Fast, Right, or Best?

Algorithms for Practical Optimization Problems

Willem Feijen

Fast, Right, or Best?

Algorithms for Practical Optimization Problems

ILLC Dissertation Series DS-202X-NN

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Science Park 107
1098 XG Amsterdam
phone: +31-20-525 6051
e-mail: illc@uva.nl
homepage: http://www.illc.uva.nl/

This research has been carried out in the Networks and Optimization group at the Centrum Wiskunde & Informatica in Amsterdam and was co-funded by DELMIA, Dassault Systèmes in 's-Hertogenbosch.

Cover design by Bert Jones.
Printed and bound by your printer.

ISBN: 90–XXXX–XXX–X

Fast, Right, or Best?
Algorithms for Practical Optimization Problems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. P.P.C.C. Verbeek
ten overstaan van een door het College voor Promoties ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op maandag 9 december 2024, te 17.00 uur

door Willem Feijen
geboren te Bemmel

Promotiecommissie

Promotor:	prof. dr. G. Schäfer	Universiteit van Amsterdam
Copromotor:	prof. dr. R.D. van der Mei	Vrije Universiteit Amsterdam
Overige leden:	prof. dr. U. Endriss	Universiteit van Amsterdam
	prof. dr. J.A.S. Gromicho	Universiteit van Amsterdam
	prof. dr. S.I. Birbil	Universiteit van Amsterdam
	dr. ir. S.H.M. van Zwam	Dassault Systèmes
	dr. ir. R.A. Sitters	Vrije Universiteit Amsterdam
	prof. dr. C. Büsing	RWTH Aachen University

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Contents

Chapter 1

Introduction

Algorithms are everywhere. Imagine the day of a millennial in Amsterdam. The shortest route is shown on their phone, the 'recommended for you' music list plays on their headphones while they head off to an appointment with the bank for their mortgage. Without realizing it, they have come across three different algorithms already in the early morning, and they are no exception. People's lives are intertwined with algorithms nowadays, with some algorithms more prominently present than others. Moreover, the word 'algorithm' has also become a part of people's lives, almost anyone has heard of it. We think the book *Introduction to Algorithms* by Cormen et al. [2022], often bestowed the title 'algorithm bible', rightfully claims that the word algorithm appears somewhere in the news seemingly every day. By the Oxford English Dictionary [2024], an algorithm is defined as *a set of rules that must be followed when solving a particular problem*, which allows a great length of flexibility. This thesis showcases this flexibility of algorithms: we use a variation of techniques to address several problems, ranging from theoretical problems to practical real-world planning puzzles.

The algorithms that we consider in this thesis are designed to solve combinatorial optimization problems. With the example of determining the shortest route to the appointment at the bank in mind, let us digest what it actually means to find a solution to an optimization problem. A *solution* represents some plan or set of decisions. In the example, a solution is defined as one of the possible routes from our location to the appointment. We say that a solution is *feasible* if it satisfies the conditions for being a valid solution to the optimization problem. We need for example that the route starts at our location and ends at the location of the appointment, otherwise the solution is useless. To decide if one solution is better than another, we introduce an objective. An *objective function* quantifies the quality of each solution. It can be regarded as a cost function, which indicates the quality or cost of each solution in a single number, the *objective value*. In the routing example, the objective value of a solution can be defined as the length of the route. An *instance* of an optimization problem is defined by a set of feasible

solutions and an objective function. The goal of an optimization problem is to find a feasible solution such that there exists no other feasible solution that has a better objective value. We call this solution an *optimal solution*. To accomplish this goal, we need algorithms. An *algorithm* takes an instance of an optimization problem as input and returns a solution. In the example, an algorithm would take the road network as input and would output a route that leads you from your location to the appointment.

Now, we question ourselves what defines a good algorithm. This thesis explores three possible criteria for answering that question. The first criterion checks whether an algorithm finds the *right* solutions, i.e., if it always returns a feasible solution to the optimization problem. In most cases, we would only accept an algorithm that finds feasible solutions. However, sometimes, especially when it is hard to find feasible solutions, it might also be interesting to consider slightly infeasible solutions.

The second scale along which we can measure how well an algorithm works is the quality of the computed solutions, that is, does an algorithm return a *good*, or even the *best*, solution? Some algorithms are guaranteed to find the optimal solution. In the shortest path example, that means the algorithm would return the shortest route, guaranteeing that the distance of any other route to the appointment cannot be shorter. In practice, a solution with an objective value close to the optimal solution's objective is often satisfactory, and usually easier to compute. Therefore, there also exist algorithms with weaker guarantees, like algorithms for which any returned solution has an objective value that is only a bounded factor worse than the optimal. In the routing example, this would mean that the algorithm returns a route together with a guarantee that the optimal route is, for example, at most 5% shorter than the returned route. Algorithms with guarantees like that are called *approximation algorithms*. There are also algorithms that do not offer any guarantee on the solution value, called *heuristics*. Heuristics aim to find satisfactory solutions, often by simplifying the problem, making educated guesses, or following rules of thumb. For example, if you know your destination in the routing problem is north of you, a heuristic solution can be computed by taking the road that points in that direction at every crossroad. Even though this could lead to a sensible solution, there is no guarantee about the length of this route compared to the optimal one. Oftentimes, heuristics run faster than algorithms with performance guarantees, since it is not required to prove any optimality guarantee.

That leads us to the third way of assessing an algorithm: time. That is, can an algorithm find a solution *fast*? The time that an algorithm runs can be measured in multiple ways. A straightforward procedure would be to measure the number of seconds that an algorithm takes. Even though this method is pragmatic and easy to measure, unfortunately, it is also machine-dependent and makes comparisons between machines hard. A more robust way to measure algorithm times is to count the number of elementary operations a computer would need to run the

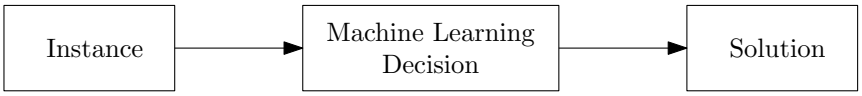


Figure 1.1: Paradigm 1: a combinatorial problem is solved with machine learning directly from the input instance.

algorithm. Often, we consider an asymptotic lower bound for the number of operations an algorithm would need to solve a worst-case instance, measured in the input size. However, considering the worst-case instance might give a pessimistic view of the running time, so sometimes it also makes sense to consider the running time of an average-case instance. If the number of computing steps that an algorithm takes is a polynomial in the size of the problem input, we say that an algorithm runs in polynomial time, or the algorithm is efficient. Intuitively this means that the running time of the algorithm grows polynomially whenever the problem size grows. In contrast, if the running time is for example exponential in the input size, the running time might increase significantly whenever large instances need to be solved. Sometimes the search for a polynomial algorithm remains unfruitful and you might be better off proving that the problem at hand is too hard to expect an efficient algorithm. In this case, you might be able to prove that the problem is at least as hard as a lot of well-studied problems, namely the set of NP-hard problems. This is a large set of problems that are considered *hard*, meaning that no polynomial time algorithm is known for any of them, and moreover, would an algorithm be known for a single one of these problems, all problems in the class can be solved in polynomial time.

Machine Learning for Combinatorial Algorithms. To summarize, this thesis studies how we can improve algorithms in order to perform their job right, well, and fast. In three chapters of this thesis, we try to improve algorithms along these measures by enhancing them with *machine learning* (ML) techniques. Machine learning focuses on teaching computers to perform specific tasks based on algorithms and statistical models. Instead of executing exact instructions, machine learning models learn patterns from data or experience. Machine learning and, more generally, artificial intelligence, have made a huge rise recently. In fact, this rise is probably the reason why many people have heard of algorithms nowadays. Success stories include contributions to health care, natural language processing, image recognition, board games, etc. (see, e.g., [Chugh et al., 2021, Lu and Weng, 2007, Otter et al., 2020, Qayyum et al., 2020, Silver et al., 2018, 2017]).

We believe that the combinatorial optimization community can benefit from the recent advances in machine learning by integrating algorithms from both fields. Combining machine learning and optimization algorithms is a hot topic, and can be approached in many different ways. An overview is given by Bengio

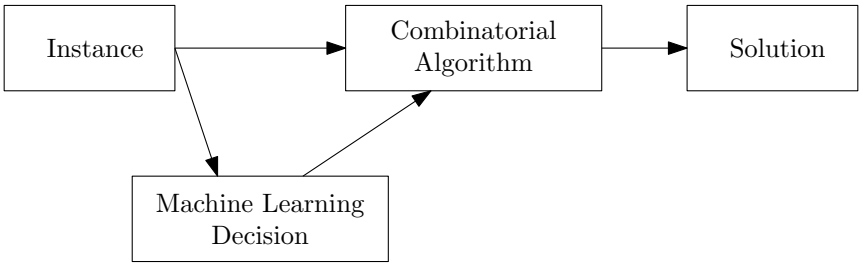


Figure 1.2: Paradigm 2: machine learning is consulted once by a combinatorial algorithm.

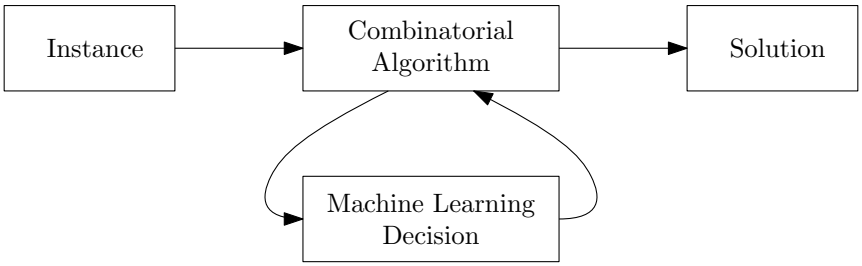


Figure 1.3: Paradigm 3: machine learning is consulted repetitively by a combinatorial algorithm.

et al. [2021], who distinguish three different paradigms of combinations between ML and combinatorial optimization. The first of these paradigms, schematically shown in Figure 1.1, is to leverage machine learning to solve combinatorial optimization problems directly from the input instance. As opposed to this first paradigm, in which ML replaces the combinatorial algorithm, the second and third paradigms use ML predictions alongside combinatorial methods. Paradigm 2 consults a machine learning prediction a single time, as schematically shown in Figure 1.2, and paradigm 3 consults machine learning predictions repetitively, as schematically shown in Figure 1.3.

We can use the classification by Bengio et al. [2021] to illustrate what we believe are the limitations and possibilities of using ML predictions to solve combinatorial problems. We believe it is a rather demanding task for an ML model to solve a combinatorial problem directly from the instance, as is expected from algorithms that follow the first paradigm. Moreover, we believe that these algorithms do not leverage the power of known combinatorial algorithms. In some sense, you could argue that designing these algorithms is like reinventing the wheel. We therefore believe that the impact of ML on combinatorial algorithms is limited if an algorithm follows the first paradigm.

