This is because, at the beginning of the search, many different states exist, and thus the number of nodes the dominance is checked with is small. Towards the end of the search, where the number of different states gets less, also the number of possible actions $|U|$ gets less; thus, fewer than βn nodes are created, and again only some collisions of states happen. Therefore, we argue that the reached check needs significantly less time than $O((\beta n)^2 t_{\mathrm{dom}})$ in the average case.

The dominance check checks, in the worst case, the dominance of all nodes in V_{ext} with all nodes in B. As $|V_{ext}| \leq \beta|U|$ and $|B| \leq \beta$, time $O(\beta^2|U|)$ is needed.

Algorithm 3.1: General BS Algorithm

Data: β (max. number of nodes in a layer), instance I, number of layers n, heuristic function h, dominance_enabled denotes if dominance check is enabled
Result: best terminal node

```
1  initialize root r
2  add r to B_0
3  for l ∈ {1, ..., n} do
4      V_ext ← ∅
5      foreach node v ∈ B_{l-1} do
6          foreach valid action a ∈ U(v) do
7              create node u from v and a
8              dominated = ⊥
9              if dominance_enabled;                          // optional
10             then
11                 remove all nodes w ∈ V_ext with o(u) = o(w) or where u dominates
                   w
12                 if ∀w ∈ V_ext with o(u) = o(w) : ¬(w dominates u) ∧w ≠ u then
13                     add u to V_ext
14                 end
15             else
16                 add u to V_ext
17             end
18         end
19     end
20     sort V_ext according to f(·) + h(·)
21     while |B_i| < β do
22         remove first node v of V_ext
23         if dominance_enabled and ∀w ∈ B_i : ¬(w dominates v) ; // optional
24         then
25             remove all w ∈ B_i dominated by v
26             add v to B_i
27         end
28         if ¬dominance_enabled then
29             add v to B_i
30         end
31     end
32     /* do postprocessing for LBS                              */
33 end
34 return best node from B_n (according to f(·))
```

Reference Method from Pourhejazy et al. [Pou+20]

We take the work of Pourhejazy et al. [Pou+20] as a reference method. To the best of our knowledge, it is the first work aiming to solve the NWFSP-RT. It provides a mathematical formulation of the NWFSP-RT for solving it via MILP (mixed-integer linear programming) based on the completion time of each job in the solution sequence on each machine, i.e., the C matrix. Further, to solve the NWFSP-RT, it suggests a BS approach, which selects the ξ successors with the lowest idle time of each node of the beam and improves them via random local search (LS). The idle time for a partial solution s is calculated by the sum of the completion times of job s_{k-1} on machine i minus the starting times of job s_k on machine i, for all $i \in M$. Thus, the idle time after the last job in s is finished on machine i is not respected. However, idle time incurred before the first job starts on some machine i is respected.

These ξ nodes are improved by LS and then added to V_{ext}. They are sorted by a lower bound F_{pour} of the completion time. The β best nodes are then added to the beam. The lower bound $F_{\text{pour}}(v)$ of partial solution v is calculated by

$$F_{\text{pour}}(v) = \max\left(f(v), \min_{j \in U} r_j\right) + \sum_{j \in U} p_{m,j}. \tag{3.1}$$

First, the maximum of the current objective value and the minimum release time of all unscheduled jobs is taken. Then, the processing time of all unscheduled jobs on the last machine is added (later in this work denoted as P_m^U). Clearly this is a LB as $P_m^U = \sum_{j \in U} p_{m,j}$ time is taken in any completion of solution s additionally to $f(v)$. Also, the minimum release time of all unscheduled jobs must be awaited before starting the next job. In Section 4.2 we present tighter LBs than this. The two LS procedures proposed are one that exchanges two jobs in a partial solution sequence and another which randomly removes one job and inserts it into all possible positions. They are randomly applied. It is not described in which order these LS procedures are used or how they are selected. The maximum number of LS executions is limited by nmk where k is a strategy parameter balancing between solution quality and time needed, which was set to $k = 0.1$ for $n \leq 400$ and $k = 0.01$ for $n \geq 500$. The number of successors per node in the beam ξ, in their work denoted by α, was evaluated and set to $\xi = 2$ by them.

3.3 Machine Learning for Optimization Problems

Scheduling problems, including the FSP, were also tackled with machine learning. An early work using NNs for solving scheduling problems is [PKL00], which uses a neural network to learn input parameters for some solution algorithm depending on the characteristics of an instance.

A learned weight matrix is used in [LS00] to solve a variant of the real-time FSP, where fixed types of jobs arrive over time. The weight matrix indicates the desirability of one

job type following the other. The weight matrix is trained for a specific set of job types on training instances labeled with their optimal sequences.

In [ACE06] an adaptive learning strategy is used to solve the FSP. In every iteration of the algorithm, the jobs operation times are multiplied by a learned weight per operation, and a constructive heuristic, like, NEH or CDS, is executed with the product of each operations time with its corresponding weight. If the results on the test instances improve during training, the weights are reinforced else the weights are altered randomly. If no improvement happens after several rounds, backtracking to the best weights is done.

The authors of [Ram+11] used a back-propagation NN with two hidden layers to generate an initial solution for a Genetic Algorithm (GA) and a heuristic proposed by [Sul00] to solve the FSP to minimize the makespan. The NNs were trained on completely enumerated small instances (the number of jobs $n \in \{5, 6, 7\}$, the number of machines $m = 5$). The trained model is dependent on the number of machines but independent of the number of jobs. The input features for the NN contain the jobs processing times, the average processing times, and the standard deviation of the job lengths per machine, all of them on each machine. To solve an instance, the jobs are sequentially provided to the NN and sorted by the output values of the NN. Their approach outperformed two other established heuristics for generating the initial solution, namely: NEH [NEH83], and Suliman Heuristic [Sul00] initialized with CDS [CDS70].

The work of [GML20] compared different training algorithms for NNs for solving the FSP with the objective to minimize the makespan. The training setup was the same as in [Ram+11]. The learned networks were applied to instances with up to 100 jobs. The NNs trained with the Levenberg-Marquardt (L-M) algorithm approximated the objective best.

Nevertheless, having optimal results for the training data at hand is not always possible. Therefore, AlphaGo Zero, which is an agent excelling in the games of Go, chess, and shogi, [Sil+17] uses reinforcement learning via self-play to train the NN. It is based on a Monte Carlo tree search in which a deep NN is used to evaluate game states. Training data is continuously generated by simulating games against itself. The approach was able to achieve superhuman performance in the sense that it beat a previous approach, which was able to beat the best human Go players in numerous simulated games.

In recent works, the FSP has been solved via RL approaches. In [RYY21] the FSP is tackled by an RL approach in which a NN is trained, which should output for every machine the best next action given a state. In total, ten actions are designed, which use different heuristic rules for selecting the next job, such as shortest/longest processing time first, or a function computing a critical ratio. The actions even include a neighborhood search, i.e., an action which swaps two neighboring jobs if this reduces the idle time. Input features consist of the maximum, minimum, and mean of the makespan, remaining operations, and the load of machines. To provide immediate reward during training, idle time is used, as increased idle time leads to a worse solution, according to the authors. Their NN was trained for $m = 5$ and used 24 nodes, three hidden layers, and a logistic

activation function. The input layer has 45 nodes, nine nodes per machine, and the output layer has 50 nodes, ten nodes per machine, representing the probability of a specific action chosen on one machine. The NN was trained using training samples from instances with $n \in \{5, 10, 20, 30\}$ to increase adaptability, labeled with Branch and Bound or GA. It was applied to instances with $m = 5$ and $n \in \{5, 10, \ldots, 80\}$. One paper that uses machine learning also for bigger instances with hundreds of jobs and twenty machines is [Ni+21], which tackles the Hybrid Flow Shop Scheduling Problem (HFSP), in which the sequential machines of the FSP are replaced by sets (called stages) of parallel machines. The goal is still minimizing the total makespan. First, an initial solution is generated by some heuristic. The proposed framework then uses RL, i.e., a Markov-Decision-Process with search operators as actions, to train the model. It reformulates the problem state into a multi-graph data structure, in which the job to machine assignment of every stage is modeled as a graph. Therefore, a node for every job is created. All jobs assigned to the same machine in the respective layer are connected via directed edges in the executed order. Node features include idle time, waiting time, arriving time, starting time, completion time, and processing time. These graphs are then processed by a graph NN. To overcome the issue that stages might have different dimensions, an Attention-based Weighted Pooling approach is used, which uses a Multi-layer perceptron. The problem is then solved by unsupervised training of the framework, which should map graphs to specific actions by classifying them. Tests were performed on large real-world datasets, and, according to the authors, good-quality solutions were obtained within a limited time.

As the NWFSP can be reduced to ATSP, some machine learning approaches for the ATSP shall also be referred to here. The authors of [KHW19] propose an encoder/decoder approach to solve the TSP and related problems like the Vehicle Routing Problem (VRP). In this approach, an encoder produces an embedding of all input nodes of a single instance. These embeddings are then used by the decoder together with an input mask, masking cities that were already visited and context information, providing information about the first and the last city in the solution sequence. The decoder outputs probabilities about which city should be visited next. These probabilities are used to greedily construct a solution sequence by always visiting the city with the highest probability next. The decoder contains certain considerations for efficiency. The model is trained via self-critical training using a greedy rollout as a baseline for estimating the total cost. This is similar to the self-play improvement of [Sil+17]. Also the authors of [Jos+21] use a encoder/decoder approach. Nevertheless, to speed up the computation, they explicitly sparsify the graph by considering only the k-nearest neighbors of each node. Further, beam search with different widths is compared with greedy search as well as supervised learning and reinforcement learning approaches. The ability of the models and approaches to generalize larger unseen instances than they have been trained on, is evaluated on TSP instances with up to 200 cities. The euclidean version of the TSP is tackled in [JLB19] by using a graph convolutional network. The authors propose to use the k-nearest neighbors graph during computation to reduce the computational complexity.

3.4 The Learning Beam Search Framework

The LBS framework was introduced by [HR21] and the description in this section follows their work. LBS is a general framework for learning a heuristic function to guide a BS. The framework uses a reinforcement learning (RL)-like approach where a BS with the currently learned heuristic function is used to label training data during self-learning. Further, it suggests an ML model and defines the interactions needed during training between the ML model, the BS and its other components like the instance generation and the replay buffer. The authors showed for the Longest Common Subsequence Problem (LCS), where the goal is to find the longest common supersequence of a set of strings, and related problems that the approach is highly competitive with the state-of-the-art of manually designed beam search heuristics. They could find several new best-known solutions so far. For solving the LCS, the NN gets the sorted remaining string lengths and the minimum number of remaining letters as input features.

The detailed description of the LBS Framework follows [HR21]. The pseudo-code of the algorithm for learning the heuristic function is provided in Algorithm 3.2. The training happens in iterations, whereas in each iteration, first, a random representative problem instance is generated, a BS with training data generation is performed, and finally the (re-)training of the ML model is performed. The total number of iterations z performed is given and used as a stopping criterion. Generating training instances is problem specific and needs to be defined for the problem at hand. Nevertheless, the generated instances should represent the problem instances that should be solved later. The BS calls generate in every iteration α training samples on average. To do so, nodes are selected as training data at an adaptive probability α/n_{nodes}. The number of training samples α, which should be generated in each iteration, is therefore divided by the number of generated non-dominated nodes n_{nodes} during one whole BS run. Initially, for the first BS run $n_{nodes} = 0$ and thus, no training data is generated, but the number of produced nodes is counted, and n_{nodes} is updated for successive iterations. In every following iteration, n_{nodes} is again updated to reflect the average number of all nodes produced over all so far performed LBS iterations. The nodes selected as training samples are labeled by a nested BS (NBS) call, again using the currently learned heuristic function as guidance, starting at the given node, i.e., using it as root node r. The NBS is performed with the beam width β'. The pseudo-code for data generation is shown in Algorithm 3.3. This algorithm is included into the general BS (Algorithm 3.1) at Line 32 when called with data generation. It is assumed that all additionally needed arguments are passed to the BS algorithm and are thus available and that Inst(v) creates an instance starting at node v. All labeled training samples are added to a replay buffer R, which can hold a given number of samples ρ. It is realized as a ring buffer. Thus if the number of training samples exceeds ρ, the oldest samples are overwritten. The training of the ML model starts when the buffer contains a minimum number of samples γ. As an ML model, we propose to use a NN, which we train by randomly selected mini-batches from R. In every iteration, one training step is performed, and all samples of R are included in randomly selected mini-batches. In general, any appropriate ML model can be used within the

Algorithm 3.2: General LBS Algorithm [HR21]

Data: beam width β, beam width for NBS β', number of layers n, number of training iterations z, size of the replay buffer ρ, buffer size with which training starts γ
Result: learned guidance function for BS

```
1  h ← initialize guidance function
2  R ← ∅ // replay buffer:  size ρ ring buffer
3  for z iterations do
4  |  I ← randomly generated problem instance
5  |  BS with training data generation on instance I
6  |  if |R| ≥ γ then
7  |  |  (re-)train h with data from R
8  |  end
9  end
10 return h
```

Algorithm 3.3: Data generation for LBS [HR21]. This piece of code is included in Algorithm 3.1 at line 32, if BS is called with data generation. For simplicity, it is assumed that all arguments additionally needed are passed to the BS algorithm.

Data: beam width for NBS β', replay buffer R, number of samples to generate α, number of expected nodes n_{nodes}, guidance function for the BS h

```
1  N = V_ext ∪ B_i
2  for v ∈ N do
3  |  if BS with data generation ∧ rand() < α/n_nodes then
4  |  |  t ← BS without data generation (β', Inst(v), n − |v|, h) // NBS call
5  |  |  add training sample (v, f(t)) to R
6  |  end
7  end
```

framework. The training approach must, of course, be adapted appropriately.

The computational complexity of one NBS call depends on the numbers of nodes that are expanded, which are at most β', the problem-specific time for expanding and evaluating one node t_{node} and the height of the search tree, which is n. Thus, one NBS call requires time $O(\beta' n t_{\text{node}})$. The overall time complexity for one LBS run is dependent on the number of iterations z, their beam width β, and the number of NBS calls performed per iteration, which is equal to α on average. This yields an average time complexity of $O(z(\beta + \alpha\beta')nt_{\text{node}})$.

CHAPTER 4

Solving the NWFSP-RT with Beam Search

In this section we provide the missing parts to solve NWFSP-RT with the general BS algorithm from Section 3.2. Therefore, in Section 4.1 we describe states and nodes for the NWFSP-RT, the branching scheme, and give a dominance definition for the case when two solutions have an equal state. We present our novel lower bound together with three other lower bounds for the NWFSP-RT in Section 4.2.

4.1 Beam Search for the NWFSP-RT

To solve the NWFSP-RT with BS, we define all parts needed for the general BS framework presented in Section 3.2.

State: As *state* for a partial solution v of the NWFSP-RT we take the set of unscheduled jobs $U(v)$ and the last scheduled job $a(v)$, which yields $o(v) = (a(v), U(v))$. Thus, the state is independent of the concrete permutation of the jobs in v. Further, multiple solutions map to the same state if they already scheduled the same jobs and have the same time on all machines relative to the start time at the first machine, which is ensured by having $a(v)$ in the state. Note that as we only use $a(v)$ and not the completion times of $a(v)$ relative to the completion time of $a(v)$ on the first machine, it might happen that two jobs theoretically end in the same state, but are mapped to different ones in our framework. Nevertheless, this is unlikely, and our dominance filters such nodes. Therefore, we take the performance advantage of storing only a single value.

Node: A node v contains following necessary parts:

- The state $o(v)$ of the current solution:

  - $a(v) \in J \cup 0$ ... action leading from the parent to node v, i.e., the job actually scheduled last. Job 0 is only used in the root node.
  - $U(v) \subseteq J$ ... set of still unscheduled jobs. In the implementation stored as a bit vector.

- parent(v) ... a pointer to its parent node or $\perp$ if it is the root node.
- $g(v) = f(v)$ ... the costs of the respective partial solution so far, which is the makespan of the partial solution.
- $h(v)$ ... a heuristic value that should estimate the cost-to-go, i.e., the expected further increase of the makespan for scheduling the remaining jobs in $U(v)$ in an optimal fashion.

If it is clear from the context to which node we refer, we only write a, U, g, and h.

If two nodes are in an equal state, we keep the one with the smaller objective value $g(\cdot)$. This selection does not degrade the overall solution quality, as only the best node in each state is needed to reach the optimal solution. If both have an equal makespan $g(\cdot)$, an arbitrary one is kept.

For the *root* node r we set $U(\mathrm{r}) = J$, parent(r) $= \perp$, $a(v) = 0$, $g(\mathrm{r}) = 0$, and $h(\mathrm{r})$ calculated respectively.

A node is denoted a *terminal node* or *terminal*, if $|U(v)| = 0$, i.e., all jobs are scheduled. Terminal nodes only exist at layer n, where all jobs are scheduled. Layer n might have multiple terminal nodes, as not all nodes map to the same state due to the state containing the last scheduled job $a(\cdot)$.

An efficient method to calculate $f(\cdot)$ is provided in Section 4.1.2. The definition of the heuristic function $h(\cdot)$ remains open as we want to learn it with the LBS framework. Nevertheless, an example would be using a lower bound as done in [Pou+20].

4.1.1 Branching

In our implementation, we branch every node over its unscheduled jobs and append them to the end. Note that an alternative branching scheme suggested in [JSW06] would be to insert the next job out of an ordered list of unscheduled jobs into every possible position of the current partial solution sequence. Nevertheless, we argue that BS does not benefit from this alternative branching. This is because during a search, two jobs yielding a good objective value might reside next to each other, which might not occur in any optimal sequence next to each other. It would be even worse if two jobs are in the correct order for an optimal solution, but that are missing some other jobs in between them, are yielding a bad objective value. Thus, the BS heuristic function might have a hard time filtering out partial solutions which are locally optimal but yield a worse objective value when completed than other partial solutions. When branching over all

unscheduled jobs and appending at the end, locally optimal solutions might also happen. However, it is at least guaranteed that these locally optimal parts must be included in the final solution of these nodes. An example, were we branch over all unscheduled jobs of every node in the beam is provided in Figure 3.1.

4.1.2 State Transition

To branch from node u to node v by scheduling job $j \in U(u)$ we set $\text{parent}(v) = u$ and $a(v) = j$. In theory, these operations are enough to build up the search tree for the BS. Nevertheless, our nodes also store auxiliary data, so we need to update it depending on the parent node's values. Thus, we set $U(v) = U(u) \setminus j$. For calculating the objective value $g(v)$ we use the following equation:

$$g(v) = g(u) - p^{\text{tot}}_{a(u)} + \max(\delta^1_{a(u),a(v)}, r_{a(v)} - g(u) + p^{\text{tot}}_{a(u)}) + p^{\text{tot}}_{a(v)}. \tag{4.1}$$

Note that this calculation can be performed in constant time $O(1)$, other than calculating a new row for the C matrix that needs time $O(m)$. The used heuristic function calculates the value for $h(v)$.

If disregarding the calculation of $h(v)$, the state transition is performed in time $O(n)$, as U must be copied and has $|U| \leq n$ entries.

4.1.3 Dominance Definition for the NWFSP-RT

The dominance check needs to cover cases where nodes are on the same layer, i.e., the partial solutions have the same length. We define dominance for two nodes u and v of the NWFSP-RT only for the case where they have equal $U(\cdot)$.

Definition 2 (Dominance for partial solutions of the NWFSP-RT)**.** *Partial solution u dominates partial solution v iff $u \neq v \wedge U(u) = U(v) \wedge g(v) - g(u) \geq \psi_{a(u),a(v)}$.*

This check can be done in $O(1)$ time except for the comparison of $U(u)$ and $U(v)$. As we use bit vectors to represent the unscheduled jobs, we may assume that also the latter can be done efficiently in practice, although the time complexity is in $O(n)$.

To perform the dominance check on a whole set of solutions V_{ext} obtained as extensions of the current beam, we store the nodes in a hash map with $U(\cdot)$ as keys. In this way, all groups of nodes with the same unscheduled nodes are determined efficiently, and $O(n)$ time pairwise dominance checks must be done only within each group.

4.2 Lower Bounds for a Beam Search Node

This section describes lower bounds for the NWFSP-RT used in our implementation. In general, it needs to be said that every lower bound for the NWFSP and the FSP is also a lower bound for the NWFSP-RT, as those problems generally yield smaller objective

values for the same p-Matrix than the NWFSP-RT [LH05; PSW91]. Nevertheless, such LBs perform worse than bounds specifically designed for the NWFSP-RT as the second can use all constraints of the problem.

We will start with the LB used in [Pou+20] for the NWFSP-RT and continue by adapting the LB from [Tai93] used for generating hard FSP instances. Then we present an LB resulting from the idea of reducing the NWFSP to the ATSP, and finally, we combine the ideas of the former LBs into a new LB denoted as ITLB.

4.2.1 LB from Pourhejazy et al. [Pou+20] (PLB)

This section refers to the LB on the completion time presented in [Pou+20] formula 12. The provided formula is translated into our notation. The idea of this LB is to take the minimum release time and add up the time all unscheduled jobs take on the last machine. The formula is presented in Equation 3.1. It can be calculated in time $O(|U|) \leq O(n)$.

4.2.2 LB from Taillard (TLB) [Tai93]

The LB proposed in [Tai93] was used for generating random test instances for the PFSP which have large gaps between this LB and the best-known solution determined with different approaches. In this section, we adapt it to the NWFSP-RT and refine it by incorporating release times and, to a small extent, also delays, but do not consider the no-wait aspect yet. The bound consists of two parts that both yield valid LBs, from which we take the maximum. The first part calculates a bound for a shortest possible schedule. More specifically, let η_i be the minimum time needed before machine i starts to work relative to $g - p_a^{\text{tot}} + p_{1,a}$, i.e., from the time when a finished on machine one, and λ_i the minimum amount of time that machine i remains inactive after it finished all jobs. Note that $q_{1,j} = p_{1,j}, \forall j \in J \cup \{0\}$. Before we start describing the bound we define how to calculate the job-specific release time relative to the starting time of job $a(v)$ of solution v:

$$r_j^{\text{rel}}(v) = \max(0,\ r_j - (g(v) - p_{a(v)}^{\text{tot}})) \tag{4.2}$$

To calculate η_i for node v we use:

$$\eta_i = \max(\min_{j \in U}(q_{i,j} - p_{i,j} + \max(0, r_j^{\text{rel}} - p_{1,a})),\ q_{i,a} - p_{1,a}) \tag{4.3}$$

The above formula's maximum consists of two parts. In the first part, we aim to find the shortest possible time any job $j \in U$ needs until it starts on machine i. If release times are ignored, this is $q_{i,j} - p_{i,j}$. To incorporate release times we use $\max(0, r_j^{\text{rel}} - p_{1,a})$. In this formula the release time has to be shifted such that it is also relative to $g - p_a^{\text{tot}} + p_{1,a}$, i.e., does not include $p_{1,a}$. The maximum with zero is taken for the cases where the release time has already passed. The second part calculates the time when machine i becomes available in the current state, i.e., when job a finishes on machine i. As we defined to calculate the lower bound from the time where a finishes on machine one, we

have to subtract the time a takes on the first machine $p_{1,a}$ from $q_{i,a}$ as it is included there.

The minimum amount of time that machine i remains inactive after it finished all jobs is calculated for partial solution v according to the following formula:

$$\lambda_i = \min_{j \in U} p_j^{\text{tot}} - q_{i,j} \tag{4.4}$$

The calculation of λ_i uses the pre-computed values of $p_j^{\text{tot}} - q_{i,j}$ which denote the time machine i is idle after executing job j if job j is the last job in the solution sequence.

