Czech Technical Univeristy in Prague

Faculty of Electrical Engineering

Department of Computer Science

Master's thesis

Accelerating an exact scheduling algorithm using machine learning

David Procházka

Supervisor: doc. Ing. Přemysl Šůcha, Ph.D.

Study Program: Open Informatics

August 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Procházka** Jméno: **David** Osobní číslo: **474606**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Zrychlení exaktního algoritmu pomocí metod strojového učení

Název diplomové práce anglicky:

Accelerating an exact scheduling algorithm using machine learning

Pokyny pro vypracování:

Machine learning (ML) techniques are a modern approach to solving many types of problems nowadays. Nevertheless, for ML techniques, it is not easy to guarantee the correctness of the result. This fact limits their use in exact algorithms for solving combinatorial problems, and thus, it is crucial to devise a good synergy between ML techniques and exact algorithms. The thesis addresses exact approaches for scheduling problem 1|rj|sum Uj and has the following tasks:
1) review existing approaches for scheduling problem 1|rj|sum Uj,
2) review existing scheduling algorithms exploiting ML techniques,
3) select a suitable algorithm for problem 1|rj|sum Uj that can be improved using ML,
4) implement the algorithm, design the ML part and integrate it into the algorithm,
5) compare the achieved results with the literature.

Seznam doporučené literatury:

[1] Philippe Baptiste, Laurent Peridy, Eric Pinson, A branch and bound to minimize the number of late jobs on a single machine with release time constraints, European Journal of Operational Research, Volume 144, Issue 1, 2003, Pages 1-11.
[2] Roman Václavík, Antonín Novák, Přemysl Šůcha, Zdeněk Hanzálek, Accelerating the Branch-and-Price Algorithm Using Machine Learning, European Journal of Operational Research, Volume 271, Issue 3, 2018, Pages 1055-1069.
[3] Michal Bouska, Antonin Novak, Premysl Sucha, István Módos, Zdenek Hanzálek: Data-driven Algorithm for Scheduling with Total Tardiness. Proceedings of the 9th International Conference on Operations Research and Enterprise Systems, ICORES 2020, Valletta, Malta, February 22-24, 2020, Pages 59-68.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Přemysl Šůcha, Ph.D. katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **28.01.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

doc. Ing. Přemysl Šůcha, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instruction no. 1/2009 for observing the ethical principles in the preparation of university theses.

Prague, 15. 8. 2022 . . . .

David Procházka

Acknowledgements

I would like to express my utmost thanks to my thesis consultant Ing. Michal Bouška, without whose guidance I would have a very difficult time writing this work. Furthermore, I want to thank my supervisor, doc. Ing. Přemysl Šůcha, Ph. D., for his valuable inputs. Lastly, I want to express gratitude to my parents, family and my girlfriend for supporting me throughout my studies.

Abstract

This thesis aims to study the application of machine learning in combinatorial optimization. Our goal is to take an existing scheduling algorithm for the $1 \mid r_j \mid \sum U_j$ and improve it using machine learning, which has not yet been attempted. Firstly, we formally define the problem and review the state-of-the-art literature in the scheduling and machine learning fields. Afterwards, we describe the scheduling algorithm we would like to improve in detail. In the subsequent machine learning chapter, we propose a model based on the LSTM architecture, which predicts which jobs in an input instance will be tardy, a piece of information that the scheduling algorithm can use. During the evaluation, we first perform hyperparameter optimization, which produces a model which correctly classifies nearly 95% of jobs in instances of size up to 100 and generalizes well to instances of size 200. It still correctly labels more than 93% of jobs. The results of the integration of this model into the combinatorial algorithm are unimpressive, which led us to develop a heuristic algorithm based on our trained model. This heuristic provides good results; it achieves an average optimality gap of 1.8% on instances of size 100 and 7% on instances of size 200, with an average runtime of 4.2 seconds for the largest instances.

Keywords: Combinatorial optimization, scheduling, number of tardy jobs, machine learning, LSTM

Abstrakt

Tato diplomová práce se zabývá aplikací strojového učení v kombinatorické optimalizaci. Naším cílem je vzít stávající rozvrhovací algoritmus pro $1 \mid r_j \mid \sum U_j$ a vylepšit jej pomocí strojového učení, o což se dosud nikdo nepokusil. Nejprve formálně definujeme problém a provedeme přehled nejnovější literatury v oblasti rozvrhování a strojového učení. Poté podrobně popíšeme rozvrhovací algoritmus, který bychom chtěli vylepšit. V následující kapitole o strojovém učení navrhujeme model založený na architektuře LSTM, který předpovídá, které úlohy ve vstupní instanci budou mít zpoždění, což je informace, kterou může plánovací algoritmus využít. Při vyhodnocování nejprve provedeme optimalizaci hyperparametrů, jejímž výsledkem je model, který správně klasifikuje téměř 95 % úloh v instancích o velikosti do 100 úloh a dobře zobecňuje i na instance o velikosti 200. Stále správně označuje více než 93 % úloh. Výsledky integrace tohoto modelu do kombinatorického algoritmu jsou však nevýrazné, což nás vedlo k vývoji heuristického algoritmu založeného na našem natrénovaném modelu. Tento heuristický algoritmus poskytuje dobré výsledky; dosahuje průměrné mezery optimality 1,8 % u instancí o velikosti 100 a 7 % u instancí o velikosti 200, přičemž průměrná doba běhu u největších instancí je 4,2 sekundy.

Klíčová slova: Kombinatorická optimalizace, rozvrhování, počet pozdních úloh, strojové učení, LSTM

Contents

Chapter 1

Introduction

We begin this chapter by explaining the motivation that led us to write this thesis. Subsequently, we describe the contribution we bring to the existing research fields. Finally, we lay out the structure of this thesis.

1.1 Motivation

Problems in the field of combinatorial optimization are solved daily in nearly any practical setting imaginable. Be it delivery companies looking for the best routes for their vehicle fleet, surgery room scheduling, or finding the most cost-effective way of building a power grid, underneath all these formulations lies a combinatorial optimization problem.

However, solving these problems is often computationally demanding, even for small instances. In practice, optimal algorithms resort to space state search, which extensively uses sophisticated heuristics and pruning to minimize the number of nodes explored.

These parts of the combinatorial approaches are the hardest to design, while at the same time, they must be easy to compute because they are called repeatedly through the state space search. Another problem is that the solution often requires using a general integer linear programming solver, which is costly for enterprises.

This setting has led to the creation of machine learning (ML) solutions, which aim to replace these problematic parts of combinatorial algorithms with a learned model that performs these computations instead. Although not intuitive at first glance, techniques such as random forests, neural networks, and reinforcement learning have found their use even when approaching discrete problems. ML has been used to provide complete end-to-end solutions to combinatorial problems as well as enhance existing solutions.

This thesis examines a scheduling problem that has been well studied in the past but not so much in the last decade. Firstly, we familiarize ourselves with the state-of-the-art approaches from the literature, which rely mainly on branch and bound techniques with extremely efficient state space pruning to overcome the strong NP-hardness and associated computational complexity of the problem. The research on these approaches is extensive and seems to have reached its potential, which is why we would like to improve it by using machine learning.

1.2 Contribution

We aim to enhance an existing approach for solving a combinatorial problem with the newest insights from the field of machine learning to improve its performance. Specifically, we will attempt to train a neural network that will provide information relevant to state space exploration.

Because the number of jobs in the instance of our scheduling problem can be arbitrary, a key focus of our research is to design a machine learning approach that will be able to handle variable length input. Furthermore, we would like to output more than a single value, such as an estimate of the criterion; instead, we aim to predict information about each input job. In our case, we would like to estimate which jobs will miss their deadlines.

1.3 Outline

Firstly, we quickly refresh the basics of the scheduling field and define the problem we are studying. We then review related work in scheduling and machine learning, focusing on literature combining methods from these two areas.

Subsequently, we present the algorithmic approach, which we aim to improve. We first describe its building blocks and then explain how they all come together. Afterwards, we propose our machine learning solution and lay out all the steps we believe are necessary for achieving good results.

We then show the results of several experiments that illustrate the performance of our trained ML approach under changes in various parameters and its integration into the combinatorial algorithm. Furthermore, we use the trained neural network to create a heuristic Finally, we discuss the results in the concluding chapter.

Chapter 2

Problem Statement

In this chapter, we first briefly review basic scheduling concepts. Afterwards, we describe our scheduling problem. For clarity, we also present a small instance of the problem and two of its possible solutions. Lastly, we describe an integer linear programming formulation of the problem.

2.1 Scheduling review

Let us begin with a brief recapitulation of the field of scheduling. For more detailed information on the topic, we refer to the book [1], from which this brief introduction draws. The scheduling field has its roots in industry and services, where it has its main applications to this day. Scheduling problems traditionally arise in manufacturing and transportation, but also in customer services, computer operating systems, and many other locations.

The objective of scheduling is to optimally allocate jobs to resources given properties, conditions, and the criterion of the problem. The variability of these three attributes creates a wide variety of problems that can be studied, which has led to the adoption of Graham's [2] notation as a standard to describe a specific problem. It is denoted in the form $\alpha \mid \beta \mid \gamma$ and allows for the representation of virtually any scheduling problem. The first field describes the number and properties of the machines, the second one represents the jobs' properties and conditions, and the final field is the criterion function. Let us have a deeper look at these three fields.

2.1.1 Machines definition

The first field, in the literature referred to as α, defines our resources, also called machines or processors. Common values are:

- **1**, describes a problem where we execute all jobs on a single machine.
- **Pm**, in this scenario, we may process the jobs in parallel on m machines, all of which work with the same speed.
- **Qm**, represents m machines, each with its own speed v_i, on which jobs may be processed in parallel
- **Fm**, is called a *flow shop* with m machines. These machines are in series; each job must be processed on each machine for a certain amount of time, before moving on to the next.

- **Jm**, called a *job shop* with m machines, is a generalization of the flow shop. Each job has an ordered set of operations, which must be processed on a given machine for a given amount of time before moving on to the next machine for the next operation.

2.1.2 Properties and conditions

The second field, called β in the literature, describes the properties of the problem. Common examples include:

- $\mathbf{p_j}$, called *processing time*, signifies how long the job j has to be processed on a machine before it is finished.
- $\mathbf{p_j} = k$, a special case of the above, where all jobs have the same processing time. Most often, the case $k = 1$ is studied.
- $\mathbf{p_{ij}}$, the processing of the job j differs when assigned to two different machines i_1 and i_2. This is typical in manufacturing, where different types of machines might be able to perform the same job.
- $\mathbf{r_j}$, *release time*, only from this time forward may the processing of this job begin. This is again a property which occurs in practice a lot. For example, we are waiting for the shipment of material.
- $\mathbf{d_j}$, *due date*, the time we would like the processing of the job to be completed, we might finish it after this time, but it will most likely lead to a worse criterion value.
- $\mathbf{\bar{d}_j}$, a *hard deadline*, the job has to be completed before this time. As described later, a schedule where the job would be completed after this time is not considered feasible.
- **pmtn**, *preemption*, the ability to pause the execution of a job on a machine and start another job instead. Implicitly it is not allowed.
- **prec**, given a precedence relation $\prec$, if $i \prec j$, then job i has to be completed before job j, when the precedence relation is of a particular form, a different word may be used e. g. **chains**, **tree**.
- **batch(b)**, machines can process up to b jobs at once in a batch, all the jobs have to be ready at the time of the batch start, the batch is completed after the job with the largest processing time is finished.

2.1.3 Criterion

The last field defines the objective function, i. e. our goal. We will always assume that we want to minimize this function in the following section. Firstly, we need to define s_j as the *start time* of a job and c_j as the *completion time* of job j. Consult Fig. 2.1 for a visual illustration of some of the most common criteria regarding a single job. Below the figure, these criteria are described in the context of the whole instance.

- $-$ no goal is defined. The problem consists simply of producing a feasible schedule
- $\mathbf{C_{max}} = \max(c_1, c_2, ..., c_n)$, we want to finish all the jobs as soon as possible, also called minimizing the makespan
- $\mathbf{L_{max}} = \max(L_1, L_2, ..., L_n)$, maximum lateness, $L_j = c_j - d_j$
- $\sum(\mathbf{w_j})\mathbf{c_j}$, (weighted) sum of completion times

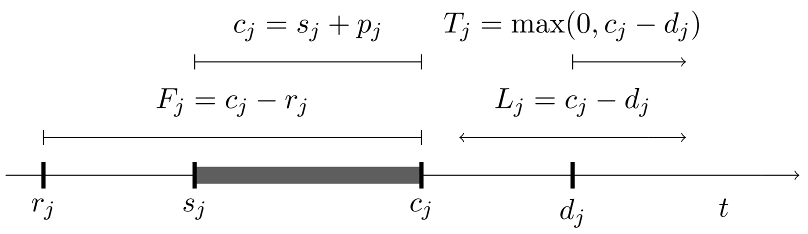


Figure 2.1: Basic scheduling criteria for a single job

- $\sum(\mathbf{w_j})\mathbf{L_j}$, (weighted) sum of lateness
- $\frac{1}{n}\sum(\mathbf{w_j})\mathbf{c_j}$, (weighted) average of completion times
- $\frac{1}{n}\sum(\mathbf{w_j})\mathbf{L_j}$, (weighted) average of lateness

Analogous goals might be defined for $F_j = c_j - r_j$ called flow time, tardiness $T_j = \max\{0, c_j - d_j\}$ or even earliness $E_j = \max\{0, d_j - c_j\}$ and finally U_j defined as

$$U_j = \begin{cases} 1, & \text{if } c_j > d_j; \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

which is not used in its averaged version.

Lastly, it should also be noted that some values in specific fields might imply others. For example, when we want to minimize lateness or the number of jobs that finish after their deadlines, both imply that a job has a due date, and some authors may choose to omit d_j from their problem description.

2.1.4 Solution

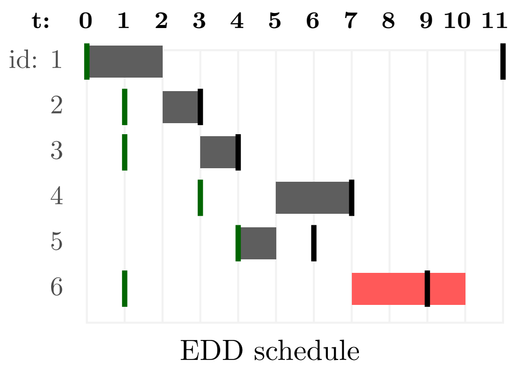
Given a scheduling problem in the format $\alpha \,|\, \beta \,|\, \gamma$ and a set of jobs $\mathcal{J}$, we define a *schedule* as an assignment of jobs to machines, with each job having a defined start time on the given machine. Generally speaking, we need more information for certain problems to describe a schedule fully. For example, when preemption is allowed, we need to specify more start times for a single job, possibly on different machines when its execution was paused.

A *feasible schedule* is a schedule whose assignment of jobs to machines satisfies all the properties and constraints in β. For example, each job is allocated on the machines for the length of its processing time, all the jobs are finished before their hard deadline, precedence constraints and all other conditions are fulfilled. The goal of scheduling is to find an *optimal schedule*, the schedule with the lowest criterion value γ among all feasible schedules.

2.2 Problem statement

In Graham's notation, our problem can be described as $1 \,|\, r_j, d_j \,|\, \sum U_j$. We are scheduling on one machine, and our jobs have both a release and a due date. We aim to minimize the number of tardy jobs, the ones that we are not able to process fully before their deadline. This problem has applications both in manufacturing and service settings. In production, missing a due date might lead to a lower price being paid for the order. In services, not being able to serve a customer will decrease overall customer satisfaction.

Our problem has been proven strongly NP-hard by Lenstra in [3]. However, there exist polynomial algorithms for certain special cases. To better illustrate the nature of this problem, let us present a small instance with two different schedules.



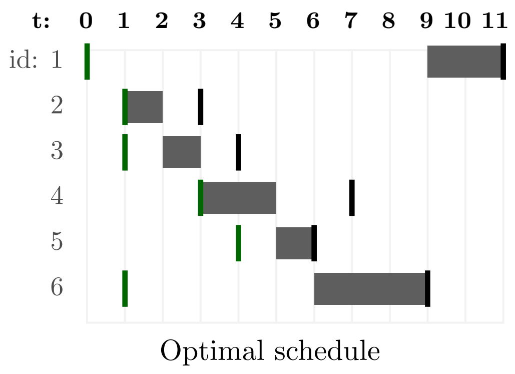


Figure 2.2: Two different schedules for the same set of jobs

In Fig. 2.2, we can observe two different schedules, which both provide a feasible solution to an instance of this problem. The green vertical line in each row is the release time, whereas the black one represents the due date. The grey rectangle represents when the given job is scheduled. On the right is an example of a schedule in which each job is completed before its deadline; on the left is a schedule following the Earliest Due Date (EDD) dispatch rule, which generates a suboptimal solution, job id 6 misses its deadline.

Note that when a job is late, the objective value does not worsen the later the job is scheduled. We are only interested in the number of tardy jobs, not "how late" these jobs are, which means that all tardy jobs might be scheduled arbitrarily late. Moreover, when we believe a job will be late, we might remove it from the problem and solve it for the remaining jobs, adding the late job at the very end.

2.3 ILP formulation

The problem might also be formulated using *integer linear programming* (ILP) in several equivalent formulations. Let us consider the set of all jobs $\mathcal{J}$, and for each job $j \in \mathcal{J}$, let p_j, r_j, d_j represent the processing time, release time and due date, respectively. The set $\mathcal{T}$ defines the set of all time instants up to the last possible time a job might be theoretically started.

$$\min \sum_{j\in\mathcal{J}} U_j$$

$$\begin{aligned}
\text{s. t. } & x_{jt} && = 0, && \forall x, \forall j \in \mathcal{J}, t < r_j && (1)\\
& U_j + \sum_{t \le d_j - p_j + 1} x_{tj} && = 1, && \forall j \in \mathcal{J} && (2)\\
& \sum_{t'=\max(0,t-p_j+1)}^{\min(t+1,d_j-p_j)} x_{t'j} && \le 1, && \forall j \in \mathcal{J}, \forall t \in \mathcal{T} && (3)\\
& U_j && \in \{0,1\}, && \forall j \in \mathcal{J} && (4)\\
& x_{tj} && \in \{0,1\}, && \forall t, \forall j \in \mathcal{J} && (5)
\end{aligned}$$

Figure 2.3: ILP formulation of $1 \mid r_j, d_j \mid \sum U_j$

The model in Fig. 2.3 aims to minimize the sum of variables U_j, which represent whether a job j is tardy. Decision variables x_{jt} are introduced for each combination of a job and time, representing whether the job j starts at time t. Both of these variables are binary as defined in constraints (4) and (5). Constraint (1) forbids the start of a job before its release time. Constraint (2) defines that a job is either started soon enough that it is on time, or it is late. Finally, constraint (3) specifies that there might be only one running job at each time.

Chapter 3

Related Work

In this chapter, we are going to review the literature in the areas relevant to this thesis. We begin by examining the field of scheduling, where a significant amount of research has been done concerning our problem and its simplified variants. We analyze the techniques that are often used for optimally solving scheduling problems, as well as present heuristic solutions.

Afterwards, we will discuss the applications of machine learning in combinatorial optimization, of which scheduling is a part. We show that much recent research has been dedicated to leveraging machine learning techniques to replace or enhance existing combinatorial algorithms.

The reviewed approaches from these two areas will provide us with crucial insights when designing our machine learning solution for the scheduling problem. Scheduling literature will help us build a baseline combinatorial algorithm, whereas the machine learning review will show us the possible ways of incorporating machine learning into purely combinatorial approaches.

3.1 Scheduling

For a comprehensive survey of all current literature on the topic of scheduling with the goal of minimization of tardy jobs, we refer to the survey [4], which contains over one hundred references.