On the contrary, we believe there is much potential in algorithms that follow the second and third paradigms. For many combinatorial problems, well-working algorithms without ML already exist. We believe ML predictions can improve these algorithms even further by replacing subroutines in them with new ML

techniques. In this way, we can exploit some of the existing qualities of combinatorial algorithms, while improving designated tasks within the algorithm with ML, resulting in a well-working interplay between ML and combinatorial algorithms. In this thesis, we study the impact of ML on combinatorial algorithms following the second and third paradigms both from a practical and a theoretical point of view.

On the practical side, we study which tasks in an existing algorithm can be identified to be replaced with ML. Specifically, ML can have a great impact on tasks that benefit from learning from data or experience. Next to identifying the suitable subroutines in existing algorithms, we study how the ML model should be trained. In ML, it is crucial to learn from the right data, and it turns out that if ML is used alongside a combinatorial algorithm, it is not straightforward which data samples should be learned from. In this thesis, we present guidelines and techniques that advise how to train ML models when they are used alongside combinatorial algorithms.

On the theoretical side, we distinguish three properties, which are defined to measure the impact of ML predictions in combinatorial algorithms. First, *consistency* expresses how an algorithm behaves in case of perfect predictions, i.e., if the predictions are error-free. The consistency therefore indicates the potential of using ML predictions in an algorithm. Second, *robustness* expresses how the algorithm behaves if the predictions are arbitrarily bad. It is used to measure how much the algorithm degrades by incorrect or imprecise predictions. Lastly, we study the behavior of an algorithm given that predictions are within an error of η, for a suitably defined error parameter η.

Practice versus Theory. All problems considered in this thesis are motivated by practical applications. However, this does not prevent us from considering these applications through a theoretical lens. I.e., next to studying well-working algorithms for practical problem instances, we aim to find algorithms with provable properties. We give some examples of results presented in this thesis that highlight the different viewpoints that we take. On the one hand, we show in this thesis that ML can speed up practical algorithms for real-world problems by exploiting patterns in the data. On the other hand, we show theoretical bounds, as defined earlier, for using ML in classical shortest-path algorithms. On one side, we show how to exploit the characteristics of specific instances of NP-hard problems to solve them to optimality in tractable time, but on the other side, we prove that there is an algorithm for the same problem that computes solutions that are arbitrarily close to the optimal solution with an arbitrarily small violation of the constraints. We believe it is important to consider the theoretical perspective for practical applications because the gained insights give solid reasons and more intuition on why and how methods work. In the remainder of this introduction, we introduce the practical applications addressed in this thesis and elaborate on

the research questions that we aim to answer.

SSMTSP. Imagine you arrive in a new city and want to go to a supermarket, but you do not prefer any supermarket over the other. You want to find the route to the closest supermarket since you are in a rush. Formally, this problem can be defined as finding the shortest path from a given source node to any of several designated target nodes in a given directed graph with non-negative edge weights. It is known as the single-source many-targets shortest-path problem (SSMTSP).

There exist algorithms for this problem that run in polynomial time and guarantee to find an optimal solution, if it exists. For example, the classical shortest-path algorithm for the single-source single-target shortest-path problem by Dijkstra [1959] can be used to solve the SSMTSP, and an improved version tailored to many targets is presented by Bast et al. [2003]. During the execution of this latter algorithm, the value of the best-known shortest path to any of the target locations is maintained. This solution is stored and updated not only to return it upon the algorithm's completion but also to accelerate the optimization process. Specifically, any other path encountered during scanning that is longer than the best-known solution can safely be discarded, ultimately speeding up the optimization. Clearly, the sooner such a solution is found the better, since this allows for more paths to be discarded.

We study if it is possible to predict the length of the shortest path after just a few iterations and use this prediction as a best-known solution value; discarding any encountered longer path during scanning the graph. Since we want to keep the guarantee of returning an optimal solution, a problem arises when a prediction is too low, since this could lead to the problematic discard of the optimal solution. We give an algorithm that is guaranteed to find the optimal solution, even in the case of these so-called *underpredictions*. We study both theoretically and empirically how much our algorithm saves.

In our earlier introduced terminology for assessing algorithms, we answer the following question:

- Can we make the algorithm for the SSMTSP *faster*, while maintaining the guarantee that it finds the *best* solution?

Casting Problem. The second application considered in this thesis is the *Casting Problem*, motivated by a practical real-world problem in a metal foundry. In such a foundry, metal objects of different weights are cast out of larger metal heats. An instance of this problem can be defined by a set of knapsacks and a set of items, where each knapsack has a given capacity corresponding with the size of a heat and each item has a given weight. A feasible solution to this problem is an assignment of all items to knapsacks, such that the total weight of items assigned to a knapsack is at most the capacity of the knapsack. The efficiency of a solution can be quantified by considering the utilization ratio for the knapsacks. For a

given solution the *utilization ratio* of a knapsack is defined by the total weight of items assigned to the knapsack divided by the knapsack's capacity. The goal of the casting problem is to find a feasible solution that maximizes the sum of the knapsacks' utilization ratios.

We show that the problem of finding a feasible solution to the casting problem is already NP-hard. Therefore, we do not expect to find a polynomial time algorithm or even a polynomial time approximation algorithm if we desire a feasible solution. In this thesis, we explore what we can achieve for this NP-hard problem using two different perspectives.

First, we take a practical point of view and consider a given set of empirical instances. We show that by considering a disaggregated problem formulation, we can decrease the running time, ultimately leading to solving the considered problem instances to optimality. Second, we address the problem with a theoretical point of view and study what is possible if we relax one of the feasibility constraints by considering bicriteria approximation algorithms. Informally, a bicriteria approximation algorithm for the casting problem tries to find an almost optimal solution that violates the knapsack capacity constraints by only a bounded factor.

Using our earlier introduced terminology for algorithm assessment, we answer the following two questions about the casting problem:

- Even if it seems theoretically unlikely, can we exploit large-scale practical instances such that we can find the *right* and *best* solutions *fast*?

- If finding *right* solutions *fast* does not seem possible, or at least unlikely, can we find *good*, almost *right*, solutions *fast*?

CVRPTW. We consider an example of supermarket planning again but approach it from the supermarket's perspective this time. Home deliveries of groceries have become more frequent, which challenges supermarkets to solve large planning puzzles every day. The planners at the supermarket need to solve the problem of deciding which groceries are delivered by what vehicle, and what order the groceries are delivered in. They need to take constraints into consideration involving the time windows of customers and the capacity of the vehicles. The goal of this problem is to find the most efficient solution, where efficiency is often defined based on traveled distance and the number of used vehicles. One can imagine that both for minimizing cost, but also for minimizing environmental footprint, it is important to find efficient solutions. These problems are known as *capacitated vehicle routing problems with time windows* (CVRPTW).

A universal approach that has proven to be highly efficient for solving these routing problems is the *Large Neighborhood Search* (LNS) heuristic [Pisinger and Ropke, 2010]. LNS iteratively improves a solution by destroying and repairing a part of the solution, while leaving the rest of the solution untouched. To illustrate,

for the application to the CVRPTW, LNS considers a subset of routes in each iteration and tries to find an improvement within this subset of routes. In order to quickly obtain solutions with a good objective value, it is crucial to destroy smartly, since the repair routine of LNS is usually expensive, both in computing time and computing power. Given that it is easy to create exemplary iterations where we can observe which destroy steps are effective and which are not, we aim to learn from these previous iterations. Our idea is that if we can improve the destroy step by learning from previous iterations, we can quickly obtain better solutions.

In our terminology, this boils down to answering the following question:

- *By improving the destroy step of the LNS algorithm, can we learn how to become* faster *at finding* good *solutions?*

1.1 Collaboration with Dassault Systèmes

This thesis is the result of a collaboration between CWI and the DELMIA Quintiq department of Dassault Systèmes. DELMIA Quintiq specializes in optimization and supply chain planning solutions. The R&D department of DELMIA Quintiq is constantly aiming to improve the algorithms that function as the engine of these planning solutions. The applications range from manufacturing and workforce planning to routing planning problems. The collaboration with DELMIA Quintiq inspired the research presented in this thesis, both in a direct manner as well as in more subtle ways. There is a clear direct influence on the last two chapters of this thesis. The Logistics Planner is a tool that is used daily by clients of DELMIA Quintiq to solve last-mile routing problems. Together with the R&D department of DELMIA Quintiq we have studied the neighborhood creation routines in the Logistics Planner application. These routines are an essential part of the application, so there was an obvious desire to improve them, if possible. We have tried to improve the neighborhood creation method by introducing two new techniques. We were able to test these techniques using benchmark instances provided by DELMIA Quintiq. Subsequently, we have translated this research to the well-studied world of vehicle routing problems. The translation to vehicle routing problems allowed us to make the code and the entire framework that our methods were embedded in publicly available.

A different project with Dassault Systèmes inspired the research in this thesis more indirectly. In this project, we predicted the outcome of an optimization run with ML after running only a small fraction of the iterations. The concept of predicting the outcome of an optimization run based on its start seemed more generally applicable, and specifically well to the single-source many-targets shortest-path problem, as explained above. Ultimately, this led to the results now presented in Chapter 3.

1.2 Overview and Publications

Chapter 2. In the next chapter, we introduce the notation used in the rest of the thesis and introduce concepts used in later chapters. There is a section about graphs, after which there is a section about optimization problems, complexity, (approximation) algorithms, and heuristics. Moreover, we explain the basics of machine learning methods used in this thesis.

Chapter 3. The work in this chapter is based on the findings in the following paper:[WF1]

[WF1]: Add correct bib entry

In this chapter, we study the SSMTSP, as defined in the introduction. We enhance an existing adaptation of Dijkstra's algorithm with an ML procedure, which predicts the shortest path distance after a few iterations, based on the so-called trace of the algorithm. This prediction is used to prune the search and reduce the number of queue operations on the underlying priority queue. First, we present extensive experimental results on random instances showing that we can save on computing time significantly. The rest of the chapter studies a theoretical lower bound on the number of saved operations of our proposed algorithm. Crucially, we require that our algorithm returns an optimal solution, even if predictions are off. This means that for some worst-case instances, we do not save any queue operations, even if the predictions are perfect. In general instances, however, our algorithm may save a significant number of priority queue operations. In fact, we give a closed-form expression of the expected number of saved queue operations on random instances.