To get the first part of the TLB we take the maximum of the per machine sum of $\eta_i + \lambda_i$ together with the total processing time $P_i^U = \sum_{j \in U} p_{i,j}$ on machine i for all unscheduled jobs. We further need to correct LB_1 by $-(p_a^{\text{tot}} - p_{1,a})$ such that it can be directly added to $g(v)$ for solution v:

$$\text{LB}_1 = \max(0, \max_{i \in M}(\eta_i + P_i^U + \lambda_i) - (p_a^{\text{tot}} - p_{1,a})) \tag{4.5}$$

This is a valid LB as all jobs need to be processed on any machine i. Further, also the minimum times to start computation on machine i and minimum idle time afterward, and the minimum release time are always implied. The correction by $-(p_a^{\text{tot}} - p_{1,a})$ is necessary as we need to account for the fact that η_i starts after the finishing of a on machine one. Through this correction, LB_1 can be directly added to g. Lower bounds are generally defined to yield values ≥ 0, so we take the maximum with zero.

The second part of the LB calculates the sum of the total processing times, and release times and takes the maximum over all unscheduled jobs:

$$\text{LB}_2 = \max(0, \max_{j \in U}(p_j^{\text{tot}} + \max(0, r_j^{\text{rel}} - p_{1,a})) - (p_a^{\text{tot}} - p_{1,a})) \tag{4.6}$$

Again, this is a valid LB as every job's operations need to be processed sequentially. One operation cannot start before the preceding one is finished, and every job can only be scheduled when its release time is exceeded. As above, the relative release time needs to be corrected, and the result corrected such that we can directly add it to g.

An improvement of LB_2 follows from Lemma 1. We can take the maximum of r_j^{rel} and $\delta_{a,j}^1$, as any job $j \in U$ must always follow job a somewhere later in the sequence:

$$\text{LB}_2' = \max(0, \max_{j \in U}(p_j^{\text{tot}} + \max(r_j^{\text{rel}}, \delta_{a,j}^1) - p_{1,a}) - (p_a^{\text{tot}} - p_{1,a})) \tag{4.7}$$

Because $\delta_{a,j}^1$ always includes $p_{1,a}$, which we defined not to be contained in the bound, we have to subtract $p_{1,a}$ from the dominating factor of release time and delay.

Finally, by combining these two bounds we get

$$\text{TLB} = \max(\text{LB}_1, \text{LB}_2'). \tag{4.8}$$

Note that for the root node r, all p_a^{tot} and $p_a^{\text{tot}} - q_{1,a}$ are zero as we defined $a(\text{r}) = 0$ and 0 denotes the artificial job. Further, note that all processing times of the artificial job 0 are zero. For improvements upon this LB, see ITLB in Section 4.2.4

Runtime Analysis: Calculating η_i and λ_i is in $O(|U|) \leq O(n)$ as the minimum of all $j \in U$ is taken. Thus calculating LB_1 takes time $O(nm)$ as it iterates over all machines. Calculating $\text{LB}_2^{'}$ takes $O(|U|) \leq O(n)$ as iterating over all jobs is necessary. Note that calculating P^U takes time $O(nm)$, which could be reduced by storing it in the node and calculating it in the state transition incrementally in time $O(m)$. Thus, the total time needed is $O(nm)$.

4.2.3 Delay-Based LB (DLB)

The idea for DLB stems from the reduction of NWFSP to the TSP [Chr72]. The delays and the artificial job play a central role in this reduction. Looking at the reduction to the ATSP designed by us earlier, we aim to sum up the minimum cost of all incoming arcs for every $j \in U \cup \{0\}$ and all outgoing arcs $j \in U \cup \{a\}$. When adding release times to this bound, one must consider carefully that if the minimum starting time difference $\delta^1_{a,j}$ is $\geq r_j$, no additional idle time is introduced by scheduling job j next. In theory, any machine's minimum starting time differences could be taken, and the calculation altered accordingly. More specifically, for the NWFSP the makespan of solution s represented by node v is calculated as $\sum_{k=1}^{|s(v)|-1} \delta^1_{s(v)_k, s(v)_{k+1}} + \delta^1_{a,0}$.

Regarding the case when $v = \text{r}$, we highlight that $a(\text{r}) = 0$. We define the temporary distance matrix δ' to use release times when they exceed the delays and subtract $p_{1,j}$ as we want to consider the operation times separately by adding P_i^U later:

$$\delta'_{j,k} = \begin{cases} \max(\delta^1_{j,k}, r^{\text{rel}}_k) - p_{1,j} & \text{if } a = j \\ \delta^1_{j,k} - p_{1,j} & \text{otherwise} \end{cases}, \forall j,k \in J \cup \{0\} \tag{4.9}$$

A bound for the NWFSP-RT considering incoming arcs which can be directly added to g would be:

$$\text{LB}_{\text{in}} = \max(0, \sum_{j \in U \cup \{0\}} \min_{k \in U \cup \{a\} \setminus \{j\}} \delta'_{k,j} + P_1^U - (p_a^{\text{tot}} - q_{1,a})) \tag{4.10}$$

Subtracting $p_a^{\text{tot}} - q_{1,a}$ in the above formula is necessary as $\delta'_{j,k}$ is calculated on machine one and starts at the time when a finishes on machine one. If doing this calculation e.g. on machine m the subtraction is not needed as it would result in $-p_a^{\text{tot}} + q_{m,a}$ with $q_{m,a} = p_a^{\text{tot}}$.

Further, we define a second bound based on the outgoing arcs when thinking in terms of an ATSP:

$$\text{DLB} = \max(0, \sum_{j \in U \cup \{a\}} \min_{k \in U \cup \{0\} \setminus \{j\}} \delta'_{j,k} + P_1^U - p_a^{\text{tot}} + p_{1,a}) \tag{4.11}$$

This time, all $j \in U \cup \{a\}$ have outgoing arcs which need to be considered and which might lead to unscheduled jobs, including the artificial job but excluding self-loops $k \in U \cup \{0\} \setminus \{j\}$.

To compute the final bound we take the maximum of the above parts:

$$\mathrm{LB}_{\mathrm{delay}} = \max(\mathrm{LB}_{\mathrm{in}}, \mathrm{LB}_{\mathrm{out}}) \tag{4.12}$$

The lower bound $\mathrm{LB}_{\mathrm{delay}}$ can be directly added to g.

Note that this bound can be further improved by calculating it on all machines $i \in M$. Additional considerations for the release times must be taken. Further, $q_{i,a}$ must be substracted instead of $q_{1,a}$ or $p_{1,a}$, and the maximum over the resulting bounds on all machines must be taken.

Runtime Analysis: To calculate $\mathrm{LB}_{\mathrm{in}}$ and $\mathrm{LB}_{\mathrm{out}}$ we iterate over all combinations of unscheduled jobs including the artificial job. Therefore, calculating $\mathrm{LB}_{\mathrm{delay}}$ takes time $O(|U|^2) \leq O(n^2)$.

4.2.4 Improved Taillard Lower Bound (ITLB)

This section aims to improve TLB by incorporating a bound on the minimum delays on each machine. The ITLB can be seen as using DLB (with adaptions) within the calculation of TLB once for every machine. The bound is calculated from the end of job a on machine one until the end of all jobs on machine m. Thus, we correct it by subtracting $(p_a^{\mathrm{tot}} - q_{1,a})$ such that it can be directly added to g. Note that the artificial job 0 is not used in calculating this bound, as we consider the processing times for finishing the last job by λ_i.

We adapt the calculation of the TLB for η_i accordingly for partial solution v to not add up the whole release time but only the part of the release time that exceeds the necessary delay. Further, we add the minimum delay on this machine calculated by $d(v, i, j)$ stated later. In the second part of the maximum function, we calculate the minimum delay over all jobs, as it is not given which job will be the next one scheduled. All other parts of the formula remain the same as for the TLB.

$$\eta_i = \max(\min_{j \in U}(q_{i,j} - p_{i,j} + \max(\delta^1_{a,j}, r^{\mathrm{rel}}_j) - p_{1,a} + d(v,i,j)), \\ q_{i,a} - p_{1,a} + \min_{j \in U} d(v,i,j)) \tag{4.13}$$

The calculation of λ_i is not altered, i.e.,

$$\lambda_i = \min_{j \in U} p_j^{\mathrm{tot}} - q_{i,j}. \tag{4.14}$$

To calculate $d(v, i, j)$ we use the following formula to first calculate a delay bound $d^{in}(v, i, j)$ on the incoming arcs. This bound only considers the additional time needed (delay) on the respective machine but ignores the processing times as they are added

later. Thus, the processing times must always be subtracted from the minimum starting time differences. We assume that every unscheduled job has an incoming arc, where we consider job j separately as it must follow job a:

$$d^{\text{in}}(v,i,j) = \sum_{j' \in U \setminus \{j\}} \min_{k \in U \setminus \{j'\}} \delta^i_{k,j'} - p_{i,k} \tag{4.15}$$

To calculate a bound considering only outgoing arcs $d^{out}(v,i,j)$, we assume that every unscheduled job has one outgoing arc, except the last job in the sequence. To correct the delay the last job in the sequence introduces, we subtract the maximum of the delays summed up before. Again, the processing times are subtracted from the delays:

$$d^{\text{out}}(v,i,j) = \sum_{j' \in U} \min_{k \in U \setminus \{j,j'\}} (\delta^i_{j',k} - p_{i,j'}) - \max_{j' \in U} \min_{k \in U \setminus \{j,j'\}} (\delta^i_{j',k} - p_{i,j'}) \tag{4.16}$$

In the above formula it is again clear that job j must follow job a and j is therefore considered separately. The second part of the formula calculates the sum over all outgoing arcs for all unscheduled jobs. Note that we now added one outgoing edge too much. To correct this we remove the most expensive outgoing edge counted before by subtracting it.

Our final delay bound is thus:

$$d(v,i,j) = \max(d^{\text{in}}(v,i,j), d^{\text{out}}(v,i,j)) \tag{4.17}$$

To get the first part of the LB we proceed as for the TLB and derive a bound which can be directly added to g:

$$\text{LB}_1 = \max(0, \max_{i \in M} \eta_i + P^U_i + \lambda_i - p^{\text{tot}}_a + p_{1,a}) \tag{4.18}$$

The second part of the TLB, LB'_2, was already improved before and therefore stays the same.

Finally, we get:

$$\text{ITLB} = \max(\text{LB}_1, \text{LB}'_2) \tag{4.19}$$

Note that in the case of the root node $p^{\text{tot}}_{a(\text{r})}$ and $q_{1,a}$ evaluate to zero as the duration of the artificial job which comes first is zero on all machines.

This bound clearly dominates TLB as it contains all its components. It also dominates DLB, as it contains all delays. It was also tested on 100 randomly generated instances (n=10, m=20) with a BS where TLB, DLB, and ITLB were calculated for every node that ITLB outperforms the others.

To speed up the calculation of $d(v,i,j)$ and calculate it for all $j \in U$ in time $O(n^2)$ together, we do the following:

- For calculating $d^{\text{in}}(v, i, j)$ for all $j \in U$ at once: First, we compute the minimum $x_k = \delta^i_{k,j'} - p_{i,k}, \forall k \in U$ while discarding self loops (i.e. $\delta^i_{j,j} - p_{i,j}$) and then take the sum of it which we denote here by sum. To compensate for counting job j too much in the sum we now calculate sum $- x_j, \forall j \in U$. The whole computation happens in time $O(n^2)$ for all $j \in U$.

- For calculating $d^{\text{out}}(v, i, j)$ for all $j \in U$ at once: We compute the two minimum delays $x_{j'} = \delta^i_{j',k} - p_{i,j'}, \forall k \in U$ and store them together with their indices denoted by $\text{index}_{j'}$. The minimum is denoted by $x_{j',1}$ and the second smallest value by $x_{j',2}$, similarly for indices. We then take the sum $\text{sum} = \sum_{j \in U} x_{k,1}$ and store it in a repeated way as vector $\text{sum}_j, \forall j \in U$. We further copy $y = x_{:,1}$. Now we have to account for the cases where the job j is also the minimum object and thus contained in sum_j. Therefore, we do $\text{sum}_{\text{index}_{j,1}} = \text{sum}_{\text{index}_{j,1}} - x_{j,1} + x_{j,2}$ for all $j \in U$. To account for the subtraction of the maximum values we compute $y_{\text{index}_{j,1}} = \max y_{\text{index}_{j,1}}, x_{j,2}$. This calculation happens in time $O(n^2)$ for all $j \in U$.

Note that due to performance reasons and to stay within reasonable time limits, we will only compute d^{in} during evaluation. Even though the theoretical time complexity of d^{in} and d^{out} is the same, it takes a significantly higher time to compute d^{out} in our implementation.

Runtime Analysis: Calculating λ_i is in $O(|U|) \leq O(n)$ as the minimum of all $j \in U$ is taken as for the TLB. Calculating $d(v, i, j)$ for all $j \in U$ at once is in $O(|U|^2) \leq O(n^2)$ and thus calculating η_i takes time $O(|U|^2)$. Further, all computations must be done for every machine. This finally yields that calculating ITLB takes time $O(m|U|^2) \leq O(mn^2)$.

CHAPTER 5

Learning Beam Search for the NWFSP-RT

To apply the LBS framework to the NWFSP-RT, we define and describe all missing parts of the general LBS framework as described in Section 3.4. We aim to learn a heuristic function $h(\cdot)$, approximating the minimum makespan increase when completing a partial solution. Therefore, we first describe some general details in Section 5.1. Then, the training data generation is described in Section 5.2. Section 5.3 proposes three NN types used, and Section 5.4 presents the used features and states the idea behind each. Those features are combined to observations in Sections 5.5.

5.1 General Details

The description of the time complexity of one LBS run in Section 3.4 leaves the description of the time expanding and evaluating one node takes t_{node} open as it is problem specific. This time is dependent on the time the state transition described in Section 4.1.2 takes which is $O(n)$ as well as the time needed for evaluating it t_{eval}. Further, each node might have up to n children and thus $t_{\text{node}} = O(n^2) + t_{\text{eval}}$. The evaluation time t_{eval} is dependent on the time needed to evaluate $h(\cdot)$, and the time needed to compute the used features which are further described in Sections 5.3, and 5.4.

5.2 Training Instance Generation

We generate the training instances to match the characteristics of the benchmark instances of [VRF15]. We note again that we assume the jobs to be different from each other such that they can not be clustered into types of jobs. This assumption was experimentally verified by giving a tolerance area around each processing time on each machine and

trying to find jobs for which all their processing times on all machines fit into the tolerance area. It was found that the assumption also holds if the criterion is ignored for up to two machines using a tolerance area of 40. Thus, we assume that the times of all operations of all jobs $p_{i,j}$ are equally distributed over $[0, 99]$ and independently chosen. Further, the release times r_j for an randomly created instance are calculated by the formula of [ZLC15]:

$$r_j = 0.05 \sum_{j \in J} p_{j,1}. \tag{5.1}$$

Note, that Formula 5.1 was also used to adapt the benchmark sets we use from the FSP to the NWFSP-RT in [Pou+20].

5.3 Neural Networks

We propose different types of NNs to be used within the LBS framework. The inputs used for these NNs are described in Section 5.4 and combined to observations that are later used during evaluation in Section 5.5.

5.3.1 Multi-Layer Perceptron

The use of a MLP within LBS for approximating the costs for completing a partial solution is proposed in [HR21]. We use a dense network of neurons with two hidden layers and ReLu activation functions in our work. The output layer consists of a single neuron without an activation function. The number of nodes in the hidden layers is set to 20 as it was done in [HR21] and as preliminary tests with $\{10, 20, 30\}$ showed no significant difference. Preliminary tests showed no significant difference in the results when adapting this value. When ignoring the weights between the input layer and the first hidden layer, this network has 420 learnable weights. It is trained using the mean squared error (MSE) as a loss function.

5.3.2 Aggregated Job Neural Network ($\mathbf{AJNN_{add}}$)

The AJNN_{add} consists of an internal MLP, further denoted as M_1. An overview of its structure is given in Figure 5.1. The observation vector consists of a global part x^{glob} and job $j \in U$ specific features x_j^{job}. The job specific part x_j^{job} is then split further into $x_j^{\text{job}} = (x_j^{\text{job,glob}}, x_j^{\text{job,in}}, x_j^{\text{job,out}})$. The first part $x_j^{\text{job,glob}}$ contains specific features for job j. The other parts $x_j^{\text{job,in}}, x_j^{\text{job,out}}$ each contain κ features dependent on the nearest pre- and succeeding jobs of j.

The nearest jobs are determined by the ω matrix, containing idle time. For every job j in $|U|$ the κ nearest jobs must be selected, which takes time $O(|U| \min(\kappa, \log |U|)$ for one job. Note that when κ gets big enough, it is more efficient to sort all jobs in U instead of iteratively selecting the nearest one. Thus selecting the κ nearest pre- and succeeding jobs for all $j \in U$ takes time $O(|U|^2 \min(\kappa, \log |U|)) \leq O(n^2 \log n)$. These 2κ nearest jobs are stored in vectors of sorted lists $\mathcal{N}_j^{\text{red,in}}$ and $\mathcal{N}_j^{\text{red,out}}$ for each job j. In

the implementation, the values in $\mathcal{N}^{\text{red,in}}$ and $\mathcal{N}^{\text{red,out}}$ refer to the jobs' indices when taking only jobs from U, sorting them by their id, and numbering them with 1 to $|U|$.

If there are too few jobs left to fill up $x_j^{\text{job,in}}$ and $x_j^{\text{job,out}}$, they are filled up with the last job's values. We do similarly for $\mathcal{N}_j^{\text{red,in}}$ and $\mathcal{N}_j^{\text{red,out}}$ by filling the values up with the last job. When only one job remains, we use it in the feature vectors $x_j^{\text{job,in}}$, and $x_j^{\text{job,out}}$ and fill $\mathcal{N}_j^{\text{red,in}}$, and $\mathcal{N}_j^{\text{red,out}}$ with ones as the index of the job left is one.

Take a sample instance with $n = 5$, $U = \{1, 2, 4\}$, $\kappa = 1$ and four job specific features $|x^{\text{job}}|$. Further, assume that $\mathcal{N}_1^{\text{red,in}} = [2]$, $\mathcal{N}_1^{\text{red,out}} = [2]$, $\mathcal{N}_2^{\text{red,in}} = [4]$, $\mathcal{N}_2^{\text{red,out}} = [1]$, $\mathcal{N}_4^{\text{red,in}} = [2]$ and $\mathcal{N}_4^{\text{red,out}} = [1]$. We also assume that $x^{\text{glob}} = (10, 20, 30)$. Further, we assume $x_{1,2}^{\text{job,in}} = (8)$, $x_{1,2}^{\text{job,out}} = (9)$, $x_{2,4}^{\text{job,in}} = (10)$, $x_{2,1}^{\text{job,out}} = (11)$, $x_{4,3}^{\text{job,in}} = (12)$, and $x_{4,1}^{\text{job,out}} = (13)$. Finally, assume $x_1^{\text{job,glob}} = (22, 33)$, $x_2^{\text{job,glob}} = (44, 55)$, and $x_4^{\text{job,glob}} = (66, 77)$. Then $x^{\text{job}} = (22, 33, 8, 9)||(44, 55, 10, 11)||(66, 77, 12, 13)$, where $||$ denotes the concatenation of vectors. The nearest job vectors are $\mathcal{N}^{\text{red,in}} = (2, 4, 2)$ and $\mathcal{N}^{\text{red,out}} = (2, 1, 1)$. In our implementation, we map those vectors to $(2, 3, 2)$ and $(2, 1, 1)$ respectively, representing the jobs' indices in U when sorted by their ids. These values can be used as indices in the feature vector x^{job} in the $\text{NN}_{\text{nearest}}$ in Section 5.3.3.

If the AJNN_{add} needs to generalize to an instance with more jobs than it was trained on, M_1 is called more often but always with the same number of input features, and all its results are summed up.

The AJNN_{add} aims to approximate the increase of the makespan when completing the solution. The AJNN_{add} is trained the same way as the MLP in Section 5.3.1 is, using the mean squared error (MSE) as a loss function. The loss is calculated between the result value of the AJNN_{add} and the label. We use for M_1 the same setup as in Section 5.3.1, a dense network of neurons with two hidden layers and ReLu activation functions. The output layer consists of a single neuron without an activation function.

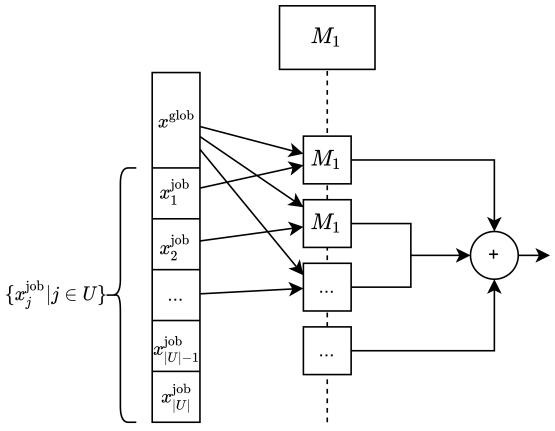

Figure 5.1: Structure Diagram of the AJNN_{add}. The shared MLP is denoted as M_1 and can itself have multiple layers like a NN.

5.3.3 Nearest Neighbors Neural Network

In this section, we describe the $\text{NN}_{\text{nearest}}$. This network takes a similar observation vector as AJNN_{add}. The parameter κ used by the $\text{NN}_{\text{nearest}}$ denotes how many preceding and succeeding neighbors should be included in the observation, similar as for the AJNN_{add}. The structure of the features passed to the NN is the same as for the AJNN_{add}. Therefore the description is omitted here.

For our NN model we consider the input vectors x^{glob}, x_j^{job}, $\mathcal{N}_j^{\text{red,in}}$ and $\mathcal{N}_j^{\text{red,out}}$ for $j \in U$. A graphical representation giving an overview is provided in Figure 5.2. Parameters are denoted by W and are learnable weight matrices. The learnable bias vectors are denoted by b, the internal states by h, and the final output value by o. The job specific features x_j^{job} are first preprocessed by following formula:

$$h_j^0 = \text{ReLu}(W^0 x_j^{job} + b^0), \forall j \in U. \tag{5.2}$$

The internal node state h_j^0 is a vector of dimension d_h.

The feature vectors of the κ nearest jobs are combined in the next layer. The κ nearest pre- and succeeding jobs for job j are denoted by $\mathcal{N}_j^{\text{red,in}}$ and $\mathcal{N}_j^{\text{red,out}}$, and thus we get as input vector:

$$h_j^{0'} = h^0_{\mathcal{N}_{j,1}^{\text{red,in}}} || \dots || h^0_{\mathcal{N}_{j,\kappa}^{\text{red,in}}} || h_j^0 || h^0_{\mathcal{N}_{j,1}^{\text{red,out}}} || \dots || h^0_{\mathcal{N}_{j,\kappa}^{\text{red,out}}}, \forall j \in U. \tag{5.3}$$

The combined internal state vectors $h_j^{0'}$ are then processed by a layer containing a ReLu activation function, biases and skip connections:

$$h_j^1 = \text{ReLu}(W^1 h_j^{0'} + b^1) + h_j^0, \forall j \in U. \tag{5.4}$$

The skip connection is implemented by adding h_j^0 at the end of the formula. Formula 5.4 yields an output of dimension d_h.