Let us first focus on the most straightforward problems, where some properties from our problem are dropped. When all release times are zero, the problem $1 \mid \mid \sum U_j$[1] can be solved by describing the recurrence relations when adding a new job to a set of jobs that have already been scheduled optimally. An efficient implementation of these rules, which constructively builds a schedule and runs in $\mathcal{O}(n \log(n))$, was first presented by Moore in [5]. He attributed this algorithm to T. J. Hodgson, and so it is nowadays most often referred to as Moore-Hodgson's algorithm.

Lawler in [6] slightly modified this algorithm to solve the weighted version of the problem $1 \mid \mid \sum w_j U_j$ under the condition that job weights and processing times are oppositely ordered, meaning that they can be sorted in a way such that: $p_1 \leq p_2 \leq ... \leq p_n$ and $w_1 \geq w_2 \geq ... \geq w_n$. The running time of the algorithm remains the same, $\mathcal{O}(n \log(n))$. When dropping the constraint of opposite ordering of weights and processing times, the problem $1 \mid \mid \sum w_j U_j$ is proven to be NP-hard by Karp [7], even when all jobs have identical due dates.

As previously mentioned, our problem in the form $1 \mid r_j \mid \sum U_j$ has been proven strongly NP-hard by Lenstra[3]. Kise et al. [8] propose an $\mathcal{O}(n^2)$ algorithm for the case where the

[1]Let us remind the reader that the criterion $\sum U_j$ implies that each job has a set due date d_j, which is omitted from the properties in Graham's notation

jobs release and due times are similarly ordered, that is $r_i < r_j \implies d_i \leq d_j$. This result is further improved by Lawler in [9], where he studies deeper the recurrent relations introduced by Moore-Hodgson's algorithm to introduce the term *tower of sets* of feasible solutions based on the idea that smaller feasible solutions are subsets of the larger ones. Moreover, he gives a nonintuitive recursive definition of so-called *effective processing time.* Using these insights, he creates an algorithm which utilizes a double-ended priority queue (DEPQ) and runs in $\mathcal{O}(n\log(n))$.

Another related problem, $1|\bar{d}_i|\sum w_iU_i$, is studied in [10]. The authors seek to schedule all the jobs, so they are finished before their deadline, while minimizing the number of jobs that are finished after their due date $d_i \leq \bar{d}_i$. Their problem is known to be NP-hard, but it is open whether or not in the strong sense. Nevertheless, their ILP-based approach is able to solve instances of up to 30 000 jobs.

A recent result in the thesis [11] builds on top of this article. The author studies the special case of *correlated instances* where there is a relationship between the processing time and the weight of a job. This problem, $1 \mid p_i = w_i + K \mid \sum w_iU_i$, is challenging to solve using the approach in [10]. A new ILP model and tighter lower bounds are proposed for this class of instances. These improvements lead to optimal solutions to problems consisting of up to 5 000 strongly-correlated jobs.

A majority of research on our problem and its weighted form relies on a branch and bound (B&B) algorithm with lower bounds to prune away unnecessary parts of the state space search tree. The first such example is by Peridy et al. [12] who attempt to improve an existing relaxation of the problem, which is commonly used as a lower bound. The time-indexed ILP program of the relaxation leads to a definition of a *time-indexed graph*, in which a path corresponds to a schedule. The authors then propose a method to search this graph using recurrence relations which can be solved using dynamic programming with a buffer called short-term memory.

The improved lower bound is then used and evaluated in a classical depth-first branch and bound algorithm, first they try to fix a job on time and then it is switched to tardy when backtracking occurs. A number of propositions such as decomposition of the problem into subproblems that do not interact with each other is used to further improve performance. They present results on a standard benchmark datasets where their approach solves 83% of 100 job instances in one hour, showing significant improvement against the baseline approach without the strengthened lower bound, which solves only 80% of 50 job instances in an hour.

Dauzère-Pérès and Sevaux propose a different type of branching [13], again building on top of previous works. They describe a theorem which lists the necessary conditions for a sequence of jobs to be a possible optimal solution. Subsequently, they define a "master sequence", which contains every sequence with these properties as its subsequence. Branching is then performed on this master sequence, reducing it to obtain a solution. Their results show that they solve 95% of 140 jobs instances in an hour of CPU time.

Baptiste et al. in [14] propose another B&B algorithm which branches by setting the jobs early and tardy. They utilize two lower bounds, one based on the relaxed version of the problem studied in [8], the second lower bound they propose themselves. They firstly define an integer linear programming (ILP) model which solves the $1 \mid r_i, pmtn \mid \sum U_i$ problem. It turns out that the solution of the dual of its Lagrangian relaxation can be evaluated efficiently by finding a solution of the maximum flow problem, which allows the use of a numerical subgradient optimization method.

The branch and bound is further sped up by evaluating propositions such as decomposition and tightening of job execution windows and using the lower bound to cut away parts of the search tree without missing optimal solutions. Their approach solves 86.7% and 94.4% of 140 job instances from their two benchmark datasets within one hour of

CPU time.

Another example of a B&B approach which solves the weighted version of our problem, that is $1|r_j|\sum w_jU_j$, is described by M'Hallah and Bulfin [15]. The authors use a surrogate relaxation (SR) of the original LP of the problem. The dual form of this relaxation has a structure close to a multiple-choice knapsack problem (MCKP), so it is adjusted to correspond to MCKP. As shown in [16], Lemma 5.4, the best solution of the MCKP, which is also feasible for the scheduling problem, is the best solution of the scheduling problem.

However, obtaining the surrogate multipliers which yield the "least relaxed" SR is not straightforward. The authors propose to start with a feasible schedule and utilize a subgradient optimization method to find them, which is not a fast procedure. The declared experimentally measured runtime is approximately $0.000016n^3$, or around 1.5 minutes for the 175 jobs instances.

A branching scheme on the MCKP is then proposed. The linear relaxation of the MCKP is used as a lower bound. A mean time of 20-30 seconds of CPU time is reported for their instances of 175 jobs generated by the procedure introduced by Dauzère-Pérès and Sevaux in [17]. Afterwards, the approach is evaluated on the problems that the approach of Baptiste et al. [14] was unable to solve in an hour. Interestingly, this approach solves all of the 44 previously unsolved 200 job instances.

Moreover, the authors propose a greedy heuristic to obtain an initial solution. This heuristic runs very fast in $\mathcal{O}(n^2)$. Nevertheless, it relies on coefficients from the transformed MCKP model, which need to be calculated by a subgradient method whose runtime dwarfs the time needed to evaluate the heuristic. This heuristic seems to perform very well on the unweighted variant of the problem; it is reported to nearly always return the optimal solution for 100, 150 and 175 jobs. On the other hand, it struggles in the instances of 50 jobs, which might indicate that the job generation produces simpler instances for larger numbers of jobs. On the weighted problems, it returns a good initial solution.

Sadykov in [18] proposes a slightly different approach called branch and check. He uses an optimization approach known as *Bender's decomposition*, where the variables that do not appear in the objective function are not considered. When the optimal solution of such a relaxed problem is found, it is then checked whether a feasible solution exists for the previously ignored variables. If yes, an optimal solution is found. If not, a "cut" is added to the master problem, which removes the infeasible found solution, and the optimization process is started anew. This approach has managed to solve all of the instances of 140 jobs from the benchmark datasets under one hour, with the maximum solve time on an instance of 1110 seconds and an average of 33 seconds.

One of the latest results by Briand and Ourari [19] from 2009 defines the term *top*, which is a job that does not fully contain an execution window of another job in its execution window. To each top corresponds a *pyramid*, a set of jobs whose execution windows fully contain the execution window of the top. The notion of *dominance* between job sequences is also introduced. A sequence σ_1 dominates sequence σ_2 when the feasibility of σ_2 implies the feasibility of σ_1. The authors show that a dominant sequence to the optimal solution of the problem can be constructed from all pyramids in a given instance. Based on these insights, the authors propose a lower and an upper bound of the problem as an ILP program. They then evaluate these bounds, showing that running them only once often provides an optimal solution of the problem. However, these bounds are computationally very expensive. Their runtime is close to the runtime of the algorithm in [14], which produces optimal solutions.

3.2 Machine learning

As Bengio et al. note in the review [20], two situations often happen when dealing with combinatorial optimization (CO) problems. Firstly, an algorithm may rely on *expert knowledge*, a rule based on the researcher's intuition that consistently gives good numerical results, but its justification is hard to prove and or it is computationally demanding. Secondly, procedures such as state space search require decisions to be made throughout the run of the algorithm. Similarly, making these decisions is a difficult task with significant implications for the overall runtime of the algorithm. Both of these cases provide an opportunity for machine learning (ML) to improve or replace parts of existing algorithms.

After identifying one of these two main motivating factors for the use of ML, we need to decide on how to incorporate it into the existing combinatorial approach. Bengio et al. propose to divide the existing literature into three categories.

Firstly, ML might completely replace the CO algorithm, directly solving each input instance. Alternatively, the use of ML might be limited to running once in the beginning, providing insight regarding the structure of the instance, which is then used in a standard CO algorithm. Finally, ML can be consulted repeatedly throughout the run of the CO method to provide guidance. Let us now present the details of these three approaches.

3.2.1 End-to-end solution

In the most straightforward setting, one can develop an ML algorithm which takes the whole instance as an input and outputs a solution directly. However, the first neural network approaches, such as the multilayer perceptron, consider fixed-size input and output, limiting their usefulness on variable-length data, while the application of other ML methods on combinatorial was not much explored.

This began to change in the 1980s with the advent of *recurrent neural networks* (RNNs) which overcame the hurdle of variable-sized input. In 1985, Hopfield presented his neural network [21], which gives meaningful results for instances of Euclidean TSP with ten cities. Another early example of an RNN, sometimes even credited as the first description of a RNN is presented by Rumelhart [22]. He describes the foundations of a learning process which could, in principle, learn an arbitrary mapping from input to output, discusses its limitations and proves its usefulness by running simulations on several problems, such as XOR, parity and others.

After these initial attempts, research applying ML in CO started growing, and this trend continued throughout the 1990s. A detailed review of the literature from this period, which also discusses the most significant hurdles of the time, is provided by Smith [23]. However, the applications were still very limited and behind the state of the art.

It was only with advances in computing power that more robust models and more complex neural network architectures emerged, which spurred more research into end-to-end solution of combinatorial problems. The most prominent example of these new architectures is the *long short-term memory* (LSTM) architecture, which was first introduced in 1997 by Hochreiter and Schmidhuber [24]. They enhance the recurrent neural network with a state vector. Weights are then learned to determine which new information from the input to store, and what to forget. The state is passed throughout the recurrent evaluation of the model and updated during the processing of each input. The cell's output is generated based on a learned combination of both the input and the current state.

The vast amount of research concerning its successful application is a testament to the effectiveness of this approach. One such example is the [25], which takes the architecture even one step further. Whereas vanilla LSTM generates one output per input, the proposed sequence-to-sequence, also called the encoder-decoder approach, aims to map the input sequence to another output sequence, both of arbitrary lengths. One LSTM network, the encoder, is used to map the input to a fixed size vector and a second LSTM, the decoder, maps this vector to an output sequence. The authors achieve close to state-of-the-art results at the time on an English to French translation task from a benchmark dataset.

The introduction of models capable of mapping between sequences of arbitrary length while first completely reading the input sequence, only then beginning to produce the output, is very important for CO. Many problems, such as the TSP, can be formulated this way. The application of these architectures quickly gained traction.

One such example also uses the idea introduced by Bahdanau et al. [26], who connects the encoder to the decoder via an *attention* layer, which can be thought of as mimicking the idea of human attention. The goal of this module is to learn weights which will decide which parts of the input in the given layer should we attend to more and which parts can be ingored. The authors report state-of-the-art results on a standard English to French translation dataset.

The Pointer Network architecture [27] of Vinyals et al. addresses the limitation of the sequence-to-sequence model. It can only return an output with a length equal to the input. The authors use this architecture to obtain approximate results for three CO problems, most notably Euclidean TSP, showing that the architecture learns to solve problems of up to 50 nodes, giving reasonably good approximations.

An interesting idea regarding the training of the ML approach is shown by Bello et al. [28]. They aim to solve the TSP by finding a policy on where to go next from each node based on a pointer network architecture. The goal is to predict such policies that lead to short tours. The authors argue that learning from labelled data is inefficient for NP-hard problems and opt to use reinforcement learning (RL) instead to optimise the parameters of the neural network. Experimental results show close to optimal results for 2D Euclidean TSP instances of up to 100 nodes.

A similar approach is used by Kool and Welling in [29], except that instead of an RNN, they use a *graph neural network* using attention mechanisms to encode the input instances. An advantage of such a model is that it is invariant to the permutations of the input nodes. Once more, the neural network is trained using reinforcement learning. Experiments show that this approach gives solutions with a slightly better optimality gap than in articles that use RNNs.

Finally, a few years after the newest state-of-the-art transformer architecture had been introduced [30] based on the previously mentioned concept of attention, Bresson and Laurent applied it to the TSP [31]. Their idea is to turn the TSP into a translation problem from the language of instances into the language of tours. They describe their attention-based encoding and decoding and show that such a model can predict a reasonable TSP solution. They report an optimality gap of 0.98% for the TSP100 dataset.

To sum up, in some instances, ML-based methods are comparable to or even outperform state-of-the-art methods. In other areas, they show promising results and potential for improvement. Let us also stress that given the probabilistic nature of ML, an algorithm relying solely on ML models will never be able to guarantee the optimality of the solution. Finally, the literature on end-to-end combinatorial problem solutions using ML models other than neural networks is minimal. Because of their weaker expressive power and quick evaluation, they are better suited for guiding existing combinatorial approaches, as described in the following sections.

3.2.2 Algorithm configuration

The second approach consists of using ML first to gain insight into the structure of a specific instance. Based on this step which mimics the expert knowledge often used in CO algorithms, a slightly modified version of an algorithm is run, or a completely different approach to solve a particular instance might be used. This field is called *algorithm* *configuration* or *selection*, and it has received a lot of attention lately. A thorough survey of algorithm selection in combinatorial optimization, which answers basic questions like from which set of algorithms to select, what and how to select and what information to use for the selection, is provided by Kotthoff [32].

A typical example of algorithm selection can be found in the well-known mixed-integer linear programming (MILP), where some variables are integers, and others are left unconstrained. As it turns out, in some instances, when the problem is reformulated using the Dantzig-Wolfe decomposition [33], the resulting model may be stronger, allowing current solvers to solve it more easily. However, it is not easy to tell in which cases the decomposition will lead to a model which is easier to solve.

Kruber et al. in [34] aim to address this problem by using machine learning to decide this question. Their article is based on the hypothesis that decomposition will work on a MILP model well if it has worked well on a similar model, which leads them to several ML classifiers: Nearest Neighbors, Support Vector Machines (SVM) with RBF kernel and Random Forests. Their final method relies on two existing MILP solvers, where one tries to use the Dantzig-Wolfe decomposition, and the other is a general MILP solver. Their experimental results indeed show an improvement in a particular class of instances while slowing the rest by an insignificant amount.

A similar situation arises in *mixed-integer quadratic programming* (MIQP), which solves the problems in the form:

$$\min\left\{\frac{1}{2}x^TQx + c^Tx : Ax = b, \quad l \leq x \leq u, x_j \in \{0,1\} \forall j \in I\right\} \tag{3.1}$$

Where Q is symmetric. As it turns out, this problem might be linearized in certain cases by rewriting the quadratic term, allowing for the use of standard *mixed-integer linear programming* (MILP) solvers. This approach seems to improve the performance overall, but for some instances, the novel methods designed for general MIQP perform better.

Researchers in the article [35] aim to learn a classifier to predict when to linearize a given instance. The machine learning methods they use are SVM, Random Forests, Extremely Randomized Trees and Gradient Tree Boosting. Authors show that it is possible with a reasonable degree of accuracy to predict when to linearize a given instance and propose further research of this problem and possible integration into MIQP solvers.

A problem very close to algorithm configuration is *algorithm selection.* We have a portfolio of algorithms which all solve a given problem and our goal is to choose the one that will perform the best on a given instance. Authors in [36] consider the *Bid Evaluation Problem* in Combinatorial Auctions, which can be solved using Constraint Programming (CP) or ILP. They decide to use Decision Trees and describe in detail their training process, which leads to a model can select the best performing algorithm in 90% of the instances.

3.2.3 Continous guide

Lastly, ML might act as a guide for the combinatorial algorithm. We can query it repeatedly during the algorithm's run to help us continue the state space search in the most promising way. This way, branch and bound (B&B) approaches often lead to finding good candidate optimal solutions, which then allow for pruning large parts of the search tree. This is of particular interest in mixed-integer linear programming (MILP), where we solve the following optimization problem:

$$\min\left\{c^Tx : Ax \leq b, x \geq 0, \forall i \in I : x_i \in \mathbb{Z}\right\} \tag{3.2}$$

Current state-of-the-art methods rely mainly on B&B. In each node, a linear relaxation, a simplified problem where the integrality constraints are dropped, is solved. Based on the results, a variable that should be integer is chosen, and the problem splits into two subproblems. Deciding on which variable to branch might significantly impact the runtime of the algorithm.

Furthermore, since B&B is a state space search algorithm, it keeps several open nodes stored in some data structure. Choosing which one to start processing next is equally important to the overall runtime. As Lodi and Zarpellon note in their survey [37], current methods often deal with these problems using heuristics, which are supported mainly by computational results instead of rigorous mathematical approaches.

This is precisely the environment suited for ML. Instead of relying on *expert knowledge* backed by computational results, we can learn a classifier, which can guide us in each node in the state space, providing answers to the two key questions. Alvarez et al. in [38] aim to imitate an existing good but computationally demanding branching rule known as *strong branching* with a fast approximation based on ML. They describe their feature encoding, which turns examples from analysed runs of the algorithm into input-output vector pairs on which they learn Extremely Randomised Trees. Experiments on a standard dataset show that the approach slightly underperforms the state-of-the-art branching solutions but is on par with strong branching.

The problem of branching in MILP solvers is of great importance, and it has been studied extensively, Gasse et al. [39] propose a graph neural network to imitate strong branching. With this architecture, they outperform previous ML approaches to branching as well as the default branching policy of one of the state-of-the-art MILP solvers on several typical CO problems.

Gupta et al. [40] discuss the previous approach's drawbacks and how to overcome these hurdles. One minor issue they note is that ML-based branching rules tend to only perform well on the class of the problems they were trained on. However, they argue that this is reasonable since we are usually only interested in one class of problems at a time in practice.

The bigger problem they describe is that using GNNs for inference is computationally expensive, requiring high-end GPUs to achieve reasonable speed. Their goal is to create an ML approach that will retain the expressiveness of GNNs, avoid the need for GPUs and still provide good branching results. They achieve this by creating a hybrid model, which uses a GNN only in the root node and uses a multilayer perceptron throughout the branching. They show that using CPU only, their approach still attains a 26% reduction in solving time on benchmark datasets compared to one of the state-of-the-art MILP solvers.

One of the most recent and advanced ML approaches to speed MILP is a collaboration between scientists from Deep Mind and Google Research [41], who propose two different speed-ups. Firstly, their *neural diving* uses neural networks to generate high-quality partial assignments for the integer variables, quickly leading the solver to a possible solution. Secondly, the proposed *neural branching* uses a similar idea to the previous approaches. It aims to mimic the strong branching.

These methods use a bipartite graph representation of MILP as an input into a type of GNN called a *graph convolutional network*. Their approach is evaluated both on real-world datasets and on datasets from the Mixed Integer Programming Library (MIPLIB), a collaborative project that provides several benchmark MILP datasets. Their selection methodology is described in [42]. To experimentally evaluate their approach, they implemented their improvements into the SCIP solver.

The solver augmented with these techniques provides a much better primal-dual gap on a dataset of hard instances with large time limits. It also reaches a 10% gap five times faster on another dataset, whereas no improvement is reported on the final one. On some unsolved problems from the benchmark dataset, it also returned the best variable assignments found so far. One slight drawback of this approach is what has already been mentioned in this review, and that is that the GNNs need access to GPUs in order for their ability to scale to larger instances in a reasonable time.