Chapter 4. The results in the following paper serve as the basis for the work in this chapter and the next chapter:[WF2]

[WF2]: Add correct bib entry

Chapter 4 and Chapter 5 study the *casting problem*, as defined in the introduction. In Chapter 4, we show that the casting problem is NP-hard. However, we show that it remains possible to solve empirical instances of the casting problem to optimality, even if they grow in size. First, we show that adding a decision variable in the formulation of the problem enables mathematical programming solvers to solve the problem faster. Second, we introduce a disaggregated formulation of the problem and show that we can create variables for the disaggregated formulation in such a way that any feasible solution with them is automatically optimal. The disaggregated formulation allows us to solve all the instances introduced in Deb and Myburgh [2017], with up to 100 million knapsacks, to optimality, taking at most $\sim$75 minutes. As a comparison, the algorithm in Deb and Myburgh [2017] needs more than 6 days to solve the largest instance, with no optimality guarantee.

Chapter 5. In this second chapter about the casting problem, we let go of the empirical viewpoint and instead study what is possible in terms of polynomial-time algorithms for the NP-complete casting problem. We present two bicriteria approximation algorithms in this chapter, which allow an infeasible solution, provided that the infeasibility is bounded by a constant factor. The first bicriteria approximation algorithm that we present returns solutions that have an objective value that is at least as good as the optimal feasible solution, while the capacity constraints are violated by at most a factor $3/2$. The second bicriteria approximation algorithm that we present is a polynomial time bicriteria approximation scheme, i.e., a family of algorithms that yields a solution that has a solution value that is arbitrarily close to the optimal solution and for which the capacity constraint violation can be made arbitrarily small. However, the execution time of the approximation algorithm increases as the errors decrease in size.

[WF3]: Add correct bib entry

Chapter 6. This chapter is based on the following paper:[WF3]

The last two chapters of this thesis each introduce an improvement of the destroy step in an LNS algorithm. The new subroutines aim to leverage machine learning to improve their ability to identify which parts of the solution to destroy. In Chapter 6 we introduce a subroutine called *Learning-Enhanced Neighborhood Selection* (LENS). It functions within an LNS algorithm applied to the CVRPTW. LENS considers multiple sets of routes in each iteration and predicts which set of routes has the most potential to improve the solution. This set of routes is then selected, destroyed, and repaired, after which the solution has possibly improved. The crucial idea is that the predictions by LENS favor those sets of routes for which it is worth running the expensive repair routine. We illustrate the workings of LENS on synthetic CVRPTW instances.

[WF4]: Add correct bib entry

Chapter 7. This chapter's work is founded on the results presented in the following article:[WF4]

In this chapter, we present a second subroutine that can be used as a destroy step of an LNS algorithm. In this second approach, called *smart neighborhood creation* (SNC), we take the concepts introduced in Chapter 6 a step further. Instead of *selecting* the set of routes from a list of options of sets as LENS does, in SNC the set is *created* by adding routes one by one. In SNC, the decision of which route to be added to the neighborhood is made by a trained ML policy. Moreover, we enable the ML models to self-adapt to changes in the problem structure by training them with reinforcement learning. We apply SNC to two different applications. First, we show the workings of SNC on real-world last-mile pick-up and delivery problem instances, coming from the Logistics Planner application by DELMIA Quintiq. Second, we apply SNC to the same synthetic CVRPTW instances as in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we introduce the key concepts, background information, and essential terminology necessary for understanding the subsequent chapters of the thesis. The section about graphs is based on Schrijver et al. [2003] and the section about algorithms and complexity is based on Cormen et al. [2022]. We refer to these publications for a more detailed description of the introduced concepts.

We use $\mathbb{N}$ to refer to the natural numbers, i.e., the positive integers, and $\mathbb{N}_0$ to refer to $\mathbb{N} \cup \{0\}$. We use $\mathbb{R}$ to refer to the real numbers and $\mathbb{R}_{\geq 0}$ to refer to the non-negative reals. For $n \in \mathbb{N}$, we use the notation $[n]$ to denote $\{1, \ldots, n\}$.

2.1 Graphs

A *graph*, denoted by $G = (V, E)$, is a tuple consisting of a set of *nodes*, V, and a set of *edges*, $E \subseteq V \times V$. A graph can be directed or undirected. In a *directed graph*, the edges point from one node $u \in V$ to another node $v \in V$, and are denoted with (u, v). In directed graphs, we also call the nodes *vertices* and the edges *arcs*. An edge in an *undirected graph* is denoted with $\{u, v\}$. An example of a directed, bipartite graph and an undirected, complete graph are given in Figure 2.1. If the vertices in a graph can be partitioned in two sets, say L and R, such that every edge in the graph has one endpoint in L and the other endpoint in R, a graph is called *bipartite*. If there is an edge between every pair of nodes, a graph is called *complete*.

A *walk* in a graph is defined by a sequence of alternating vertices and edges, ending and starting with a vertex, say $P = (v_0, e_1, v_1, \ldots, e_m, v_m)$, such that e_i is an edge from v_{i-1} to v_i, for $i = 1, \ldots, m$. If all the vertices in a walk are distinct, P is called a *path*. We call a path that starts in s ($v_0 = s$) and ends in t ($v_m = t$) an s-t-path. The edges in a graph sometimes have an associated *weight* (or cost) in the form of a weight function $w : E \mapsto \mathbb{R}$. The weight or cost of a path is defined as the sum of the cost of the edges on the path.

Figure 2.1: Example of a directed, bipartite graph (left) and an undirected, complete graph (right).

2.2 Algorithms and Complexity

In this thesis, we consider combinatorial optimization problems.

Definition 2.2.1. A *combinatorial optimization problem* (or *optimization problem*) Π can be defined as a triple $(\mathcal{I}, F, c)$, where

- $\mathcal{I}$ is a set of instances. An instance is a specific scenario or set of parameters for the problem.
- $F(I)$ is a set of feasible solutions for each instance $I \in \mathcal{I}$. Oftentimes, $F(I)$ is only given implicitly, e.g., as all the $s-t$-paths in a given graph for some given nodes s and t.
- $c(I, s)$ is the cost of a feasible solution $s \in F(I)$.

If the problem instance $I \in \mathcal{I}$ is given, or clear from the context, we sometimes denote F and $c(s)$ instead of $F(I)$ and $c(I, s)$, respectively. The goal of a combinatorial optimization problem is to find an *optimal solution* for a given instance $I \in \mathcal{I}$. An optimization problem can be a *minimization problem* or a *maximization problem*. If problem Π is a minimization problem, an optimal solution for $I \in \mathcal{I}$ is a solution $s^* \in F(I)$ such that

$$c(I, s^*) \leq c(I, s) \quad \forall s \in F(I).$$

Similarly, if problem Π is a maximization problem, an optimal solution for $I \in \mathcal{I}$ is a solution $s^* \in F(I)$ such that

$$c(I, s^*) \geq c(I, s) \quad \forall s \in F(I).$$

The problem of deciding if a feasible solution exists for a given instance $I \in \mathcal{I}$, or equivalently, deciding if $F(I)$ is not empty, is called the *feasibility problem* of I. The answer to the feasibility problem is either *yes* or *no*. We call problems with either a yes or a no answer a *decision problem*. An instance for a decision problem that has a yes answer (no answer, resp.) is called a *yes-instance* (*no-instance*, resp.).

For each optimization problem, there is a specific decision problem that we call the *decision problem related to the optimization problem*. For an instance I of a minimization problem $\Pi = (\mathcal{I}, F, c)$ and a given number K, the decision problem related to Π outputs *yes* if there is a feasible solution $s \in F$ such that $c(s) \leq K$ and *no* otherwise.

2.2.1 Algorithms and Running Time

The optimization problem, feasibility problem and decision problem defined above are all examples of *computational problems*. A computational problem is nothing more than an input/output relationship. An *algorithm* takes an input and describes a specific procedure of computing a corresponding output. An algorithm can therefore be used to tackle computational problems. For a combinatorial optimization problem, the input of an algorithm consists of an instance, the output could be a feasible solution. If an algorithm terminates with the correct output for every input instance, we say that an algorithm is *correct*, or it *solves* the computational problem. In contrast, an incorrect algorithm might not terminate at all, or it might terminate with a wrong answer. In Chapter 5 of this thesis, we see that even incorrect algorithms can be of interest if we can control the size of the error.

Two algorithms solving the same computational problem might differ tremendously in their efficiency. Analyzing the efficiency of algorithms is often done by measuring the resources that the algorithm requires. Sometimes resources like memory usage or computer hardware are used as a comparison, but mostly computational time is used to measure an algorithm's efficiency.

To make machine-independent claims about the computation time of algorithms, we consider a generic model of computation called a *random-access machine* (RAM). The RAM model contains several elementary operations, inspired by the design of real computers, that form the building blocks of all algorithms. The elementary operations are arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, copy, store), and control (subroutine call, return, (conditional) branching). In the RAM model, each such operation takes a constant amount of time. For storing data, the RAM model has the data types integer and floating point. In order to prevent that we can abuse the amount of data stored in one data entry and perform constant time operations on it, we limit the size of each stored entry. We typically assume that integers are represented by at most $c \log n$ bits, where n is the size of the input and $c \geq 1$ is some constant. In this way, we can store n in one entry, but the size of an entry cannot grow arbitrarily since c is constant.

Generally speaking, the time that an algorithm takes is dependent of the instance and its size, so we often describe the running time of an algorithm as a function of the input size. First, we clarify what we mean by *input size* and *running time*. It is problem-dependent what the best notion for input size

is. Oftentimes we use the number of items in the input, for example, the total number of nodes and edges in a graph. Sometimes the number of bits necessary to describe the input is used as the input size. For each problem of which we study the running time we indicate what measure of input size we use. The running time of an algorithm can then be described as the number of elementary operations in the RAM model executed before it terminates, expressed as a function of the size of the input.

Oftentimes, it is not necessary to determine the exact relation between the running time of an algorithm and the size of the input. It suffices to compute the rate at which the running time increases whenever the size of the input increases. This rate yields a good comparison between the performance of algorithms on large instances. This can be accomplished using the *big $\mathcal{O}$ notation*, essentially an asymptotically upper bound of a function:

Definition 2.2.2. Let f and g be two real-valued functions, then

$$f(n) = \mathcal{O}(g(n))$$

if there exists $c > 0$ and $n_0 > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0.$$

For the sake of completeness, we also introduce the asymptotic lower bound notation Ω (we say *big omega*), and the combination of the asymptotic upper and lower bound notation Θ (we say *big theta*).

Definition 2.2.3. Let f and g be two real-valued functions, then

$$f(n) = \Omega(g(n))$$

if there exists $c > 0$ and $n_0 > 0$ such that

$$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0.$$

Definition 2.2.4. Let f and g be two real-valued functions, then

$$f(n) = \Theta(g(n))$$

if both $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.