The global features are preprocessed by a feed-forward network with one hidden layer, including ReLu activation functions, biases, and d_h output nodes. The number of internal nodes is 20. The output vector of this network is denoted as h^{glob}.

Next, we combine the job specific features with the global features, by using a dense layer with ReLu activation functions, biases, and a skip connection and another layer with biases reducing the results to dimension one:

$$h_j^{1'} = \left[h^{\text{glob}} || h_j^1\right] h_j^3 = W^3 \left(\text{ReLu}\left(W^2 h_j^{1'} + b^2\right) + h_j^{1'}\right) + b^3, \forall j \in U. \tag{5.5}$$

Note that the addition of $h_j^{1'}$ results from using a skip connection The output h_j^3 is defined as a single value. To achieve this, we use an additional weight matrix W^3. In the above equation, W^3 is designed to be a learnable weight matrix. Nevertheless, it might be enough if W^3 is designed as a static matrix with all entries equal to 1 only used for reducing dimensionality.

These single per job values are finally combined with each other and the result is a single value which is the output of the $\text{NN}_{\text{nearest}}$:

$$o = \sum_{j \in U} h_j^3 \tag{5.6}$$

We do not use weights when summing up the results of all job-individual features, as the jobs do not have an intrinsic order and thus, using weights is not applicable.

The size of the mentioned dimensions d_h and the number of internal nodes for preprocessing the global features is a parameter that influences the number of overall weights in the network and is subject to optimization. In our experiments we use $d_h = 10$ and 20 internal nodes in the hidden layers as done for the MLP in Section 5.3.1. The selection of $d_h = 10$ follows preliminary tests where $d_h \in \{10, 20, 30\}$ were compared on the VRF-small test instance set, see Section 6.1, and $d_h = 10$ showed to work best.

5.4 Feature Sets

This section describes the features used later in the observations. Each feature is motivated, and the time complexity for calculating it is provided. All features are calculated depending on the state and the instance. In Section 5.4.1 global features are presented. These are features that represent the whole instance. The features presented in Section 5.4.2 are dependent on a individual unscheduled job j.

5.4.1 Global Features

The global features of a partial solution v are denoted by $x^{\text{glob}}(v)$. Note that we omit the explicit mentioning of v if it is clear from the context. Following global features are used within the work:

1. number of unscheduled jobs $|U|$.
   Motivation: This feature should provide the NN with information about how many jobs are unscheduled such that the output can be adapted accordingly. Time complexity: $O(n)$ (Can be reduced to $O(1)$ when storing the current layer in the state.)

2. minimum sum delay incoming $\delta_{\min}^{\text{in,m}}$: $\sum_{j \in U} \min_{k \in U \cup \{a\} \setminus \{j\}} \delta_{k,j}^m - p_{m,k}$.
   Motivation: This feature should provide the NN with the minimum delay expected when scheduling any job $j \in U$ and might be used to calculate a (bad) lower bound. Nevertheless, as one delay matrix is enough to make a reduction to the ATSP, we take the delays on machine m here. Time complexity: $O(n^2)$

3. minimum sum delay outgoing $\delta_{\min}^{\text{out,m}}$: $\sum_{j \in U \cup \{a\}} \min_{k \in U \setminus \{j\} \cup \{0\}} \delta_{j,k}^m - p_{m,j}$.
   Motivation: See the motivation for the incoming delay. Time complexity $O(n^2)$

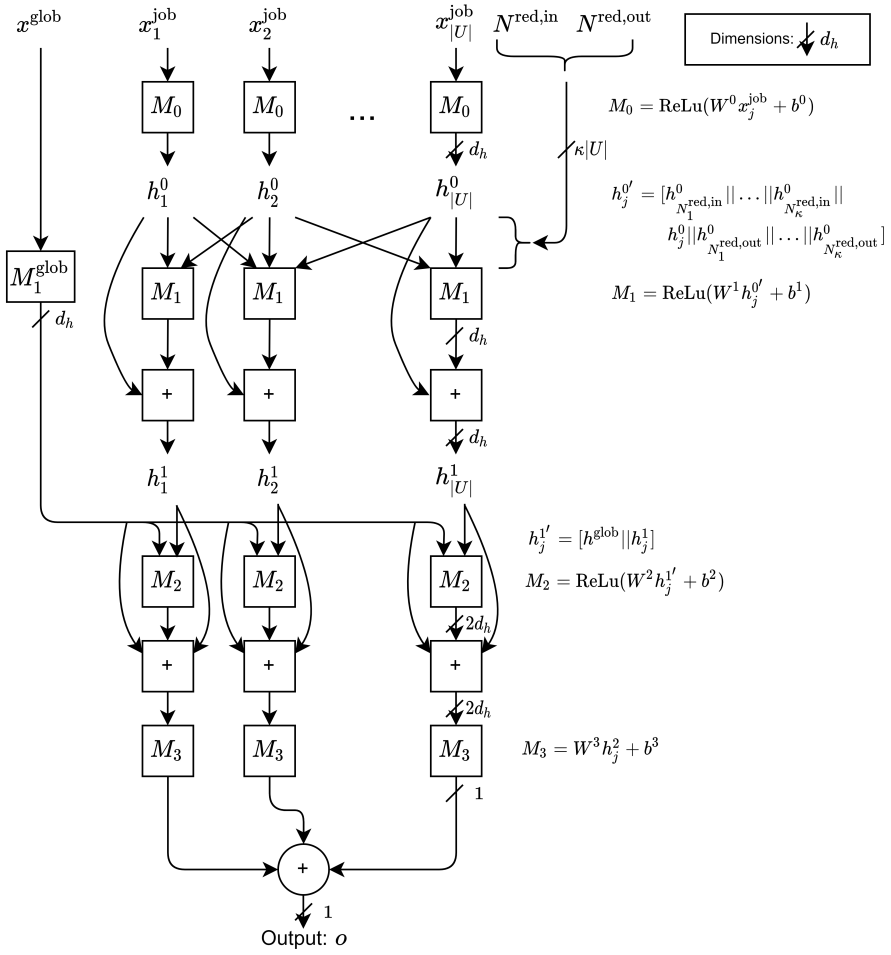

Figure 5.2: Graphical representation of the internal structure of the $\text{NN}_{\text{nearest}}$. Arrows denote data flow, and boxes denote components like layers and aggregation functions. Details about the structure are provided in Section 5.3.3.

4. maximum total job time: $\max_{j \in U} p_j^{\text{tot}}$
   Motivation: The maximum total job time is part of a lower bound to the resulting solution. Even if it can not be added directly to g, it might be used to identify solutions still missing long jobs to be scheduled. Time complexity: $O(n)$ when precomputing p^{tot}.

5. minimum release time left: $\min_{j \in U} \max(0, r_j^{\text{rel}} - p_{1,a})$
   Motivation: This feature should provide the NN with information about the release time relative to the earliest scheduling time on machine one. The minimum is provided as guidance if there are jobs that can be scheduled soon. The maximum to provide a lower bound about how much additional time is at least needed. Time complexity: $O(n)$

6. maximum release time left: $\max_{j \in U} \max(0, r_j^{\text{rel}} - p_{1,a})$
   Motivation: See the motivation for the minimum release time left.

7. minimum sum idle time incoming per machine $\omega_{\text{min}}^{\text{in}}$: $\sum_{j \in U} \min_{k \in U \cup \{a\} \setminus \{j\}} \omega_{k,j}/m$.
   Motivation: This feature should provide the NN with the anyway minimum incurred idle time per machine. The sum of the minimum idle times is divided by m. This division makes the feature independent of a change in the number of machines. Time complexity: $O(n^2)$

8. minimum sum idle time outgoing $\omega_{\text{min}}^{\text{out}}$: $\sum_{j \in U \cup a} \min_{k \in U \setminus \{j\} \cup \{0\}} \omega_{j,k}/m$.
   Motivation: See the motivation for the incoming idle times.

9. total job time on machine m: P_m^U
   Motivation: The total execution time of all unscheduled jobs on machine m yields a lower bound and is always contained in the makespan. Time complexity: $O(n)$

10. Lower Bound: LB = ITLB (without the calculations for d_out due to performance reasons)
    Motivation: A (tight) LB should help the NN to not underestimate the makespan. If the lower bound is stable in the sense that all its results are off by the same factor independent if a partial solution leading to a better or worse final state is evaluated, it might give good guidance. Time complexity: $O(mn^2)$

5.4.2 Job-Individual Features

The job-individual features of a partial solution are denoted by $x^{\text{job}}(v)$, whereas the specific features for each job $j \in U$ are denoted by $x_j^{\text{job}}(v)$. Note that we do not mention partial solution v if it is clear from the context. The job-individual features for job j consist of a global part $x_j^{\text{job,glob}}$ only dependent on job j and features dependent on the nearest neighbors $x_j^{\text{job,in}}$ and $x_j^{\text{job,out}}$. The features $x_j^{\text{job,in}}$ and $x_j^{\text{job,out}}$ are only provided when $\kappa > 0$. Following global job-individual features $x_j^{\text{job,glob}}$ are used:

1. minimum delay incoming $\min_{k \in U \cup \{a\} \setminus \{j\}} \delta^m_{k,j} - p_{m,k}$
   Motivation: This feature should provide the NN with information how much additional time is at least needed in scheduling this job. Time complexity: $O(n)$

2. minimum delay outgoing $\min_{k \in U \cup \{0\} \setminus \{j\}} \delta^m_{j,k} - p_{m,j}$
   Motivation: This feature should provide the NN with information how much additional time is at least needed in scheduling this job. Time complexity: $O(n)$

3. minimum idle time incoming per machine: $\min_{k \in U \cup \{a\} \setminus \{j\}} \omega_{k,j}/m$.
   Motivation: The minimum idle time for incoming nodes gives a measure how well job j fits after the other unscheduled jobs and the last scheduled job a. Time complexity: $O(n)$

4. minimum idle time outgoing per machine: $\min_{k \in U \cup \{0\} \setminus \{j\}} \omega_{j,k}/m$.
   Motivation: The minimum idle time for outgoing nodes gives a measure how well job j fits before the other unscheduled jobs. Time complexity: $O(n)$

5. release time left: $\max(0, r^{\text{rel}}_j - p_{1,a})$
   Motivation: When scheduling job j next the release time must be respected, thus it is also provided to the NN. Time complexity: $O(1)$

6. the job duration on machine m: $p_{m,j}$
   Motivation: We use the time on machine m, as it can be directly added to the total makespan, yielding a LB. This does not hold for processing times on the other machines. Time complexity: $O(1)$

Following job-individual features depending on the nearest preceding and succeeding jobs $x^{\text{job,in}}_j$, $x^{\text{job,out}}_j$ are included if $\kappa > 0$:

1. for each $k = 1, \ldots, \kappa$ features $x^{\text{job,in}}_{j,k}$ in respect to $\mathcal{N}^{\text{red,in}}_j[k]$:

   - Delay: $\delta^m_{k,j} - p_{m,k}$
     Motivation: This feature gives the delay when job k is scheduled directly before job j, which is reasonable as k is one of the nearest preceding jobs of j. Time complexity: $O(1)$
   - Idle Time: $\omega_{k,j}/m$
     Motivation: This feature gives the idle time introduced when job k is scheduled directly before job j, which is reasonable as k is one of the nearest preceding jobs of j. The idle time is divided by m to be independent of the number of machines m. Time complexity: $O(1)$

   If $|\mathcal{N}^{\text{red,in}}_j| < \kappa$, we fill the features up to size κ by copying the last ones. Thus, we always have the same number of features for each job j independent of the layer of the BS the observation is made at.

2. for each $k = 1, \ldots, \kappa$ features $x_{j,k}^{\text{job,out}}$ in respect to $\mathcal{N}_j^{\text{red,out}}[k]$:

   - Delay: $\delta_{j,k}^m - p_{m,j}$
     Motivation: This feature gives the delay when job k is scheduled directly after job j, which is reasonable as k is one of the nearest succeeding jobs of j. Time complexity: $O(1)$
   - Idle Time: $\omega_{j,k}/m$
     Motivation: This feature gives the idle time introduced when job k is scheduled directly after job j, which is reasonable as k is one of the nearest succeeding jobs of j. The idle time is divided by m to be independent of the number of machines m. Time complexity: $O(1)$

   If $|\mathcal{N}_j^{\text{red,out}}| < \kappa$, we fill the list up to size κ by copying the last job. Thus, we always have the same number of jobs and features for each job j independent of the layer of the BS the observation is made at.

5.5 Observations

In the following sections, the observations evaluated later are described. We provide one observation for the AJNN_{add} and the $\text{NN}_{\text{nearest}}$ and two similar observations for the MLP.

5.5.1 Observation O_{glob}

This observation is designed for use with the MLP. It uses the global features described in Section 5.4.1 with the exception that ITLB is omitted. The observation can be computed in time $O(n^2)$.

5.5.2 Observation $O_{\text{glob,itlb}}$

Similar as O_{glob}, observation $O_{\text{glob,itlb}}$ uses the global features described in Section 5.4.1 and is designed for the MLP. Nevertheless, this time also ITLB is included in the features. Thus calculating this observation takes time $O(mn^2)$.

5.5.3 Observation $O_{\text{AJNN,add}}$

This observation is designed for the AJNN_{add}. It consists of all mentioned global features presented in Section 5.4.1 and all job-individual features presented in Section 5.4.2. Note that when setting $\kappa = 0$, the feature sets $x_j^{\text{job,in}}$ and $x_j^{\text{job,out}}$ would not be included. Nevertheless, as it is mentioned in Section 6.1, we set $\kappa = 3$ for all of our tests. Thus $x_j^{\text{job,in}}$ and $x_j^{\text{job,out}}$ are always included. The observation can be computed in time $O(n^2 \max(m, \log n))$, where the additional time compared to $O_{\text{glob,itlb}}$ results from calculating $\mathcal{N}^{\text{red,in}}$ and $\mathcal{N}^{\text{red,out}}$.

5.5.4 Observation O_{nearest}

The observation O_{nearest} is designed for the $\text{NN}_{\text{nearest}}$ and is similar to $O_{\text{AJNN,add}}$ as it contains all global features of Section 5.4.1 and all job-individual features presented in Section 5.4.2. Due to the specific structure of the AJNN_{add}, we require this observation to have $\kappa > 0$ and thus always include the feature sets $x_j^{\text{job,in}}$ and $x_j^{\text{job,out}}$. Further, this observation includes $\mathcal{N}_j^{\text{red,in}}$ and $\mathcal{N}_j^{\text{red,out}}$ itself for all $j \in U$ as a feature. Note that due to simplicity during the evaluation of the AJNN_{add} the job IDs in $\mathcal{N}_j^{\text{red,in}}$ and $\mathcal{N}_j^{\text{red,out}}$ refer to the positions in the unscheduled jobs, i.e., the unscheduled jobs are sorted by their original IDs and then their indices in the resulting sequence starting at one to $|U|$ are used. This observation can be computed in time $O(n^2 \max(m, \log n))$.

CHAPTER 6

Results

In this chapter, we present the results of our computational study. Before doing so, we describe the test setup and the benchmark instances used in Section 6.1. After that, in Section 6.2, we conduct tests on the same instance classes the NNs are trained, and perform statistical tests to check whether the NNs perform better than the reference approach ITLB as heuristic function (h_{itlb}). Section 6.3 experimentally evaluates how well the NNs generalize over m and Section 6.4 evaluates the generalization over n alone and both n and m. In Section 6.5 the approximation errors of the NNs over the layers of a BS are evaluated. Finally, we compare the best approaches with the state-of-the-art method from Pourhejazy et al. [Pou+20] in Section 6.6.

6.1 Test Setup and Benchmark Instances

All benchmark tests are run with Julia 1.7.0 on a computing cluster on Intel Xeon E5540, 2.53 GHz Quad-Core nodes in single-threaded mode. Each neural network is trained ten times independently of the others but with the same settings. The tests are performed with all these ten trained models, and results are reported as boxplots if not described differently. Deterministic methods like when using ITLB as guidance function for the BS are only run once. In our work, we use four different benchmark sets, all adapted for the NWFSP-RT in [Pou+20]. The first benchmark set is denoted by VFR and can be split into small and large instances. The set of small instances VFR-small consists of instance-classes of all combinations of $n \in \{10, 20, \ldots, 60\}$ and $m \in \{5, 10, 15, 20\}$ and VFR-large of $n \in \{100, 200, \ldots, 800\}$ and $m \in \{20, 40, 60\}$. Each instance class consists of ten individual instances. Thus, there are 240 VFR-small and 240 VFR-big instances. Note that when evaluating a NN on a per instance-class basis, the boxplots contain 100 values as we perform ten runs per instance. A smaller test set called REC consists of three instances of sizes $(n = 20, m \in \{5, 10, 15\})$, $(n = 30, m \in \{10, 15\})$, $(n = 50, m \in \{10\})$, and $(n = 75, m \in \{20\})$, thus, in total 21 instances. The TA-small

instance set consists of ten instances of $n \in \{20, 50, 100\}$ and $m \in \{5, 10, 20\}$ and bigger instances of $(n = 200, m \in \{10, 20\})$ and $(n = 500, m = 20)$. Thus, it has 120 instances. We describe the instance set TA-large for completeness even if we did not test on it due to its big instance sizes. The set TA-large consists out of ten instances of sizes $n \in \{1000, 1500, 2000\}$ and $m = 20$, thus, in total 30 instances.

As baseline we use the results provided in [Pou+20] for their BS with local search from [Pou+20] ($\text{BS}_{\text{LS,pour}}$) approach which are also the so far best known results per instance. If not mentioned differently, we report the relative percentage difference (RPD) to the currently best known results $\text{BS}_{\text{LS,pour}}$. The RPD is calculated by the following formula, in which x denotes the length of solution u on a specific instance and x_{best} the result of $\text{BS}_{\text{LS,pour}}$ on the same instance:

$$\text{RPD} = (x - x_{\text{best}}) * 100/x_{\text{best}} \tag{6.1}$$

We experimentally evaluate four configurations in the later sections. These are one configuration per observation presented in Section 5.5:

- $\text{MLP}_{\text{basic}}$: The MLP as described in Section 5.3.1 together with Observation O_{glob}.
- MLP_{ITLB}: The MLP as described in Section 5.3.1 together with Observation $O_{\text{glob,itlb}}$.
- AJNN_{add}: The aggregated job neural network as presented in Section 5.3.2 with $\kappa = 3$ and Observation $O_{\text{AJNN,add}}$.
- $\text{NN}_{\text{nearest}}$: The nearest Neighbors network as presented in Section 5.3.3 with $\kappa = 3$ and Observation O_{nearest}.

All NNs are trained on instances $n \in \{10, 20\}$ and $m \in \{10, 20\}$, in total on four different instance sizes. The LBS was performed with the following settings: number of iterations $z = 3000$, minimum number of observation to start learning $\gamma = 3000$, size of the replay buffer $\rho = 5000$, beam width of the NBS calls $\beta' = 50$, expected number of samples $\alpha = 60$. Resulting from the settings approximately $180 \cdot 10^3$ training samples are created and the loss function is called approximately $14.5 \cdot 10^6$ times. This is far more than the number of weights contained in our proposed networks.

6.2 Comparison on Same Instance Sizes as Trained

In this section we compare all trained NNs on separate test instance with the same size m and n as the NNs were trained on. This is done to evaluate whether the NNs are able to improve the results compared with h_{itlb} (using ITLB as guidance function). As beam width we use $\beta \in \{1, 10, 20, 50\}$. Due to the lack of being able to reproduce the results of [Pou+20] we can not run their BS with smaller beam widths and thus only

have results for $\beta = 600$. Therefore, we use h_{itlb} as reference method here. Figures 6.1,6.2, 6.3, and 6.4 show the results as boxplots for the instances $n \in \{10, 20\}$ and $m = \{10, 20\}$. To check whether the results provided by the NNs are significantly better than h_{itlb} we perform a Wilcoxon rank-sum test with a significance level of $\alpha = 0.05$ and the alternative hypotheses that the compared method is better, i.e. yields smaller results, than h_{itlb}. The results of the statistical tests are provided in Table 6.1.

The Figures and the significance tests show that the MLP_{basic} does not outperform h_{itlb}. For the MLP_{ITLB} the results are mostly quite similar to h_{itlb} and only the configuration trained on $n = 20$ and $m = 10$ was able to significantly outperform h_{itlb} when run with $\beta \geq 10$. The $AJNN_{add}$ and $NN_{nearest}$ are able to outperform h_{itlb} significantly on ten and nine of sixteen tested configurations and performed best overall. Also note that some of the tested methods yield results better than the best known results on individual instances even if run with small beam widths of at most $\beta = 50$ compared to $\beta = 600$ which was used for $BS_{LS,pour}$. Especially for the instance class $n = 10, m = 10$ the tested methods outperformed $BS_{LS,pour}$ on average. When visually comparing the results of BS from [Pou+20] (BS_{pour}) shown as dashed lines in the figures that were evaluated with $\beta = 600$ with the trained models and h_{itlb} on $\beta = 50$ it shows that all methods outperform BS_{pour} on average. Thus, we conclude that LBS is highly effective for learning guidance functions for BS as some models outperformed the well working guidance function h_{itlb} and MLP_{basic} which does not contain ITLB is comparably or better on all instances than BS_{pour}. For a comparison of the average runtimes see Figure 6.11 in Section 6.4.

6.3 Generalization over m

This section evaluates whether the trained NNs generalize well over the number of machines m. Therefore, all NNs are evaluated on instance classes with the same number of jobs as they are trained on and $m \in \{5, 10, 15, 20\}$ of the VFR-small set. The results are shown in Figures 6.5, and 6.6. It shows that the $AJNN_{add}$ and the $NN_{nearest}$ generalize poorly over m. Especially the NNs trained on $m = 20$ perform worse than the NNs $m = 10$ on instances with $m = 5$. For instances with $n = 20$ and $m = 15$ the $AJNN_{add}$ and the $NN_{nearest}$ trained on $m = 20$ yield better results when visually compared. For instances with $n = 10$ and $m = 15$ this does not hold when visually compared. The average RPD of the MLP_{basic} gets worse when needing to generalize. Nevertheless, much less than the $NN_{nearest}$ gets worse when generalizing. The MLP_{ITLB} has a good generalization performance as the average RPD does not change much. The Wilcoxon rank-sum test performed to check whether the method trained with $m = 10$ or $m = 20$ is better than the other is given in Table 6.1 and shows that there are no significant differences on a level of $\alpha = 0.05$.