Another area where branching is also found is the Boolean satisfiability (SAT) problem. Given a propositional formula in conjunctive normal form, the question is whether there exists an interpretation of the variables that satisfies it. This problem is NP-complete; existing algorithms often rely on some form of Branch & Bound.

Selsam and Bjørner in [43] aim to improve existing SAT solvers by predicting whether the clauses of the formula form a so-called *unsatisfiable core*, a subset of formulas whose conjunction is still unsatisfiable. They employ a simple neural network with three fully-connected layers and custom update rules to predict whether each formula will be in an unsatisfiable core, which they believe can speed up the branching algorithm. In experiments, this approach outperforms the state of the art by solving up to 11% more instances from a benchmark dataset.

Another area where machine learning has been applied successfully is the *branch and price* algorithm [44], which is often used in CO to solve ILP problems with many variables. The main idea behind this method is to reformulate the original problem definition into a master problem and a pricing problem, which is then solved repeatedly throughout the run of the algorithm.

Válacvík et al. [45] present case studies which show that around 90% of the computation time is spent on the pricing problem. Moreover, they note that only a tiny part of the pricing problem changes during the computation, leading them to propose an online regression-based approach similar to SVM, which learns throughout the algorithm's run and quickly calculates a bound for the pricing problem. They then evaluate their approach on 14 different combinatorial problems and show that their improvements lead to an average 40% reduction in total CPU time.

Finally, the literature incorporating machine learning approaches into scheduling started appearing in the 1990s, a review of work from the era can be found in [46], but state-of-the-art approaches are rare. From our research, it appears that no one has so far attempted to tackle the problem we do. However, a similar problem $1 \mid \mid \sum T_j$, which aims to minimize total tardiness, has been studied. The authors use the LSTM architecture RNN to predict the criterion value of certain subproblems [47]. This value is used as a guide in a decomposition-based algorithm and provides superior results to existing state-of-the-art approaches.

Chapter 4

Algorithmic solution

In this chapter, we present the combinatorial part of our approach. We first explain what led us to choose this specific algorithm and then give a brief high-level description of how it works. Subsequently, we provide a detailed description of all the parts of the algorithm, explaining the necessary background theory along the way and providing pseudocode. Finally, the pseudocode of the whole high-level algorithm is presented in Algorithm 7.

We have decided to base our solution on the algorithm presented in [14] mainly because it is based on B&B, and the branching is performed by setting the jobs to be early or tardy, which presents an opportunity for a machine learning approach that would be able to predict this information. Given such predictions, we hope to navigate the algorithm to find excellent solutions at the beginning of its run, leading to the pruning of large parts of the search tree. Furthermore, this algorithm relies on several propositions and two lower bounds to speed up the branching, which means it can be redeveloped incrementally. Lastly, this algorithm is based on a large amount of knowledge built over several years in the previous articles of the authors and has become a benchmark B&B approach. It will be interesting to see whether it can be improved.

As previously mentioned, the algorithm is an example of a branch and bound (B&B) method. It utilizes a depth-first search (DFS) to find a feasible solution and then uses a so-called lower bound, an algorithm that solves a simplified version of the problem and its result is then compared against the original problem. If the best result that can be achieved on the simplified problem corresponding to a given node is not better than our current best solution, it is pointless to explore that node further.

The state space of our problem is the set of all assignments of the jobs into three sets: free, on-time and late, denoted by F, T and L, respectively. On-time jobs are the ones we consider will finish on time. As mentioned during the analysis of Fig. 2.2, when we consider a job late, it might as well be arbitrarily late. Therefore we might ignore the late jobs in a given node and only count them towards the criterion value. Free jobs are the ones we have not yet decided on.

This way, if we can determine whether a feasible schedule exists for a given set of on-time jobs, we can create an optimal algorithm by inductively building the on-time set. If we find no feasible schedule for a given set of on-time jobs, we backtrack to the last job added to the on-time set and place it into the late set instead.

During this branching, we can use several properties of the problem in each node to further speed up the algorithm. We might decompose the jobs into subsets which do not interact with each other and solve these subproblems separately. Furthermore, by analyzing the execution windows of different jobs, we might conclude that given the jobs we consider on time, there is simply no way of scheduling specific free jobs, and they must be late. Lastly, it might be proved that certain jobs cannot start right at the beginning of their execution window or finish at its end due to interaction with other jobs, which leads to adjustments of release and due dates.

Let us now look at the algorithm's properties and building blocks in detail. Firstly, we shall describe the simple procedure that partitions the current problem into several subproblems, which can be solved separately if such a partition exists. Afterwards, we examine two algorithms that allow us to determine whether it is possible to schedule a given set of jobs feasibly. We will then build our knowledge about the simplified version of our problem, which will lead to a formulation of a lower bound for our problem. Finally, we introduce two algorithms which take advantage of the problem properties and decrease the size of the space state. We then conclude this chapter with a high-level description of the combinatorial approach.

4.1 Decomposition

Arguably the most straightforward proposition concerns the decomposition of an instance. We might find two or even more sets of jobs that do not interact with each other. If such a thing occurs, it is advantageous to solve each subproblem separately and concatenate their results. A theorem that precisely defines when this occurs is presented in the article:

Proposition 1 *[14] Two sets of jobs I and J do not interact with each other if there exists a time t such that: $\forall i \in I : d_i \leq t$ and $\forall j \in J : r_j \geq t$.*

Indeed, it is trivial to observe that if all the jobs from the first have to finish before the jobs from the second set are even released, the two sets might be scheduled separately. The authors propose an algorithm to find such sets:

Algorithm 1 Decomposition of the problem as described in [14]

```
procedure DECOMPOSE($jobs$)
    $subproblems \leftarrow \{\}$
    $K \leftarrow \{(i, r_i) \mid i \in jobs \setminus L\} \cup \{(i, d_i) \mid i \in jobs \setminus L\}$
    sort K in ascending order, use the second element of the tuple as key,
        break ties by giving priority to times corresponding to due dates
    $d \leftarrow 0$
    $subproblem \leftarrow \varnothing$
    for $(job, time)$ in $K$ do
        if $job.d = time$ then
            $d \leftarrow d - 1$
            if $d = 0$ then
                $subproblems \leftarrow subproblems \cup \{subproblem\}$
                $subproblem \leftarrow \varnothing$
            end if
        else
            $d \leftarrow d + 1$
            $subproblem \leftarrow subproblem \cup \{job\}$
        end if
    end for
    return $subproblems$
end procedure
```

This algorithm starts with the release and due dates of the jobs sorted in ascending order and keeps track of how many execution windows are currently open. A subproblem has been identified if a deadline is found and the number of open execution windows is zero.

4.2 Schrage's schedule

An important building block of the whole approach is deciding whether a set of jobs can be feasibly scheduled. A simple and fast way to schedule a set of jobs with release times and due dates is to use the earliest due date (EDD) rule, also known as Schrage's schedule or Jackon's rule.

This algorithm often produces a feasible schedule; if not, performance guarantees exist, which leads to a branching algorithm that can correct the schedule produced by EDD to become feasible or prove that no feasible schedule exists. This branching is described in the next section.

The advantage of this approach is its speed. Sorting takes $\mathcal{O}(n\log(n))$ time. When using a priority queue, operations add and removeMin both take $\mathcal{O}(\log(n))$ time, and both will be called n times. The total time complexity of finding a Schrage schedule is $\mathcal{O}(n\log(n))$.

Algorithm 2 Schrage's EDD schedule

```
procedure SCHRAGE($jobs = \{J_1, J_2, ..., J_n\}$, sorted with ascending release times)
    $t \leftarrow 0$
    $idx \leftarrow 1$
    $schedule \leftarrow \{\}$
    $ready \leftarrow PriorityQueue()$
    for $i$ in $1, 2, ..., n$ do
        if $ready.empty()$ then
            $t \leftarrow \min(r_{J_i} | i \geq idx)$
        end if
        $ready.addAll(\{J_i \text{ with priority } d_{J_i} | i \geq idx, r_{J_i} \leq t\})$
        $idx \leftarrow \underset{i>idx}{argmin}\{r_{J_i} | r_{J_i} > t\}$
        $j \leftarrow ready.removeMin()$
        $schedule \leftarrow schedule \cup \{(t, j)\}$
        $t \leftarrow t + p_j$
    end for
    return $schedule$
end procedure
```

The algorithm first sorts all jobs by ascending release times and then moves forward in time. All the jobs released up to the given time are added into the priority queue *ready*, from which the jobs with the earliest due date are removed and processed. Time moves to the end of processing, and the loop continues until all jobs are scheduled.

4.3 Branching of Carlier

When a Schrage schedule of a set of jobs is created, it might be feasible, meaning that each job is finished before its due date. However, this is not guaranteed in the general case. In his article from 1982 [48], Carlier analyses the EDD schedule's performance and proves its useful properties. Nonetheless, this article is written from the point of view of multiple machines. His findings can be found in a neater way in [49], where the application to our problem is directly visible. Firstly, a lower bound on the maximum lateness of a set of jobs to be scheduled is presented:

Theorem 1 *For any set of jobs J, there exists the following bound for maximum lateness*

$L_{max} = \max_{j \in J} c_j - d_j$ of jobs from J:

$$L_{max} \geq \min_{i \in J} r_i + \sum_{i \in J} p_i - \max_{i \in J} d_i$$

This result on its own is not that interesting. For a strong performance guarantee, we need to define additional notation. Suppose we have a set of jobs $j_1, j_2, ..., j_n; j_i = (p_i, r_i, d_i)$ to be scheduled. Let us consider a Schrage schedule in the format T, σ, where $\sigma(i)$ defines which job will be started as i-th, and $T(i)$ represents when the i-th job will be started. We define $\sigma[i, ..., j]$ as a series of consecutive jobs $\{\sigma(i), ..., \sigma(j)\}$. A *chain* is a maximal sequence of consecutive jobs executed without idle time.

Theorem 2 *Let $\sigma(i)$ be a job with maximum lateness in T. Let $\sigma(h)$ be the head of the chain of which job $\sigma(i)$ is part. One of the following is true:*

1. *For $J = \sigma[h, ..., i]$:*
$$L_{max} = \min_{i \in J} r_i + \sum_{i \in J} p_i - \max_{i \in J} d_i$$

2. *There exists a job $c = \sigma(j)$ preceding $\sigma(i)$ in its chain such that, for $J = \sigma[j+1, ..., i]$*
$$L_{max} < p_c + \min_{i \in J} r_i + \sum_{i \in J} p_i - \max_{i \in J} d_i$$
Moreover, in an optimal schedule, job c has to be either scheduled before all the jobs in J or after them.

The first possibility tells us that the EDD schedule is optimal, the L_{max} is the lowest possible based on the lower bound, and we cannot obtain a better schedule. If the second case is true, we know that there exists a job c, which is blocking another set of jobs J and that in an optimal schedule, c has to be scheduled before or after all of the jobs from J.

Identifying which of these two cases occurs is straightforward. Firstly, we need to find the job $\sigma(i)$ with the maximum lateness and then check whether the first possibility holds on its corresponding chain $\sigma[h, ..., i]$. If not, we search from the tail of this chain towards the head, checking whether the condition from the second part of the statement holds until we find the job c. When found, all the preceding jobs we have checked did not pass the condition for c, give us the set J.

Based on this result, Carlier proposes a branching approach which produces a feasible schedule or finds that the problem has no feasible solution. In each node, a Schrage schedule is constructed. If the first option from Theorem 2 does not hold, job c and set J are identified, and two more nodes are added into the queue, one in which c has to be scheduled before J and in the second one vice versa. The first case is performed by bringing the due date of c closer by setting it to $d_{\sigma(i)} - \sum_{i \in J} p_i$. In the second, the release time is moved to later by setting it to $\min_{i \in J} r_i + \sum_{i \in J} p_i$.

This branching is performed until an optimal schedule is found. When this happens, we check whether all jobs meet their original due dates. If yes, we have found a feasible schedule for all the jobs. If not, this set of jobs cannot be feasibly scheduled.

4.4 Moore-Hodgson's algorithm

As a next component of the algorithm, we would like to know when it is no longer helpful to explore a partially constructed solution further. In order to do so, we would like to obtain a lower bound by quickly solving a simplified version of the problem. Let us now look at the case where all the jobs are available at the beginning. The problem $1 \mid \mid \sum U_j$ becomes much easier to solve. An algorithm to solve it was first presented by Moore in [5], where he attributes it to T. J. Hodgson. Hence it became known as Moore-Hodgson's algorithm.

Algorithm 3 The algorithm of Moore and Hodgson [5]

```
procedure MOORE($jobs = \{J_1, J_2, ..., J_n\}$, sorted with ascending due dates)
    $S \leftarrow \{\}$
    $t \leftarrow 0$
    for $i$ in $1, 2, ..., n$ do
        $S \leftarrow S \cup \{J_i\}$
        $t \leftarrow t + p_{J_i}$
        if $t > d_{J_i}$ then
            $l \leftarrow \arg\min_{j \in S} p_j$
            $S \leftarrow S \setminus \{l\}$
            $t \leftarrow t - p_l$
        end if
    end for
    return $S$
end procedure
```

The algorithm builds up the set S from jobs sorted by due dates. If adding a job would violate its deadline, the job with the longest processing time is removed from S. It can be proved inductively that this procedure will produce the optimal result. When using a priority queue to keep track of the job with the longest processing time, the time complexity of $\mathcal{O}(n \log(n))$ is achieved.

This algorithm provides one idea for a suitable lower bound for our problem, setting all the release times to zero. However, in practice, this relaxes the problem too much, so the lower bound would not be as strong. Nevertheless, this algorithm shows an instructive way to solve simple scheduling problems, which we will use in the following algorithm.

4.5 Lower bound

One of the lower bounds proposed by Baptiste et al. in [14] is to relax our problem to $1 \mid r_i, d_i; r_i < r_j \implies d_i \leq d_j \mid \sum U_j$. This is done by sorting the jobs by the ascending order of their release times and then iterating through them. If the due date of the job is lower than the maximal one found so far, it is relaxed to the maximal value. If the job's due date is later than the maximal one found so far, the maximum is updated to the current job's due date, as shown in the following pseudocode:

Algorithm 4 Relaxation of jobs to satisfy $r_i < r_j \implies d_i \leq d_j$

```
procedure RELAX($jobs = \{J_1, J_2, ..., J_n\}$)
    sort jobs by ascending order of release times
    $d_{max} \leftarrow 0$
    for $i$ in $1, 2, ..., n$ do
        if $d_{J_i} < d_{max}$ then
            $d_{J_i} \leftarrow d_{max}$
        else
            $d_{max} \leftarrow d_{J_i}$
        end if
    end for
end procedure
```

As mentioned during the literature review, this problem was first solved by Kise et al. [8] in $\mathcal{O}(n^2)$. The idea of the algorithm is similar to Moore's algorithm in that it iteratively adds new jobs into a set S, and when the new job's deadline is violated, a job is removed based on a given criterion.

Algorithm 5 Lower bound based on [9]

```
procedure LOWER_BOUND($jobs = \{J_1, J_2, ..., J_n\}$)
    $q \leftarrow Map(), q(S) \leftarrow 0$
    $S \leftarrow DEPQ(), S' \leftarrow PriorityQueue()$
    for $j$ in $1, 2, ..., n$ do
        $r \leftarrow 0$ if $j = 0$ else $r_{J_j} - r_{J_{j-1}}$
        while $r > 0$ do
            $i \leftarrow S.removemin()$
            $q(S) \leftarrow q(S) - q_i$
            if $q_i \leq r$ then
                $S'.insert(i$ with priority $q_i)$
                $r \leftarrow r - q_i$
            else
                $q_i \leftarrow q_i - r$
                $S.insert(i$ with priority $q_i)$
                $q(S) \leftarrow q(S) + q_i$
                $r \leftarrow 0$
            end if
        end while
        $q_{J_j} \leftarrow p_{J_j}$
        $S.insert(J_j$ with priority $q(J_j))$
        $q(S) \leftarrow q(S) + q_{J_j}$
        if $q(S) > d_{J_j} - r_{J_j}$ then
            $l \leftarrow S.deletemax()$
            $q(S) \leftarrow q(S) - q_l$
        end if
    end for
    return $S \cup S'$
end procedure
```

Lawler [9] improved this algorithm by further studying the recursive relationships when adding a job. A set S is enlarged job by job as well, but when the insertion of the next job violates its deadline, the job with the largest *effective processing time* is removed. These processing times, denoted q, are derived from the recurrence relations and kept up to date efficiently by the Algorithm 5.

4.6 Dominance properties

To speed up the branching, Baptiste et al. propose and prove several properties related to the structure of the problem. First of these are called *dominance properties*, which allow us to deduce that specific jobs must be on time and others late.

Proposition 3 *(Dominance property) [14] There is an optimal schedule where, for any pair of jobs i,j such that (1) $i < j$, (2) $r_j + p_j \geq r_i + p_i$ and (3) $d_j - p_j \leq d_i - p_i$, job i is on-time if j is on-time.*

This property, proven in the article, shows that when two jobs fulfil these properties, job j being on time implies that job i will be on time as well. This leads us to the following proposition, which decides what happens when we cannot schedule both of these jobs.

Proposition 4 *Let i and j be two jobs in F verifying Proposition 2. If $(r_i + p_i + p_j > d_j)$ and $(r_j + p_j + p_i > d_i)$ then $U_j = 1$.*

The conditions in Proposition 4 imply that we will be unable to schedule both of the jobs. Therefore as an application of Proposition 3, we set the "less interesting" job to be late. These properties might be checked by simply iterating over all possible pairs of jobs in $\mathcal{O}(n^2)$.

4.7 Tightening of job windows

Another way to speed up branching proposed by Baptiste et al. is to tighten the job windows of certain jobs. This is based on a previous article by Carlier and Pinson [50], where this process was first described, in the literature concering this problem, these propoisitions are also kown as *elimination rules.*

Firstly, the notion of precedence is described: we denote by $i \rightarrow j$ the fact that in every optimal schedule, job i will be scheduled before job j. Then it can be proven:

Proposition 9 *For any pair of on-time jobs i,j such that $r_j + p_j + p_i > d_i$ we have $i \rightarrow j$.*

Furthermore, this can be formulated even more powerfully when taking into account subsets of jobs and when defining:

$$r(B) = \min_{i \in B} r_i, p(B) = \sum_{i \in B} p_i, d(B) = \max_{i \in B} d_i$$

Authors prove that:

Proposition 10 $\forall c \in T \cup F$ *and* $\forall K \subseteq T - \{c\}$

- *if* $\min(r_c, r(K)) + p(K) + p_c > d(K)$ *then* $(\forall j \in K, j \rightarrow c)$,
- *if* $r(K) + p_c + p(K) > \max(d_c, d(K))$ *then* $(\forall j \in K, c \rightarrow j)$.

As the authors describe, these precedence relations translate to tightening of job release and due dates in the following ways:

Proposition 11 *For any job $i \in T$ and for any job $j \in T \cup F$:*

$$(i \to j) \Rightarrow \begin{cases} r_j = \max(r_j, r_i + p_i) \\ d_i = \min(d_i, d_j - p_j) \end{cases}$$

Correspondingly for the set version:

Proposition 12 *For any job $c \in T \cup F$ and for any subset $K \subseteq T - \{c\}$:*

$$(K \to c) \Rightarrow r_c = \max(r_c, r(K) + p(K))$$
$$(c \to K) \Rightarrow d_c = \min(d_c, d(K) - p(K))$$

The article proposed by Carlier and Pinson [50] provides two algorithms to find all precedence relations based on Proposition 9 and Proposition 10. The reported runtime of both of these algorithms is $\mathcal{O}(n \log(n))$ which, considering the power of Proposition 12 to simplify the whole problem, is undoubtedly of great significance to the overall runtime of the algorithm. The algorithm which performs all adjustments corresponding to Proposition 11 is straightforward, whereas the one implementing adjustments based on Proposition 12 is relatively complex, requiring the use of sophisticated custom data structures. Therefore we consider it beyond the scope of this thesis.