We have established how to measure the running time of an algorithm based on the input size, but we have not yet covered which instances should be used for this aim. Oftentimes, we take a worst-case perspective and consider the *worst-case running time*, i.e., the longest running time for any input of size n. Doing so provides an upper bound of the running time, and a guarantee that the running time cannot be more, for every other instance.

In combinatorial optimization, we are striving for efficient algorithms in the worst case. An algorithm is *efficient* if it runs in polynomial time in the input size.

Definition 2.2.5. An algorithm is *efficient*, or runs in *polynomial time*, if its running time is upper bounded by a polynomial expression in the size of the input, i.e., if the size of the input is n, there exists a constant $c > 0$ independent of n such that the running time is $\mathcal{O}(n^c)$.

The worst-case running time might sometimes give a too pessimistic representation of the running time and it might be insightful to consider random instances to compute the *average case running time* or *expected running time*. To accomplish this it is necessary to define a random model and define a probability distribution over possible instances. In Chapter 3, we perform such a computation, called a *probabilistic analysis*.

2.2.2 Complexity

In this section, we aim to give an intuition of the complexity classes P and NP, we refer to Cormen et al. [2022] or Garey and Johnson [1979] for a formal definition of the complexity classes using formal language theory.

We can classify decision problems based on their complexity. The first class that we consider, P, can be regarded as the class of easy decision problems: all problems for which an efficient algorithm exists.

Definition 2.2.6. A decision problem belongs to the complexity class P if an efficient algorithm exists to solve it.

The second set of decision problems that we consider is the set of NP-complete problems. Before we define what it means to be NP-complete, we give some intuition on these problems. Contrary to the problems in P, it is unknown if an efficient algorithm exists for the problems that are NP-complete. At the same time, no one has ever been able to prove that an efficient algorithm cannot exist. Moreover, the set of NP-complete problems has the astonishing property that the existence of an efficient algorithm for *any* NP-complete problem means that there is an efficient algorithm for *all* problems that are NP-complete.

To introduce what it means for a decision problem to be NP-complete, we need the notion of a verification algorithm and the related complexity class NP. A *verification algorithm* takes as input, next to the instance, an extra piece of information called the *certificate*. A verification algorithm A *verifies* a problem if, given any yes-instance of the problem, there is a certificate y that A can use to prove that it is a yes-instance. Moreover, for a no-instance of the problem, no such certificate may exist.

Definition 2.2.7. A decision problem belongs to the complexity class NP if a verification algorithm exists that verifies the problem in polynomial time, using a certificate of polynomial size.

It follows that all decision problems that belong to P automatically belong to NP, since for every instance of a decision problem in P a solution can be obtained in polynomial time, which can serve as a certificate to verify if it is a yes-instance or not. The NP-complete problems mentioned above are also members of NP, in fact, they are considered to be the hardest problems in NP. To be able to define a hardness relation between problems we introduce the notion of a reduction.

Definition 2.2.8. A decision problem Π_1 is *polynomial-time reducible* to a decision problem Π_2 if there exists an efficient algorithm that translates yes-instances of Π_1 into yes-instances of Π_2 and no-instances of Π_1 into no-instances of Π_2.

If a decision problem Π_1 is reducible to a decision problem Π_2, then problem Π_2 can be regarded to be at least as hard as problem Π_1. To see this, observe that an algorithm for Π_2 can be used to solve Π_1, by using the polynomial-time reduction to translate an instance of Π_1 to an instance of Π_2, and then using the algorithm for Π_2 to solve the translated instances, returning yes if and only if the original instance of Π_1 was a yes-instance.

The polynomial reduction can be composed: if problem Π_1 is polynomial-time reducible to problem Π_2, and problem Π_2 is polynomial-time reducible to problem Π_3, then problem Π_1 is polynomial-time reducible to problem Π_3. This allows us to define the notion of NP-complete problems.

Definition 2.2.9. A decision problem $\Pi \in$ NP is called *NP-complete* if all decision problems in NP can be polynomial-time reduced to Π.

Observe that this definition proves what we claimed earlier, namely if there is an efficient algorithm for one NP-complete problem, there is an efficient algorithm for every NP-complete problem. In fact, it would mean there is an efficient algorithm for every problem in NP. To illustrate this, suppose there is an efficient algorithm for an NP-complete problem Π_1. Since Π_1 is NP-complete, any problem Π_2 in NP can be reduced to Π_1 and therefore be solved efficiently.

The first problem shown to be NP-complete was the boolean satisfiability problem (SAT) [Cook, 1971]. This problem deals with the question whether it is possible to find an assignment of variables such that a given conjunction of several disjunctions of these variables becomes true. Since then, many more problems have been shown to be NP-complete. In order to show that any problem, say Π, is NP-complete, it suffices to reduce any known NP-complete problem to Π. NP-complete problems lie at the heart of the famous P = NP problem, one of the Millenium Prize Problems selected by the Clay Mathematics Institute in 2000. The relation is illustrated in the following theorem.

Theorem 2.2.10. *If a polynomial-time algorithm exists for any NP-complete problem, then P = NP. Equivalently, if an NP-complete problem exists that is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.*

Proof:
Suppose problem $\Pi_1 \in$ P and Π_1 is NP-complete. Then any $\Pi_2 \in$ NP can be reduced to Π_1, since Π_1 is NP-complete. Since Π_1 can be solved in polynomial time, this means that also Π_2 can be solved in polynomial time, which proves the first statement. The second statement in the theorem is a direct consequence of the first statement. □

As explained above, all NP-complete problems are in NP and moreover, they have the property that the existence of an efficient algorithm for one of them proves the existence of an efficient algorithm for all problems in NP. Other problems can also have this property, without necessarily being in NP. We call those problems NP-hard, since intuitively they are as hard as NP-complete problems. In order to define them, we use the notion of a *hypothetical subroutine*, a routine that an algorithm may call to solve another problem, without proving that such a subroutine actually exists.

Definition 2.2.11. A problem Π_1 is NP-hard if there exists a polynomial time algorithm A for an NP-complete problem Π_2, where A may make use of a hypothetical subroutine that solves Π_1 in polynomial time.

The definition illustrates that if an efficient algorithm exists for an NP-hard problem, then an efficient algorithm exists for an NP-complete problem. Subsequently, we have seen that this means an efficient algorithm exists for every problem in NP.

Some problems are NP-hard only if their input numbers grow exponentially large, other problems are NP-hard even for smaller input numbers. To formalize this, we will consider a subset of instances of a problem in which the numbers are bounded. Given a problem $\Pi = (\mathcal{I}, F, c)$, let $|I|$ be the binary encoding length of $I \in \mathcal{I}$ and let $\text{NUMBER}(I)$ be the largest number appearing in $I \in \mathcal{I}$. Then for some non-negative function $f : \mathbb{R} \mapsto \mathbb{R}$, the *restricted problem instances* are defined as

$$\mathcal{I}_f = \{I \in \mathcal{I} \mid \text{NUMBER}(I) \leq f(|I|)\}.$$

This allows us to define strongly NP-hard problems. Intuitively, these are the problems that remain NP-hard even if all numbers are polynomial.

Definition 2.2.12. A problem $\Pi = (\mathcal{I}, F, c)$ is *strongly NP-hard* if there exists a non-negative polynomial function $p : \mathbb{R} \mapsto \mathbb{R}$ such that $(\mathcal{I}_p, F, c)$ is NP-hard.

2.2.3 Approximation Algorithms

For problems for which no efficient algorithm is known, we can sometimes defer to algorithms that do not have the guarantee of finding the optimal solution, but instead guarantee a solution that is close to the optimal one. Such an algorithm is

called an *approximation algorithm*, parameterized by an approximation factor α. Recall that we use s^* to denote an optimal solution of an optimization problem.

Definition 2.2.13. Let $\Pi = (\mathcal{I}, F, c)$ be a maximization problem. An algorithm is an *α-approximation algorithm* with $0 \leq \alpha \leq 1$ for Π if for every instance $I \in \mathcal{I}$ of size n it computes a feasible solution $s \in F$ of cost $c(s) \geq \alpha \cdot c(s^*)$ in time that is polynomially bounded in n.

The definition for a minimization problem is the same, except $\alpha \geq 1$ and for the cost of the solution found it must hold that $c(s) \leq \alpha \cdot c(s^*)$.

Sometimes it is possible to find approximation solutions with a solution value within an arbitrarily small factor of the optimal solution. A polynomial-time approximation scheme (PTAS) finds solutions like these, moreover, it runs in polynomial time for any fixed approximation ratio.

Definition 2.2.14. A family of algorithms A_ϵ is called a *polynomial time approximation scheme* (PTAS) if it gives an ϵ-approximation for each value of $\epsilon > 0$. The algorithm must run in polynomial time in the input size for any fixed value of ϵ.

An approximation scheme with a running time that is not only polynomial in the input size, but also polynomial in $1/\epsilon$, is called a fully polynomial time approximation scheme (FPTAS).

2.2.4 Large Neighborhood Search

For algorithms that solve practical problems, the need for optimality guarantees is often less prominent. In practice, *heuristics* are often used to solve problems, which aim to find satisfactory solutions by, for example, simplifying the problem, making educated guesses, or following rules of thumb. A heuristic is a practical solving method that does not guarantee an optimal solution but often offers a sufficient solution. Since a heuristic does not need to prove (approximate) optimality of a solution, it often runs faster than an algorithm that is guaranteed to find an optimal solution.

An example of such a heuristic is Large Neighborhood Search (LNS) (see, e.g., Pisinger and Ropke [2010] and Shaw [1998]). LNS is a universal approach that can be applied to a great deal of combinatorial optimization problems. It forms the basis of the algorithms that we study in the last two chapters of this thesis. We describe the approach in more detail below (see also Algorithm 1).

Let $\Pi = (\mathcal{I}, F, c)$ be the optimization problem under consideration, and let $I \in \mathcal{I}$ be the considered instance. In the explanation of LNS, for ease of notation, we omit the problem instance when denoting the set of feasible solutions and write F if we mean $F(I)$, and write $c(s)$ if we mean $c(I, s)$. LNS starts with an arbitrary feasible initial solution $s \in F$ as input.