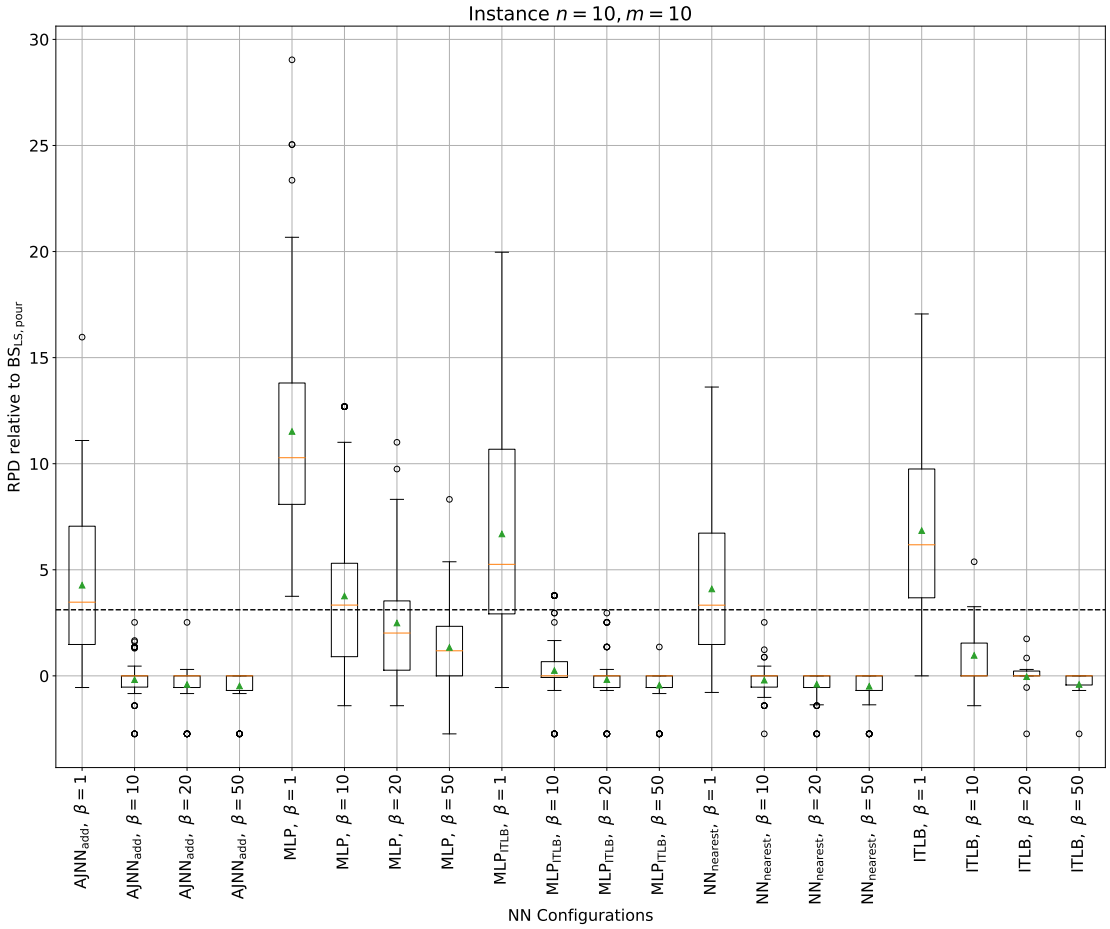

Figure 6.1: The figure shows the results of the NNs on the same instance sizes for n and m as they were trained on and the results for ITLB relative to $\mathrm{BS}_{\mathrm{LS,pour}}$ for $n = 10$ and $m = 10$. Each boxplot of the NNs represents results over ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots indicate the median and the triangles the mean results. Instances from the VFR-small set are used. The dashed line shows the average RPD of $\mathrm{BS}_{\mathrm{pour}}$.

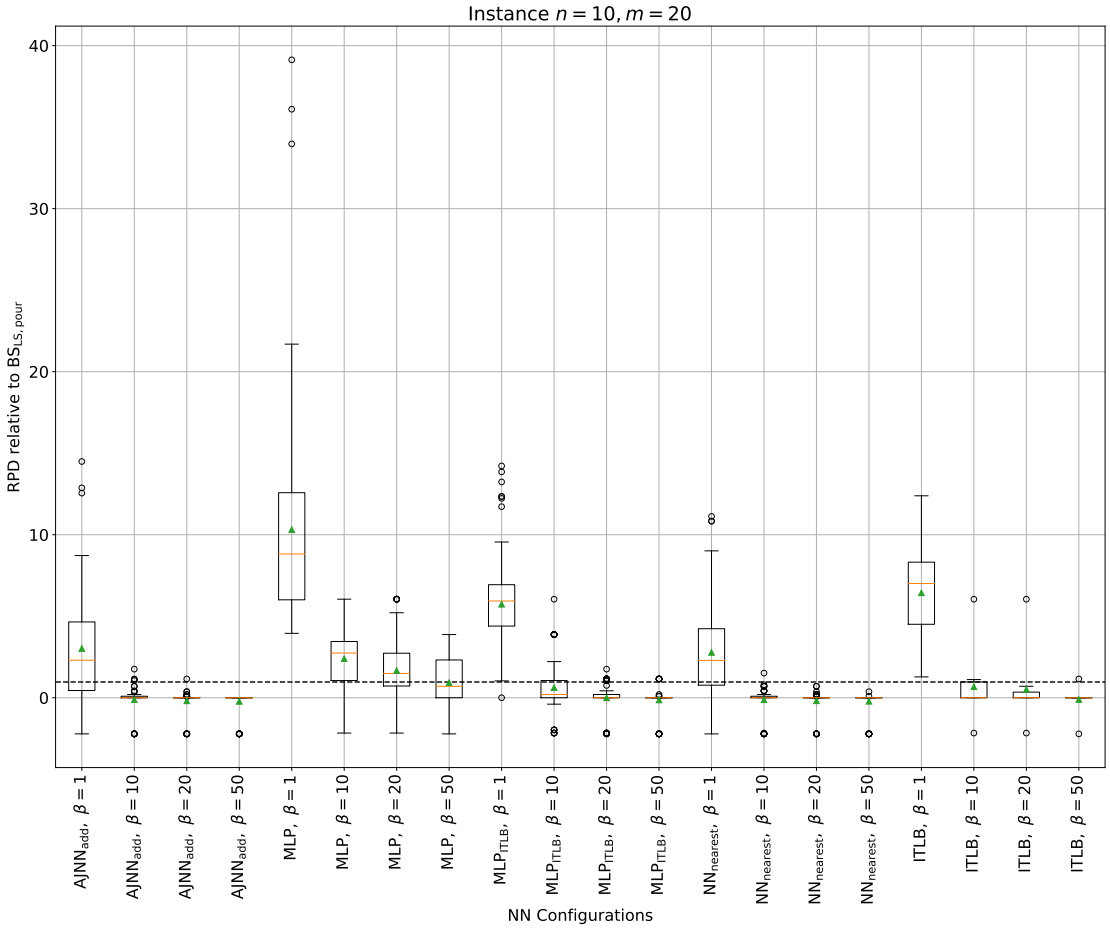

Figure 6.2: The figure shows the results of the NNs on the same instance sizes for n and m as they were trained on and the results for ITLB relative to $\mathrm{BS}_{\mathrm{LS,pour}}$ for $n = 10$ and $m = 20$. Each boxplot of the NNs represents results over ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots indicate the median and the triangles the mean results. Instances from the VFR-small set are used. The dashed line shows the average RPD of $\mathrm{BS}_{\mathrm{pour}}$.

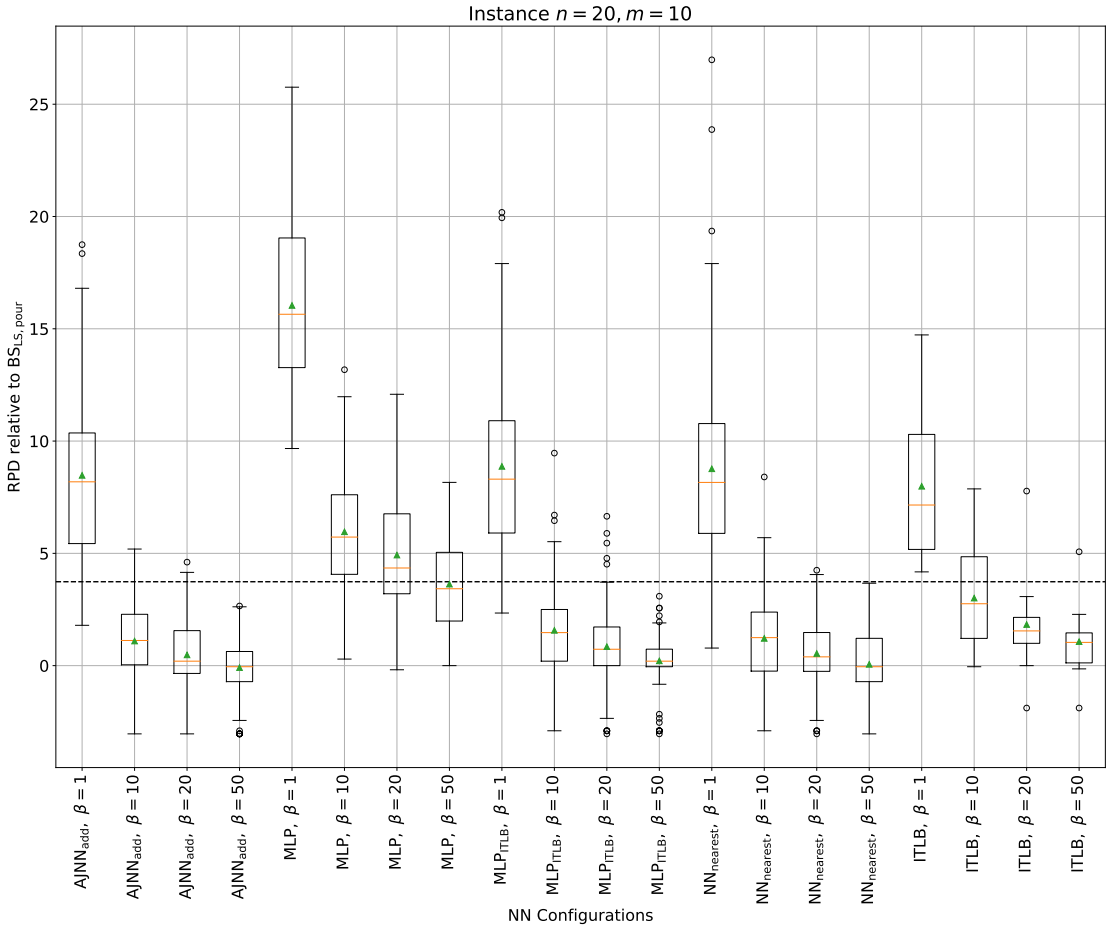

Figure 6.3: The figure shows the results of the NNs on the same instance sizes for n and m as they were trained on and the results for ITLB relative to $\mathrm{BS}_{\mathrm{LS,pour}}$ for $n = 20$ and $m = 10$. Each boxplot of the NNs represents results over ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots indicate the median and the triangles the mean results. Instances from the VFR-small set are used. The dashed line shows the average RPD of $\mathrm{BS}_{\mathrm{pour}}$.

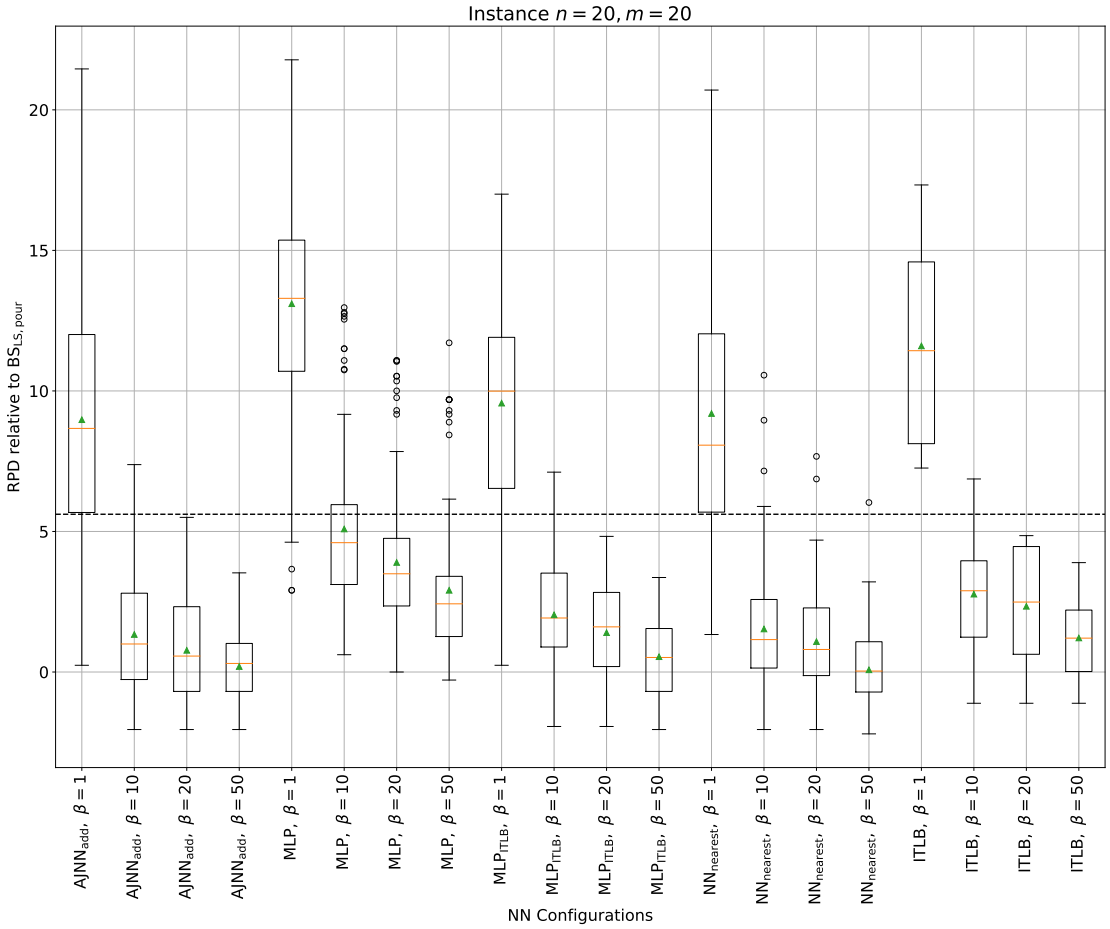

Figure 6.4: The figure shows the results of the NNs on the same instance sizes for n and m as they were trained on and the results for ITLB relative to $\mathrm{BS}_{\mathrm{LS,pour}}$ for $n = 20$ and $m = 20$. Each boxplot of the NNs represents results over ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots indicate the median and the triangles the mean results. Instances from the VFR-small set are used. The dashed line shows the average RPD of $\mathrm{BS}_{\mathrm{pour}}$.

Table 6.1: The table shows the results of Wilcoxon rank-sum tests when testing whether h_{itlb} is worse in terms of the resulting objective value than the NNs when run on the same instances as trained. As an alternative hypothesis, we use that the compared method is better than h_{itlb}. Further, $\alpha = 0.05$ is used. The tests are performed on runs with the same β values. Note that the results for h_{itlb} are ten times the same per instance as it is a deterministic method. Significant results where $H0$ can be rejected are written in bold and marked with a *****.

Methods	$\beta = 1$	$\beta = 10$	$\beta = 20$	$\beta = 50$
MLP	1.00	1.00	1.00	1.00
MLP , $n = 10,\ m = 20$	1.00	1.00	0.96	0.86
MLP , $n = 20,\ m = 10$	1.00	1.00	1.00	1.00
MLP , $n = 20,\ m = 20$	1.00	0.99	0.95	0.99
MLP_{ITLB} , $n = 10,\ m = 10$	0.28	0.08	0.20	0.75
MLP_{ITLB} , $n = 10,\ m = 20$	0.21	0.42	0.62	0.32
MLP_{ITLB} , $n = 20,\ m = 10$	0.90	***0.02**	***0.00**	***0.01**
MLP_{ITLB} , $n = 20,\ m = 20$	***0.01**	0.12	0.17	0.24
$AJNN_{add}$, $n = 10,\ m = 10$	***0.00**	***0.00**	0.13	0.32
$AJNN_{add}$, $n = 10,\ m = 20$	***0.00**	0.28	***0.04**	0.57
$AJNN_{add}$, $n = 20,\ m = 10$	0.79	***0.01**	***0.00**	***0.02**
$AJNN_{add}$, $n = 20,\ m = 20$	***0.00**	***0.03**	***0.03**	0.07
$NN_{nearest}$, $n = 10,\ m = 10$	***0.00**	***0.02**	***0.03**	0.77
$NN_{nearest}$, $n = 10,\ m = 20$	***0.00**	0.30	***0.02**	0.24
$NN_{nearest}$, $n = 20,\ m = 10$	0.76	***0.00**	***0.00**	***0.00**
$NN_{nearest}$, $n = 20,\ m = 20$	***0.01**	0.17	0.13	0.10

Table 6.2: The table shows the results of Wilcoxon rank sum tests, when testing for two models trained on the same n but different m the one is yields better results than the other. In the table e.g. $m = 10 < m = 20$ denotes the results when testing whether the model trained with $m = 10$ is better than the model trained with $m = 20$. The table shows that there are no significant differences on a level of $\alpha = 0.05$.

	$n = 10$		$n = 20$	
Methods	$m = 10 < m = 20$	$m = 20 < m = 10$	$m = 10 < m = 20$	$m = 20 < m = 10$
MLP	0.11	0.89	0.72	0.28
MLP_{ITLB}	0.58	0.42	0.47	0.53
$AJNN_{add}$	0.16	0.84	0.89	0.11
$NN_{nearest}$	0.40	0.60	0.65	0.35

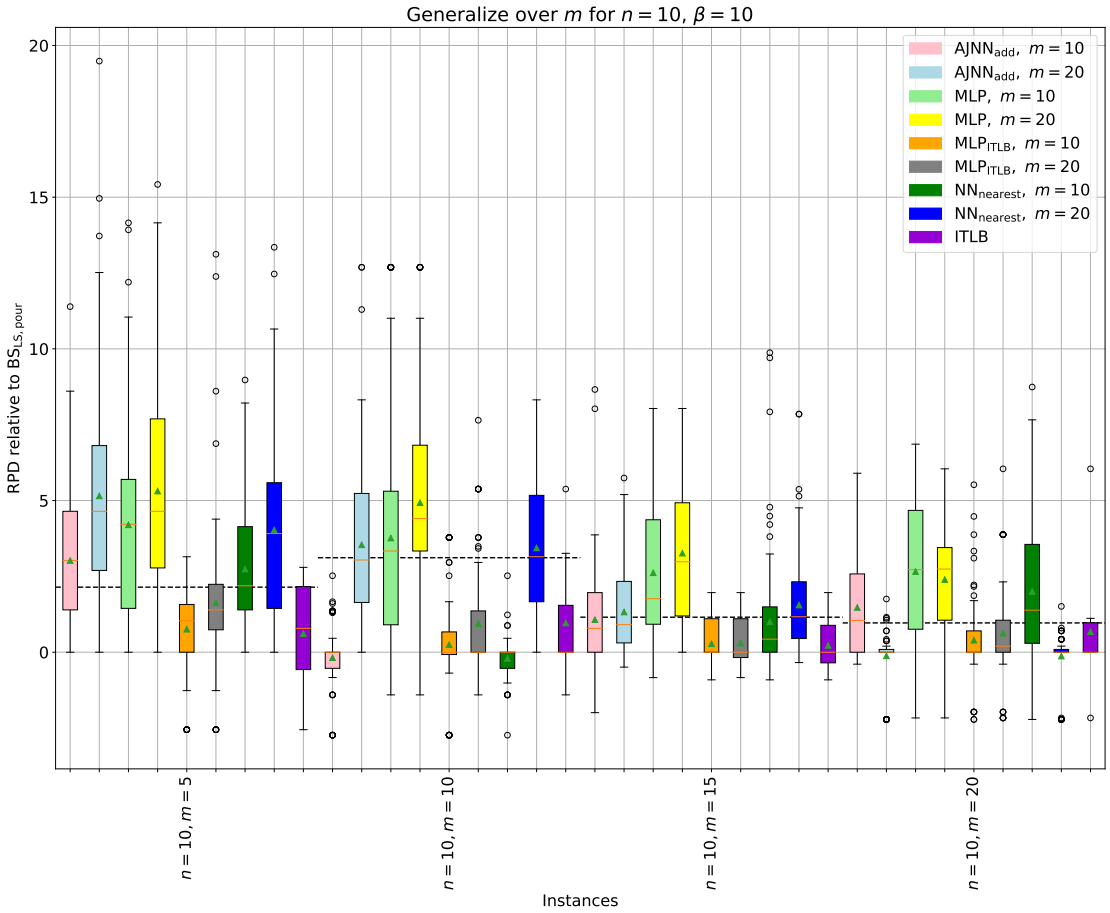

Figure 6.5: The figure shows the results of the NNs on instances with the same number of jobs as they were trained on $n = 10$ and $m \in \{5, 10, 15, 20\}$. The results for ITLB are included as reference. For evaluation $\beta = 10$ is used. All results are relative to BS_{pour}. Each boxplot of the NNs contains results for ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots show the median and the triangles the mean results. Instances from the VFR-small set are used. The dashed line shows the average RPD of BS_{pour}.

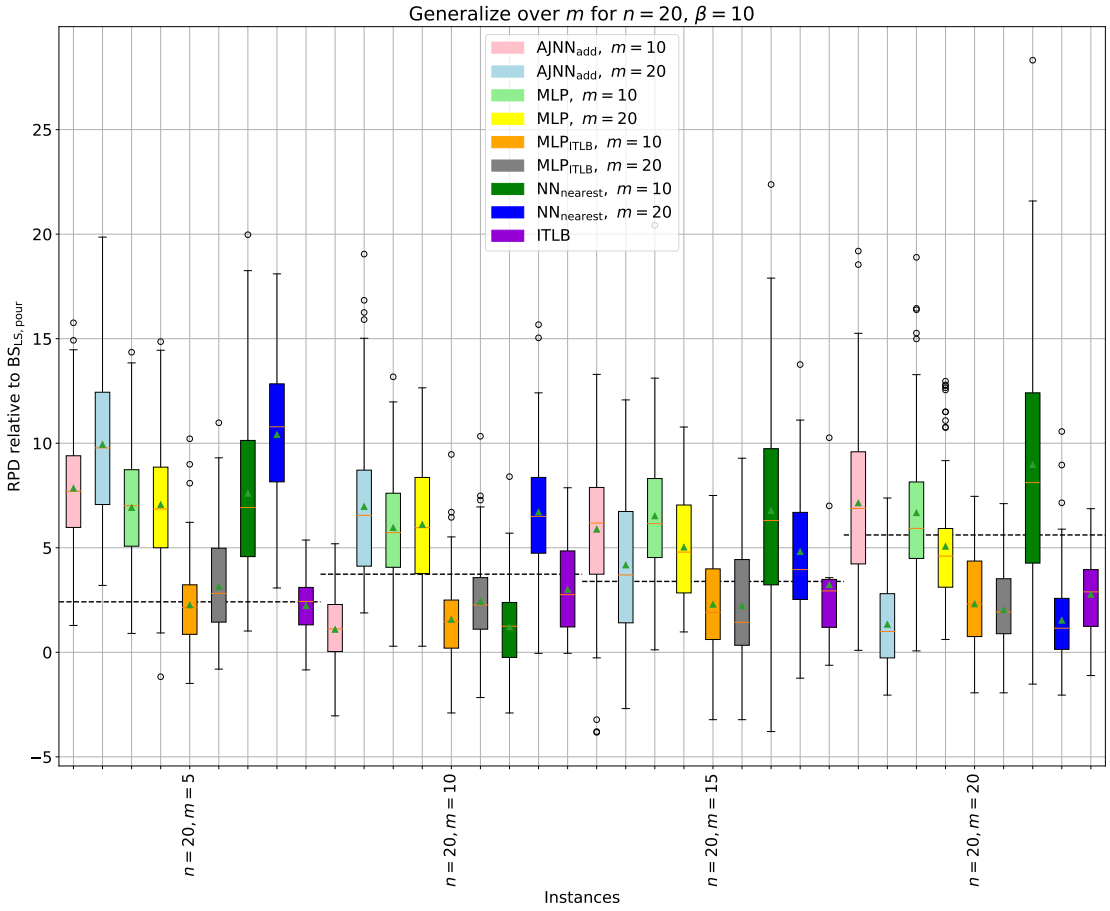

Figure 6.6: The figure shows the results of the NNs on instances with the same number of jobs as they were trained on $n = 20$ and $m \in \{5, 10, 15, 20\}$. The results for ITLB are included as reference. For evaluation $\beta = 10$ is used. All results are relative to BS_{pour}. Each boxplot of the NNs contains results for ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots show the median and the triangles the mean results. Instances from the VFR-small set are used. The dashed line shows the average RPD of BS_{pour}.