In [50], an algorithm which immediately adjusts the heads of the jobs is presented:

Algorithm 6 Adjustment of release and due dates $r_i < r_j \implies d_i \leq d_j$

```
procedure ADJUST($jobs = \{J_1, J_2, ..., J_n\}$)
    $\mathcal{L}_1 \leftarrow \{J_1, J_2, ..., J_n\}$ sorted by ascending $r_i + p_i$
    $\mathcal{L}_2 \leftarrow \{J_1, J_2, ..., J_n\}$ sorted by ascending $d_i - p_i$
    while $\mathcal{L}_1 \neq \emptyset$ and $\mathcal{L}_2 \neq \emptyset$ do
        $j_1 \leftarrow \mathcal{L}_1.min$
        $j_2 \leftarrow \mathcal{L}_2.min$
        if $j_1 == j_2$ then
            continue
        end if
        if $r_{j_1} + p_{j_1} \leq d_{j_2} - p_{j_2}$ then
            $\mathcal{L}_1.removemin()$
        else
            $\mathcal{L}_2.removemin()$
            $r_{j_1} \leftarrow max(r_{j_2} + p_{j_2}, r_{j_1})$
            $d_{j_2} \leftarrow min(d_{j_1} - p_{j_1}, d_{j_2})$
        end if
    end while
end procedure
```

Building a priority queue is performed in $\mathcal{O}(n \log(n)$, and it is followed by a maximum of $2n$ removals from the priority queue, which take $\mathcal{O}(\log(n)$ each. Therefore, the whole algorithm runs in $\mathcal{O}(n \log(n)$.

4.8 Final algorithm

Finally, having all the background knowledge and necessary tools, we might present the overall design of the algorithm. The pseudocode of the complete combinatorial approach is shown in the following Algorithm 7

As the input, we take a set of jobs to be scheduled. These are transformed into the initial state, with empty sets for both on-time and late jobs, all the jobs are free. The DFS nature of the algorithm is assured by the use of a first in, first out queue, to which new states are appended. The algorithm performs operations on the current, possibly adding new states into the queue. This continues until the queue is not empty, indicating that we have searched all relevant parts of the tree.

Algorithm 7 High level description of the B&B algorithm

```
procedure SOLVE($jobs$)
    $q \leftarrow empty\_queue()$
    $init\_state \leftarrow State([], [], [jobs])$
    $q.add(init\_state)$
    $best\_criterion \leftarrow \infty$
    $best\_solution \leftarrow null$
    while $\neg q.empty()$ do
        $state \leftarrow q.remove()$
        if $lower\_bound(state) \geq best\_criterion$ then
            continue
        end if
        $subproblems \leftarrow decompose(state)$                    ▷ Prop. 1 from [14]
        if $size(subproblems) > 1$ then
            $solution \leftarrow []$
            for $subproblem$ in $subproblems$ do
                $solution.append(solve(subproblem))$
            end for
            return $solutions$
        end if
        $state \leftarrow propq\_3\_4(state)$                         ▷ Prop. 3, 4 from [14]
        $state \leftarrow elimination\_rules(state)$                 ▷ Prop. 11, 12 from [14]
        $schedule \leftarrow schrage(state.on\_time)$
        if $feasible\_edd(schedule)$ then
            $chosen \leftarrow choose(state.free)$
            $q.append(State(state.t, state.l \cup \{chosen\}, state.f \setminus \{chosen\}))$
            $q.append(State(state.t \cup \{chosen\}, state.l, state.f \setminus \{chosen\}))$
        else
            $carlier(q, state, schedule)$
        end if
    end while
    return $best\_solution$
end procedure
```

In each state, or equivalently, node, we first evaluate the lower bounds. If any of these is greater than or equal to the best solution we have found so far, we might ignore this state as we will not receive a better criterion value. Afterwards, the selected properties are applied to the current state. The dominance properties allow us to deduce that some jobs have to be late and others on time, while the tightening of the jobs' time windows reduces the complexity of the remaining problem.

Afterwards, we have to determine whether a feasible schedule exists for the jobs we have set as on time, which we described in detail in Section 4.3. We do this by first creating Schrage's schedule and checking whether it is feasible. It turns out that it very often is. If not, however, the branching proposed by Carlier [48] has to be performed, which then fully decides this problem.

If we can feasibly schedule all on-time jobs, we choose a free job based on defined criteria and branch on it, adding two new states into the queue, one where it is late and the other where we consider it on time. It is preferred first to explore the node where the job is on time, to quickly gain a solution to the whole problem, which might be compared to the lower bounds in subsequent nodes.

The selection of the job to branch on in the original article is performed by a simple heuristic with the idea that it is good to schedule on-time a job with a large execution window and small processing time. This part of the algorithm is what we would like to enhance with a machine learning based heuristic. We believe that scheduling first the jobs the ML model will be confident are on time will lead to less backtracking and thereby faster runtime.

Chapter 5

Machine learning solution

In this chapter, we shall describe our proposed machine learning solution, which aims to speed up the combinatorial algorithm from the literature. Given an instance of the $1 \mid r_j \mid \sum U_j$ problem, our goal will be to predict which jobs will be early and which will be tardy in an optimal solution. Based on this prediction, we will start branching on the jobs the ML is most confident about.

The very first step of this task is to gather enough data. We will need large datasets with solutions on which we can perform training. Afterwards, we will need to find a way to transform the datasets composed of tuples of instances and their optimal solutions into vector representation suitable for machine learning. This step is called feature extraction.

Subsequently, we will need to design an ML model with enough expressive power to handle this task. Our goal is not only to achieve reasonable accuracy on the training and testing datasets, but we would also like the model to generalize well, providing good results even on instances with more jobs than it was trained on.

During the design phase, our architecture will be thoroughly tested. We will assess several metrics on all our datasets, thereby proving the validity of our approach. Finally, we shall incorporate this prediction framework into the combinatorial algorithm and evaluate its effectiveness.

All of these steps require careful consideration as each of them can potentially impact the accuracy of the final framework. Only by thorough experimentation and verification of the results of each step can we obtain excellent results. Let us now describe these steps in detail.

5.1 Dataset generation

In our scenario, training examples are tuples composed of an instance of the $1 \mid r_j \mid \sum U_j$ and an optimal solution, which immediately creates a dilemma because an instance may have more than one optimal solution. Which of them should we include in the training dataset?

The answer we believe is correct is that one solution is enough. Using multiple or even all of them is very computationally demanding, and presenting several possible targets for the same example may ”confuse” the model and lead to worse results.

As for the actual generation of problem instances, we follow the process first described by Baptiste et al. in [51]. Firstly, the job's processing time is generated randomly from a uniform distribution on $[p_{min}, p_{max}]$ and the release date from a normal distribution $(0, \sigma)$. The due date is then calculated as $d_j = r_j + p_j + m_j$, where the slack m_j is drawn from a uniform distribution $[0, m_{max}]$.

However, we cannot simply generate instances job by job. Consider a case where there are only a few jobs to be scheduled in a large window of time; this problem will be fairly easy. On the other hand, when we want to schedule a large number of jobs in a small amount of time, the complexity of the problem is much greater. To control this, we define the *load* of an instance as:

$$load = \frac{\sum\limits_i p_i}{\max\limits_i d_i - \min\limits_i r_i} \tag{5.1}$$

The authors show that when generating jobs in this fashion, the expected load can be calculated as:

$$load \sim \frac{n(p_{min} + p_{max})}{2(4\sigma + p_{max} + m_{max})} \tag{5.2}$$

Because of this, the datasets are usually described by the tuple $(n, (p_{min}, p_{max}), m_{max}, load)$, and the σ used in the generation is calculated accordingly. Based on these rules, we have generated many instances with differing numbers of jobs, load, and m_{max}. These instances were then solved optimally using an ILP solver, thus giving us the information on which jobs are late and on time in each instance. This is the last step of dataset generation and gives us complete datasets for training and testing.

5.2 Feature extraction

Once we have the training and testing datasets, we need to transform them into vectors, so we can input them into a mathematical ML model. Our data is in the form of tuples consisting of a list of jobs in (p_i, r_i, d_i) format and a one-hot enconding of tardy jobs. Unsurprisingly, the targets are in a perfect form for machine learning and even the features seem to suitable for machine learning.

However, we would like to enrich the input data to capture more information than just the three times for each job. Specifically, we want to propagate information from other jobs into each input job. We believe that this adds expressive power to our final model, as some of the features we are handcrafting would not be easy for the machine learning model to emulate.

In total, beside the triplet of times, we add 23 additional features to each job used in our solution. For documentation purposes, let us list all 26 of the features we extract. We use the standard notation, that for job i we have the processing, release and due times p_i, r_i, d_i, the symbol $\sum\limits_j$ refers to summing over all individual jobs in the given instance. We also define the maximum time which occurs in an instance: $t_{max} = \max(\max\limits_j p_j, \max\limits_j d_j)$.

The first twelve of our selected features for job i are: p_i, r_i, d_i, $p_i / \sum\limits_j p_j$, $r_i / \sum\limits_j p_j$, $d_i / \sum\limits_j p_j$, p_i / t_{max}, r_i / t_{max}, d_i / t_{max}, $p_i / \max\limits_j d_j, r_i / \max\limits_j d_j$, $d_i / \max\limits_j d_j$. These all aim to put together the relation of the individual jobs to the instance as a whole. Afterwards, we add features which represent relationship between pairs of jobs. We sort the jobs first by due date and add the features $p_{i+1} > p_i$, $r_{i+1} > r_i$, $p_{i-1} < p_i$, $r_{i-1} < r_i$, as one/zero representing true and false. A default value is used for jobs which have no preceding or following job. We also add analogous features when sorting the jobs by processing times: $d_{i+1} > d_i$, $r_{i+1} > r_i$, $d_{i-1} < d_i$, $r_{i-1} < r_i$. Default values are again used. As the two last features, we have decided to add whether the previous job with the current one and the current one with the next job satisfy the dominance property from Proposition 3 described in Section 4.6. Again as one/zero representing true and false.

Finally, we create the EDD schedule and add a binary representaion of whether the job is on-time in it as well as a representation of how much it is late as a fraction of its processing time, or zero if it is on-time. Analogously, we add two more of these features after constructing a schedule based on the shortest processing time dispatch rule[1].

5.3 Machine learning model

Now that we have an input vector of features extracted by hand, we begin designing the ML approach. We have decided to use a neural network, as it seems to have the most expressive power based on previous approaches in the literature. It already performs well guiding an algorithm on a similar scheduling problem [47] and is able to guide several other combinatorial algorithms [41][43][45].

Since we need to predict a vector with a length equal to the number of input jobs represented by feature vectors, and we would like to make such predictions considering the instance as a whole, we need an architecture that can provide such outputs. In our case, we will use either a recurrent neural network (RNN) or a graph neural network. Since we do not see an intuitive graph structure in this problem, we decided to use an RNN.

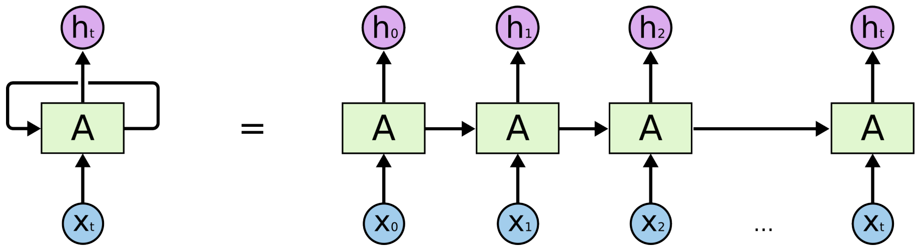


Figure 5.1: RNN repeatedly applies the same cell[1]

As illustrated by Fig. 5.1, an RNN is formed by a *cell*, which is recursively applied to each input vector and aside from creating an output, it updates its inner state. In this way, the architecture can pass information throughout the evaluation of the model and handle variable-length inputs.

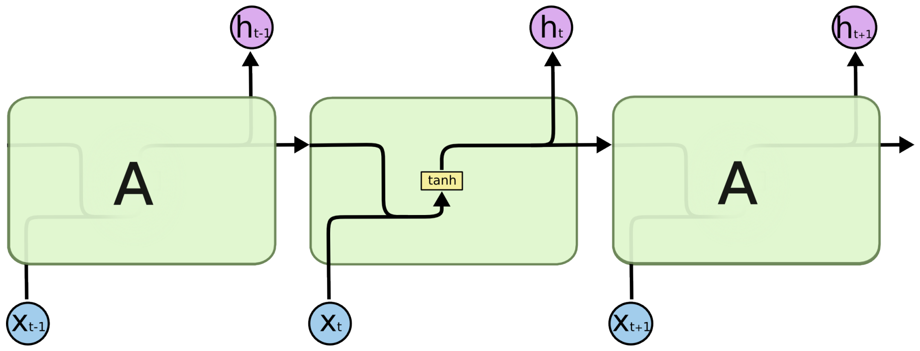


Figure 5.2: A simple recurrent neural network[2]

In the simplest case, as shown in Fig. 5.2, an RNN cell can contain only one *tanh* layer whose output is at the same time used as the input of the next cell. However, more complex architectures have been described, which obtain better results. We are going to show an example in the upcoming Section 5.3.2.

[1]`https://colah.github.io/posts/2015-08-Understanding-LSTMs/img/RNN-unrolled.png`, from [52]. Used with permission from the author.

[2]`https://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-SimpleRNN.png`, from [52]. Used with permission from the author.

5.3.1 Embedding

However, before we employ such a network, an *embedding* is usually performed. It can be though of as an analogue to feature extraction, but it is performed by the neural net and learned over time. The idea is to propagate information between distant jobs and obtain vectors representing the instance in this aggregated form.

We have decided to use 1D convolutional layers [53] with kernel size of three for this task. Kernel size defines the number of elements of the vectors to be used in the weighted combination that produces each output. In this case, the value of the output will be a linear combination with the learned weights of the value of the previous, current and next values. Furthermore, one output calculated in this way is called a *channel*, we can calculate more of these channels in parallel to produce different convolutions which focus on different aspects of the jobs.

In order to propagate information between more distant jobs, we stack several of these 1D convolutional layers. In the beginning, the number of input channels is equal to the number of features, that is 26, and the number of channels for output is 32. We lift the dimension three more times, convoluting from 32 channels to 64, then from 64 to 128 and finally from 128 to 256 before dropping the dimension back down to the width of the LSTM.

5.3.2 LSTM

After embedding, we are ready to input the vectors into an RNN model. We have chosen the LSTM architecture because it is a model that has proved its usefulness on various learning tasks, and most importantly, has been successfully applied to scheduling before [47]. Firstly, let us quickly describe how LSTM works and afterwards, we will thoroughly describe how we implement it into our complete model.

The design of an LSTM is shown in Fig. 5.3. The main idea [24] is to keep the cell's state, which allows the network to make future decisions based on previously seen data. The upper horizontal line represents the state in Fig. 5.3, and as expected, it runs from the beginning to the end throughout the whole chain.

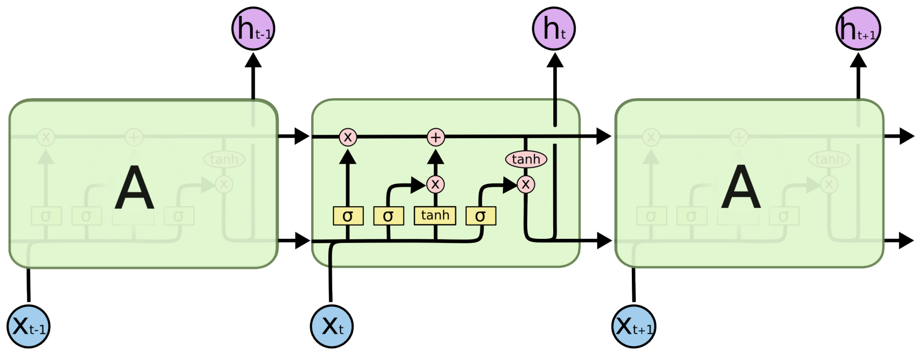


Figure 5.3: Structure of an LSTM cell[3]

When a new input is fed to the cell, it concatenates it with the output of the previous step and then puts this extended input into three so-called gates. Firstly, the *forget gate* decides which parts of the information stored in the cell's state to remove. The extended input is passed through a linear layer with a sigmoid activation function, and the result is multiplied by the previous state vector. This multiplication allows the cell to zero out certain elements stored in the state while keeping others [24].

[3] `https://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png`, from [52]. Used with permission from the author.

Subsequently, we would like to update the state of the cell. To do this, we apply another linear layer with sigmoid activation and multiply it with a linear layer with tanh activation. The result of this *input gate* is then added to the state of the cell.

Finally, the output of the cell in the given step is the result of the multiplication of the enhanced input passed into a linear layer with a sigmoid activation function and the tanh of the current state vector [24].

The parameters to consider are mainly the length of the state vector, which is called the width of the LSTM, and it is achieved by choosing the appropriate dimensions of the matrices in the linear layers inside the three gates. Furthermore, it is possible to stack multiple of these LSTM layers on top of each other, considering the outputs of one layer as the inputs of another. Lastly, it is possible to define a bidirectional layer, where after applying the LSTM described here, another layer is applied from the end to the beginning, possibly receiving the state of the last cell of the previous layer as its initial state. The resulting vectors of these two layers are then concatenated [24].

As a final remark, it is essential to note that the architecture of the gates defined here is not the only one that exists. A notable example is the *gated recurrent unit* (GRU) introduced by Cho et al. [54]. Greff et al. [55] compare eight different LSTM variants and find that their performance does not differ significantly, which is why we did not consider these variations in our choice of a model.

In our model, the LSTM layer receives a vector for each input job, with a length defined by the embedding output length. We use the bidirectional version of the LSTM architecture, which means that one LSTM cell is applied recursively from beginning to the end, while another cell with its own weights and biases is applied recursively from the end to the beginning. An output for each input is then generated by concatenating output vectors of the forward and backward pass. The idea behind this is obvious; the network can not only use information from the first inputs to abstract and use in the later applications, but also vice versa. This approach has found wide use in NLP. One such example is by Huang et al. [56].

We also can stack more than one LSTM cell for each direction behind each other. As Goldberg [57] notes, the theoretical justification of such architectures is not entirely clear. However, they have been empirically found to perform very well on specific problems. We shall experimentally show whether this applies in our case.

5.3.3 Attention

As mentioned during the literature review, the *attention* mechanism is another process used to improve the performance of recursive networks [26][58]. In a nutshell, it works similarly to the human concept of attention. We add modules, which learn a set of weights to decide which elements of the input vectors are worth paying more attention based on other elements of the input sequence, or another input data.

The attention mechanism may be applied in various ways [58]. However, the more complicated methods aim to create the concept of attention given a pair of inputs, each from a different set of objects. Unfortunately, given our problem, we do not see an opportunity for such usage. In our case, the only option is to apply the so-called *self-attention.* This method predicts the relevance of parts of a single input vector given the rest of the sequence.

The self-attention layer works by using three matrices $\mathbf{W_Q} \in \mathbb{R}^{n \times d_q}$, $\mathbf{W_K} \in \mathbb{R}^{n \times d_k}$, $\mathbf{W_V} \in \mathbb{R}^{n \times d_v}$. These are initialized in the beginning, and their values are learned throughout the training. Based on the batched input vectors $\mathbf{X} \in \mathbb{R}^{n \times d}$, the matrices $\mathbf{Q} = \mathbf{X}\mathbf{W_Q}$, $\mathbf{K} = \mathbf{X}\mathbf{W_K}$ and $\mathbf{V} = \mathbf{X}\mathbf{W_V}$ are computed. These three matrices can then be passed into an attention mechanism, we propose to use the *dot-scaled attention*, which is defined as:

$$A = softmax\left(\frac{\mathbf{QK^T}}{\sqrt{d_q}}\right)\mathbf{V} \tag{5.3}$$

Furthermore, we propose to use a *multi-head attention* [30], which calculates several attention modules in parallel, which are then concatenated and linearly transformed to required length, which allows to attend to different parts of the input sequence differently. One set of attention weights can be learned for short-term dependencies, whereas another one for long-term dependencies. We believe that by using attention, the model might learn to take advantage of hidden relationships in the input data, and thereby the accuracy of our model might improve, which we experimentally test in the following chapter.