Algorithm 1 Large Neighborhood Search

```
1: Input: feasible solution s ∈ F
2: s^best = s
3: repeat
4:     η = SelectNeighborhood(s)
5:     s^temp = Repair(Destroy(s, η))
6:     if Accept(s^temp, s) then s = s^temp
7:     if c(s^temp) < c(s^best) then s^best = s^temp
8: until StoppingCriterion is met
9: return s^best
```

The algorithm keeps track of the best solution s^{best} encountered so far. In each iteration, a (small) part of the solution s, called *neighborhood*, which is then destroyed and repaired (or rebuilt) again. The former and latter are done by a so-called *destroy method* and *repair method*, respectively. The goal of alternating the destroy and repair operations is to compute a new solution in each iteration with an improved objective value. With this aim, an *accept method* is used to determine whether the improvement of the newly created solution is significant enough (e.g., in terms of a decrease in objective function value) to be used subsequently. Furthermore, the best-known solution s^{best} is updated if necessary. The algorithm continues this way until a predefined *stopping criterion* is met.

The LNS algorithm does not specify how the respective stopping criterion and the neighborhood selection, destroy, repair, and accept methods are defined, as this is problem-specific. The selection of a neighborhood can for example be random, based on low-quality parts of the current solution, or based on easy interchangeability. The destroy method simply destroys the neighborhood that it gets as input. In practice, many advanced and well-working procedures are used for the implementation of the repair method, e.g., integer linear programming solvers or advanced path-building algorithms. The accept method could be a simple hill-climbing procedure, in which only improving solutions are accepted, but also simulated annealing can be used. A stopping criterion could be based on, e.g., time, number of iterations, or the value of the best-known solution.

2.3 Machine Learning

Machine learning focuses on teaching computers to perform specific tasks based on algorithms and statistical models. Instead of executing exact instructions, machine learning models learn patterns from examples and data.

Definition 2.3.1. *Machine learning* (ML) is a subfield of artificial intelligence in which algorithms are created that allow computers to learn behavior based on

data. A machine learning system aims to improve its performance on a *task* T, based on empirical data or *experience* E, measured with a *performance measure* P.

We explain the notions of a task, experience and performance measure with the help of an example. Suppose we are given a graph $G = (V, E)$ and two vertices in the graph, $s, t \in V$, and we want to predict the shortest path distance from s to t by a quick inspection of the graph. Instead of running a dedicated algorithm tailored to do this, suppose we want to make an educated guess about the shortest path distance, quickly and fairly accurately. Suppose we have a dataset of graphs with designated start and end nodes, which we have considered in the past, and therefore know the shortest path distance between s and t in these graphs.

The *task* T, the objective or the problem at hand, is to predict the shortest path distance from s to t. The *experience* E is the data that we can learn from, in this case, the set of graphs in the database and the corresponding shortest path distances. The *performance measure* P evaluates the accuracy of the prediction, and weighs the predicted distance against the real shortest path distance. In our example, we can for example take the squared difference between the prediction and the actual shortest path distance as an error measure.

Predicting the shortest path distance given two nodes in a graph is an example of *supervised learning*. In supervised learning, the algorithm learns from labeled examples in a dataset. Each entry in this dataset is a pair consisting of the features describing the object on one side, and the desired label on the other. Suppose the dataset consists of n datapoints, each with p features, then the data can be represented by matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$. The i'th row of the matrix, denoted by $\mathbf{x}_i$, corresponds to the i'th data entry, and consists of the features $[x_{i1}, x_{i2}, \ldots, x_{ip}]$. Each data in the database has a given label, which together make up the feature vector $\mathbf{y}$. In case of a *classification* problem, the labels are categorical, so for example $\mathbf{y} \in \{0, 1\}^p$. If the labels are numerical, for example, $\mathbf{y} \in \mathbb{R}^p$, we are dealing with a *regression* problem. By examining many of the data entries in $\mathbf{X}$ with their corresponding labels in $\mathbf{y}$, an algorithm might be able to learn a pattern between the features and the labels. Afterward, when the algorithm is presented with new, unlabeled, entries, it can predict the corresponding label based on the pattern that it learned.

A model's ability to predict well on data entries that it did not encounter before is called *generalization*. A model that generalizes well captures the essence of the pattern in the data, but ignores uninformative noise. In the unfortunate case that a model also learns to interpret noise as information, *overfitting* might happen. An overfitted model performs well on training data, but fails to generalize to unseen data points. One of the goals of machine learning is to find a balance between the risk of overfitting and the complexity of a model to enhance generalization.

As an alternative to supervised learning, a second type of machine learning is

unsupervised learning. In this learning paradigm, the learning data is unlabeled. Instead of learning to predict a label for each data entry, the aim is to learn the structure that is present within the data. If we consider the example of the graph again, an unsupervised learning task could be to cluster the nodes in the graph based on their proximity.

A third paradigm of machine learning is *reinforcement learning*, in which an algorithm learns by interacting with an environment. The algorithm consists of a so-called *agent* that makes an action based on a policy. The environment gives feedback based on the action in the form of rewards, after which the agent might adapt its policy. The goal of the agent is to adapt its policy in such a way that it maximizes the total received reward. The example of finding the shortest path distance between two nodes in a graph can illustrate how reinforcement learning works. Suppose an agent starts at node s, and travels through the graph, trying to find the shortest route to node t. After traversing an edge, the agent receives feedback from the environment indicating how much the distance between the agent and the target node t decreased. The feedback can help the agent to adjust its policy such that edges that bring the agent closer to t will be favored. Ultimately, after many tries, the agent will learn what edges to take to travel from s to t quickly.

The rest of this section consists of a brief explanation of the machine learning models that we use throughout this thesis: linear regression, neural networks and random forests.

2.3.1 Linear Regression

Linear regression is a type of supervised learning. Suppose we are given a dataset, $[\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$ that consists of n datapoints, where each datapoint $\mathbf{x}_i \in \mathbb{R}^p$ contains p features. The label vector is denoted by real-valued p-dimensional vector, $\mathbf{y} \in \mathbb{R}^p$, so that we are indeed dealing with a regression problem.

In linear regression, the aim is to find a linear relation between the features and the target value. The assumption in linear regression is that the relation between the features and the label for the i'th entry can be expressed in the following equation:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \epsilon_i.$$

Or in matrix form:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}.$$

Here, $\boldsymbol{\beta} \in \mathbb{R}^{p+1}$ is a vector of coefficients that describe the relation between the features and the target value. We allow a slight abuse of notation and use $\mathbf{X}$ for the $n \times p + 1$ matrix consisting of the rows $[1, x_{i1}, x_{i2}, \ldots, x_{ip}]$, in contrast with the previous section, where $\mathbf{X}$ was defined as a $n \times p$ matrix. $\boldsymbol{\epsilon} \in \mathbb{R}^n$ is an

n-dimensional vector representing the *error* or *residual*, defined as the deviation between the true labels and the prediction.

The goal of linear regression is to find the coefficient vector $\boldsymbol{\beta}$ that is the best fit to model the observations in the data set. This can be achieved by taking the sum of squared residuals as a performance measure and minimizing this measure. The sum of squared residuals is defined as follows:

$$P(\mathbf{X}, \mathbf{y}, \boldsymbol{\beta}) = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon} = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) .$$

The optimal coefficients that minimize this performance measure can be obtained by solving the following set of equations, to which a unique solution exists if $\mathbf{X}^T\mathbf{X}$ is invertible:

$$\boldsymbol{\beta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{y}.$$

After computing $\boldsymbol{\beta}$, the model can be used to predict the label $\hat{\mathbf{y}}$ for new data as follows:

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}.$$

2.3.2 Neural Network

Neural networks are inspired by the way that a brain works. They consist of multiple layers of nodes, where each node takes an input, manipulates it and outputs the result. By doing so, complex patterns can be recognized in the data through training.

Many different types of neural networks exist, like recurrent networks or convolutional networks. We refer to Goodfellow et al. [2016], Haykin [2009], Bishop and Nasrabadi [2006] for complete overviews. One of the basic neural network structures is a *multilayer perceptron*. We explain it in more detail here. A multilayer perceptron consists of several layers, each consisting of nodes called *neurons*. The following layers are present in a network:

- *Input layer*. The nodes in the input layer represent the features in the data.
- *Hidden layers*. Usually, several layers that receive data from the previous layer, manipulate the data, and send it to the next layer.
- *Output layer*. The nodes in the output layer compute the output of the network. In the case of a classification problem, the output consists of a distribution over the classes. In the case of a regression problem, the network outputs the predicted target value.

Each neuron computes a weighted sum of its inputs, adds a bias term, and applies an activation function to the result. Formally, the output of a neuron j

in layer l is computed as follows:

$$a_j^{(l)} = \sigma\left(\sum_{i=1}^{n_{l-1}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right).$$

Here, n_l is the number of neurons in the l'th layer, $a_j^{(l)}$ is the output of the j'th neuron in layer l, $w_{ij}^{(l)}$ is the weight of the edge between neuron i in layer $l-1$ and j in layer l, $b_j^{(l)}$ is the *bias term* in the j'th neuron of layer l and $\sigma(\cdot)$ is the *activation function* for the j'th neuron of layer l.

The activation function serves as a way to introduce non-linearity into the neural network. This enables the network to learn complex structures in the data. Usual activation functions are for example:

- *Sigmoid*: $\sigma(x) = \frac{1}{1+e^{-x}}$,
- *ReLU* (Rectified Linear Unit): $\sigma(x) = \max(0, x)$,
- *Tanh*: $\sigma(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

Deciding how many layers a neural network should have, how many nodes there should be per layer, and which activation to use is part of the design of the neural network. Often several designs are tested to decide which one works best. The values of the weights and biases in the network are not part of the design, but they are optimized by exposing the neural network with data, as explained below.

As for the linear regression model, a neural network can be used to compute a prediction $\hat{\mathbf{y}}_i$ based on a feature vector $\mathbf{x}_i$. This is done with a process called *forward propagation*, in which the feature vector serves as input for the input layer. Sequentially, the data is passed through the layers in the network, ultimately computing an output in the output layer. The output serves as the prediction $\hat{\mathbf{y}}$.

Given the prediction $\hat{\mathbf{y}}$ and the true value $\mathbf{y}$, the *loss function* can be calculated. For regression tasks, this can, e.g., be the Mean Squared Error (MSE), which is the sum of squared residuals divided by the number of observations, formally defined as:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i).$$

For binary classification tasks, this can be, e.g., the average log-loss, defined as follows:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{n}\sum_{i=1}^{n} (y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)).$$

The structure of a neural network enables an efficient way of training, called *back-propagation*. In this process, given a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$ and labels $\mathbf{y}$,

the weights and biases in the network are adjusted in order to minimize the loss function. *Gradient descent* is used to update the weights and biases, by following these steps:

1. Forward pass. For each input vector $\mathbf{x}_i$ perform the forward propagation as explained above to obtain $\hat{y}_i$.