6.4 Generalization over n and m

In this section we evaluate whether the NNs generalize well over n and also over n and m at the same time. We evaluate all trained NNs on all instances in VFR-small with $\beta = 100$. The results for instances with $m \in \{5, 10, 15, 20\}$ are reported in Figures 6.7, 6.8, 6.9, and 6.10.

Generalization over n with $m = 5$:
The results in Figure 6.7 show that MLP_{ITLB} and h_{itlb} generalize well and are able to constantly improve the best known results for some instances, even if only run with $\beta = 100$. The $\text{MLP}_{\text{basic}}$ performs second best on average and shows only a moderate growth of its average RPD with increasing n

Generalization over n with $m = 10$:
The results in Figure 6.8 show that MLP_{ITLB} generalizes again well over increasing n and looks slightly better than h_{itlb}. Nevertheless, the statistical test results in Table 6.3 show that this difference is not significant. Further, AJNN_{add} and $\text{NN}_{\text{nearest}}$ trained on $n = 20, m = 10$ generalize well over increasing n as they only have a moderate growth of their average RPD. Surprisingly this does not hold for their configurations trained on $n = 10$.

Generalization over n with $m = 15$:
The results in Figure 6.9 show that MLP_{ITLB} and h_{itlb} again perform well and also the results of $\text{MLP}_{\text{basic}}$ have a moderate RPD growth.

Generalization over n with $m = 20$:
The results in Figure 6.10 show that again MLP_{ITLB} and h_{itlb} perform best. Further, $\text{MLP}_{\text{basic}}$ performs good. For this instances AJNN_{add} and $\text{NN}_{\text{nearest}}$ trained on $n = 20, m = 20$ generalize well.

Overall Results:
Overall MLP_{ITLB} and h_{itlb} show excellent performance in generalization over n and also good performance when generalizing over n and m at the same time. We observe that training on bigger instances regarding n yields better results in generalizing to bigger instances. Further, $\text{MLP}_{\text{basic}}$ shows well but not excellent performance. Nevertheless, we want to highlight that this method is the fastest and thus could in theory use a higher β and terminate in the same time as the methods containing ITLB. The average runtimes over instances of increasing n and $m = 20$ are shown in Figure 6.11. Figures for $m \in \{5, 10, 15\}$ are omitted as they show the same trend with smaller runtimes. The figure shows that AJNN_{add} has the highest average runtime, followed by $\text{NN}_{\text{nearest}}$. Both have a steep increase in runtime. The average runtimes of MLP_{ITLB} and h_{itlb} are similar, and also have a steep increase. The average runtimes of $\text{MLP}_{\text{basic}}$ increase only slowly and are the lowest. For AJNN_{add} and $\text{NN}_{\text{nearest}}$ we observed that they benefit from training on bigger instances, i.e., $n = 20$. As already observed in Section 6.3 these network types do not perform well in generalizing over m. The AJNN_{add} and $\text{NN}_{\text{nearest}}$ trained with $n = 20$ show a well generalization behavior when evaluated on instances with the same

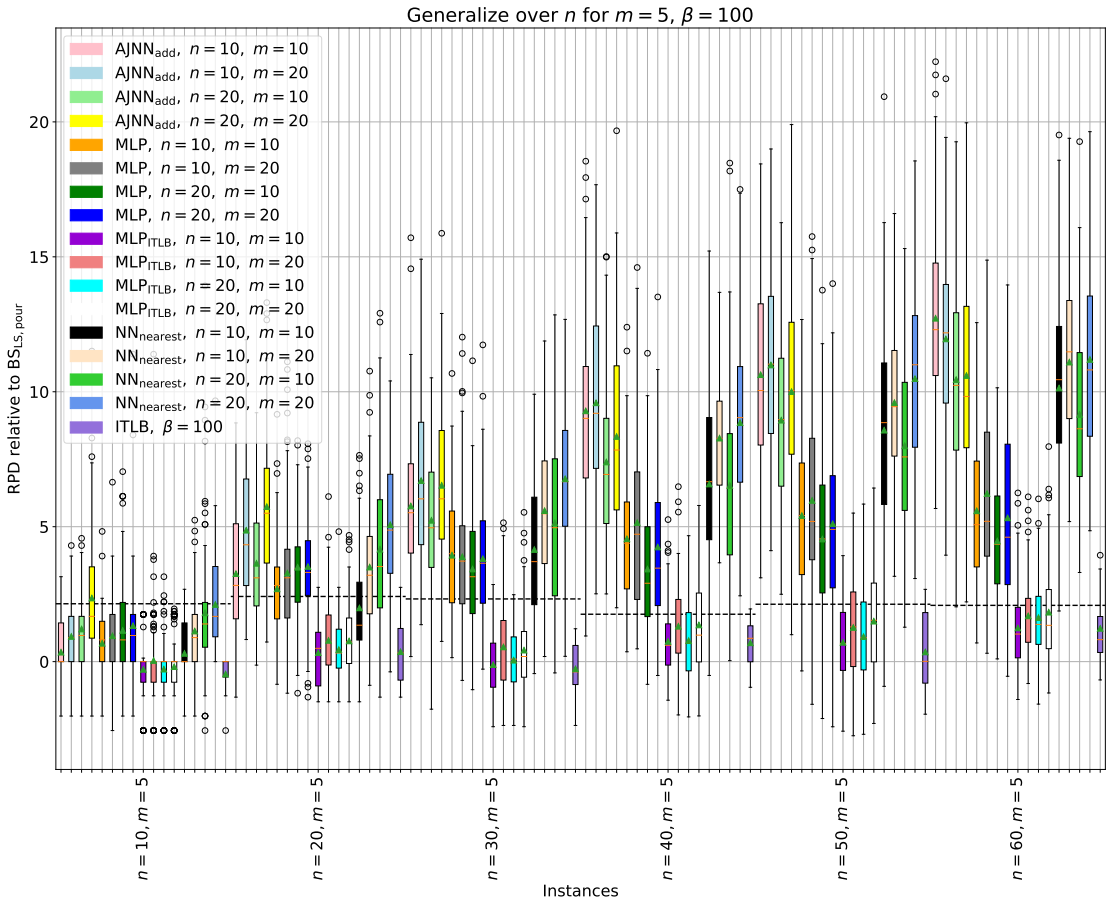

Figure 6.7: The figure shows the results of the NNs when generalizing over m and n. The results for ITLB are included as a reference. For evaluation $\beta = 100$ is used. All results are relative to BS_{pour}. Instances from the VFR-small set with $m = 5$ are used. The dashed line shows the average RPD of BS_{pour}.

number of machines m they were trained with. Nevertheless, neither configuration is statistically significantly better than h_{itlb}, when compared over all instances or instances with the same m used for training. The results of the statistical test are provided in Tables 6.4, and 6.3.

6.5 Approximation Errors over Layers

This section evaluates how well the NNs approximate the remaining cost on the different layers in the BS. The evaluation is done for all four NNs types trained on $n = 20, m = 20$ as these configurations scaled well over the number of jobs n.

Results are shown in Figures 6.12, 6.13, 6.14, 6.15, and 6.16 for $n \in \{10, 20, \ldots, 50\}$.

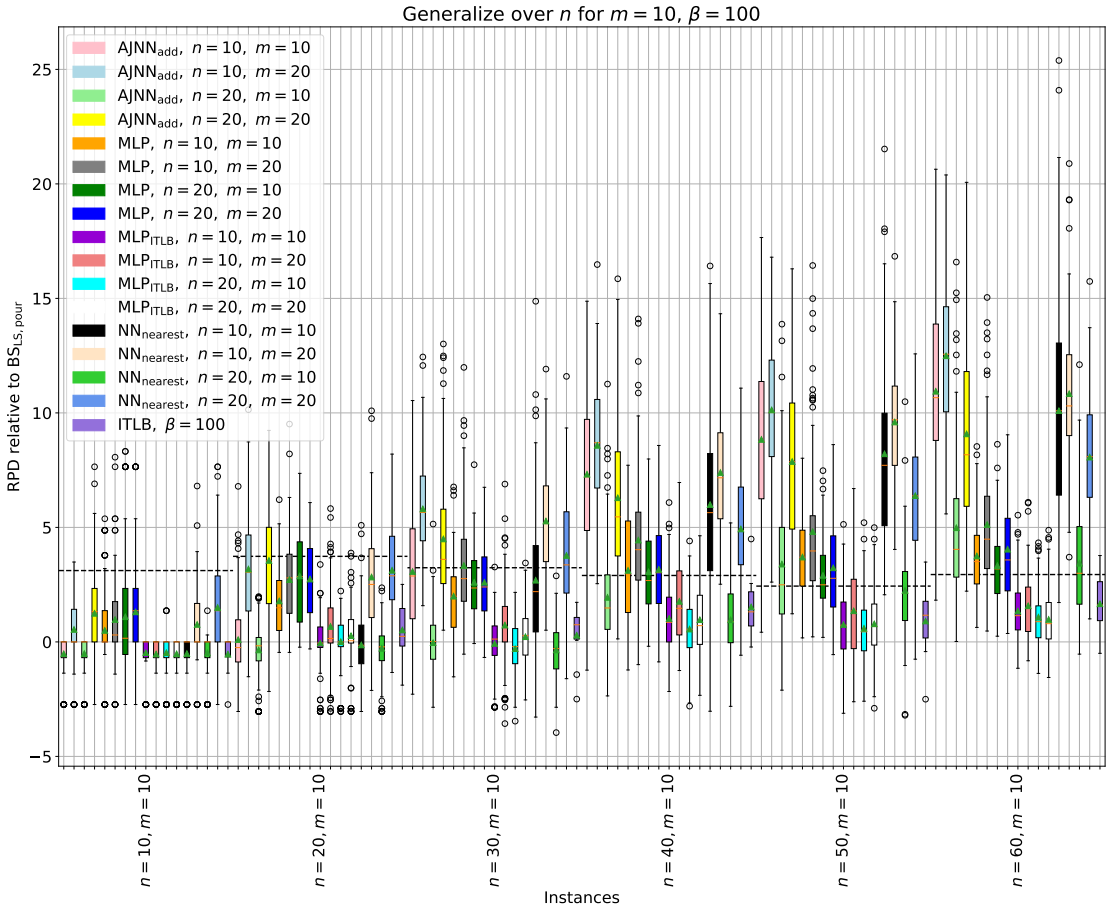

Figure 6.8: The figure shows the results of the NNs when generalizing over m and n. The results for ITLB are included as a reference. For evaluation $\beta = 100$ is used. All results are relative to BS_{pour}. Instances from the VFR-small set with $m = 10$ are used. The dashed line shows the average RPD of BS_{pour}.

Table 6.3: The table shows the results of the Wilcoxon rank-sum test when testing whether h_{itlb} is worse in terms of the resulting objective value than the NNs when evaluated over all instances of the VFR-small set which have the same m as the model was trained with and $n \in \{10, 20, \ldots, 60\}$. As an alternative hypothesis, we use that the compared method is better than h_{itlb}. In the table, one row represents the statistical test results for one NN type, and each column represents the instance class it was trained. The table shows that none of the NNs is significantly better than h_{itlb}.

Methods	$n = 10, m = 10$	$n = 10, m = 20$	$n = 20, m = 10$	$n = 20, m = 20$
MLP	0.96	0.98	0.62	0.55
MLP_{ITLB}	0.36	0.52	0.46	0.16
AJNN_{add}	1.00	0.95	0.69	0.60
$\text{NN}_{\text{nearest}}$	1.00	0.92	0.71	0.52

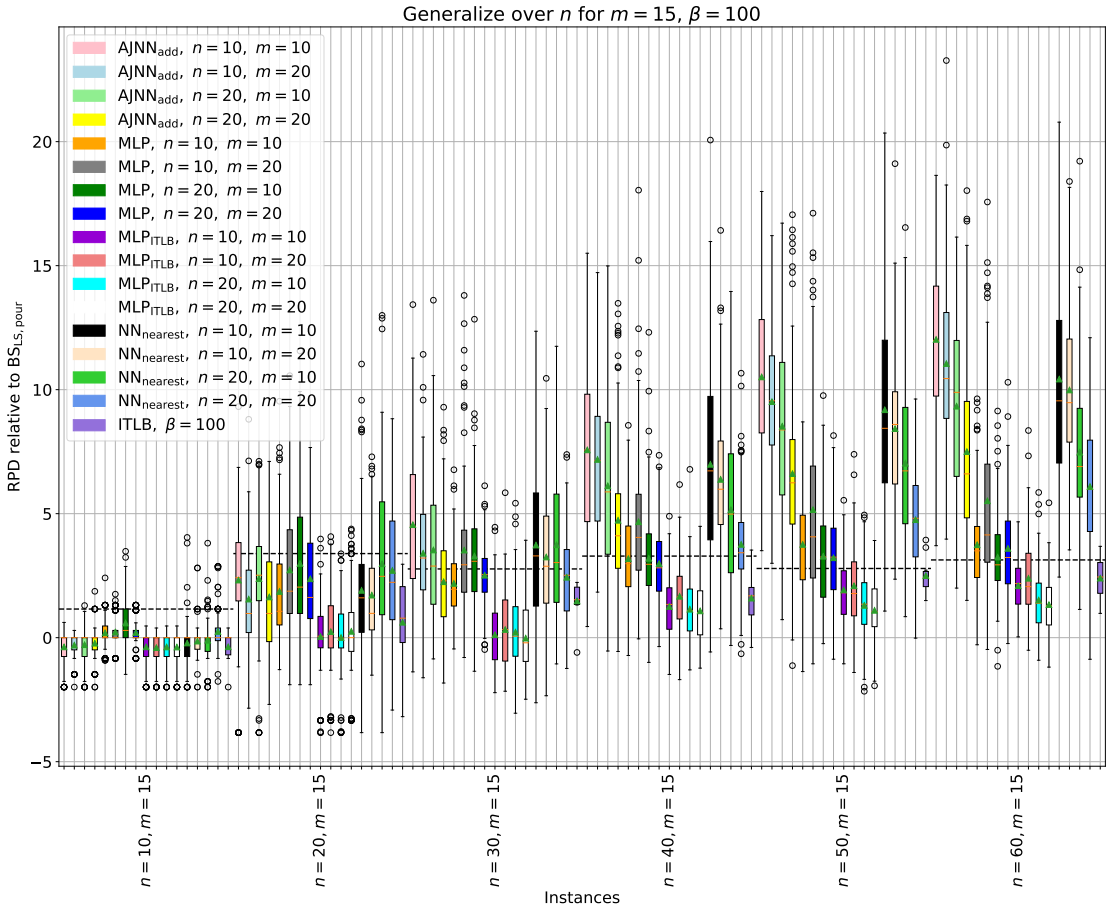

Figure 6.9: The figure shows the results of the NNs when generalizing over m and n. The results for ITLB are included as a reference. For evaluation $\beta = 100$ is used. All results are relative to BS_{pour}. Instances from the VFR-small set with $m = 15$ are used. The dashed line shows the average RPD of BS_{pour}.

Table 6.4: The table shows the results of the Wilcoxon rank-sum test when testing whether h_{itlb} is worse in terms of the resulting objective value than the NNs when evaluated over all instances of the VFR-small set. As an alternative hypothesis, we use that the compared method is better than h_{itlb}. In the table, one row represents the statistical test results for one NN type, and each column represents the instance class it was trained. The table shows that none of the NNs is significantly better than h_{itlb}.

Methods	$n = 10, m = 10$	$n = 10, m = 20$	$n = 20, m = 10$	$n = 20, m = 20$
MLP	0.79	0.94	0.78	0.95
MLP_{ITLB}	0.36	0.37	0.23	0.67
AJNN_{add}	1.00	1.00	1.00	1.00
$\text{NN}_{\text{nearest}}$	1.00	1.00	0.99	0.98

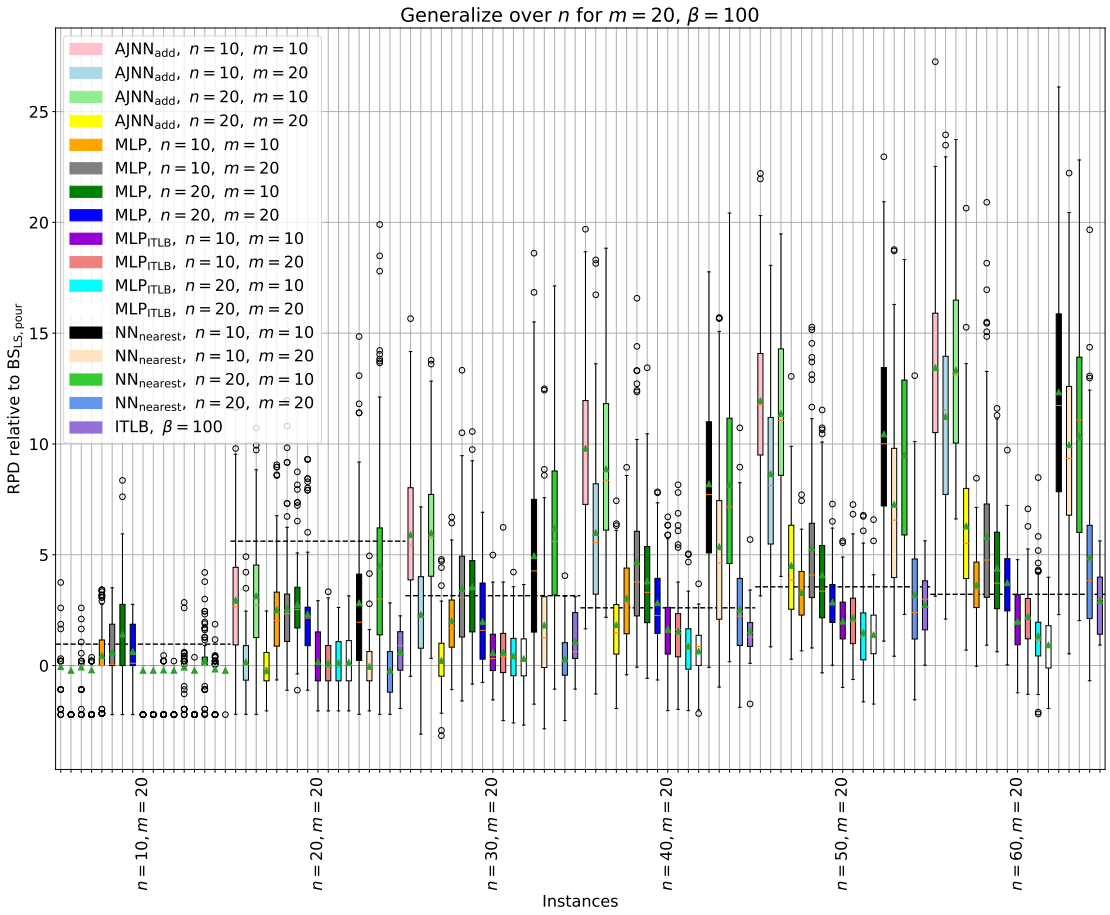

Figure 6.10: The figure shows the results of the NNs when generalizing over m and n. The results for ITLB are included as a reference. For evaluation $\beta = 100$ is used. All results are relative to BS_{pour}. Instances from the VFR-small set with $m = 20$ are used. The dashed line shows the average RPD of BS_{pour}.

Note that layer c yields instances of size $n - c$, which the NNs must approximate. A NN trained on $n = 20$ therefore must generalize if instances of sizes $n - c \geq 20$ are inputted. For AJNN_{add} and $\text{NN}_{\text{nearest}}$ it shows that for $n \in \{10, 20, , 30\}$ the boxplots without outliers stay into a $+/-25\%$RPD range and also most outliers stay within this range. For $n = 40$, still, most of the boxplots stay within the $+/-25\%$RPD range. However, the layers one to ten outliers are approximately within a $+/-50\%$RPD. The observed behavior seems reasonable, as the models were not trained on sizes of instances of $n \in \{31, 32, \ldots, 40\}$. Starting from layer 11, the figure is similar to the figure of $n = 30$. For $n = 40$ AJNN_{add} seems to underestimate the results, while $\text{NN}_{\text{nearest}}$ is closer at zero. For $n = 50$ the trend of the evaluation on $n = 40$ continues. Again, the approximation results for layers where the size of the instances is bigger than the models are trained, i.e., layers one to 20, have a higher approximation error than on the sizes where they

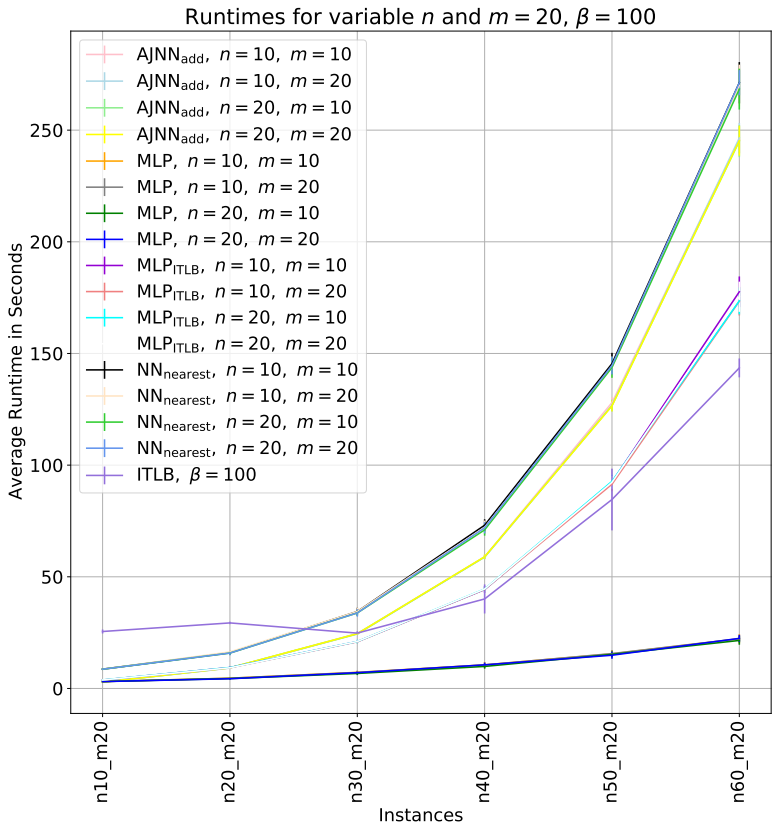

Figure 6.11: The figure shows average runtime in seconds over instances with increasing n and $m = 20$. Even though the data is not continuous, we connect the data points of the instance classes to visualize the trend.

were trained. The range of the boxplots and outliers is approximately within $[-75, 100]$. On average, the AJNN$_{\text{add}}$ again underestimates the results.

The figures show for the MLP$_{\text{basic}}$ and MLP$_{\text{ITLB}}$ that they approximate the results well for layers yielding instance sizes where they are not trained. Nevertheless, starting with $n = 40$, they tend to underestimate the results for lower layers. For higher layers, the approximation errors get higher for both MLP$_{\text{basic}}$ and MLP$_{\text{ITLB}}$. For MLP$_{\text{basic}}$ the approximation errors on the last layers in every figure become almost infinitely high, which is not shown in the figures.