5.3.4 Final classifier

The recurrent LSTM layer outputs for each input vector a vector with a length equal to the length of the state of the LSTM[4]. Moreover, if attention is used, we receive an attention vector for each input. However, we need only a number between 0 and 1, representing the probability we assign to the given job to be tardy. We achieve this by introducing a fully connected layer with an input width corresponding to the output of the LSTM and an output width of 256. When we use attention, we concatenate the output of the attention to the output of the LSTM and feed this enhanced vector into a wider input linear layer. We use the ReLu activation function in all cases. Afterwards, we use two fully connected layers to scale down the result from 256 to 128 and then from 128 to 1. Finally, to obtain the desired probability, we pass this number through a sigmoid function.

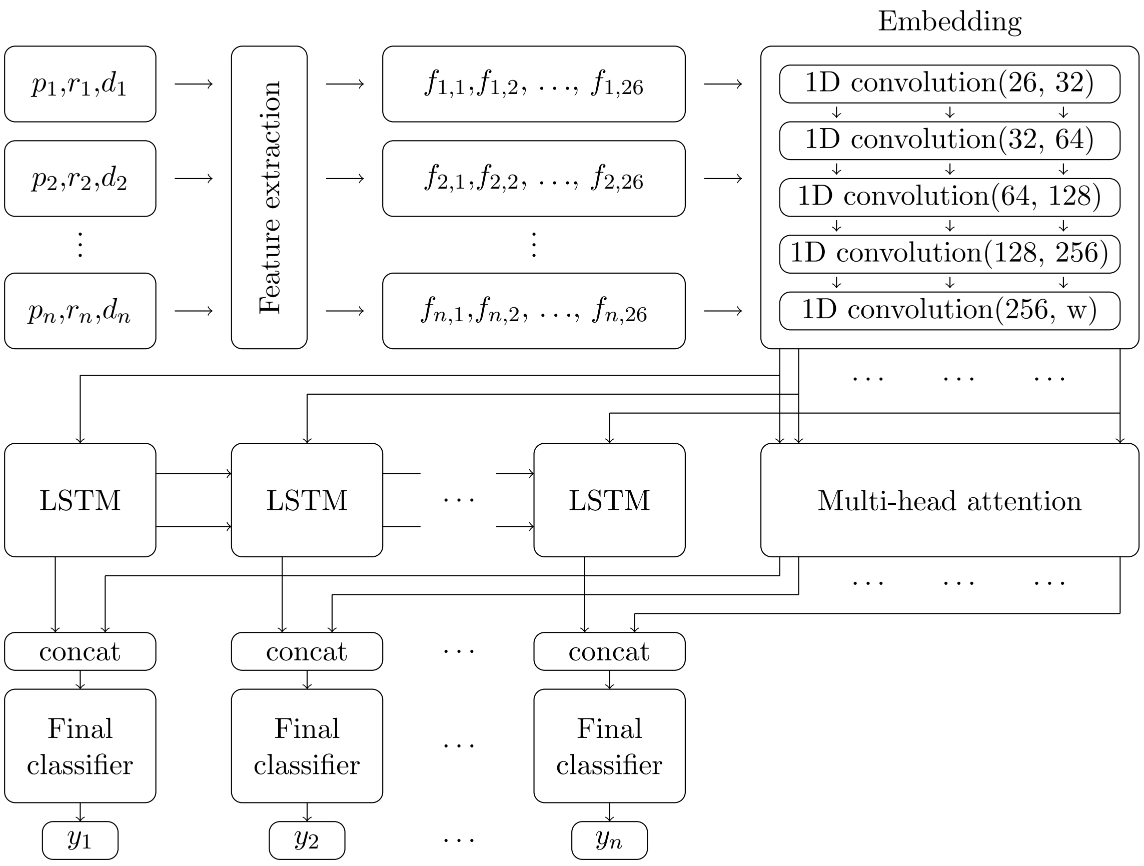


Figure 5.4: Propsed machine learning solution

[4]Or double the width if the LSTM is bidirectional

5.4 Training

The target vector for each input is the one-hot encoding of the tardy jobs. The training is performed using the binary cross-entropy (BCE) loss defined as:

$$BCE = \frac{1}{n}\sum_{i=1}^{n} y_i \log(p(y_i)) + (1 - y_i)\log(1 - p(y_i)) \tag{5.4}$$

Unless otherwise specified, we use a set of 100 000 of our generated instances, each with a calculated optimal solution. The number of jobs in the instances ranges from 10 to 100, load values used are $load \in \{0.5, 0.8, 1.1, 1.4, 1.7, 2.0\}$ and $mmax$ used is $mmax \in \{5, 10, 15, ..., 50\}$, which gives us around 17 instances for each combination of the parameters. For the training itself, we use the Adam optimizer [59], which is an extension of stochastic gradient descent which computes individual learning rates for different parameters during the learning, among other improvements.

5.5 Architecture overview

To sum up, we begin with a dataset of generated instances of the $1 \mid \mid \sum U_j$ problem. We extract 26 features chosen by hand for every job in each instance. On this dataset of extracted features, we train a neural network composed of an embedding part, a recurrent LSTM layer and a final classifier using BCE loss. We present a diagram of our architecture in Fig. 5.4. However, this architecture is not final and allows for certain slight modifications. Details such as the size of the training dataset, the number and depth of the LSTM layers, or whether to use the embedding are open for debate, and we shall explore them further in the next chapter.

Chapter 6

Experiments

In this chapter, we shall briefly comment on how we implemented our proposed solution. Afterwards, we show an experiment which demonstrates the validity of our approach. We then evaluate our architecture for different values of several hyperparameters, which leads us to the best-performing model. We then attempt to use this model to guide the deterministic algorithm. Finally, as an alternative to the deterministic solution, we propose a simple heuristic based on the neural network's prediction.

6.1 Implementation

We implemented both the deterministic algorithm and the machine learning model in the Python programming language, version 3.7.11. The deterministic part of the algorithm requires very few additional packages aside from a priority queue data structure. The ML part was implemented using the `PyTorch` framework, which provides standard implementations of the basic linear and sigmoid layers, as well as a more complicated LSTM and multi-head attention modules.

We used the `MongoDB` database to keep track of our instances and results. We managed to keep track of the many metrics we collected during training with the help of an online experiment tracking service. The training and evaluation was performed on the Czech Institute of Informatics, Robotics and Cybernetics computer cluster, where each training process was assigned one Nvidia Tesla V100 GPU.

6.2 Training results reporting

In order to perform the experimentation, which will give us an understanding of the effect each hyperparameter has on the accuracy of the model, we define a baseline model with the following parameters: width of the LSTM state vector (w) 64 for each direction (bidirectional LSTM is used), number of LSTM layers (d) 1, size of the training dataset (train size) 100 000 instances, multi-head attention with one head (h), learning rate (lr) of 0.0005, feature extraction, convolutional embedding and the final classifier as described in the previous chapter. In all further figures, when describing a model, all parameters except those stated otherwise are always set to the baseline value.

Firstly, we show the train and test accuracy of five different architectures as well as some additional metrics. These architectures are chosen to differ in a different hyperparameter from the baseline, and we aim to show that we can train a well-performing model. However, such metrics only consider the average results on the training and testing datasets. To evaluate the performance, we take approximately 200 000 completely unseen instances, again generated as described in Section 5.1. These instances range in size from 5 to 200, giving us around a thousand per size. For each hyperparameter, we evaluate the accuracy of the models which differ in its value on these instances. Results are reported as the mean and standard deviation per instance size.

6.3 Training examples

In the following Fig. 6.1 we present the accuracy of five different architectures on the training set.

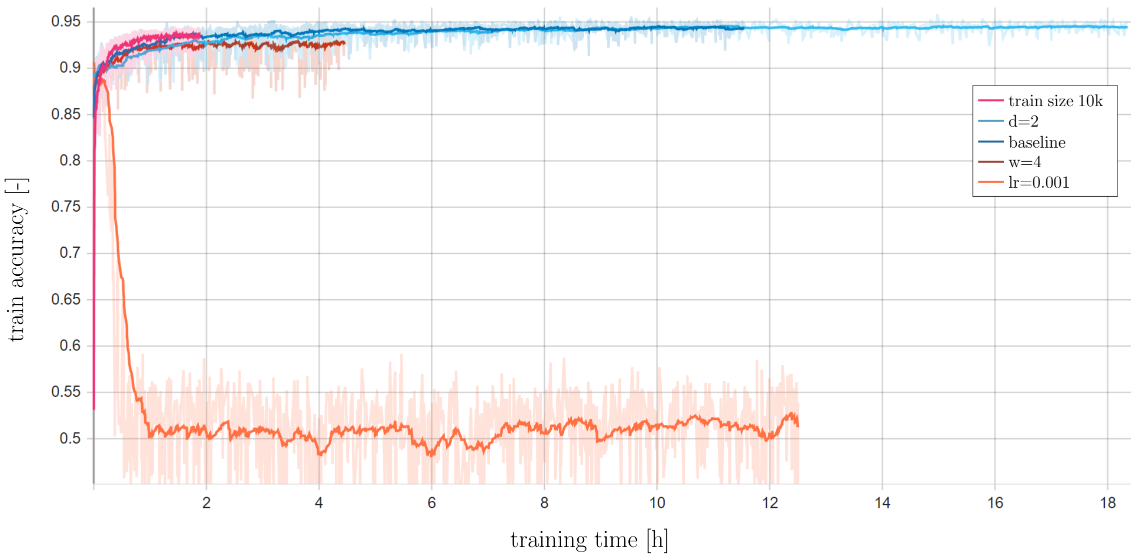


Figure 6.1: Accuracy of various architectures on the training dataset

As expected, the more complex a model is, the longer it takes to train. All models except for one achieve good accuracy on the training set. The model with a higher learning rate achieves its best results during the first epoch; afterwards, its accuracy decreases to the point of random decision. We believe such a learning rate is too big, and the learning algorithm completely diverges.

Furthermore, we can observe that the successful models already achieve very reasonable accuracy during the first hours of the training. Afterwards, the accuracy improves only very slowly. Considering the Fig. 6.2, we can see that even the test accuracy follows this pattern, which questions whether these latter parts of the training are necessary.

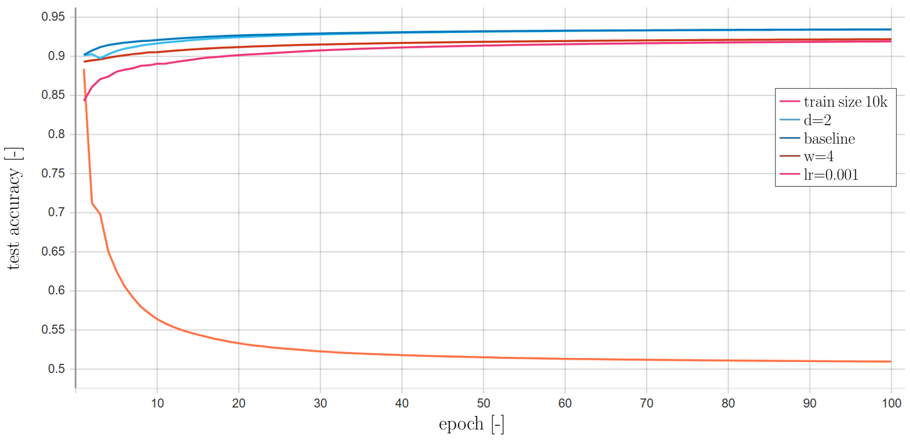


Figure 6.2: Accuracy of various architectures on the test dataset

However, to further illustrate the structure of the results of our approach, we define two metrics, $determined_p$ (shortened as det_p) and $accuracy_p$ (shortened as acc_p) as follows:

$$determined_p = \frac{\sum\limits_{y_i \in output} [\![y_i \in [0,p] \vee y_i \in [1-p,p]]\!]}{length(target)}$$

$$accuracy_p = \frac{\sum\limits_{y_i \in output} [\![y_i = \hat{y}_i \wedge (y_i \in [0,p] \vee y_i \in [1-p,p])]\!]}{length(target)}$$

The determined metric tells us how many jobs are near the edges of the zero-one interval. We can interpret it as a percentage of jobs classified with a corresponding degree of confidence. The accuracy metric does the same but only for correctly predicted data. Fig. 6.3 shows that as the training progresses, the proportion of outputs very close to 0 and 1 increases. This behaviour also shows in the $acc_{0.05}$ metrics and in metrics with a bigger threshold p. In other words, the network is becoming more confident with its outputs.

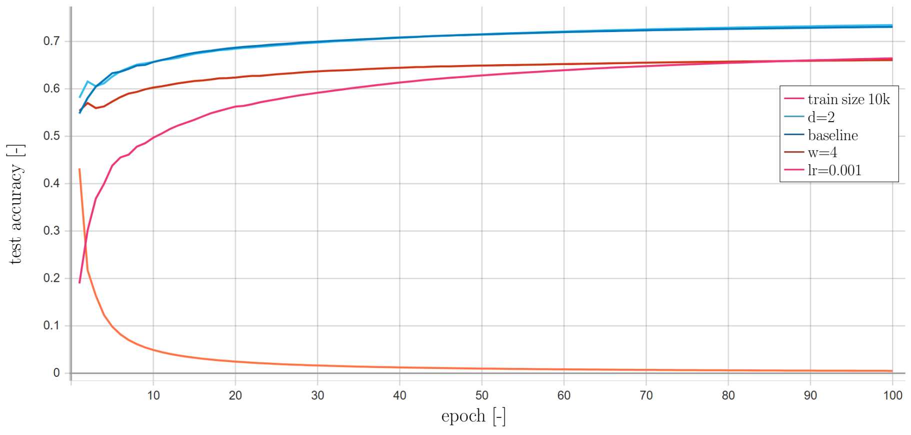


Figure 6.3: $determined_{0.05}$ of various architectures on the test dataset

In Fig. 6.4, we show detailed results of these two metrics for the architectures from the previous graphs on the training dataset for the last training epoch.

	baseline		train size 10k		d=2		w=4		lr=0.001	
p	det_p	acc_p	det_p	acc_p	det_p	acc_p	det_p	acc_p	det_p	acc_p
0.5	100	94.3	100	93.4	100	94.5	100	92.8	100	51.4
0.4	97.0	92.7	96.4	91.4	97.0	92.8	96.1	90.6	0	0
0.3	93.7	90.5	92.3	88.7	93.8	90.7	91.8	87.9	0	0
0.2	89.6	87.4	87.2	84.9	89.7	87.6	86.6	84.0	0	0
0.1	82.8	81.5	79.1	77.8	82.9	81.7	78.4	77.0	0	0
0.05	74.6	73.9	69.9	69.2	74.8	74.2	69.4	68.7	0	0

Figure 6.4: Metrics on the training dataset (in percent)

We can see fairly good results for the four correctly performing architectures. It is important to note that when the neural network is confident that its output is in $[0, 0.05] \cup [0.95, 1]$, then for example the baseline model correctly labels around 99% of the jobs $(73.9/74.6 = 0.9906)$. Moreover, such a situation occurs for 74.6% of the jobs.

We present the same results for the test dataset in Fig. 6.5.

	baseline		train size 10k		d=2		w=4		lr=0.001	
p	det_p	acc_p	det_p	acc_p	det_p	acc_p	det_p	acc_p	det_p	acc_p
0.5	100	93.4	100	91.7	100	93.3	100	92.1	100	51.2
0.4	96.7	91.5	95.6	89.3	96.7	91.5	95.66	89.7	2.6	2.1
0.3	93.0	89.2	90.9	86.3	93.0	89.2	90.9	86.7	1.6	1.5
0.2	88.4	85.9	85.0	81.9	88.4	85.9	85.2	82.3	1.4	1.3
0.1	81.1	79.6	75.6	73.9	81.2	79.7	76.0	74.5	1.1	1.0
0.05	72.5	71.7	65.2	64.3	72.8	71.9	65.7	64.9	0	0

Figure 6.5: Metrics on the test dataset (in percent)

As in the previous graphs, good performance on the training dataset translates well to the test dataset. Interestingly, the model with one additional LSTM layer (d=2) has nearly identical performance to the baseline. Also, on the most confident predictions in the set $[0, 0.05] \cup [0.95, 1]$ it again has an accuracy of nearly 99% ($71.9/72.8 = 0.9876$).

6.4 Hyperparameter optimisation

Now that we have shown that models based on our proposed solution provide good results, we go over the individual hyperparameters and evaluate several models which differ in one of them on nearly 200 000 previously unseen instances. All other parameters are left as defined in our baseline. We report the results as described, a mean and standard deviation of the $acc_{0.5}$ metric per instance size.

6.4.1 Training dataset size

Firstly, we would like to see how the size of the training dataset influences the results. In Fig. 6.6 shows the mean of $acc_{0.5}$ of models trained on datasets of six different sizes.

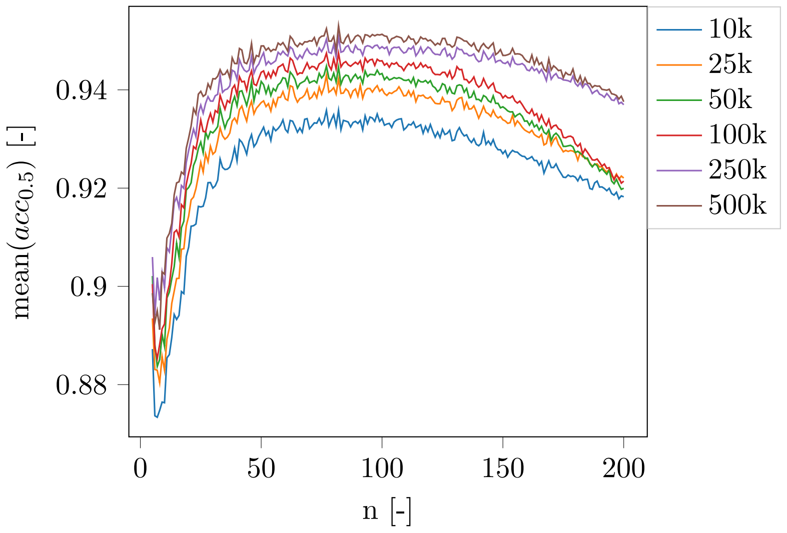


Figure 6.6: Mean of $acc_{0.5}$ for different train dataset sizes

The results show that even as little as ten thousand instances is enough to learn reasonably good results. However, we need more training data to obtain a model that generalizes very well to instances of size 200. Interestingly, while the models trained on smaller datasets provide worse results in the class of instances they were trained on, they seem to lose accuracy slower when generalizing on larger instances. The most prominent example is the model trained on 100 000 examples, which reaches nearly the same accuracy as the one trained on 10 000 when predicting inputs of length 200. However, it is worth noting that the absolute differences between the accuracies of the models are tiny, only around 2% between the best and worst ones.

In the following Fig. 6.7 we can see that the standard deviation of the accuracy is very small, which means that the model gives reasonable results consistently, without much variation between different instances.