2. Loss calculation. Calculate the loss for each $\mathbf{y}, \hat{\mathbf{y}}$ pair.

3. Backward pass. Compute the gradient of the loss with respect to the parameters in the network, namely the weights and biases. The chain rule is used to compute the loss through the network.

4. Parameter update. The computed gradient of the loss can be used to update the weights and biases in the network, usually with an algorithm like gradient descent. To illustrate, gradient descent updates the weight $w_{ij}^{(l)}$ as follows:
$$w_{ij}^{(l)} = w_{ij}^{(l)} - \eta \frac{\partial L}{\partial w_{ij}^{(l)}}.$$
Here, η is the so-called *learning rate* of gradient descent and $\frac{\partial L}{\partial w_{ij}^{(l)}}$ is the gradient of the loss with respect to this specific weight.

2.3.3 Random Forest

A *random forest* is an ML model that consists of several decision trees, hence the name, which together decide the outcome of a prediction. Random forests can be used both for classification and regression tasks.

Random forests are a type of *ensemble learning*. Ensemble learning is based on the principle of the wisdom of the crowd, since multiple base learners are combined to create the prediction method. In the case of random forests, these base learners are decision trees. Suppose there are T decision trees in the forest, denoted by $\{h_1, h_2, \ldots, h_T\}$. Each tree h_i in the forest is a classification or regression tree, depending on the problem, trained on a subset of the training data.

A *decision tree* is a supervised learning algorithm that can be used for both classification and regression problems. A decision tree is a tree-like model, with a root node at the top, and internal nodes connecting the leaf nodes at the bottom to the root node. The root node of a decision tree corresponds to all of the data in the data set. At each of the internal nodes, including the root node, the data is split into subsets based on one or several of the features and a threshold value. The leaf node represents the final decision for the subset of the data that corresponds to that node. In classification problems, a leaf node gets a class label, in regression problems, the leaf node gets a prediction value. The construction of a tree is done by recursively splitting the dataset based on some features such that a *splitting criterion* is maximized until a *stopping criterion* is met.

Algorithm 2 TrainRandomForest($\mathbf{D}, T$)

1: **input dataset D, number of trees** T**,**
2: **for** $t = 1, \ldots, T$ **do**
3: draw a bootstrap sample $\mathbf{D}_t$ from $\mathbf{D}$
4: $h_t = \text{BuildTree}(\mathbf{D}_t)$ ▷ *train a decision tree* h_t *based on* $\mathbf{D}_t$
5: **return** $(h_1, \ldots, h_T)$

In the algorithm for training random forests, the concept of *bootstrapping* is used. Suppose a dataset $\mathbf{D} = [\mathbf{X}, \mathbf{y}]$ consists of n samples. Then bootstrapping, or *bagging*, means to take a subset of the dataset, with replacement. I.e., a bootstrap sample of the given dataset also consists of n samples, but some of the samples of the original dataset might be repeated and others might be left out.

The algorithm for training a random forest is given in Algorithm 2. In the algorithm, a decision tree is created for each of the T bootstrap samples. The decision trees created for a random forest often use *random feature selection*, i.e., a randomly chosen subset of the features is used to decide how to split the data in the creation of the decision tree. The resulting forest consists of a collection of the created trees.

When a random forest is used to create a prediction for a feature vector $\mathbf{x}_i$, the predictions made at the leaf nodes of the T decision trees need to be aggregated into a single prediction. For classification, this final prediction is computed by majority voting, i.e., the final prediction is the class predicted by most of the decision trees. For regression, the final prediction is the average of the predictions of the separate decision trees.

Compared to a single decision tree, random forests reduce the risk of overfitting since each tree in the forest is based on only a subset of the data. For the same reason, random forests are more robust to noise and are better able to handle large amounts of data.

Chapter 3

Dijkstra's Algorithm with Predictions for the Single-Source Many-Targets Shortest-Path Problem

3.1 Introduction

In recent years, techniques from machine learning (ML) have proven extremely powerful to tackle problems that were considered to be very difficult or even unsolvable. Success stories include, as mentioned in the introduction of this thesis, contributions to health care, natural language processing, image recognition, board games, etc. (see, e.g., [Chugh et al., 2021, Lu and Weng, 2007, Otter et al., 2020, Qayyum et al., 2020, Silver et al., 2018, 2017]). As such, machine learning is intimately connected with optimization because many learning algorithms are based on the optimisation of some loss function over a large set of training samples. Even though optimization techniques play a vital role in the design of ML approaches, the reverse direction of using ML techniques to improve optimization algorithms is much less explored.

In this chapter, we investigate the use of predictions in the context of a fundamental shortest-path problem, which is known as the *single-source many-targets shortest-path problem (SSMTSP)*. Given a directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}_{\geq 0}$, a source node $s \in V$ and a subset $T \subseteq V$ of designated target nodes, the goal is to compute a shortest path from s to one of the target nodes in T. Note that the SSMTSP problem generalizes the single-source single-target shortest-path problem (for which $T = \{t\}$ for a given target node $t \in V$) and the single-source all-targets shortest-path problem (for which $T = V$). The attentive reader will have noted that the SSMTSP problem

can be reduced to a single-source single-target shortest-path problem simply by adding a new target node t and connecting each node in T to t with a zero-weight edge. Unless stated otherwise, we adopt the viewpoint of having a set of target nodes throughout the chapter. These shortest-path problems are among the most fundamental optimization problems with various applications in practice. Further, both the assignment problem and the maximum weight matching problem in bipartite graphs can be reduced to the solving of n SSMTSP problems, where n is the maximum number of nodes on each side of the bipartition (see, e.g., [Bast et al., 2003] for details). Clearly, this emphasizes the importance of deriving efficient algorithms for this problem.

The SSMTSP problem can be solved exactly in (strongly) polynomial time by using a slight adaption of the well-known shortest path algorithm due to Dijkstra [1959], referred to as DIJKSTRA in the remainder of this chapter. Basically, DIJKSTRA grows a shortest path tree rooted at the source node s by iteratively adding new nodes to the tree by increasing distances, until the first target node from T is included. DIJKSTRA guarantees a worst-case running time of $O(m + n \log n)$, where n and m denote the number of nodes and edges, respectively. In order to achieve this running time, crucially the underlying priority queue data structure must be implemented through *Fibonacci heaps* (see [Fredman and Tarjan, 1987]). In terms of worst-case running time, DIJKSTRA is the best-known algorithm that runs in strongly polynomial time for the shortest-path problem with *arbitrary* non-negative edge weights (see, e.g., [Cormen et al., 2022]). If the edge weights are known to be non-negative integers from a restricted range, there are better algorithms (see also the related work section).

Given that the SSMTSP problem can be solved very efficiently by Dijkstra's algorithm, it is unclear how predictions can help to further improve the running time of the algorithm. Reading the input instance alone takes time $\Theta(m + n)$, thus, the best we can hope for is to reduce the $O(n \log(n))$ term in the running time of DIJKSTRA to $O(n)$. In this chapter, we focus on exact algorithms for the SSMTSP problem. More specifically, we require that the algorithm computes a shortest path together with a certificate that proves optimality, even if the predictions are arbitrarily bad. A common approach to exhibit such a certificate is by means of a corresponding dual solution to the linear programming formulation of the problem. The single-source single-target shortest-path problem has a natural (flow-based) linear programming formulation (see, e.g., [Papadimitriou and Steiglitz, 1998]). The dual of this linear program associates a dual variable $\pi(u)$ with every node $u \in V$ and reads as follows:[1]

$$\begin{aligned} &\text{maximize} && \pi(t) - \pi(s) \\ &\text{subject to} && \pi(v) \leq \pi(u) + w(u,v) \quad \forall (u,v) \in E. \end{aligned} \tag{3.1}$$

[1]Here, for the sake of conciseness, we state the dual linear program adopting the (equivalent) viewpoint of having a single target node.

In this formulation, we can fix $\pi(s) = 0$ without loss of generality. The respective dual values then have a natural interpretation as shortest path distances. By applying the complementary slackness condition, a dual feasible solution $\pi = (\pi_u)_{u \in V}$ proves the optimality of an s,t-path if and only if every edge $e = (u, v)$ on that path is *tight*, i.e., $\pi(v) = \pi(u) + w(u, v)$. Most exact shortest path algorithms known in the literature provide a certificate of optimality by constructing such an optimal dual solution (or, equivalently, shortest path distances).

Chen et al. [2022] recently studied algorithms with predictions for the single-source shortest-path problem with arbitrary edge weights under the assumption that a (possibly infeasible) dual solution $\hat{\pi} = (\hat{\pi}(u))_{u \in V}$ is available as a prediction. Note that this prediction consists of $n = |V|$ dual values. Building on earlier work by Dinitz et al. [2021], they show that such predictions can be used to obtain an algorithm to solve this problem in time $O(m \min\{\|\hat{\pi} - \pi^*\|_1 \cdot \|\hat{\pi} - \pi^*\|_\infty, \sqrt{n} \log(\|\hat{\pi} - \pi^*\|_\infty)\})$, where π^* is an optimal dual solution to the problem. In particular, the worst-case running time improves as the difference (evaluated with respect to the L_1- and L_∞-norm) between the predicted duals $\hat{\pi}$ and the optimal duals π^* decreases. Clearly, such results are appealing as they provide a fine-grained running time guarantee depending on the (cumulative) additive error of the predictions. On the negative side, however, assuming that one has access to the entire dual solution $\hat{\pi}$ might be a rather strong assumption in certain settings—especially because the number of nodes (and thus the number of values to predict) can be very large in practice.

In this chapter, we therefore consider the other end of the spectrum: We assume that our algorithm has access to a *single* predicted value only, namely the shortest path distance of a target node. Said differently, we assume that we have access to a prediction of the objective function value $\hat{\pi}_t$ of (3.1) (which might not necessarily coincide with the optimal objective function value π^*_t). The question that we address here is whether such a 'minimalistic prediction' suffices to still achieve a running time improvement of Dijkstra's algorithm.

Clearly, exploiting a prediction of the shortest path distance only is much more restrictive than assuming that one has access to an entire dual solution—in fact, the latter can be used to derive the former. Intuitively, it is clear that it is more challenging to improve Dijkstra's algorithm using such an (inferior) prediction. The following observation lends further support to this intuition. Recall that we require that our algorithm provides a certificate of optimality. If the entire dual solution is predicted perfectly, this becomes trivial: the dual itself provides such a certificate and is readily available in this case. On the other hand, this remains a challenging task in our setting: even if the correct shortest path distance is known, it is non-trivial (in terms of computational work) to compute a corresponding dual solution that proves optimality. This latter observation will be made more rigorous in Section 3.6.