6.6 Comparison with State-of-the-Art

In this section we compare the NNs with BS$_{\text{LS,pour}}$ and BS$_{\text{pour}}$. The comparison with BS$_{\text{LS,pour}}$ is important as BS$_{\text{LS,pour}}$ represents the current state-of-the-art. However, BS$_{\text{pour}}$ uses the same guidance function as BS$_{\text{LS,pour}}$ but without local search and will therefore be our primary reference when evaluating the quality of the learned functions.

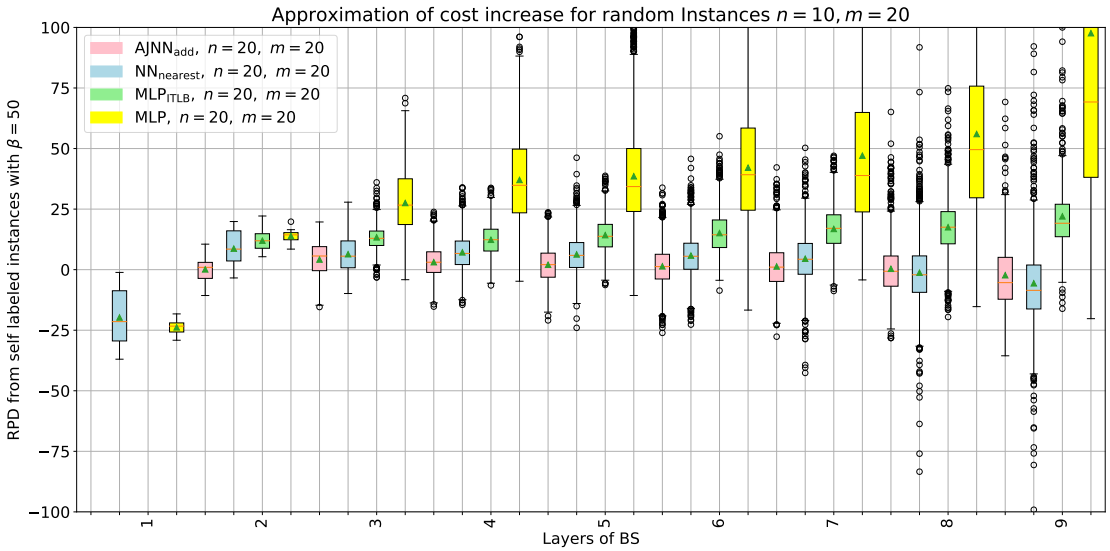

Figure 6.12: The figure shows the resulting approximation errors made by the machine learning models for $n = 10$ and $m = 20$. In total, 1000 random samples on different layers of the BS were created for each method independently by using the implementation of our replay buffer together with a fully trained NN. The beam width during the generation used for the NBS calls is $\beta' = 50$, samples are taken from BS runs with $\beta = 100$. For each configuration, the training samples and labels were generated once using the configuration itself for labeling and then evaluated once with every trained model, i.e., ten times. Each boxplot consists of a different number of samples due to the randomness of the generation. In total, there are $10 \cdot 10^3$ samples per configuration. On the vertical axis, the charts show the RPD to the labels. Thus, if values greater than zero are reported, the NN reported a higher objective value than it was, i.e., it overestimated the result. Values smaller than zero denote that the values reported by the NN are lower than the labels. The horizontal axes are fixed to a range from 100 to −100 and thus might not show all data.

Note that we frequently compare our models evaluated with $\beta = 100$ with $\text{BS}_{\text{LS,pour}}$ and BS_{pour} that were evaluated with $\beta = 600$. This is done to limit the runtime our approaches need.

The comparison on the VFR-small test instances in Table 6.5 and 6.6 show that all NNs could outperform BS_{pour} on at least eight out of 24 instance classes on average with $\beta = 100$. The best performance is shown by MLP_{ITLB} $m \in 10, 20$ which outperformed BS_{pour} on all instance classes on average and $\text{BS}_{\text{LS,pour}}$ on seven and five instance classes. Equally well performance is also shown by h_{itlb} which also outperformed BS_{pour} on average on all instance classes. Observe that AJNN_{add} and $\text{NN}_{\text{nearest}}$ outperform BS_{pour} on nine and ten instance classes on average and are able to outperform $\text{BS}_{\text{LS,pour}}$ on two to five instance classes. This behavior seems reasonable, as there are in total 12 instances with $n \leq 30$ and thus in a range where the AJNN_{add} and $\text{NN}_{\text{nearest}}$ trained on

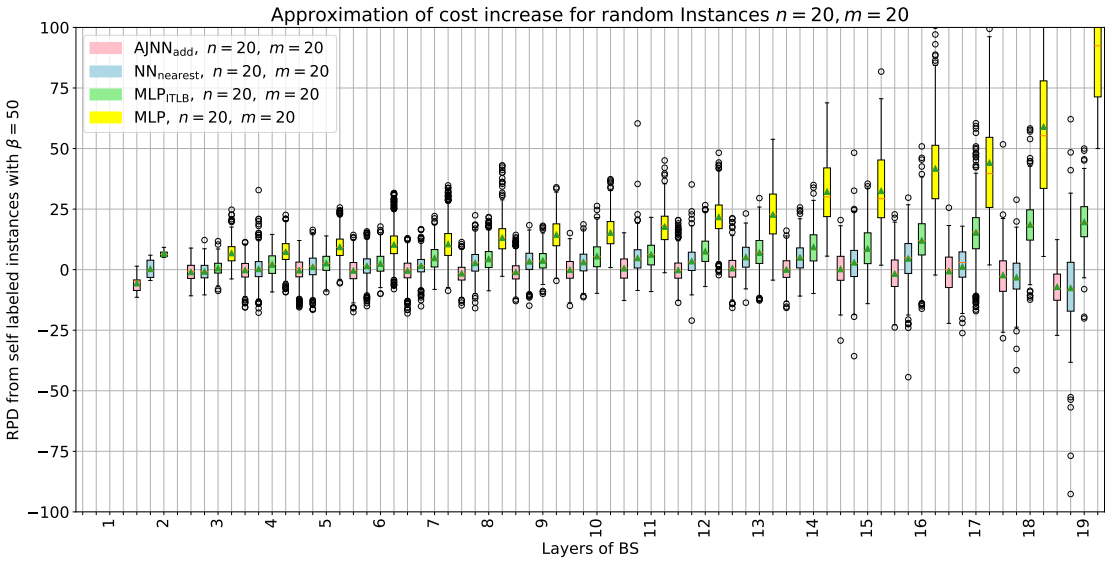

Figure 6.13: The figure shows the resulting approximation errors made by the machine learning models for $n = 20$ and $m = 20$. In total, 1000 random samples on different layers of the BS were created for each method independently by using the implementation of our replay buffer together with a fully trained NN. The beam width during the generation used for the NBS calls is $\beta' = 50$, samples are taken from BS runs with $\beta = 100$. For each configuration, the training samples and labels were generated once using the configuration itself for labeling and then evaluated once with every trained model, i.e., ten times. Each boxplot consists of a different number of samples due to the randomness of the generation. In total, there are $10 \cdot 10^3$ samples per configuration. On the vertical axis, the charts show the RPD to the labels. Thus, if values greater than zero are reported, the NN reported a higher objective value than it was, i.e., it overestimated the result. Values smaller than zero denote that the values reported by the NN are lower than the labels. The horizontal axes are fixed to a range from 100 to −100 and thus might not show all data.

$n = 20$ perform well. Even though $\text{MLP}_{\text{basic}}$ is not among the top performers in this comparison, it yields staple results by outperforming BS_{pour} on eight and 12 instance classes on average while having the lowest runtime with less than 23 seconds on average for $n = 60$ $m = 20$.

The results for TA-small in Table 6.7 and Figure 6.17 and the results for REC in Table 6.8 and Figure 6.18 show similar results as already observed for the VFR-small data set. The MLP_{ITLB} trained on $n = 20$ and $m = 20$ outperformed BS_{pour} on nine out of 12 tested TA-small instance classes and on all REC instance classes. The same holds for h_{itlb}.

Results for selected instance classes of the VFR-large instance set are shown in Figure 6.19, and Table 6.9. Due to the size of the instances and the resulting runtime, we

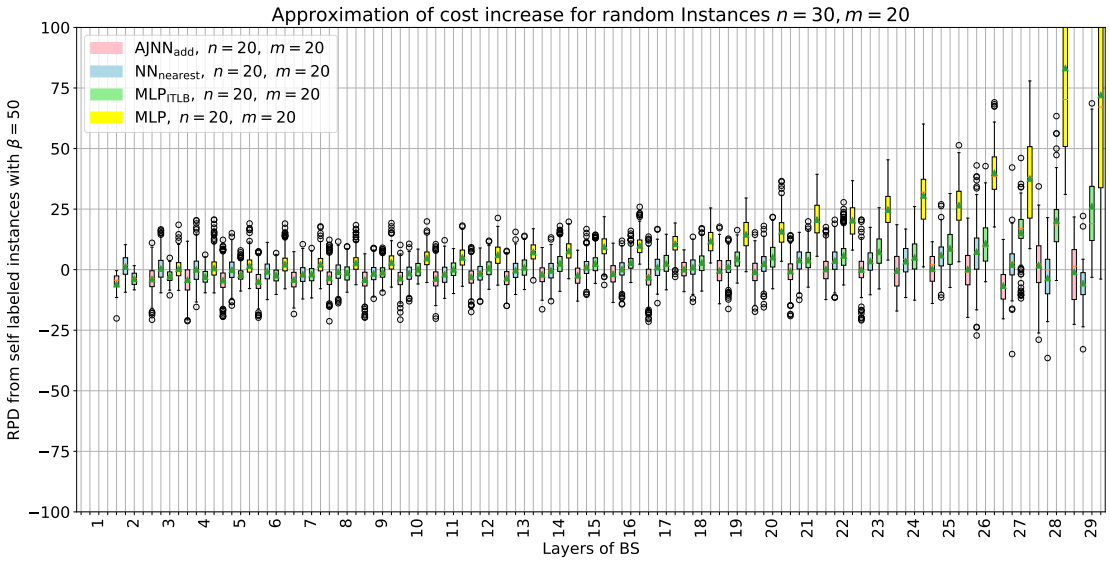

Figure 6.14: The figure shows the resulting approximation errors made by the machine learning models for $n = 30$ and $m = 20$. In total, 1000 random samples on different layers of the BS were created for each method independently by using the implementation of our replay buffer together with a fully trained NN. The beam width during the generation used for the NBS calls is $\beta' = 50$, samples are taken from BS runs with $\beta = 100$. For each configuration, the training samples and labels were generated once using the configuration itself for labeling and then evaluated once with every trained model, i.e., ten times. Each boxplot consists of a different number of samples due to the randomness of the generation. In total, there are $10 \cdot 10^3$ samples per configuration. On the vertical axis, the charts show the RPD to the labels. Thus, if values greater than zero are reported, the NN reported a higher objective value than it was, i.e., it overestimated the result. Values smaller than zero denote that the values reported by the NN are lower than the labels. The horizontal axes are fixed to a range from 100 to −100 and thus might not show all data.

performed tests only on instances up to $n = 400$ and $m \in \{20, 40, 60\}$. The $\text{MLP}_{\text{basic}}$ was only tested on instances with $n = 100$. On these instances MLP_{ITLB} could on average outperform BS_{pour}. Further, MLP_{ITLB} also performs better than h_{itlb} with $\beta = 100$ on average. However, $\text{MLP}_{\text{basic}}$ could not outperform BS_{pour} on average. This might be due to the difference in beam widths, that might have a bigger impact the bigger the instances get.

Overall we could outperform $\text{BS}_{\text{LS,pour}}$ on average on 10 out of 24 VFR-small instance classes, six out of 12 TA-small instance classes, four out of seven REC instance classes, and on zero out of 12 tested VFR-large instance classes. Further, we outperformed BS_{pour} on all instances of VFR-small, TA-small and REC, except for two TA-small instances with $n =\in \{100, 500\}$ with $\beta = 100$.

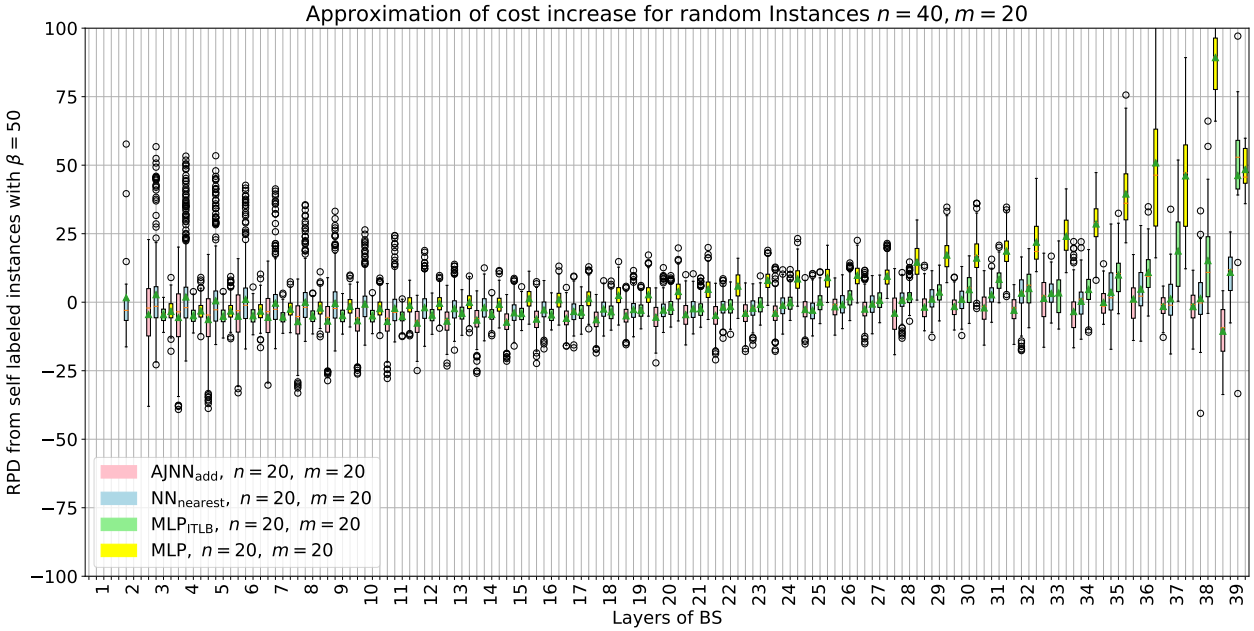

Figure 6.15: The figure shows the resulting approximation errors made by the machine learning models for $n = 40$ and $m = 20$. In total, 1000 random samples on different layers of the BS were created for each method independently by using the implementation of our replay buffer together with a fully trained NN. The beam width during the generation used for the NBS calls is $\beta' = 50$, samples are taken from BS runs with $\beta = 100$. For each configuration, the training samples and labels were generated once using the configuration itself for labeling and then evaluated once with every trained model, i.e., ten times. Each boxplot consists of a different number of samples due to the randomness of the generation. In total, there are $10 \cdot 10^3$ samples per configuration. On the vertical axis, the charts show the RPD to the labels. Thus, if values greater than zero are reported, the NN reported a higher objective value than it was, i.e., it overestimated the result. Values smaller than zero denote that the values reported by the NN are lower than the labels. The horizontal axes are fixed to a range from 100 to -100 and thus might not show all data.

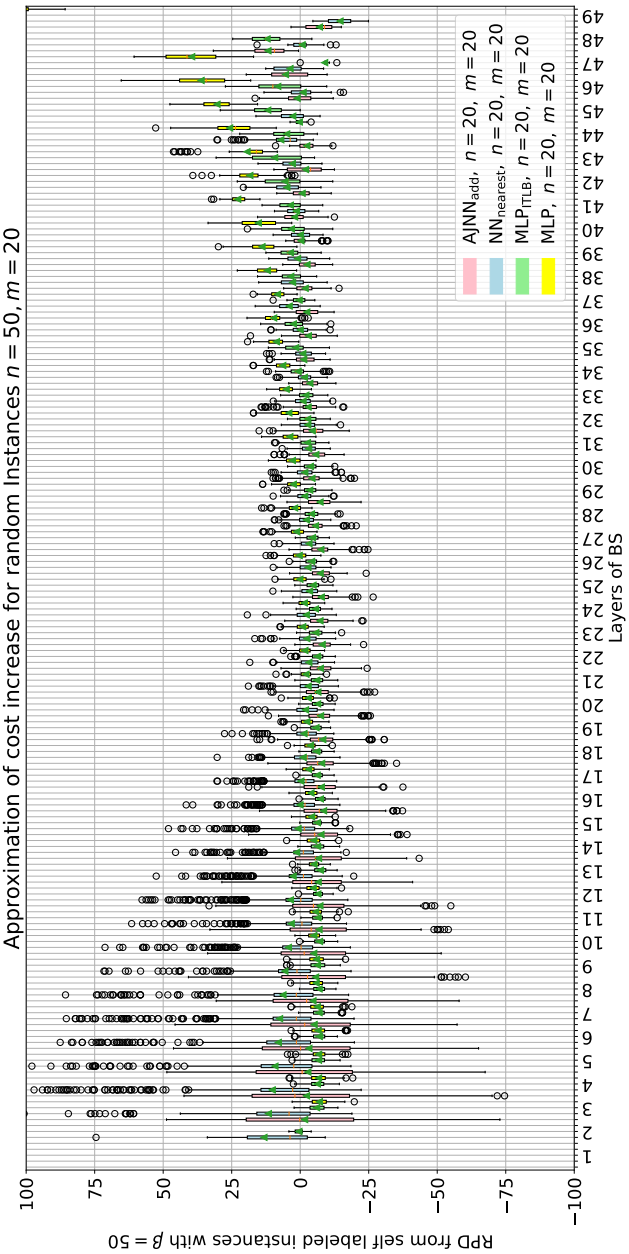

Figure 6.16: The figure shows the resulting approximation errors made by the machine learning models for $n = 50$ and $m = 20$. In total, 1000 random samples on different layers of the BS were created for each method independently by using the implementation of our replay buffer together with a fully trained NN. The beam width during the generation used for the NBS calls is $\beta' = 50$, samples are taken from BS runs with $\beta = 100$. For each configuration, the training samples and labels were generated once using the configuration itself for labeling and then evaluated once with every trained model, i.e., ten times. Each boxplot consists of a different number of samples due to the randomness of the generation. In total, there are $10 \cdot 10^3$ samples per configuration. On the vertical axis, the charts show the RPD to the labels. Thus, if values greater than zero are reported, the NN reported a higher objective value than it was, i.e., it overestimated the result. Values smaller than zero denote that the values reported by the NN are lower than the labels. The horizontal axes are fixed to a range from 100 to −100 and thus might not show all data.

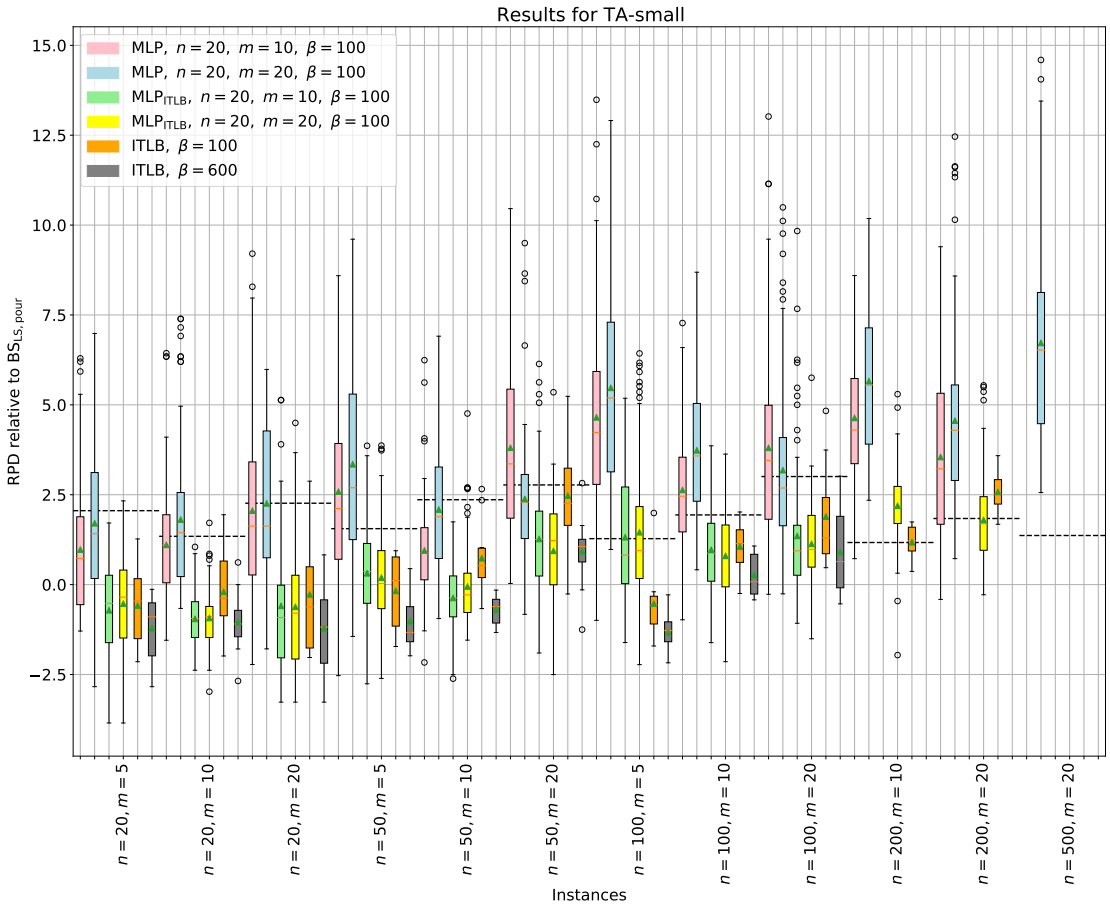

Figure 6.17: The figure visualizes the results from Table 6.7. Each boxplot of the NNs represents results over ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots indicate the median and the triangles the mean results. Instances from the TA-small set are used. The dashed line shows the average RPD of BS_{pour}.

We did not test our approach on VFR-large and TA-large instances because the runtimes exceeded our time limits.

Table 6.5: This Table shows the first part of the results shown in Figures 6.7, 6.8, 6.9, and 6.10. It contains the average objective values, the standard deviations, and the runtimes. Only results for NNs trained on instance sizes $n = 20$ and $m \in \{10, 20\}$ are shown as they performed in general better than the NNs trained with $n = 10$. The results were produced with $\beta = 100$. The instances are from the VFR-small instance set. For easier comparison we added the results of h_{itlb} as well as BS_{pour} and $BS_{LS,pour}$. The best results in each row are marked with ** whereas we mark results that are the best except for $BS_{LS,pour}$ with *. If an approach is the overall best approach, it is only marked by **. The last row of the table shows how often the individual approach was able to perform at least as good as $BS_{LS,pour}$ and BS_{pour}. The Table continues in Table 6.6.