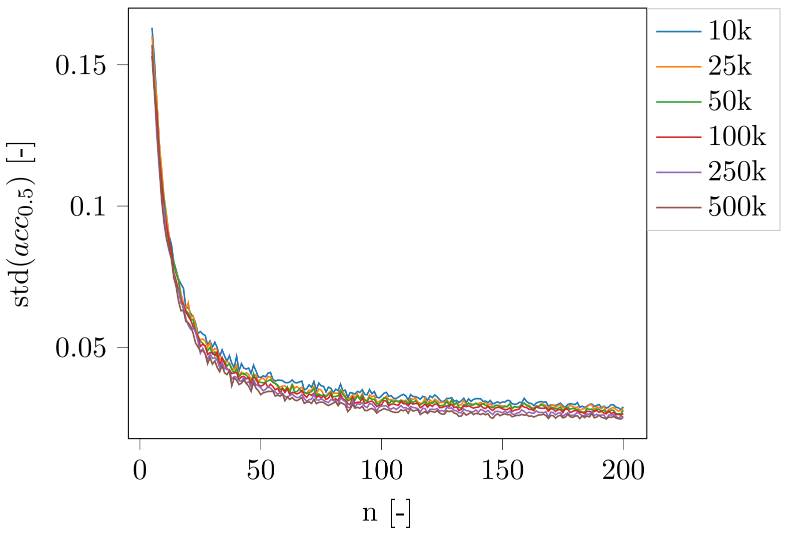


Figure 6.7: Standard deviation of $acc_{0.5}$ for different train dataset sizes

6.4.2 Width of LSTM

We chose the width of the LSTM as the next hyperparameter to evaluate. The width plays a vital role as it defines the length of the internal state of the LSTM, thereby influencing its capacity and the overall expressive power of the model. The mean of $acc_{0.5}$ is presented in Fig. 6.8.

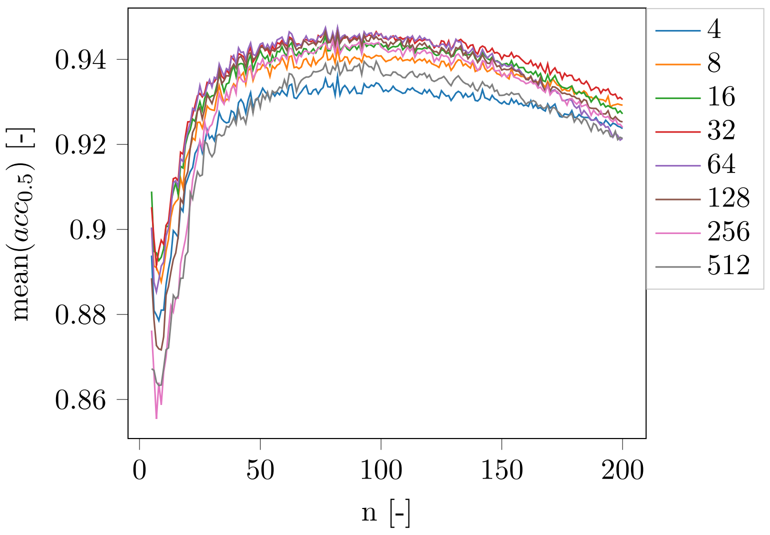


Figure 6.8: Mean of $acc_{0.5}$ for different widths of the LSTM

As expected, with bigger width comes slightly better accuracy on instances of size up to 100, which holds with the only exception of the model with the widest LSTM state vector, which suddenly starts producing results comparable with the simplest architecture. Interestingly, the width of 32 seems to produce the model which generalizes the best. The standard deviation of the accuracy is presented in Fig. 6.9, and it is again very small.

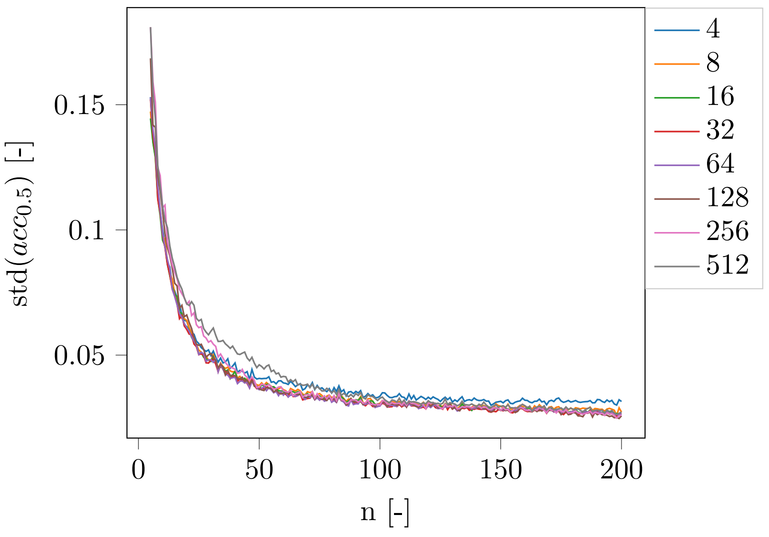


Figure 6.9: Standard deviation of $acc_{0.5}$ for different widths of the LSTM

6.4.3 Number of LSTM layers

In this experiment, we follow up on what was discussed in Chapter 5, where we presented the idea from the literature that for specific problems, stacking more layers of the LSTM on top of each is essential in achieving good results, whereas for others, it provides a slight advantage. The results for the mean of $acc_{0.5}$ are shown in Fig. 6.10.

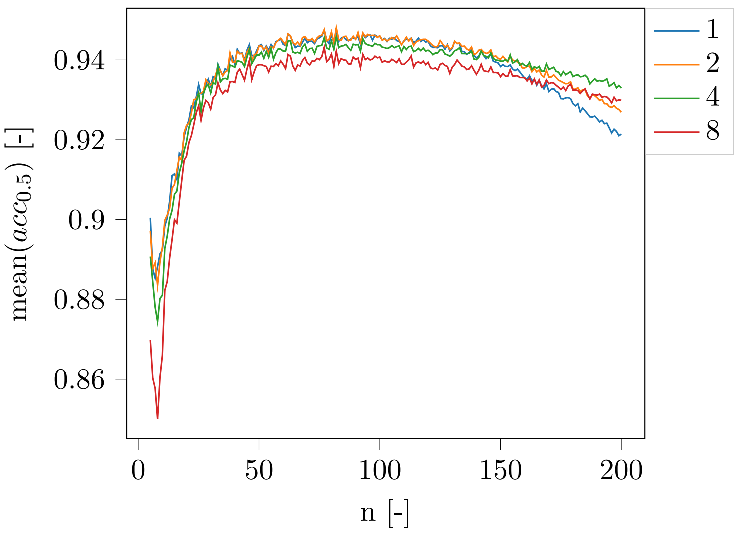


Figure 6.10: Mean of $acc_{0.5}$ for different number of layers of the LSTM

In our case, the idea of stacking more LSTM layers seems to be true, but only to a very limited extent. Even though the architecture with four layers provides some of the best results on instances of sizes up to 100 and generalizes the best, the difference in performance is under one per cent for most of the input sizes. Once more, as shown in Fig. 6.11, changing the number of LSTM layers does not increase the standard deviation of the accuracy.

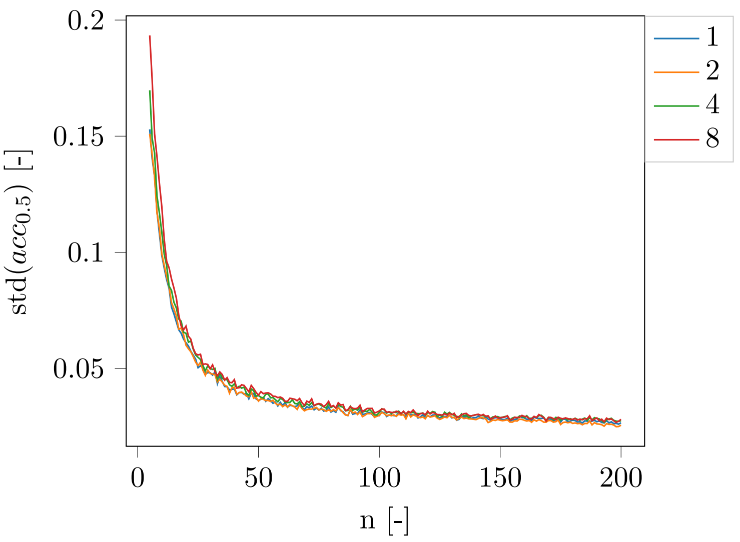


Figure 6.11: Standard deviation of $acc_{0.5}$ for different number of layers of the LSTM

6.4.4 Number of attention heads

Another hyperparameter in our model is the number of different heads used in the multi-head attention module. More heads mean running several attention mechanisms in parallel and then combining their results, which allows learning different weights for attention between elements that are closer together or further apart. The results are shown in our standard form in Fig. 6.12.

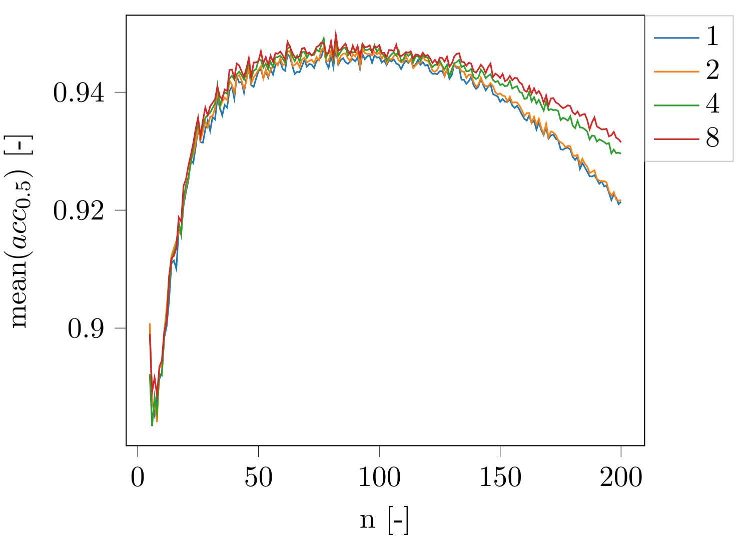


Figure 6.12: Mean of $acc_{0.5}$ for different number of attention heads

The results correspond to our theoretical expectations. While the accuracy is nearly identical in the instances of sizes up to 100, using more heads leads to models that generalize better. Once again, the standard deviation shown in Fig. 6.13 is minimal.

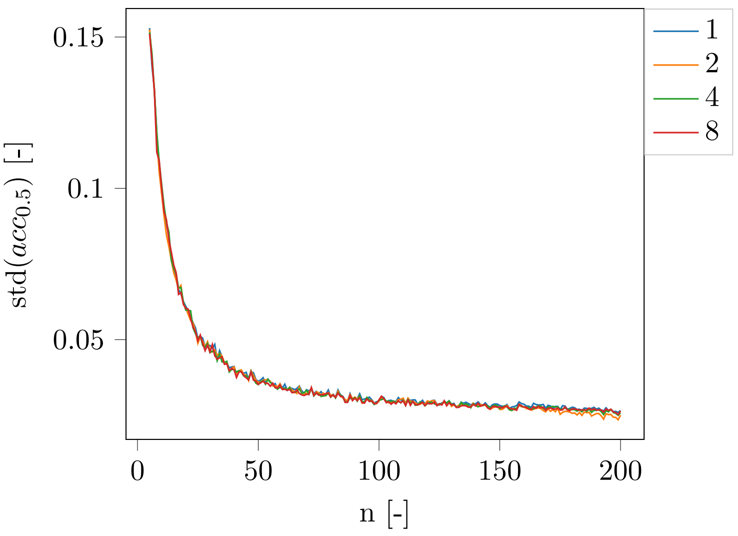


Figure 6.13: Standard deviation of $acc_{0.5}$ for different number of attention heads

6.4.5 Feature extraction, embedding, attention and final classifier

There are still a few parts of our proposed model that we want to test empirically. In our solution, we propose manually extracting 26 features for each job and then using a 1D convolutional embedding layer. Furthermore, we propose to use attention and a final classifier composed of three linear layers with one ReLU between them.

We evaluate a model without this component or with its simpler equivalent for each of these parts. Instead of complicated feature extraction, we only use the triplet p_i, r_i, d_i as the input. Instead o 1D convolutional embedding, we use a simple linear layer mapping from input size to the width of the LSTM. In the third model, we remove the attention altogether and finally, instead of using more linear layers in the final classifier, we use just one. The results of these models are shown in our standard form in Fig. 6.14.

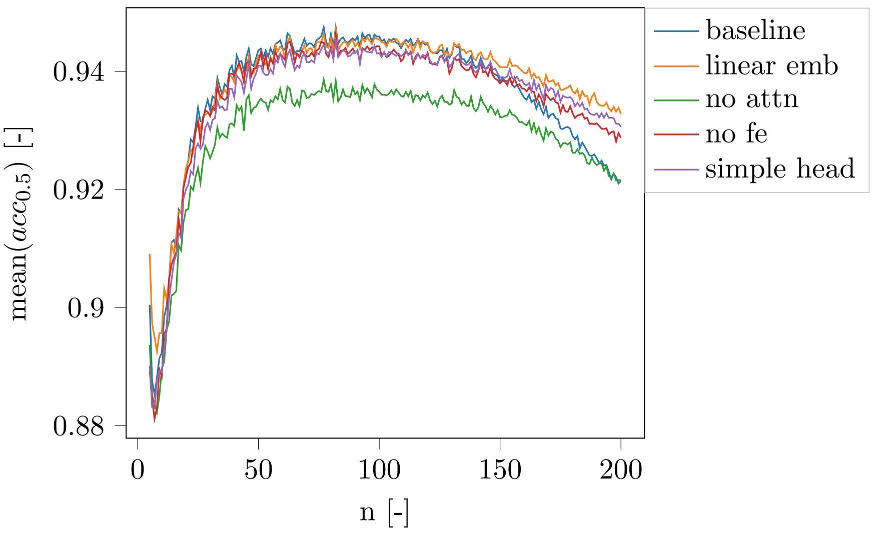


Figure 6.14: Mean of $acc_{0.5}$ given changes of the complexity of the model's components

From the results, we can observe that attention improves the model's accuracy on smaller instances but does not help it generalize that much. On the other hand, using a linear embedding instead of a convolutional one and simplifying the final classifier creates a model that generalizes better. The same behaviour is observed when not using feature extraction, although to a slightly smaller degree. These results were surprising to us, but sometimes, simpler is better. As always, the standard deviation of the accuracy of the models shown in Fig. 6.15 is tiny.

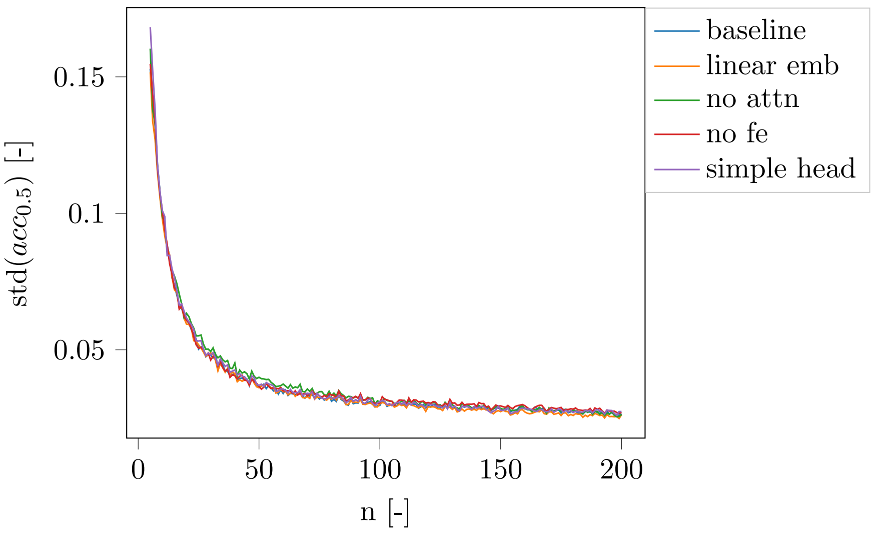


Figure 6.15: Standard deviation of $acc_{0.5}$ given changes of the complexity of the model's components

6.4.6 No LSTM

In our final experiment in which we train neural networks, we want to prove the usefulness of the recurrent architecture. We train a model in which we pass the output of embedding of each input job into one linear layer, which attempts to predict the output before we pass it through a sigmoid function to normalize it between 0 and 1. We keep using the same weights for each job, mapping the linear layer over the input vectors. We test out different widths of the output of embedding, which is equal to the input width of the linear layer.

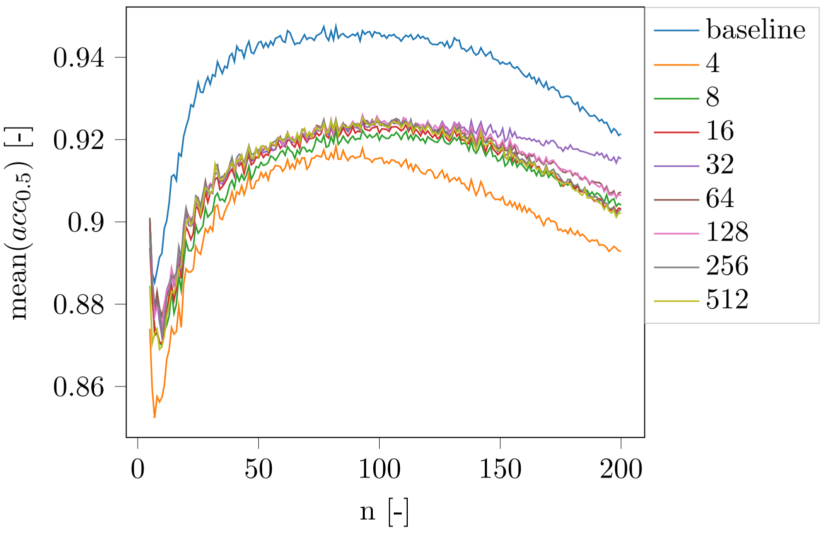


Figure 6.16: Mean of $acc_{0.5}$ when not using the LSTM architecture

The results obtained by a simple, purely linear model are surprisingly good, given the nature of the problem. However, only when considering the instance as whole can better models that generalise well can be trained. As always, the standard deviation of the accuracy shown in Fig. 6.17 is very small.

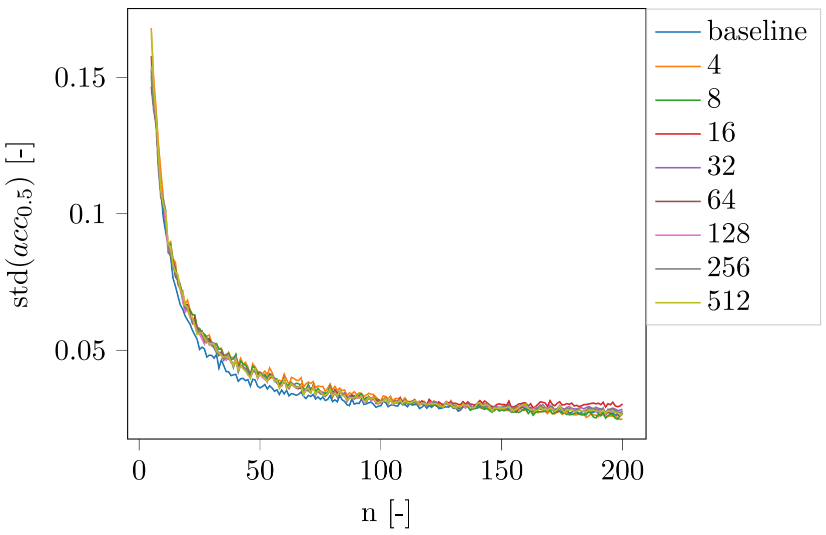


Figure 6.17: Standard deviation of $acc_{0.5}$ when not using the LSTM architecture

6.4.7 Best result

Based on the experiments, we define the values of hyperparameters, which should provide the best accuracy and generalization. We train the following neural network: width of the LSTM state vector (w) 32 for each direction (bidirectional LSTM is used), number of LSTM layers (d) 2, size of the training dataset (train size) 250 000 instances, multi-head attention with four heads (h), learning rate (lr) of 0.0005, feature extraction as described in the previous chapter, no convolutional embedding, only a linear layer and a simplified final classifier, consisting only of one linear layer and a sigmoid function. The results of this architecture are shown in Fig. 6.18.