Our algorithm is based on a heuristic improvement of Dijkstra's algorithm

proposed by Bast et al. [2003] (referred to as DIJKSTRA-PRUNING). The key idea of their algorithm is to exploit that Dijkstra's algorithm might encounter many target nodes in T before the actual target node (defining the shortest path distance) is added to the tree: DIJKSTRA-PRUNING simply keeps track of the smallest distance B to a node in T that was encountered so far. The algorithm then *prunes* each edge that leads to a node whose distance exceeds B—clearly, these edges are irrelevant for the final shortest path. A more detailed description of this algorithm will be given in Section 3.2. Although the pruning idea is very simple, it can significantly improve the running time of the algorithm. In the worst case, however, DIJKSTRA-PRUNING has the same running time as DIJKSTRA. Therefore, Bast et al. [2003] investigate the effectiveness of DIJKSTRA-PRUNING on random instances, both analytically and experimentally. They show that the pruning of irrelevant edges significantly reduces the number of executed priority queue operations on these instances.

3.1.1 Our Contributions

The main contributions presented in this chapter are as follows:

1. We combine an ML approach with the pruning idea above to obtain a new algorithm for the SSMTSP problem. Basically, our algorithm (referred to as DIJKSTRA-PREDICTION) computes a prediction P of the final shortest path distance after a few iterations, and then uses this prediction P together with the pruning trick of Bast et al. [2003] to further reduce the search space it explores. On the ML side, one of the challenges is to define features capturing the essence of the Dijkstra run which can be used to arrive at a good prediction. On the algorithm-design side, we need to tackle the problem that the prediction might be an underestimation of the actual shortest path distance. We note that our algorithm works independently of the specific method that is used to arrive at the prediction P.

2. We prove that our new algorithm DIJKSTRA-PREDICTION always computes an exact solution and has a worst-case running time of $O(m + n \log n)$ (independent of the prediction error). In particular, DIJKSTRA-PREDICTION retains the best worst-case running time and always provides a certificate of optimality. That is, while our algorithm will never use more priority queue operations than the adapted Dijkstra algorithm, it can potentially save many queue operations additionally.

3. We report on our extensive experimental studies that compare our new algorithm DIJKSTRA-PREDICTION to the existing ones and evaluate different prediction algorithms. Our experiments show that DIJKSTRA-PREDICTION, which combines a neural network ML approach to compute the prediction

with an UPDATE-PREDICTION procedure to handle possible underestimations, significantly outperforms all other algorithms (i.e., combinations of different prediction algorithms).

4. We establish a lower bound (both in expectation and with high probability) on the number of edges pruned by our new algorithm DIJKSTRA-PREDICTION. Our bound depends on the number of *relevant* edges (leading to nodes whose distances exceed the shortest path distance) and increases as the prediction error decreases. Our bound applies to arbitrary instances as long the weights of the relevant edges are chosen at random—we refer to this setting as the *partial random model*. Further, we show that no improvement is possible in the worst case, even if the prediction is perfect—this also justifies the use of our partial random model.

5. We then derive a bound on the expected number of queue operations saved by DIJKSTRA-PREDICTION in comparison to DIJKSTRA-PRUNING and DIJKSTRA on random instances. While DIJKSTRA-PRUNING already significantly improves over DIJKSTRA, we show that DIJKSTRA-PREDICTION further reduces the number of nodes which are inserted but never removed from the queue. More specifically, we consider Erdös-Rényi random graphs with average degree c and uniform edge weights in $[0, 1]$, where the source node is chosen uniformly at random and each node is selected as a target node with probability q (formal definitions are given below). If D denotes the shortest path distance and ε denotes the (additive) error of the prediction with $D < 1 - \varepsilon$, we show that the number of nodes inserted but never removed from the priority queue by DIJKSTRA-PREDICTION, DIJKSTRA-PRUNING and DIJKSTRA, respectively, is at most

$$\frac{1}{q}\left(1 + \ln(c-1) - \ln\left(\frac{1-D}{\varepsilon}\right)\right), \quad \frac{1}{q}(1 + \ln(c-1)) \quad \text{and} \quad \frac{c-1}{q}.$$

Here the latter two bounds were established by Bast et al. [2003]. Technically, this is the most challenging part of our analysis as we need to estimate the savings incurred by the pruning bound B as well as the prediction bound P in one probabilistic argument.

3.1.2 Related Work

Algorithms with predictions. Using ML techniques in combinatorial algorithms has been studied intensively recently. As also mentioned in the introduction of this thesis, Bengio et al. [2021] give an overview of leveraging ML to solve combinatorial optimization problems. In this survey, three different approaches of using ML components in combinatorial optimization algorithms are given. Our approach falls into the second of the three approaches, in which meaningful properties of the optimization problem are learned and used to augment the algorithm.

The line of research known as *Algorithms with Predictions* falls into this second approach and aims to achieve near-optimal algorithms when the predictions are good, while falling back to the worst-case behavior if the prediction error is large (see, e.g., [Mitzenmacher and Vassilvitskii, 2022, Lattanzi et al., 2020] and the references provided above). This idea is applied to optimization problems like the *ski rental problem*, *caching problem*, and *bipartite matching*.

Dinitz et al. [2021] use predictions to improve the worst-case running time to solve the bipartite matching problem. Their main idea is to use predictions for the dual values as a warm start of the primal-dual algorithm. Since the predicted duals might not be feasible, they propose a rounding procedure to compute a feasible dual, close to the predicted one. Moreover, they prove that the prediction of the duals that they require for the algorithm can actually be learned, by showing that this prediction problem has low sample complexity.

Chen et al. [2022] builds further upon the results of Dinitz et al. [2021]. First, they give an improvement of the algorithm for bipartite matching, which reduces the worst-case running time even more. Secondly, they extend the idea of using predictions for primal-dual algorithms and apply it to a shortest-path problem. When the predictions are accurate enough, they achieve an almost linear running time. Further, they propose a general reduction-based framework for learning-based algorithms and extend the PAC-learnability results of Dinitz et al. [2021] beyond the bipartite matching problem.

The analysis in this chapter differs from the results of Chen et al. [2022] since our algorithm only requires a single prediction for the shortest path value, instead of a learned dual for each node. That is, our algorithm requires less in terms of prediction, on the other hand, our algorithm does not improve the worst-case running time. Instead, we can prove a lower bound on the number of expensive priority queue operations that are saved.

Classical shortest path algorithms. An extensive survey of combinatorial algorithms to solve the shortest-path problem is given by Madkour et al. [2017]. We give a short summary of their extensive report, touching upon different shortest-path techniques by grouping the methods into four categories.

As explained above, Fredman and Tarjan [1987] improve Dijkstra by introducing Fibonacci heaps. Alternative heap structures that gave further improvements are AF-heaps [Fredman and Willard, 1990a,b, 1993] and relaxed Fibonacci heaps [Driscoll et al., 1988]. An implementation based on stratified binary trees is introduced by Emde Boas [1975]. Thorup [1999] indicates there is an analogy between sorting and single-source shortest path, claiming that SSSP is no harder than sorting the edge weights. Han [2001] improves on these results. Thorup [1999] builds hierarchical bucketing structure, which is improved by Hagerup [2000].

The second category contains the *distance oracle algorithms*, introduced by Thorup and Zwick [2005], which consist of a pre-processing phase and a query

phase. In the pre-processing phase, an auxiliary data structure is constructed, which is queried in the query phase to compute the shortest path. Distance oracle algorithms can be both exact, like in [Fakcharoenphol and Rao, 2006], or approximate, like in [Elkin and Peleg, 2004]. Some methods approximate distance using a *spanner*, a subgraph that maintains the locality aspects of the original graph. Other methods approximate distances using a *landmark approach* (see [Sommer, 2014]), where each vertex stores distances to a set of chosen landmarks. All distance oracle algorithms deal with a trade-off between space complexity and query time.

Goal-directed shortest path algorithms fall in the third category. These algorithms add annotations to vertices or edges with additional information. This allows the algorithm to determine which part of the graph to search in the search phase, and which parts to prune. A well-known algorithm in this category is A^*, which, unlike Dijkstra, is an informed algorithm, since it searches the route which leads to the goal. If an admissible heuristic is used, A^* will return the optimal shortest path, but it might fail if the heuristic does not work well. Several variants and improvements to A^* have been proposed, which include landmark approaches (see [Goldberg and Werneck, 2005]) or the concept of *reach* (see [Gutman, 2004]). Intuitively, the reach of a vertex encodes the lengths of shortest paths on which this vertex lies. Other goal-directed methods include edge labels (see [Köhler et al., 2005, Schulz et al., 2000, Lauther, 2006]), the arc-flag approach (see [Möhring et al., 2007, Hilger et al., 2009, Bauer and Delling, 2010]), or pre-computed cluster distances (see [Maue et al., 2010]). In this last method, the graph is partitioned in clusters, after which the shortest connection between clusters is stored. Interestingly, also for A^*, recent research shows an interest in replacing heuristics with machine learning. In [Eden et al., 2022], estimates in A^* that were formerly done with heuristics are executed with learning techniques, based on features of the nodes. They find there is a trade-off between the amount of information used to describe a node and the improvement in the running time of the algorithm.

The last category in this non-extensive list of shortest path methods is the *hierarchical shortest path methods*. These methods are prominent for problems which naturally exhibit a hierarchical structure, like road networks. An example of a hierarchical method is the highway hierarchies, which label an edge on a shortest path as a highway if it is not in the proximity of the source or target, as done by Sanders and Schultes [2005, 2006]. Other hierarchical methods are contraction hierarchies (see [Geisberger et al., 2008, 2012]) and hub labelling (see [Gavoille et al., 2004, Thorup and Zwick, 2005]).

Approximating shortest paths using ML. Next to classical combinatorial approaches, there has also been great interest from the field of ML in finding approximates for the shortest path distance, using an ML perspective. For ex-

ample, Bagheri et al. [2008] compute shortest paths by using a genetic algorithm. Their algorithm works faster than Dijkstra, but they only test on small graphs with at most 80 nodes. Also, more recently, using ML techniques to approximate shortest path distances has led to interesting results. For example, Rizi et al. [2018] create an estimate for the shortest path distance between two nodes in a two-step procedure: first a deep learning vector embedding is applied and then a well-known landmark procedure afterwards (see, e.g., [Zhao et al., 2010, 2011]). Rizi et al. [2018] show results on large-scale real-world social networks with more than one million nodes. Their method differs from our approach in the sense that an algorithm is created to approximate shortest path distances in one specific large-scale real-world graph, opposed to an algorithm that can be used for any graph from a set of random graphs with similar properties.