Instances		$AJNN_{add}$, $m = 10$			$AJNN_{add}$, $m = 20$			MLP, $m = 10$			MLP, $m = 20$			MLP_{ITLB}, $m = 10$		
n	m	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.
10	5	815.9	45.5	2.6	825.2	48.2	2.6	815.5	46.0	3.1	817.1	49.1	3.1	804.1	47.0	3.6
10	10	1254.9	42.3	2.7	1277.3	53.5	2.7	1274.5	34.6	3.1	1277.6	45.0	3.1	1255.4	43.2	3.7
10	15	1647.2	69.4	2.8	1648.3	71.1	2.8	1661.6	69.8	3.1	1655.3	67.8	3.1	1645.6	67.8	3.8
10	20	1976.4	76.8	2.8	1973.8	74.9	2.8	2005.7	86.4	3.1	1990.4	83.4	3.1	1973.6	75.0	3.8
20	5	1538.4	102.7	6.7	1569.4	103.9	6.7	1535.8	92.6	4.4	1536.6	100.5	4.5	1490.8	93.2	7.0
20	10	**2053.5	43.5	7.8	2134.5	74.7	7.8	2120.3	60.9	4.4	2117.5	61.6	4.4	2061.0	45.7	8.0
20	15	2634.4	109.2	8.5	2614.7	87.9	8.4	2649.3	91.3	4.4	2633.4	87.3	4.4	**2573.0	85.4	8.5
20	20	3129.4	121.2	9.2	**3027.8	125.3	9.2	3118.6	130.9	4.4	3104.5	124.3	4.3	3038.9	131.1	9.4
30	5	2228.8	75.7	17.0	2256.3	85.5	17.0	2190.5	70.5	6.7	2198.8	76.4	7.0	2119.1	60.7	13.8
30	10	2824.0	81.9	19.7	2952.2	118.3	19.6	2898.7	94.9	6.6	2898.9	100.0	7.2	2816.7	82.2	16.1
30	15	3626.3	105.9	22.0	3581.7	106.0	22.0	3617.8	92.1	6.8	3590.9	78.3	7.2	3510.4	82.6	18.5
30	20	4319.1	137.0	24.4	*4085.4	119.7	24.5	4220.4	144.1	6.7	4156.8	118.5	7.1	4093.7	117.4	21.0
40	5	3036.9	131.9	37.7	3063.1	139.7	37.6	2924.4	117.2	10.2	2947.3	117.6	10.1	2849.6	104.4	24.0
40	10	3693.8	107.2	44.4	3850.9	127.7	44.4	3734.2	80.7	10.1	3738.0	90.2	10.2	*3644.1	95.0	30.1
40	15	4668.1	173.9	51.3	4607.0	157.1	51.5	4539.8	131.6	10.1	4530.6	122.7	10.4	4449.3	102.1	37.5
40	20	5548.8	229.7	58.8	5190.1	157.4	59.0	5290.2	172.2	9.9	5240.1	147.5	10.6	5141.4	160.5	44.5
50	5	3754.7	173.3	74.3	3791.2	190.0	74.3	3603.2	166.1	15.0	3622.6	169.3	15.7	3479.0	151.9	41.0
50	10	4586.2	158.2	88.5	4785.2	187.6	88.3	4564.5	108.4	15.0	4579.2	127.3	15.3	*4461.7	96.7	55.5
50	15	5763.7	196.2	106.0	5662.1	196.9	106.2	5485.4	122.9	14.8	5482.1	118.1	15.3	5380.6	116.3	72.4
50	20	6858.0	246.9	126.7	6435.4	175.9	126.5	6409.2	181.7	15.3	6333.5	121.7	14.9	6249.1	137.7	92.7
60	5	4498.7	195.6	136.5	4504.7	204.0	136.3	4254.8	149.2	22.2	4289.7	175.6	22.3	4139.0	125.7	66.3
60	10	5568.8	193.2	167.0	5786.3	245.2	165.7	5481.4	124.8	22.2	5519.0	138.1	22.8	5362.0	113.7	96.8
60	15	6829.5	259.3	203.9	6715.4	253.5	203.0	6449.0	133.3	21.7	6471.1	147.9	22.7	6341.4	141.8	133.1
60	20	8105.6	305.4	245.1	7601.8	255.8	245.3	7466.7	231.2	21.5	7419.3	166.4	22.4	7248.8	190.3	173.6
Outperformed:		$BS_{pour} = 9$, $BS_{LS,pour} = 5$			$BS_{pour} = 9$, $BS_{LS,pour} = 3$			$BS_{pour} = 8$, $BS_{LS,pour} = 0$			$BS_{pour} = 12$, $BS_{LS,pour} = 0$			$BS_{pour} = 24$, $BS_{LS,pour} = 7$		

Table 6.6: This Table shows the first part of the results shown in Figures 6.7, 6.8, 6.9, and 6.10. It contains the average objective values, the standard deviations, and the runtimes. Only results for NNs trained on instance sizes $n = 20$ and $m \in \{10, 20\}$ are shown as they performed in general better than the NNs trained with $n = 10$. The results were produced with $\beta = 100$. The instances are from the VFR-small instance set. For easier comparison we added the results of h_{itlb} as well as BS_{pour} and $BS_{LS,pour}$. The best results in each row are marked with ** whereas we mark results that are the best except for $BS_{LS,pour}$ with *. If an approach is the overall best approach, it is only marked by **. The last row of the table shows how often the individual approach was able to perform at least as good as $BS_{LS,pour}$ and BS_{pour}. The Tables first part is given in Table 6.5.

Instances		MLP_{ITLB}, $m = 20$			$NN_{nearest}$, $m = 10$			$NN_{nearest}$, $m = 20$			ITLB, $\beta = 100$			$BS_{LS,pour}$		BS_{pour}	
n	m	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	$\overline{x}$	$\sigma_{\overline{x}}$
10	5	804.8	47.0	3.6	821.0	52.0	8.6	823.2	44.6	8.5	**802.6	45.2	25.4	806.2	43.4	823.1	40.3
10	10	1254.8	42.1	3.7	1255.3	42.9	8.6	1280.2	48.0	8.6	**1254.6	41.9	25.2	1261.5	45.5	1300.5	45.9
10	15	**1645.6	67.4	3.7	1650.2	71.4	8.6	1656.2	74.5	8.6	1645.7	66.8	25.7	1652.1	73.3	1671.2	77.0
10	20	1973.4	74.9	3.8	1982.4	79.8	8.6	1974.5	74.6	8.6	**1973.4	74.9	25.5	1977.9	76.8	1997.6	95.9
20	5	1495.6	95.1	7.1	1546.0	104.1	13.6	1559.6	103.5	13.6	*1489.8	96.3	27.7	**1484.1	89.7	1519.8	92.0
20	10	2066.4	50.7	7.9	2055.0	43.4	14.3	2125.3	61.1	14.2	2071.7	53.5	28.5	2061.1	51.8	2137.9	69.3
20	15	2579.2	85.7	8.6	2649.7	120.6	15.0	2642.0	86.7	15.0	2588.9	92.1	29.3	2573.2	87.4	2661.0	122.4
20	20	3039.7	132.8	9.3	3170.8	170.1	15.8	3028.2	126.1	15.9	3055.1	146.7	29.3	3035.6	130.1	3205.7	162.9
30	5	2127.0	65.8	13.7	2226.6	83.8	25.9	2261.3	76.0	25.9	**2112.3	59.7	20.9	2118.0	57.8	2167.4	69.6
30	10	2831.6	90.5	16.0	**2815.1	82.3	28.5	2931.7	104.7	28.3	2833.2	81.0	21.9	2825.2	81.4	2917.2	109.1
30	15	**3501.9	74.7	18.7	3633.8	120.1	31.1	3587.9	85.0	31.2	3554.8	77.9	23.5	3503.0	77.5	3600.4	107.2
30	20	4088.8	113.3	21.0	4329.0	186.9	33.8	4088.0	112.9	33.8	4120.6	106.9	24.8	**4076.0	96.3	4204.5	123.0
40	5	2866.0	108.6	24.0	3011.8	134.4	50.5	3077.8	128.4	50.6	*2847.6	107.7	22.8	**2827.6	94.9	2876.8	84.9
40	10	3658.4	77.1	30.3	3658.5	91.2	57.0	3802.2	111.0	56.7	3678.7	75.3	27.1	**3624.1	91.2	3729.1	96.7
40	15	*4446.3	98.5	37.4	4625.5	154.9	63.5	4564.2	121.6	63.7	4468.6	85.0	32.5	**4398.9	85.2	4543.5	93.8
40	20	*5130.5	159.5	44.5	5512.2	247.2	71.0	5225.7	165.4	71.9	5177.2	181.6	40.1	**5097.4	151.7	5231.0	193.1
50	5	3498.9	159.8	41.3	3722.1	172.4	92.8	3807.7	180.9	93.4	*3460.2	166.0	32.3	**3447.3	142.7	3520.3	141.1
50	10	4470.4	93.1	55.6	4535.4	125.0	107.1	4719.6	167.3	106.3	4476.1	106.6	59.1	**4436.4	113.7	4544.8	135.4
50	15	*5370.1	115.1	73.0	5684.5	185.0	123.7	5563.7	129.9	124.2	5444.3	97.6	66.9	**5311.7	101.7	5459.3	93.2
50	20	*6242.7	97.8	92.3	6743.1	270.0	143.7	6354.4	189.3	144.6	6327.7	168.6	84.6	**6157.8	96.2	6376.2	105.4
60	5	4147.2	138.5	66.0	4445.7	182.8	158.9	4528.9	196.8	161.0	*4123.0	131.4	68.2	**4072.9	120.6	4157.7	123.8
60	10	*5356.9	126.6	97.5	5486.6	154.0	189.4	5732.6	173.1	190.1	5393.5	138.6	85.1	**5304.5	81.6	5460.8	108.7
60	15	*6329.8	125.8	133.3	6713.6	210.6	224.8	6627.0	173.0	226.5	6396.6	126.6	110.2	**6247.3	136.2	6441.4	90.7
60	20	*7219.5	172.4	175.4	7892.7	343.3	268.4	7502.8	318.0	271.3	7363.1	205.1	143.5	**7152.9	135.3	7382.4	149.8
Outperformed:		$BS_{pour} = 24$, $BS_{LS,pour} = 5$			$BS_{pour} = 10$, $BS_{LS,pour} = 4$			$BS_{pour} = 10$, $BS_{LS,pour} = 2$			$BS_{pour} = 24$, $BS_{LS,pour} = 5$			-		-	

Table 6.7: This Table shows the results shown in Figure 6.17. It contains the average objective values, the standard deviations, and the runtimes. Only results for $\text{MLP}_{\text{basic}}$ and MLP_{ITLB} trained on instance sizes $n = 20$ and $m \in \{10, 20\}$ are shown because of there good former performance. The results were produced with $\beta = 100$. The instances are from the TA-small instance set. For easier comparison we added the results of h_{itlb} as well as BS_{pour} and $\text{BS}_{\text{LS,pour}}$. The best results in each row are marked with **** whereas we mark results that are the best except for $\text{BS}_{\text{LS,pour}}$ with *****. If an approach is the overall best approach, it is only marked by ****. The last row of the table shows how often the individual approach was able to perform at least as good as $\text{BS}_{\text{LS,pour}}$ and BS_{pour}. Due to the high runtimes during evaluation, not all NNs are evaluated over all instances.

Instances		MLP, $m = 10$			MLP, $m = 20$			MLP_{ITLB}, $m = 10$			MLP_{ITLB}, $m = 20$			ITLB, $\beta = 100$			$\text{BS}_{\text{LS,pour}}$		BS_{pour}	
n	m	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	$\overline{x}$	$\sigma_{\overline{x}}$
20	5	1546.3	48.3	5.0	1557.6	52.6	4.9	**1520.5	44.5	7.6	1523.3	43.6	7.6	1522.5	42.7	22.5	1531.9	54.7	1563.5	61.3
20	10	2068.4	111.3	4.9	2082.8	116.5	4.9	**2025.8	98.2	8.4	2026.4	99.7	8.5	2041.6	107.2	25.9	2045.5	100.6	2072.9	102.0
20	20	3115.0	80.6	4.8	3121.0	62.1	4.8	3034.4	75.2	9.9	**3033.5	68.1	10.1	3044.2	72.6	27.3	3052.7	61.6	3121.4	66.6
50	5	3521.8	138.6	15.4	3547.8	138.1	14.8	3444.0	119.0	42.9	3439.8	116.4	43.2	**3426.7	98.5	46.9	3433.4	114.3	3486.5	111.6
50	10	4531.9	97.4	15.7	4583.0	114.4	14.9	**4473.0	108.3	58.7	4487.4	111.0	61.3	4523.1	129.1	58.8	4489.5	89.3	4595.9	114.3
50	20	6385.0	170.8	15.6	6297.5	126.3	14.7	6228.9	131.4	101.3	*6209.0	139.8	107.3	6302.7	98.4	89.2	**6151.3	105.0	6321.3	111.2
100	5	6832.9	244.8	91.7	6886.3	247.7	92.1	6615.0	180.8	345.4	6624.2	197.3	370.0	**6494.7	153.9	205.1	6528.9	132.6	6612.3	136.7
100	10	8646.4	219.0	91.8	8739.0	219.9	91.6	8506.2	180.8	614.5	*8491.7	179.5	686.1	8513.5	155.4	374.1	**8425.0	178.2	8587.9	179.9
100	20	11635.8	322.6	91.4	11566.1	286.1	91.8	11361.1	232.4	1360.9	*11336.4	169.4	1565.6	11421.4	177.3	772.4	**11209.4	102.6	11546.7	170.3
200	10	16711.5	289.2	1026.6	16875.0	328.0	1083.0	-	-	-	16321.5	195.9	13266.4	16159.7	159.9	4993.4	**15971.7	140.3	*16158.8	162.4
200	20	21637.8	480.8	1030.4	21849.6	544.7	1076.5	-	-	-	*21272.0	286.2	47018.1	21434.3	158.6	12809.4	**20896.6	183.5	21281.1	229.0
500	20	-	-	-	52014.0	1246.1	48100.6	-	-	-	-	-	-	-	-	-	**48739.4	228.9	*49405.2	324.2
Outperformed:		$\text{BS}_{\text{pour}} = 4$, $\text{BS}_{\text{LS,pour}} = 0$			$\text{BS}_{\text{pour}} = 4$, $\text{BS}_{\text{LS,pour}} = 0$			$\text{BS}_{\text{pour}} = 8$, $\text{BS}_{\text{LS,pour}} = 4$			$\text{BS}_{\text{pour}} = 9$, $\text{BS}_{\text{LS,pour}} = 4$			$\text{BS}_{\text{pour}} = 9$, $\text{BS}_{\text{LS,pour}} = 5$			-		-	

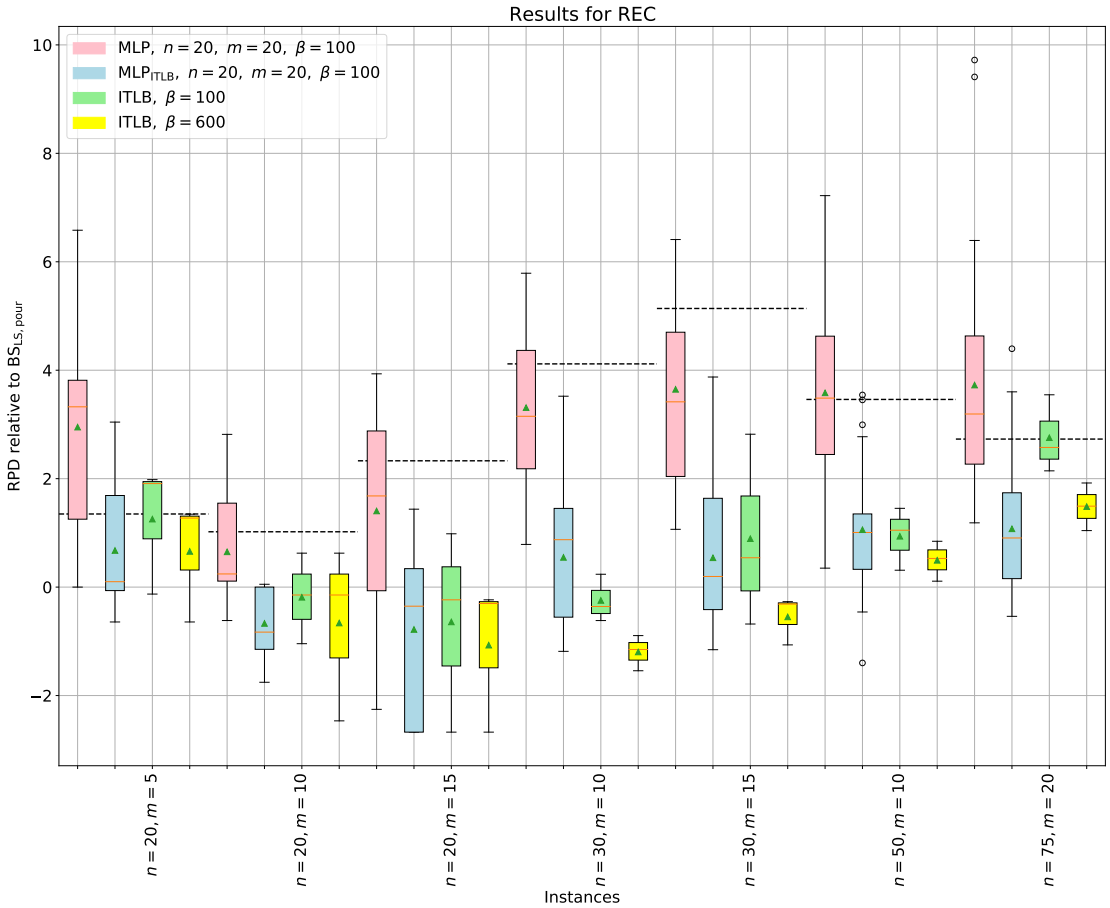

Figure 6.18: The figure visualizes the results from Table 6.8. Each boxplot of the NNs represents results over ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots indicate the median and the triangles the mean results. Instances from the TA-small set are used. The dashed line shows the average RPD of BS_{pour}.

Table 6.8: This Table shows the results shown in Figure 6.18. It contains the average objective values, the standard deviations, and the runtimes. Only results for $\text{MLP}_{\text{basic}}$ and MLP_{ITLB} trained on instance sizes $n = 20$ and $m = 20$ are shown because of there good former performance. The results were produced with $\beta = 100$. The instances are from the TA-small instance set. For easier comparison we added the results of h_{itlb} as well as BS_{pour} and $\text{BS}_{\text{LS,pour}}$. The best results in each row are marked with ** whereas we mark results that are the best except for $\text{BS}_{\text{LS,pour}}$ with *. If an approach is the overall best approach, it is only marked by **. The last row of the table shows how often the individual approach was able to perform at least as good as $\text{BS}_{\text{LS,pour}}$ and BS_{pour}.

Instances		MLP, $m = 20$, $\beta = 100$			MLP_{ITLB}, $m = 20$, $\beta = 100$			ITLB, $\beta = 100$			ITLB, $\beta = 600$			$\text{BS}_{\text{LS,pour}}$		BS_{pour}	
n	m	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	$\overline{x}$	$\sigma_{\overline{x}}$
20	5	1503.3	64.1	4.5	1470.3	62.7	7.0	1478.3	52.8	20.9	***1469.7**	53.3	27.6	****1460.7**	66.7	1480.0	59.4
20	10	2041.3	77.5	4.4	2014.7	79.4	7.8	2024.0	68.2	21.7	****2014.0**	60.1	28.6	2028.3	81.1	2048.7	74.1
20	15	2642.8	66.2	4.4	2585.7	52.7	8.5	2589.3	57.1	18.5	****2578.0**	41.0	34.0	2606.0	36.4	2667.3	90.8
30	10	2984.6	90.3	6.6	2904.4	72.4	15.8	2881.7	72.9	18.6	****2854.0**	62.5	55.7	2888.7	67.9	3007.7	90.8
30	15	3665.2	129.8	6.6	3557.4	170.8	18.0	3571.0	195.8	26.9	****3518.3**	152.5	65.0	3537.3	142.6	3720.7	223.7
50	10	4701.1	94.0	13.5	4586.4	65.3	55.3	4581.0	50.6	49.0	***4560.7**	19.7	165.0	****4538.3**	32.8	4695.3	46.8
75	20	8982.5	326.6	35.5	***8752.8**	280.5	404.3	8898.3	271.2	174.4	8788.0	254.1	1258.6	****8658.7**	222.0	8895.3	239.1
Outperformed:		$\text{BS}_{\text{pour}} = 4$, $\text{BS}_{\text{LS,pour}} = 0$			$\text{BS}_{\text{pour}} = 7$, $\text{BS}_{\text{LS,pour}} = 2$			$\text{BS}_{\text{pour}} = 6$, $\text{BS}_{\text{LS,pour}} = 3$			$\text{BS}_{\text{pour}} = 7$, $\text{BS}_{\text{LS,pour}} = 4$			-		-	

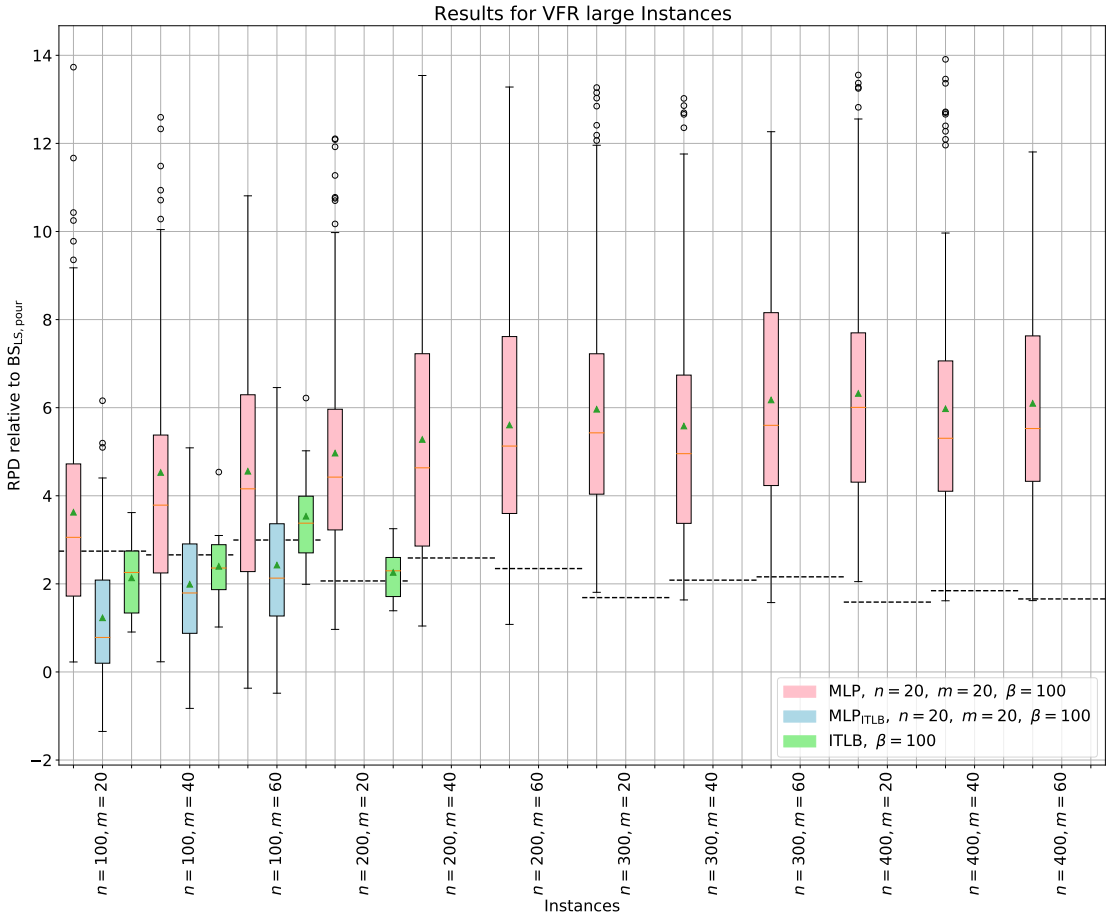

Figure 6.19: The figure visualizes the results from Table 6.9. Each boxplot of the NNs represents results over ten runs with individually trained networks over ten instances. The horizontal lines in the boxplots indicate the median and the triangles the mean results. Selected Instances from the VFR-large set are used. The dashed line shows the average RPD of BS_{pour}.