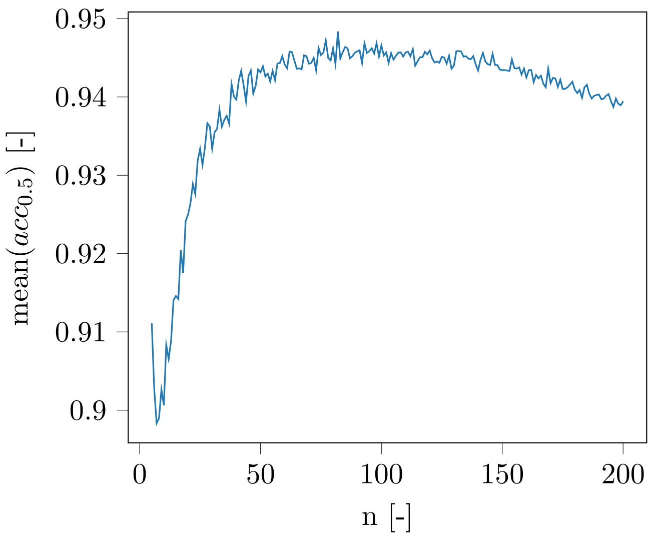


Figure 6.18: Mean of $acc_{0.5}$ for the architecture with hyperparameters selected based on the optimization process

The accuracy of this model on instances of sizes up to 100 is on par or even better compared to all of the previously trained architectures. Furthermore, this model also produces the highest average accuracy on instances of size 200. We have confirmed that using the values of hyperparameters that performed well on their own produces a very well-performing model.

However, all of the results lead to one question in particular, how is it possible that we are not obtaining better accuracies? If the accuracy reported on the training datasets was very close to 100% and the performance on the test dataset was much worse, we would think that our model is overfitting. Nevertheless, in our case, the model cannot get close to 100% accuracy on the training dataset even with many training epochs.

One possible explanation is the influence of multiple optimal solutions in some instances. The model outputs a prediction, which either corresponds to an optimal solution or leads to a solution with an excellent criterion value but in our generated instance, which was solved to optimality, we arrived at a different optimal solution. This is partially answered in Section 6.7, where we attempt to construct a simple heuristic using the neural network's output.

6.5 Prediction runtime

So far, we have only studied the performance of our proposed machine learning model. To prove its practicality, we must also be able to produce the predictions in a reasonable time frame. In this experiment, we have again taken close to 200 000 instances and measured the time it takes to output the prediction. This time contains both the normalization of the input instance into a vector form and the time it takes to compute the output of the neural network. In Fig. 6.19, we show the average time it takes to produce the prediction per input size.

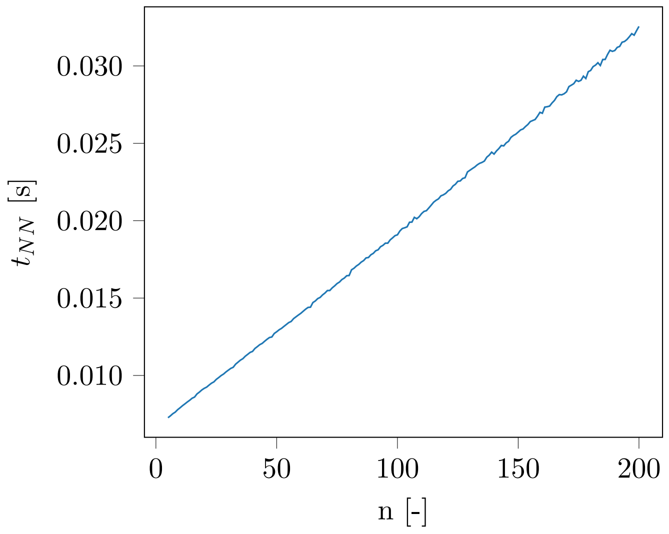


Figure 6.19: Prediction time of our best performing model

These results were produced using two cores of an Intel Xeon E5-2690 v4 CPU with 8GB of memory. Unsurprisingly, the prediction time increases linearly with respect to the input size. Furthermore, the whole prediction process is nearly instantaneous. When using the equivalent of an average home PC computing power, the prediction is returned within 35 ms, even for the largest instances, which is more than quick enough for practical purposes.

6.6 Exact solution of $1 \mid r_j \mid \sum U_j$

Now that we have a nicely performing machine learning model, which can predict which jobs in an instance of the $1|r_j|\sum U_j$ problem will be tardy and which on time, we evaluate the improvement it brings to the deterministic algorithm.

This is where we need to report a small disappointment. We have not been able to fully reimplement the algorithm of Baptiste et al. [14] on which we wanted to base our solution. We managed to implement the key parts of the algorithm, the branching and deciding whether there exists a feasible schedule for a given set of jobs, which means that our algorithm always correctly returns an optimal solution. However, the amount of research the original algorithm is based on is vast.

We were able to implement a lower bound and certain properties, which reduce the state space size correctly, but afterwards, we concluded that the amount of time further attempts at reimplementation would require is beyond the time frame of this thesis. However, this in no way hampers the possible integration of the ML model into the algorithm. On the contrary, since we have a weaker and worse performing deterministic algorithm, there is more room for improvement for the ML model to bring.

We have taken 10 000 instances, divided evenly between sizes 10, 20, ..., up to 100 and evaluated on all of these our reimplementation of the deterministic algorithm with and without the integration of our ML model. We again used two cores of an Intel Xeon E5-2690 v4 CPU with 8GB of memory. All the experiments were performed with a timeout of one hour. We report how many instances timed out, the average CPU time, the number of nodes that were considered during the algorithm's run, and the number of fully explored nodes that are not cut away by the lower bound. Results for the baseline algorithm are shown in Fig. 6.20, whereas the results of the algorithm with the ML part incorporated are shown in Fig. 6.21.

n	10	20	30	40	50	60	70	80	90	100
timed out [%]	0	0	0	0	0	0.9	4.4	3.3	7.9	10.6
avg CPU [s]	0.002	0.01	0.21	1.17	12.3	22.3	100	260	229	177
avg nodes [-]	20.4	48.6	313	1k	10k	22k	34k	80k	64k	31k
avg nodes expl. [-]	13.4	35	235	791	8k	13k	28k	62k	50k	24k

Figure 6.20: Baseline results before the integration

n	10	20	30	40	50	60	70	80	90	100
timed out [%]	0	0	0	0	0.9	1.7	4.4	11.6	16.5	22.0
avg CPU [s]	0.01	0.03	0.22	1.29	7.57	44.4	139	317	329	237
avg nodes [-]	17.2	53.0	333	1k	7k	31k	76k	168k	110k	64k
avg nodes expl. [-]	10.1	33.7	218	1k	5k	21k	51k	115k	74k	43k

Figure 6.21: Results after the integration

Unfortunately, these results are very underwhelming. We started the work on this thesis believing that the number of nodes of the state space search can be decreased using the information provided by the neural network's prediction. However, it seems that it is not the case. One more important fact is that Baptiste et al. [14] report a drastically lower number of nodes, 80 on average for 100 job instances. Their algorithm prunes the state space with extreme efficiency. We believe that even if we improved the performance of the algorithm we implemented significantly, it would be questionable whether it would even show when incorporated into the entire original algorithm since the room for improvement is minimal.

6.7 Heuristic results

Given that we have not managed to improve the deterministic algorithm, we decided to make the most of what we have. In this final set of experiments, we propose a simple heuristic based on the ML model's prediction and an algorithm to determine whether a schedule exists in which all the given jobs are on time.

In Section 4.3, we describe the branching first proposed by Carlier. This algorithm is able to branch on the schedule made by the simple EDD rule to decide whether all the jobs can be scheduled on time. We can now define a simple heuristic by taking all the jobs in the instance and running this algorithm. If the jobs cannot all be scheduled on time, we remove the one the neural network is the most likely to be late and then repeat this process.

Furthermore, we introduce two variables that allow us to fine-tune the heuristic. When we remove a job, the instance changes, it might be beneficial to recalculate the predictions for this smaller instance, we do not have to do this after removing each job, instead, we define a parameter rerun (re), which defines after how many repetitions we should recalculate the output of the network. Rerun of zero means that we only calculate the predictions once in the very beginning of the run of the whole algorithm. Secondly, we introduce a threshold (th), above which we consider all the jobs late[1], which allows us to remove a portion of the jobs right in the beginning, which further speeds up the algorithm.

To evaluate how quickly we can remove one job from the instance, that is, rerun the neural network if necessary and decide whether all jobs from a given set can be scheduled on time, we run our proposed heuristic with different values of the rerun and threshold (th) parameters. In Fig. 6.22, we report the average time it takes to decide a given set of jobs, per instance size.

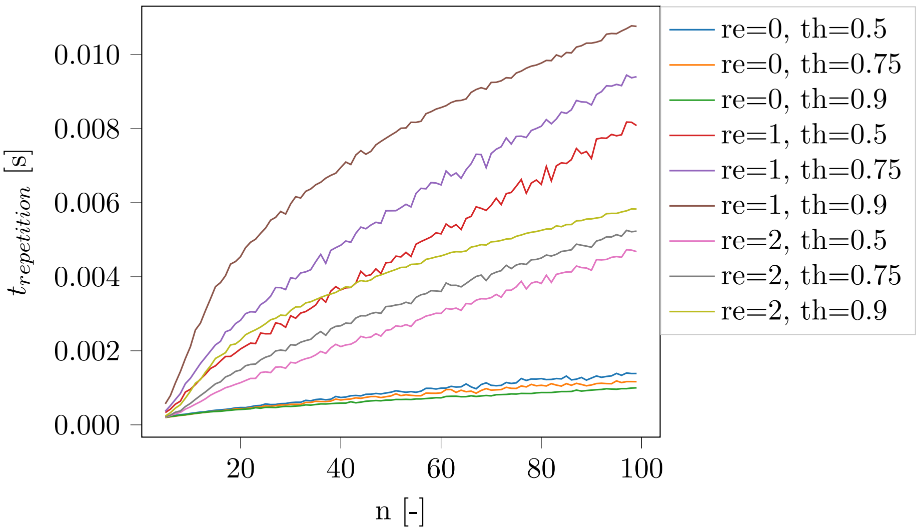


Figure 6.22: Average time of one repetition of the heuristic algorithm

Firstly, if we do not recalculate the neural network's prediction, we can decide on a given set of jobs very quickly, in just under a millisecond for 100 jobs, even when setting the late threshold very leniently. When rerunning the neural network, the time needed for the removal of one job rises significantly, all the way up to 10 milliseconds for 100 jobs, when the neural network is rerun in each iteration and the threshold is set to 0.9. As expected, rerunning the network more often leads to slower times, as does a higher

[1]We would like to quickly remind the reader that we use the one-hot encoding of tardy jobs as our prediction normalization, output close to 1 means the model believes the job is tardy.

threshold, which makes the algorithm start with more jobs. Of these two, the effect of running the model more often is more substantial.

Let us now evaluate the performance of this proposed heuristic using a measure known as the *optimality gap*. Given that our heuristic obtains a result with the criterion c and the optimal criterion is c^*, since we are minimizing, we define the gap as:

$$\textit{optimality gap} = \frac{c - c^*}{c^*} \tag{6.1}$$

In Fig. 6.23, we present the average optimality gap per instance size for nine different parameter configurations.

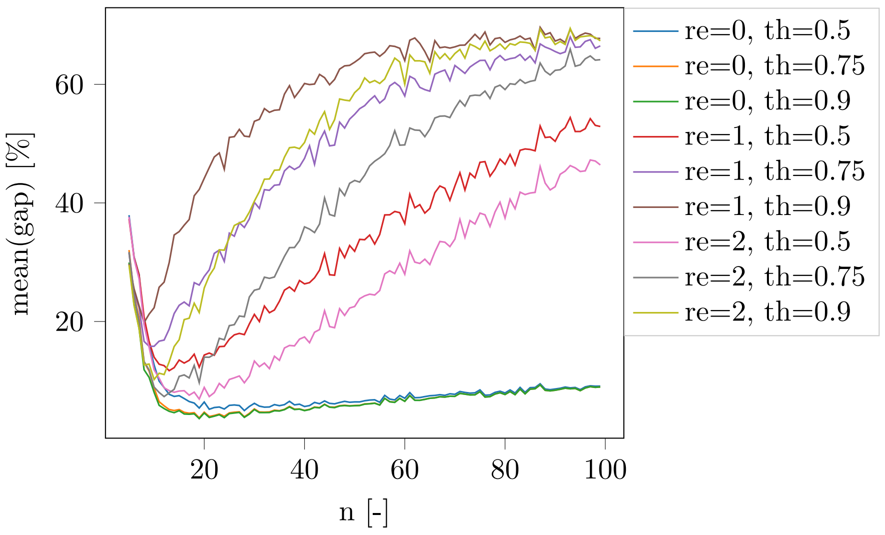


Figure 6.23: Optimality gap of the algorithm with different parameters

These results clearly show that the idea of running the model more than once does not translate well into practice. Apparently, recalculating the predictions without the already removed jobs leads to very different predictions than when the whole input instance is considered. Since rerunning the model takes more time and produces worse results, we will not consider this parameter in future experiments and instead only calculate the predictions once in the beginning.

One more idea we can use to improve the performance of our heuristic is the so-called *limited discrepancy search* (LDS) [60]. The idea behind this method is not complicated and can even be inferred from the name. We introduce a new parameter called discrepancy. Say we start the heuristic with a discrepancy of five and decide that all the jobs below the threshold cannot be scheduled on time. We need to remove the job the neural network considers the most likely to be late. However, with LDS, we might also decide to keep this job and instead remove the second most likely to be late but only continue our search with discrepancy 4. Alternatively, we might keep both these jobs, remove the third one and carry on the search with discrepancy 3. Once we search with discrepancy 0, we always have to remove the job that is most likely late. LDS creates a simple branching which can explore many more combinations of jobs, which should lead to better results. Of course, the price we pay is that more repetitions of our heuristic step need to be run.

In Fig. 6.24 and Fig. 6.25, we present the mean and standard deviation, respectively, of the optimality gap per input size for nine different parameter variations of our heuristic algorithm.

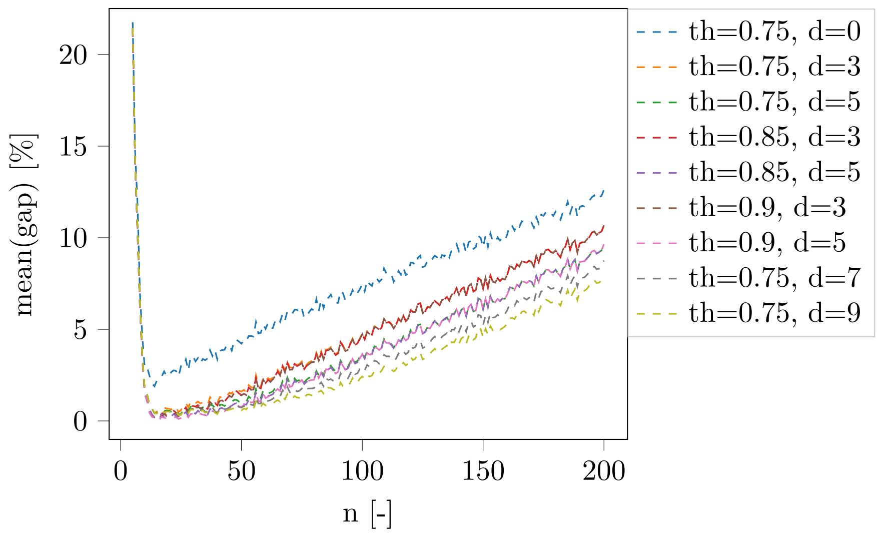


Figure 6.24: Gap of our heuristic

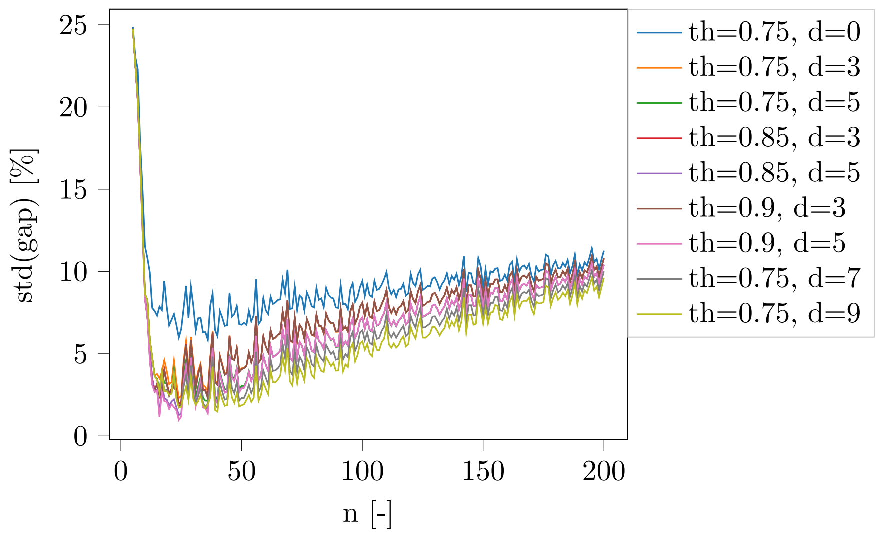


Figure 6.25: std of Gap of our heuristic

The result shows that employing LDS significantly improves the optimality gap. Without LDS, the best performing algorithm already reached a gap of 9% on instances of size 100, whereas, with it, we obtain a 7% gap for instances of size 200. We can also observe that with the use of LDS, there is no need to set excessive initial thresholds. When the model is confident that a job will be tardy, i. e. its prediction is above 0.75, it is most likely true. One result that is not so satisfying is the standard deviation, which starts to get fairly large for the largest instances with the same pace for all the combinations of the parameters.

In order to fully assess the performance of our algorithm, we would need to compare the results against the state of the art from the literature. We have found only one sophisticated heuristic for our problem. It is proposed in the article [15] mentioned during the literature review. Their results show that their heuristic always finds the optimal solution for a vast majority of the data they generate. However, they use a different method to generate their instances. They then recalculate their optimal approach on instances generated based on the article [51], which we use, but the heuristic results are not reported in this way. Let us at least present the results of our algorithm in tables

Fig. 6.26, Fig. 6.27 and Fig. 6.29, with concrete values of the average gap, standard deviation of the gap and runtime per intervals of 25 different sizes.

n [x-24, x]	25	50	75	100	125	150	175	200
d=0 [%]	4.0	3.7	5.3	6.6	8.0	9.3	10.5	11.7
d=3 [%]	2.7	1.2	2.5	3.8	5.4	6.9	8.3	9.7
d=5 [%]	2.7	0.8	1.7	2.9	4.3	5.8	7.2	8.7
d=7 [%]	2.6	0.7	1.3	2.2	3.6	4.9	6.3	7.8
d=9 [%]	2.6	0.6	1.0	1.8	3.0	4.2	5.6	7.0

Figure 6.26: Mean of the gap of the heuristic algorithm (th=0.75)

n [x-24, x]	25	50	75	100	125	150	175	200
d=0 [%]	9.2	7.3	8.0	8.4	9.1	9.5	10.0	10.5
d=3 [%]	6.7	4.3	5.9	6.9	7.9	8.7	9.4	10.0
d=5 [%]	6.4	3.4	4.8	6.0	7.2	8.1	8.8	9.6
d=7 [%]	6.4	2.7	3.9	5.2	6.4	7.4	8.3	9.2
d=9 [%]	6.3	2.3	3.3	4.5	5.8	6.8	7.8	8.7

Figure 6.27: Average standard deviation of the gap of the heuristic algorithm (th=0.75)

We can compare at least one thing with the article [15], and that is the runtime. Their heuristic runs in $\mathcal{O}(n^2)$, which is very fast, but the algorithm relies on specific coefficients, which have to be calculated by a numerical optimization method, which they experimentally estimate takes around $0.000016n^3$, or around 1.5 minutes for the 175 jobs instances. The average runtime of our heuristic per instance size is reported in Fig. 6.28.

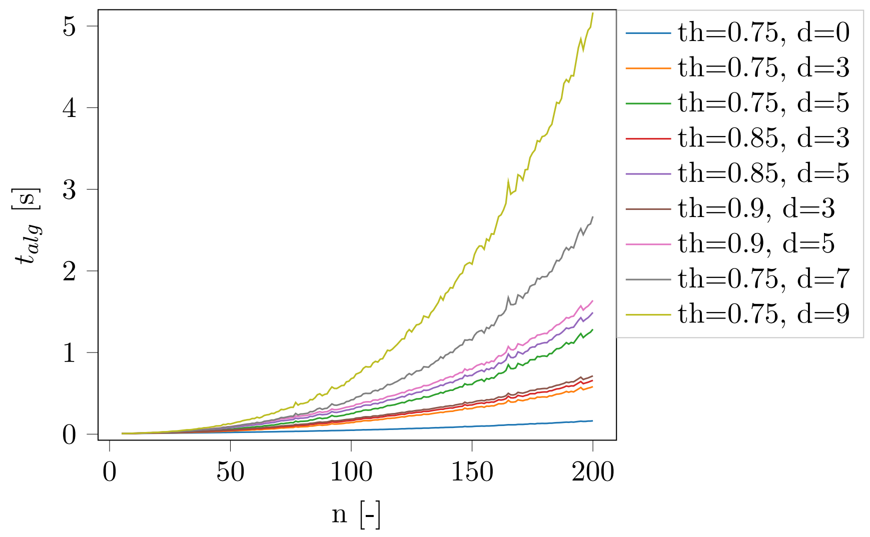


Figure 6.28: Runtime of our final heuristic

n [x-24, x]	25	50	75	100	125	150	175	200
d=0 [s]	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.1
d=3 [s]	0.0	0.0	0.1	0.1	0.2	0.3	0.4	0.5
d=5 [s]	0.0	0.0	0.1	0.2	0.3	0.5	0.8	1.2
d=7 [s]	0.0	0.1	0.2	0.3	0.6	1.0	1.5	2.2
d=9 [s]	0.0	0.1	0.2	0.5	1.0	1.7	2.8	4.2

Figure 6.29: Average runtime of the heuristic algorithm (th=0.75)

We can see that the fastest of the better performing discrepancy 5 algorithms takes just slightly over one second on average to return a result for the largest instances of size 200, which is a good result that could be used in practice where the domain changes frequently and quick although not necessary optimal recalculation is needed. The even better performing discrepancy 9 algorithm finishes on average in 4.2 seconds, which is still excellent. Overall, we have developed a heuristic algorithm which very quickly returns solutions that are fairly close to the optimum. Furthermore, its by setting the discrepancy parameter, potential users might choose between speed and expected optimality gap.

Chapter 7
Conclusion

Throughout this thesis, we have investigated how machine learning can improve combinatorial optimization algorithms. Specifically, we sought to improve an existing scheduling algorithm for the $1 \mid r_j \mid \sum U_j$ problem by using an ML model which predicts information relevant to the state space search. We have researched the relevant literature and proposed an ML model that can be integrated into a scheduling algorithm we partially reimplemented. Afterwards, we thoroughly evaluated our solution, and finally, we proposed a simple heuristic and again evaluated it experimentally.

During the literature review, we identified a state-of-the-art scheduling algorithm which solves our problem and has a place for potential improvement. The review of ML in CO has shown us that the best model architecture for predicting information about the instances of our problem is a recurrent neural network, which handles variable size input and has solid expressive power.

Afterwards, we analyzed and described the scheduling algorithm in detail. Showing its building blocks and how they tie together. We described an algorithm that can decide, given a set of jobs, whether they can all be scheduled on time and examined a lower bound based on a relaxation of the problem, which helps us to prevent the exploration of unnecessary nodes. However, we managed only partially to re-implement it. Our implementation branches as the authors describe and returns an optimal solution, but it cannot prune the state space as efficiently.

Subsequently, we proposed a machine learning model which aims to predict which job will be tardy given an instance of our problem. We begin by describing the dataset generation, feature extraction and input normalization. Afterwards, we propose the model architecture based on an LSTM recurrent neural network and then discuss the training.

Finally, we thoroughly evaluated all the aspects of the solution we have described. We start by showing that our proposed model can achieve a reasonable accuracy and then proceed to hyperparameter optimization, which leads us to a model with good accuracy that can generalize well up to the instances twice the size it was trained on. Unfortunately, the evaluation of the integration of the ML model into the algorithm did not produce the expected results. It turns out that using ML to enhance this specific algorithm might not be the best idea.

This result has made us think about other possible applications of the fairly well-performing model we have trained, and we came up with the idea to create a heuristic algorithm. After some experimentation, we develop a fast algorithm with a pretty good optimality gap even on large instances. We obtain an optimality gap of 1.8% with an average runtime of 0.5 seconds for instances of size 100 and an optimality gap of 7% with an average runtime of 4.2 seconds for instance of size 200. Furthermore, we introduce a parameter called discrepancy, which allows to shift focus between very fast runtime or better optimality gap.

References

[1] Michael L. Pinedo. *Scheduling*. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-26580-3. URL: https://doi.org/10.1007/978-3-319-26580-3.

[2] R.L. Graham et al. “Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey”. In: *Discrete Optimization II, Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*. Elsevier, 1979, pp. 287–326. DOI: 10.1016/s0167-5060(08)70356-x. URL: https://doi.org/10.1016/s0167-5060(08)70356-x.

[3] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. “Complexity of machine scheduling problems”. In: *Annals of discrete mathematics*. Vol. 1. Elsevier, 1977, pp. 343–362.

[4] Muminu O. Adamu, Aderemi O. “A survey of single machine scheduling to minimize weighted number of tardy jobs”. In: *Journal of Industrial & Management Optimization* 10.1 (2014), pp. 219–241. DOI: 10.3934/jimo.2014.10.219. URL: https://doi.org/10.3934/jimo.2014.10.219.

[5] J Michael Moore. “An n job, one machine sequencing algorithm for minimizing the number of late jobs”. In: *Management science* 15.1 (1968), pp. 102–109.

[6] Eugene L Lawler, LAWLER EL. “Sequencing to minimize the weighted number of tardy jobs.” In: (1976).

[7] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. Springer US, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[8] Hiroshi Kise, Toshihide Ibaraki, and Hisashi Mine. “A solvable case of the one-machine scheduling problem with ready and due times”. In: *Operations Research* 26.1 (1978), pp. 121–126.

[9] E.L. Lawler. “Knapsack-like scheduling problems, the Moore-Hodgson algorithm and the ‘tower of sets’ property”. In: *Mathematical and Computer Modelling* 20.2 (July 1994), pp. 91–106. DOI: 10.1016/0895-7177(94)90209-7. URL: https://doi.org/10.1016/0895-7177(94)90209-7.

[10] P. Baptiste et al. “Sequencing a single machine with due dates and deadlines: an ILP-based approach to solve very large instances”. In: *Journal of Scheduling* 13.1 (Nov. 2008), pp. 39–47. DOI: 10.1007/s10951-008-0092-6. URL: https://doi.org/10.1007/s10951-008-0092-6.

[11] Lukáš Hejl. “Single machine scheduling minimizing the weighted number of tardy jobs assuming strongly correlated instances”. Czech Technical University in Prague, 2020. URL: http://hdl.handle.net/10467/86208.

[12] Laurent Péridy, Éric Pinson, and David Rivreau. “Using short-term memory to minimize the weighted number of late jobs on a single machine”. In: *European Journal of Operational Research* 148.3 (Aug. 2003), pp. 591–603. DOI: 10.1016/s0377-2217(02)00438-1. URL: https://doi.org/10.1016/s0377-2217(02)00438-1.

[13] Stéphane Dauzère-Pérès, Marc Sevaux. “An Exact Method to Minimize the Number of Tardy Jobs in Single Machine Scheduling”. In: *Journal of Scheduling* 7.6 (Nov. 2004), pp. 405–420. DOI: 10.1023/b:josh.0000046073.05827.15. URL: https://doi.org/10.1023/b:josh.0000046073.05827.15.

[14] Philippe Baptiste, Laurent Peridy, and Eric Pinson. “A branch and bound to minimize the number of late jobs on a single machine with release time constraints”. In: *European Journal of Operational Research* 144.1 (Jan. 2003), pp. 1–11. DOI: 10.1016/s0377-2217(01)00353-8. URL: https://doi.org/10.1016/s0377-2217(01)00353-8.

[15] Rym M’Hallah, R.L. Bulfin. “Minimizing the weighted number of tardy jobs on a single machine with release dates”. In: *European Journal of Operational Research* 176.2 (Jan. 2007), pp. 727–744. DOI: 10.1016/j.ejor.2005.08.013. URL: https://doi.org/10.1016/j.ejor.2005.08.013.

[16] Radin R. L. Parker R. G. *Discrete Optimization.* en. Computer Science and Scientific Computing. San Diego, CA: Academic Press, Aug. 1988.

[17] Stéphane Dauzère-Pérès, Marc Sevaux. “Using Lagrangean relaxation to minimize the weighted number of late jobs on a single machine”. In: *Naval Research Logistics (NRL)* 50.3 (Dec. 2002), pp. 273–288. DOI: 10.1002/nav.10056. URL: https://doi.org/10.1002/nav.10056.

[18] Ruslan Sadykov. “A branch-and-check algorithm for minimizing the weighted number of late jobs on a single machine with release dates”. In: *European Journal of Operational Research* 189.3 (Sept. 2008), pp. 1284–1304. DOI: 10.1016/j.ejor.2006.06.078. URL: https://doi.org/10.1016/j.ejor.2006.06.078.

[19] Cyril Briand, Samia Ourari. “Minimizing the number of tardy jobs for the single machine scheduling problem: MIP-based lower and upper bounds”. In: *RAIRO - Operations Research* 47.1 (Jan. 2013), pp. 33–46. DOI: 10.1051/ro/2013025. URL: https://doi.org/10.1051/ro/2013025.

[20] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. “Machine learning for combinatorial optimization: A methodological tour d’horizon”. In: *European Journal of Operational Research* 290.2 (Apr. 2021), pp. 405–421. DOI: 10.1016/j.ejor.2020.07.063. URL: https://doi.org/10.1016/j.ejor.2020.07.063.

[21] J. J. Hopfield, D. W. Tank. ““Neural” computation of decisions in optimization problems”. In: *Biological Cybernetics* 52.3 (July 1985), pp. 141–152. DOI: 10.1007/bf00339943. URL: https://doi.org/10.1007/bf00339943.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning Internal Representations by Error Propagation”. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations.* Cambridge, MA, USA: MIT Press, 1986, pp. 318–362. ISBN: 026268053X.

[23] Kate A. Smith. “Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research”. In: *INFORMS Journal on Computing* 11.1 (Feb. 1999), pp. 15–34. DOI: 10.1287/ijoc.11.1.15. URL: https://doi.org/10.1287/ijoc.11.1.15.

[24] Sepp Hochreiter, Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[25] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems.* Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf.

[26] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate.* 2014. eprint: arXiv:1409.0473.

[27] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer Networks". In: *Advances in Neural Information Processing Systems.* Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf.

[28] Irwan Bello et al. *Neural Combinatorial Optimization with Reinforcement Learning.* 2016. DOI: 10.48550/ARXIV.1611.09940. URL: https://arxiv.org/abs/1611.09940.

[29] Wouter Kool, Herke van Hoof, and Max Welling. *Attention Solves Your TSP, Approximately.* 2018. eprint: arXiv:1803.08475v2. URL: https://arxiv.org/abs/1803.08475v2.

[30] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[31] Xavier Bresson, Thomas Laurent. "The Transformer Network for the Traveling Salesman Problem". In: *CoRR* abs/2103.03012 (2021). arXiv: 2103.03012. URL: https://arxiv.org/abs/2103.03012.

[32] Lars Kotthoff. "Algorithm Selection for Combinatorial Search Problems: A Survey". In: *Data Mining and Constraint Programming.* Springer International Publishing, 2016, pp. 149–190. DOI: 10.1007/978-3-319-50137-6_7. URL: https://doi.org/10.1007/978-3-319-50137-6_7.

[33] George B. Dantzig, Philip Wolfe. "Decomposition Principle for Linear Programs". In: *Operations Research* 8.1 (Feb. 1960), pp. 101–111. DOI: 10.1287/opre.8.1.101. URL: https://doi.org/10.1287/opre.8.1.101.

[34] Markus Kruber, Marco E. Lübbecke, and Axel Parmentier. "Learning When to Use a Decomposition". In: *Integration of AI and OR Techniques in Constraint Programming.* Springer International Publishing, 2017, pp. 202–210. DOI: 10.1007/978-3-319-59776-8_16. URL: https://doi.org/10.1007/978-3-319-59776-8_16.

[35] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. "Learning a Classification of Mixed-Integer Quadratic Programming Problems". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research.* Springer International Publishing, 2018, pp. 595–604. DOI: 10.1007/978-3-319-93031-2_43. URL: https://doi.org/10.1007/978-3-319-93031-2_43.

[36] Alessio Guerri, Michela Milano. "Learning techniques for Automatic Algorithm Portfolio Selection". In: *ECAI 2004: 16th European Conference on Artificial Intelligence.* IOS Press, 2004, pp. 475–479. URL: http://www.lia.deis.unibo.it/Staff/AlessioGuerri/homepage_files/Publications/ECAI2004.pdf.

[37] Andrea Lodi, Giulia Zarpellon. "On learning and branching: a survey". In: *TOP* 25.2 (June 2017), pp. 207–236. DOI: 10.1007/s11750-017-0451-6. URL: https://doi.org/10.1007/s11750-017-0451-6.

[38] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. “A Machine Learning-Based Approximation of Strong Branching”. In: *INFORMS Journal on Computing* 29.1 (Jan. 2017), pp. 185–195. DOI: 10.1287/ijoc.2016.0723. URL: https://doi.org/10.1287/ijoc.2016.0723.

[39] Maxime Gasse et al. “Exact Combinatorial Optimization with Graph Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/d14c2267d848abeb81fd590f371d39bd-Paper.pdf.

[40] Prateek Gupta et al. “Hybrid Models for Learning to Branch”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 18087–18097. URL: https://proceedings.neurips.cc/paper/2020/file/d1e946f4e67db4b362ad23818a6fb78a-Paper.pdf.

[41] Vinod Nair et al. *Solving Mixed Integer Programs Using Neural Networks*. 2020. eprint: arXiv:2012.13349.

[42] Ambros Gleixner et al. “MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library”. In: *Mathematical Programming Computation* (2021). DOI: 10.1007/s12532-020-00194-3. URL: https://doi.org/10.1007/s12532-020-00194-3.

[43] Daniel Selsam, Nikolaj Bjørner. “NeuroCore: Guiding High-Performance SAT Solvers with Unsat-Core Predictions”. In: *CoRR* abs/1903.04671 (2019). arXiv: 1903.04671. URL: http://arxiv.org/abs/1903.04671.

[44] Cynthia Barnhart et al. “Branch-and-Price: Column Generation for Solving Huge Integer Programs”. In: *Operations Research* 46.3 (June 1998), pp. 316–329. DOI: 10.1287/opre.46.3.316. URL: https://doi.org/10.1287/opre.46.3.316.

[45] Roman Václavík et al. “Accelerating the Branch-and-Price Algorithm Using Machine Learning”. In: *European Journal of Operational Research* 271.3 (2018), pp. 1055–1069. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2018.05.046. URL: https://www.sciencedirect.com/science/article/pii/S0377221718304570.

[46] H. Aytug et al. “A review of machine learning in scheduling”. In: *IEEE Transactions on Engineering Management* 41.2 (May 1994), pp. 165–171. DOI: 10.1109/17.293383. URL: https://doi.org/10.1109/17.293383.

[47] Michal Bouška et al. “A scheduling algorithm driven by machine learning for a single machine problem minimizing the total tardiness”.

[48] Jacques Carlier. “The one-machine sequencing problem”. In: *European Journal of Operational Research* 11.1 (Sept. 1982), pp. 42–47. DOI: 10.1016/s0377-2217(82)80007-6. URL: https://doi.org/10.1016/s0377-2217(82)80007-6.

[49] Jan Ellfers. “Scheduling with release times and deadlines”. TU Delft, 2014. URL: https://repository.tudelft.nl/islandora/object/uuid:5def2dbb-67d1-4672-a0b1-561d7dc1a74f/datastream/OBJ/download.

[50] J. Carlier, E. Pinson. “Adjustment of heads and tails for the job-shop problem”. In: *European Journal of Operational Research* 78.2 (Oct. 1994), pp. 146–161. DOI: 10.1016/0377-2217(94)90379-4. URL: https://doi.org/10.1016/0377-2217(94)90379-4.

[51] Philippe Baptiste, Claude Le Pape, and Laurent Peridy. “Global Constraints for Partial CSPs: A Case-Study of Resource and Due Date Constraints”. In: *Principles and Practice of Constraint Programming — CP98*. Springer Berlin Heidelberg, 1998, pp. 87–101. DOI: `10.1007/3-540-49481-2_8`. URL: `https://doi.org/10.1007/3-540-49481-2_8`.

[52] Christopher Olah. *Understanding LSTM Networks*. colah’s blog. Aug. 2015. URL: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[53] Serkan Kiranyaz et al. “1D convolutional neural networks and applications: A survey”. In: *Mechanical Systems and Signal Processing* 151 (2021), p. 107398. ISSN: 0888-3270. DOI: `https://doi.org/10.1016/j.ymssp.2020.107398`. URL: `https://www.sciencedirect.com/science/article/pii/S0888327020307846`.

[54] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: (2014). eprint: `arXiv:1406.1078`.

[55] Klaus Greff et al. “LSTM: A Search Space Odyssey”. In: (2015). DOI: `10.1109/TNNLS.2016.2582924`. eprint: `arXiv:1503.04069`.

[56] Zhiheng Huang, Wei Xu, and Kai Yu. *Bidirectional LSTM-CRF Models for Sequence Tagging*. 2015. DOI: `10.48550/ARXIV.1508.01991`. URL: `https://arxiv.org/abs/1508.01991`.

[57] Yoav Goldberg. *A Primer on Neural Network Models for Natural Language Processing*. 2015. DOI: `10.48550/ARXIV.1510.00726`. URL: `https://arxiv.org/abs/1510.00726`.

[58] Abdul Mueed Hafiz, Shabir Ahmad Parah, and Rouf Ul Alam Bhat. *Attention mechanisms and deep learning for machine vision: A survey of the state of the art*. 2021. DOI: `10.48550/ARXIV.2106.07550`. URL: `https://arxiv.org/abs/2106.07550`.

[59] Diederik P. Kingma, Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: `10.48550/ARXIV.1412.6980`. URL: `https://arxiv.org/abs/1412.6980`.

[60] William D Harvey, Matthew L Ginsberg. “Limited discrepancy search”. In: *IJCAI (1)*. 1995, pp. 607–615. URL: `http://cse.unl.edu/~choueiry/Documents/LDS.pdf`.

Appendix A

Attachments structure

In this chapter we go over the files uploaded as attachments of this thesis. The purpose of these attachments is mainly to show what we have created and that others may study them and possibly work on them.

```
thesis_attachments
├── meta
│   │ Supporting classes for the implementation of the Baptiste et al. [14] algorithm.
│   ├── chooser.py
│   ├── classes.py
│   ├── lower_bound.py
│   ├── meta.py
│   ├── oracle.py
│   └── propositions.py
├── estimators
│   ├── decomposition_estimator_meta.py
│   │   Our implementation of the Baptiste et al. [14] algorithm
│   └── early_tardy_estimator.py
│       Our proposed heuristic algorithm
├── ml
│   └── dp
│       ├── config.json
│       │   An example configuration used for the neural network training
│       ├── trained
│       │   └── best.pth
│       │       A model trained with optimized hyperparameters
│       └── model
│           ├── loss.py
│           │   Definitions of loss functions
│           ├── metric.py
│           │   Definitions of collected metrics
│           ├── model.py
│           │   Definitions of the model architecture
│           └── pytorch_modules.py
│               Classes used in the model definition
└── requirements.txt
    Requirements for Python libraries
```