Table 6.9: This Table shows the results shown in Figure 6.19. It contains the average objective values, the standard deviations, and the runtimes. Only results for $\text{MLP}_{\text{basic}}$ and MLP_{ITLB} trained on instance sizes $n = 20$ and $m \in \{10, 20\}$ are shown because of there good former performance. The results were produced with $\beta = 100$. Also h_{itlb} is shown with $\beta \in \{100, 600\}$ on selected instances. Selected instances from the VFR-large instance set are used. For easier comparison we added the results of BS_{pour} and $\text{BS}_{\text{LS,pour}}$. The best results in each row are marked with ** whereas we mark results that are the best except for $\text{BS}_{\text{LS,pour}}$ with *. If an approach is the overall best approach, it is only marked by **. The last row of the table shows how often the individual approach was able to perform at least as good as $\text{BS}_{\text{LS,pour}}$ and BS_{pour}. Due to the high runtimes during evaluation, not all NNs are evaluated over all instances.

Instances		MLP, $m = 20$			MLP_{ITLB}, $m = 20$			ITLB, $\beta = 100$			$\text{BS}_{\text{LS,pour}}$		BS_{pour}	
n	m	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	runt.	$\overline{x}$	$\sigma_{\overline{x}}$	$\overline{x}$	$\sigma_{\overline{x}}$
100	20	11559.7	289.6	92.6	***11292.4**	173.0	1391.4	11394.0	140.2	769.2	****11155.9**	113.1	11462.2	196.6
100	40	16105.8	513.0	92.5	***15714.5**	259.7	2917.4	15776.6	150.0	1469.2	****15408.3**	176.7	15817.7	236.4
100	60	19623.3	532.1	92.9	***19223.8**	336.3	4952.6	19432.2	279.3	2320.1	****18769.2**	192.6	19331.2	272.8
200	20	21840.2	522.0	1075.0	-	-	-	21277.0	186.2	11203.6	****20807.3**	180.7	***21236.6**	137.9
200	40	29487.8	844.2	1070.7	-	-	-	-	-	-	****28009.3**	161.9	***28734.3**	272.8
200	60	35485.0	923.7	1056.9	-	-	-	-	-	-	****33601.8**	190.9	***34390.4**	256.1
300	20	32065.6	825.9	5376.0	-	-	-	-	-	-	****30261.0**	222.7	***30771.9**	257.7
300	40	42586.3	1145.5	5442.1	-	-	-	-	-	-	****40334.4**	172.7	***41174.8**	221.1
300	60	51306.7	1340.4	5376.6	-	-	-	-	-	-	****48323.4**	443.6	***49366.3**	506.0
400	20	42045.4	1061.7	17794.8	-	-	-	-	-	-	****39544.8**	180.2	***40171.9**	180.3
400	40	55511.0	1486.2	17544.7	-	-	-	-	-	-	****52380.8**	161.6	***53346.8**	238.4
400	60	66500.7	1601.3	17546.4	-	-	-	-	-	-	****62679.5**	361.4	***63718.4**	386.9
Outperformed:		$\text{BS}_{\text{pour}} = 0$, $\text{BS}_{\text{LS,pour}} = 0$			$\text{BS}_{\text{pour}} = 3$, $\text{BS}_{\text{LS,pour}} = 0$			$\text{BS}_{\text{pour}} = 2$, $\text{BS}_{\text{LS,pour}} = 0$			-		-	

CHAPTER 7

Conclusion

In this work, we considered a variant of the FSP which is the NWFSP-RT. It was shown how the NWFSP could be reduced to the ATSP, and we described which auxiliary data is precomputed to speed up later the state transition in the BS and the dominance check. Further, we explained BS and presented the LBS framework. We described how these general methods can be effectively tailored towards the NWFSP-RT. More specifically this included states for the NWFSP-RT, the branching in BS, how the state transitions work, and we gave a dominance definition applicable to two BS nodes on the same layer. Following that, several lower bounds for the NWFSP-RT including a novel lower bound called ITLB were described. The novel bound combines the idea of Taillard [Tai93] and a lower bound for the ATSP. To integrate the NWFSP-RT into the LBS framework, we described how the training data is generated, and proposed NN types, including the two novel NN types AJNN_{add} and $\text{NN}_{\text{nearest}}$. The latter two aim to aggregate the individual "contribution" of each job. They follow the idea of GNNs. The AJNN_{add} uses a MLP and applies it to all job-specific and global features. The $\text{NN}_{\text{nearest}}$ extends this idea further by passing internal states of the κ nearest neighbors to each job. As a third option, we suggested using a MLP. For all three NN types, we constructed feature sets and combined them to four observations. The observations for the AJNN_{add} and the $\text{NN}_{\text{nearest}}$ consists of global parts derived from the instance and the state as a whole and job-specific parts that provide further data about each unscheduled job.

We conducted tests on how well the NN types perform when used to guide a BS on the same instance size as they were trained. The tests were performed for every NN types' four trained configurations, and the results were compared to using ITLB as a guidance function for a BS. The AJNN_{add} and the $\text{NN}_{\text{nearest}}$ performed significantly better on several configurations tested with different beam widths than ITLB.

Further, we performed tests on how well the NNs generalize when run for a different number of machines and number of jobs than there were trained. Results show that $\text{MLP}_{\text{basic}}$ and MLP_{ITLB} generalize well over the number of jobs and also over the number

of jobs and machines at the same time. The AJNN_{add} and the $\text{NN}_{\text{nearest}}$ do not perform well when the number of machines they are trained on is changed. The configurations trained on $n = 20$ generalize well over the number of jobs. Note, that no NN was able to significantly outperform using ITLB as guidance function.

To better understand why some NNs generalize well and others not, we evaluated the approximation error rates when generalizing over the number of jobs. We observed that the AJNN_{add}, the $\text{NN}_{\text{nearest}}$, and the MLP_{ITLB} have error rates within $+/-25\%$RPD for layers resulting in instance sizes they were trained. The MLP_{ITLB} and the $\text{MLP}_{\text{basic}}$ stay within the $+/-25\%$RPD range. This even holds for BS layers yielding instance sizes MLP_{ITLB} and the $\text{MLP}_{\text{basic}}$ are not trained. However, the $\text{MLP}_{\text{basic}}$ has increasing error rates when the instance sizes yielded by the layers go below nine.

During our tests, we constantly outperformed the state-of-the-art results of Pourhejazy et al. [Pou+20] on single instances even though our tests were performed with way smaller beam widths than $\beta = 600$ and without a local search. Overall when running with $\beta = 100$ our approach MLP_{ITLB} trained on instances of size $n = 20$ and $m = 20$ outperformed BS_{pour} on 43 out of 46 tested instance classes on average and was able to outperform the former state-of-the-art method $\text{BS}_{\text{LS,pour}}$ on 11 instance classes. Stable performance and guidance behavior were also shown by $\text{MLP}_{\text{basic}}$ which is also faster than MLP_{ITLB}. Overall when running with $\beta = 100$, BS_{pour} was on average outperformed on 20 out of 55 tested instance classes by $\text{MLP}_{\text{basic}}$.

CHAPTER 8

Future Work and Open Questions

It took up to multiple days to train the NNs with LBS on instances of size $n = 20, m = 20$. Most of the time was spent creating and labeling new training samples. It might be possible to parallelize the training sample generation to reduce the time needed. Further options may be to perform the nested beam search calls only up to a certain depth and then use a greedy method to complete the solution or use the trained model to approximate the remaining solution cost.

The calculation of the features used as an input for the NNs has a high theoretical computational time complexity and takes most of the time the overall beam search needs. Finding additional promising features that can be computed faster or improving the computation of the current features is important to apply the approach to instances with hundreds of jobs and is left as an open question. Another suggestion is to use a reduced graph to compute the features, i.e., a graph where only a maximum number of neighbors or arcs for each job reside. Reducing the number of neighbors was already done by Joshi, Laurent, and Bresson [JLB19] for the euclidean TSP. Reducing the time needed for computing features would also enable the possibility to train the NNs on instances with more than $n = 20$ jobs and to use them to solve instances of hundreds of jobs or more.

To further improve the solution quality, one may also combine several NN types during search and profit from their different strengths. For example, the good generalization over the number of jobs of MLP_{ITLB} and the good guidance on instance sizes trained of $\text{NN}_{\text{nearest}}$ and AJNN_{add} may be combined. This may happen based on the number of unscheduled nodes, or within another NN. Through tighter approximations it might be possible to improve the solution quality further.

A natural next step is the application of the presented approaches and features to other FSP variants. It would be interesting how the presented setup performs for them and if there is a feature set that can be used to learn guidance functions for many variants

of the FSP. If successful this might on the long term eliminate the need to design and evaluate guidance functions individually for every variant of a problem.

During our tests no GPU support was used for the evaluation and training of the NNs. This could speed up the evaluation of the NNs possibly a lot, especially when evaluating all nodes of one layer in batches. Testing the impact of batch evaluations on the GPU should therefore be considered in future works.

List of Algorithms

3.1 General BS Algorithm 17

3.2 General LBS Algorithm [HR21] 22

3.3 Data generation for LBS [HR21]. This piece of code is included in Algorithm 3.1 at line 32, if BS is called with data generation. For simplicity, it is assumed that all arguments additionally needed are passed to the BS algorithm. 22

Glossary

$\text{MLP}_{\text{basic}}$ Denotes a MLP that uses observation O_{glob}.. 44, 45, 53, 58, 60, 61, 67, 69, 71, 73, 74

MLP_{ITLB} Denotes a MLP that uses observation $O_{\text{glob,itlb}}$.. 44, 45, 53, 58–61, 67, 69, 71, 73–75

Acronyms

$\mathbf{AJNN_{add}}$ aggregated jobs neural network. 2, 3, 34–36, 41, 42, 44, 45, 53, 57–59, 73–75

$\mathbf{BS_{LS,pour}}$ BS with local search from [Pou+20]. 44–49, 58, 59, 61, 65–67, 69, 71, 74

BS_{pour} BS from [Pou+20]. 45–49, 51, 52, 54–61, 64–71, 74

h_{itlb} ITLB as heuristic function. 43–45, 50, 53–56, 59–61, 65–67, 69, 71

$\mathbf{NN_{nearest}}$ nearest neighbors neural network. 2, 3, 35–38, 41, 42, 44, 45, 53, 57, 59, 73–75

ACO Ant Colony Optimization. 14

ATSP asymmetric Traveling Salesperson Problem. 2, 9, 13, 20, 73

B&B Branch and Bound. 12

BS beam search. 2, 3, 11, 14, 21, 40, 41, 43–45, 54, 59–63, 73, 74

FSP Flow Shop Schedueling Problem. 1, 6, 11–14, 18, 19, 34, 73, 75, 76

GA Genetic Algorithm. 12, 13

GNN graph neural network. 2, 73

ITLB improved Taillard lower bound. 2, 3, 41, 43–49, 51–57, 73, 74

LB lower bound. 12, 18, 39, 40

LBS learning beam search. 2, 3, 11, 21, 22, 33, 34, 44, 45, 73, 75

LCS Longest Common Subsequence Problem. 21

ML Machine Learning. 11

MLP multilayer perceptron. 2, 34, 35, 37, 41, 44, 73

NBS nested BS. 21, 44, 59–63

NEH Nawaz-Enscore-Ham. 12

NN neural network. 3, 18, 19, 21, 33–35, 37, 40, 43–68, 70, 71, 73–76

NWFSP no-wait FSP. 6, 9, 13, 73

NWFSP-RT no-wait FSP with release times. 1, 2, 6, 11, 14, 23, 34, 43, 73

PFSP Permutation Flow Shop Schedueling Problem. 12–14, 26

RL reinforcement learning. 21

RPD relative percentage difference. 44–49, 51–57, 59–64, 68, 70

TS Tabu Search. 12, 13

VRP Vehicle Routing Problem. 20

Bibliography

[ACE06] A. Agarwal, S. Colak, and E. Eryarsoy. "Improvement heuristic for the flow-shop scheduling problem: An adaptive-learning approach". In: *European Journal of Operational Research* 169.3 (2006), pp. 801–815. DOI: 10.1016/j.ejor.2004.06.039.

[All16] A. Allahverdi. "A survey of scheduling problems with no-wait in process". In: *European Journal of Operational Research* 255.3 (2016), pp. 665–686. DOI: 10.1016/j.ejor.2016.05.036.

[And+97] E. J. Anderson et al. "Machine scheduling". In: *Local search in combinatorial optimization* 11 (1997). Publisher: Wiley Chichester, UK, pp. 361–414.

[BDG99] L. Bianco, P. Dell'Olmo, and S. Giordani. "Flow Shop No-Wait Scheduling With Sequence Dependent Setup Times And Release Dates". In: *INFOR: Information Systems and Operational Research* 37.1 (1999). Publisher: Taylor & Francis, pp. 3–19. DOI: 10.1080/03155986.1999.11732365.

[Blu05] C. Blum. "Beam-ACO—hybridizing ant colony optimization with beam search: an application to open shop scheduling". In: *Computers & Operations Research* 32.6 (2005), pp. 1565–1591. DOI: 10.1016/j.cor.2003.11.018.

[CDS70] H. G. Campbell, R. A. Dudek, and M. L. Smith. "A Heuristic Algorithm for the n-Job, m-Machine Sequencing Problem". In: *Management Science* 16.10 (1970), B–630–B–637. DOI: 10.1287/mnsc.16.10.B630.

[Chr72] N. Christofides. "Technical Note—Bounds for the Travelling-Salesman Problem". In: *Operations Research* 20.5 (1972). Publisher: INFORMS, pp. 1044–1056. DOI: 10.1287/opre.20.5.1044.

[DJ64] R. A. Dudek and O. F. T. Jr. "Development of M-Stage Decision Rule for Scheduling n Jobs through M Machines". In: *Operations Research* 12.3 (1964), pp. 471–497. DOI: 10.1287/opre.12.3.471.

[EM95] H. Emmons and K. Mathur. "Lot sizing in a no-wait flow shop". In: *Operations Research Letters* 17.4 (1995), pp. 159–164. DOI: 10.1016/0167-6377(95)00008-8.

[FVVF18] V. Fernandez-Viagas, J. M. S. Valente, and J. M. Framinan. "Iterated-greedy-based algorithms with beam search initialization for the permutation flowshop to minimise total tardiness". In: *Expert Systems with Applications* 94 (2018), pp. 58–69. DOI: 10.1016/j.eswa.2017.10.050.

[GG64] P. C. Gilmore and R. E. Gomory. "Sequencing a One State-Variable Machine: A Solvable Case of the Traveling Salesman Problem". In: *Operations Research* 12.5 (1964), pp. 655–679. DOI: 10.1287/opre.12.5.655.

[GJS76] M. R. Garey, D. S. Johnson, and R. Sethi. "The Complexity of Flowshop and Jobshop Scheduling". In: *Mathematics of Operations Research* 1.2 (1976), pp. 117–129. DOI: 10.1287/moor.1.2.117.

[GML20] J. N. D. Gupta, A. Majumder, and D. Laha. "Flowshop scheduling with artificial neural networks". In: *Journal of the Operational Research Society* 71.10 (2020). Publisher: Taylor & Francis, pp. 1619–1637. DOI: 10.1080/01605682.2019.1621220.

[Gra66] R. L. Graham. "Bounds for certain multiprocessing anomalies". In: *The Bell System Technical Journal* 45.9 (1966). Conference Name: The Bell System Technical Journal, pp. 1563–1581. DOI: 10.1002/j.1538-7305.1966.tb01709.x.

[Gra+79] R. L. Graham et al. "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey". In: *Annals of Discrete Mathematics*. Ed. by P. L. Hammer, E. L. Johnson, and B. H. Korte. Vol. 5. Discrete Optimization II. Elsevier, 1979, pp. 287–326. DOI: 10.1016/S0167-5060(08)70356-X.

[HR21] M. Huber and G. Raidl. "Learning Beam Search: Utilizing Machine Learning to Guide Beam Search for Solving Combinatorial Optimization Problems". In: *Machine Learning, Optimization, and Data Science – 7th International Conference, LOD 2021*. Vol. 11943. LNCS. to appear. Springer, 2021.

[HS96] N. G. Hall and C. Sriskandarajah. "A Survey of Machine Scheduling Problems with Blocking and No-Wait in Process". In: *Operations Research* 44.3 (1996). Publisher: INFORMS, pp. 510–525. DOI: 10.1287/opre.44.3.510.

[JLB19] C. K. Joshi, T. Laurent, and X. Bresson. "An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem". In: *arXiv:1906.01227 [cs, stat]* (2019). DOI: 10.48550/arXiv.1906.01227.

[Joh54] S. M. Johnson. "Optimal two- and three-stage production schedules with setup times included". In: *Naval Research Logistics Quarterly* 1.1 (1954), pp. 61–68. DOI: 10.1002/nav.3800010110.

[Jos+21] C. K. Joshi et al. "Learning TSP Requires Rethinking Generalization". In: *arXiv:2006.07054* (2021). DOI: 10.4230/LIPIcs.CP.2021.33.

[JSW06] F. Jin, S.-j. Song, and C. Wu. “A New Beam Search Algorithm for the Large-Scale Permutation FSP”. In: *2006 International Conference on Machine Learning and Cybernetics*. 2006, pp. 1–6. DOI: 10.1109/ICMLC.2006.258806.

[KHW19] W. Kool, H. van Hoof, and M. Welling. “Attention, Learn to Solve Routing Problems!” In: *arXiv:1803.08475* (2019). DOI: 10.48550/arXiv.1803.08475.

[KK07] P. J. Kalczynski and J. Kamburowski. “On no-wait and no-idle flow shops with makespan criterion”. In: *European Journal of Operational Research* 178.3 (2007), pp. 677–685. DOI: 10.1016/j.ejor.2006.01.036.

[KV18] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. 6th ed. Algorithms and Combinatorics. Berlin Heidelberg: Springer-Verlag, 2018. DOI: 10.1007/978-3-662-56039-6.

[LH05] T. Ladhari and M. Haouari. “A computational study of the permutation flow shop problem based on a tight lower bound”. In: *Computers & Operations Research* 32.7 (2005), pp. 1831–1847. DOI: 10.1016/j.cor.2003.12.001.

[Low76] B. T. Lowerre. “The Harpy speech recognition system”. PhD thesis. Carnegie-Mellon Univ., Pittsburgh, PA., 1976.

[LS00] I. Lee and M. J. Shaw. “A neural-net approach to real time flow-shop sequencing”. In: *Computers & Industrial Engineering* 38.1 (2000), pp. 125–147. DOI: 10.1016/S0360-8352(00)00034-6.

[NEH83] M. Nawaz, E. E. Enscore, and I. Ham. “A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem”. In: *Omega* 11.1 (1983), pp. 91–95. DOI: 10.1016/0305-0483(83)90088-9.

[Ni+21] F. Ni et al. “A Multi-Graph Attributed Reinforcement Learning based Optimization Algorithm for Large-scale Hybrid Flow Shop Scheduling Problem”. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 3441–3451. DOI: 10.1145/3447548.3467135.

[Pas00] C. A. S. Passos. “A Beam Search Based Algorithm to Solve Flowshop Scheduling Problems with Constraints on Shared Resources”. In: *IFAC Proceedings Volumes* 33.17 (2000), pp. 675–679. DOI: 10.1016/S1474-6670(17)39484-3.

[Pie60] J. Piehler. “Ein Beitrag zum Reihenfolgeproblem”. In: *Unternehmensforschung* 4.3 (1960), pp. 138–142. DOI: 10.1007/BF01963410.

[PKL00] Y. Park, S. Kim, and Y.-H. Lee. “Scheduling jobs on parallel machines applying neural network and heuristic rules”. In: *Computers & Industrial Engineering* 38.1 (2000), pp. 189–202. DOI: 10.1016/S0360-8352(00)00038-3.

[Pou+20] P. Pourhejazy et al. “Improved Beam Search for Optimizing No-Wait Flowshops With Release Times”. In: *IEEE Access* 8 (2020), pp. 148100–148124. DOI: 10.1109/ACCESS.2020.3015737.

[PSW91] C. N. Potts, D. B. Shmoys, and D. P. Williamson. “Permutation vs. non-permutation flow shop schedules”. In: *Operations Research Letters* 10.5 (1991), pp. 281–284. DOI: 10.1016/0167-6377(91)90014-G.

[Ram+11] T. R. Ramanan et al. “An artificial neural network based heuristic for flow shop scheduling problems”. In: *Journal of Intelligent Manufacturing* 22.2 (2011), pp. 279–288. DOI: 10.1007/s10845-009-0287-5.

[RR72] S. S. Reddi and C. V. Ramamoorthy. “On the Flow-Shop Sequencing Problem with No Wait in Process†”. In: *Journal of the Operational Research Society* 23.3 (1972), pp. 323–331. DOI: 10.1057/jors.1972.52.

[RYY21] J. Ren, C. Ye, and F. Yang. “Solving flow-shop scheduling problem with a reinforcement learning algorithm that generalizes the value function with neural network”. In: *Alexandria Engineering Journal* 60.3 (2021), pp. 2787–2800. DOI: 10.1016/j.aej.2021.01.030.

[SB99] I Sabuncuoglu and M Bayiz. “Job shop scheduling with beam search”. In: *European Journal of Operational Research* 118.2 (1999), pp. 390–412. DOI: 10.1016/S0377-2217(98)00319-1.

[Sil+17] D. Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676 (2017). Publisher: Nature Publishing Group, pp. 354–359. DOI: 10.1038/nature24270.

[Sul00] S. M. A. Suliman. “A two-phase heuristic approach to the permutation flow-shop scheduling problem”. In: *International Journal of Production Economics* 64.1 (2000), pp. 143–152. DOI: 10.1016/S0925-5273(99)00053-5.

[Tai90] E. Taillard. “Some efficient heuristic methods for the flow shop sequencing problem”. In: *European Journal of Operational Research* 47.1 (1990), pp. 65–74. DOI: 10.1016/0377-2217(90)90090-X.

[Tai93] E. Taillard. “Benchmarks for basic scheduling problems”. In: *European Journal of Operational Research* 64.2 (1993), pp. 278–285. DOI: 10.1016/0377-2217(93)90182-M.

[Vai03] G. L. Vairaktarakis. “Simple Algorithms for Gilmore–Gomory’s Traveling Salesman and Related Problems”. In: *Journal of Scheduling* 6.6 (2003), pp. 499–520. DOI: 10.1023/A:1026200209386.

[VRF15] E. Vallada, R. Ruiz, and J. Framinan. “New hard benchmark for flowshop scheduling problems minimising makespan”. In: *European Journal of Operational Research* 240 (2015), pp. 666–677. DOI: 10.1016/j.ejor.2014.07.033.

[Wis72] D. A. Wismer. “Solution of the Flowshop-Scheduling Problem with No Intermediate Queues”. In: *Operations Research* (1972). Publisher: INFORMS. DOI: 10.1287/opre.20.3.689.

[ZLC15] S. Zhou, X. Li, and H. Chen. “An Estimation of Distribution Algorithm for Minimizing Makespan on a No-Wait Flow-Shop”. In: *IIE Annual Conference. Proceedings* (2015), pp. 1070–1077. DOI: 10.1080/00207543.2016.1140920.