3.1.3 Organization of Chapter

The chapter is organized as follows: In Section 3.2, we formally define the problem, describe the adapted Dijkstra algorithm on which our algorithms are based on, and introduce the random graph model that we use in this chapter. In Section 3.3, we introduce our new algorithm that combines the edge pruning idea implemented by the adapted Dijkstra algorithm with shortest path predictions and prove its correctness. We also give an UPDATE-PREDICTION procedure to handle underestimations of the shortest path distance. In Section 3.4, we elaborate on different prediction methods (both ML-based and based on breadth-first search); we remark that our algorithm can be used with arbitrary prediction algorithms. In Section 3.5 we describe our experimental setup and report on the respective findings. The code that we used to obtain the experimental results reported in Section 3.5 is available at[2]

`https://github.com/w-feijen/dijkstra-predictions-for-SSMTSP`.

In Section 3.6, we prove a lower bound on the number of saved queue operations if the edge weights are chosen at random. We apply this bound to estimate the savings on sparse random graphs.

3.2 Preliminaries

The *single-source many-targets shortest-path problem (SSMTSP)* has been defined in the introduction of the chapter. We use n and m to refer to the number of nodes and edges of the underlying graph G, respectively. For every node $v \in V$, we use $\delta(v)$ to denote the total weight of a shortest path (with respect to w) from s to v; if v cannot be reached from s we adopt the convention that $\delta(v) = \infty$.

[2]We are grateful to Ruben Brokkelkamp for creating an initial implementation of our algorithm in C++, which we used as a starting point to build the rest of our code around.

Given that all edge weights are non-negative, we thus have $\delta(v) \in \mathbb{R}_{\geq 0} \cup \{\infty\}$. Note that to solve the SSMTSP problem it is sufficient to compute the shortest path distances of all nodes $v \in V$ satisfying $\delta(v) \leq D$, where D is the minimum shortest path distance of a target node, i.e., $D = \min_{t \in T} \delta(t)$. Once these distances are computed, the actual shortest path can be extracted in linear time $O(n+m)$ by computing the *shortest path tree* rooted at s (see, e.g., [Cormen et al., 2022] for more details). Throughout this chapter, we assume that there is at least one target node in T that is reachable from s, which can be easily checked in linear time $O(n+m)$, simply by running a *breadth-first search (BFS)* [Cormen et al., 2022].

3.2.1 Dijkstra-Pruning Algorithm

As mentioned in the introduction of this chapter, our algorithm combines an adaptation of Dijkstra's algorithm by Bast et al. [2003] (referred to as Dijkstra-Pruning) with an ML-prediction. We briefly review Dijkstra-Pruning here.

We first describe the standard Dijkstra algorithm (referred to as Dijkstra), adapted to many targets. Dijkstra associates a *tentative distance* $d(v)$ with every node $v \in V$ and maintains the invariant that $d(v) \geq \delta(v)$ for every $v \in V$. Initially, $d(s) = 0$ and $d(v) = \infty$ for all $v \in V$. The set of nodes is partitioned into the set of *settled* and *unsettled* nodes. Initially, all nodes are unsettled, and whenever the algorithm declares a node v to be settled, its tentative distance is exact, i.e., $d(v) = \delta(v)$. The algorithm maintains a priority queue PQ to keep track of the distance labels of the unsettled nodes $v \in V$ with $d(v) \neq \infty$. Initially, only the source node s is contained in PQ. In each iteration, the algorithm removes from PQ an unsettled node u of minimum tentative distance, declares it to be settled and scans each outgoing edge $(u, v) \in E$ to check whether $d(v)$ needs to be updated; we also say that edge (u, v) is *relaxed* (pseudocode in Algorithm 4). The algorithm terminates when a node $u \in T$ becomes settled. Dijkstra performs at most n Remove-Min, n Insert and m Decrease-Prio operations. Its running time crucially depends on how efficiently these operations are supported by the underlying priority queue data structure. In this context, *Fibonacci heaps* introduced by Fredman and Tarjan [1987] are the (theoretically) most efficient data structure, supporting all these operations in (amortized) time $O(m + n \log n)$. It is important to realize though that the actual time needed by the queue operations depends on the size (i.e., number of elements) of the priority queue. In general, a smaller queue size results in a better overall running time of the algorithm.

An example of a directed graph $G = (V, E)$ for $V = \{s, a, b, c, t_1, t_2\}$ is given in Figure 3.1, in which s is the given source node and t_1 and t_2 are two target nodes. The weights of the edges are indicated in the figure. After the first iteration of Dijkstra, the source node s is settled with $\delta(s) = 0$. The priority queue is as follows: $((a, 1), (c, 2), (t_1, 4))$, where an entry in the priority queue is denoted by

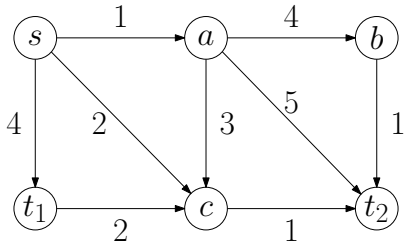


Figure 3.1: Example of a weighted directed graph with start node s and two target nodes t_1 and t_2.

$(v, \delta(v))$ for $v \in V$. At the end of the second iteration, node a is settled with $\delta(a) = 1$. The priority queue is as follows: $((c, 2), (t_1, 4), (b, 5), (t_2, 6))$. At the end of the third iteration, node c is settled with $\delta(c) = 2$. The priority queue is as follows $((t_2, 3), (t_1, 4), (b, 5))$. In the fourth iteration, the target node t_2 is removed from the priority queue and the algorithm terminates.

Dijkstra-Pruning works the same way as Dijkstra, but additionally keeps track of an *upper bound* B on the shortest path distance to a node in T. Initially, $B = \infty$ and the algorithm lowers this bound whenever a shorter path to a node in T is encountered. Crucially, $B \geq D$ always, and as a consequence, each edge $(u, v) \in E$ that leads to a finite tentative distance $d(v)$ with $d(v) \geq B$ can be discarded from further considerations; we also say that edge (u, v) is *pruned*. The pseudocode of Dijkstra-Pruning is given in Algorithm 3. Clearly, in the worst case, Dijkstra-Pruning does not prune any edges. In particular, the worst-case running time of Dijkstra-Pruning remains $O(m + n \log n)$.

For the example in Figure 3.1, Dijkstra-Pruning would update B to 4 in the first iteration during the scanning of edge (s, t_1). Bound B causes the edges (a, b) and (a, t_2) to be pruned in the second iteration. Subsequently, the nodes b and t_2 are therefore not inserted in the priority queue in that iteration, and instead the priority queue is as follows at the end of the second iteration: $((c, 2), (t_1, 4))$. At the end of the third iteration the priority queue is also smaller, namely: $((t_2, 3), (t_1, 4))$. Like Dijkstra, also Dijkstra-Pruning terminates in the fourth iteration.

3.2.2 Random Model

Bast et al. [2003] use the following random model to analyze the improved performance of Dijkstra-Pruning and show that the expected savings for these instances are significant, both analytically and empirically. The directed random graph instances are constructed using the *Erdös-Rényi random graph model* by Gilbert [1959], also known as $G(n, p)$: there are n nodes and each of the $n(n-1)$ possible (directed) edges is present independently with probability $p = c/n$, where

Algorithm 3 DIJKSTRA-PRUNING(G, w, s, T)

```
1: d(s) = 0, d(v) = ∞ for all v ∈ V \ {s}        ▷ tentative distances
2: B = ∞                                          ▷ pruning bound
3: PQ.INSERT(s, d(s))                             ▷ priority queue
4: while not PQ.EMPTY() do
5:     u = PQ.REMOVE-MIN()                        ▷ u becomes settled
6:     if u ∈ T then STOP                         ▷ stop when u is target
7:     for (u, v) ∈ E do
8:         tent = d(u) + w(u, v)                  ▷ tentative distance of v
9:         if tent > B then continue              ▷ prune edge (u, v)
10:        if v ∈ T then B = min{tent, B}         ▷ update B
11:        RELAX(u, v, tent)
```

Algorithm 4 RELAX$(u, v, tent)$

```
1: if d(v) > tent then                            ▷ lower tentative distance
2:     if d(v) = ∞ then
3:         PQ.INSERT(v, tent)                     ▷ add v to PQ
4:     else
5:         PQ.DECREASE-PRIO(v, tent)              ▷ decrease priority of v
6:     d(v) = tent                                ▷ update distance of v
```

c is (roughly) the average degree of a node. Further, each node $u \in V$ is chosen independently with probability $q = f/n$ to belong to the target set T, where f is the expected number of target nodes in T. The weight $w(e)$ of each edge e is chosen independently uniformly at random from the range $[0, 1]$.

3.3 Dijkstra's Algorithm with Predictions

Our basic idea is to further amplify the effect of the edge prunings by using a machine learning approach to obtain a prediction of the shortest path distance at an early stage. More concretely, suppose we have a PREDICTION algorithm which, based on the execution of the algorithm so far, computes an estimate of the shortest path distance D. We can then call this algorithm after a few iterations to obtain a prediction P of D and use it to prune all edges that lead to a tentative distance larger than P.

Suppose in the example given in Figure 3.1 there is a prediction $P = 3.5$ available from the start of the algorithm. This does not only prune the edges (a, b) and (a, t_2) like DIJKSTRA-PRUNING, but also edges (s, t_1) and (a, c) would be pruned because of the prediction.

Algorithm 5 DIJKSTRA-PREDICTION$(G, w, s, T, i_0, \alpha, \beta)$

```
1: d(s) = 0, d(v) = ∞ for all v ∈ V \ {s}          ▷ tentative distances
2: B = ∞, P = ∞, i = 0                             ▷ pruning bound, prediction and iteration
3: PQ.INSERT(s, d(s)), R = ∅                       ▷ PQ and reserve set R
4: while not PQ.EMPTY() and PQ.MIN-PRIO() ≤ P do
5:     u = PQ.REMOVE-MIN()                         ▷ u becomes settled
6:     if u ∈ T then STOP                          ▷ stop when u is target
7:     i = i + 1
8:     if i ≤ i_0 then x_{2i-1} = d(u), x_{2i} = B ▷ extend trace
9:     if i = i_0 then P = α · PREDICTION(x)       ▷ get prediction
10:    for (u, v) ∈ E do
11:        tent = d(u) + w(u, v)                   ▷ tentative distance of v
12:        if tent > B then continue               ▷ prune (u, v)
13:        if v ∈ T then B = min{tent, B}          ▷ update B
14:        RELAX-PREDICTION*(u, v, tent, P)        ▷ call relax routine
15: UPDATE-PREDICTION                              ▷ call update prediction routine
16: continue with while-loop
```

There are three main advantages from which our approach can (potentially) benefit when compared to the algorithms DIJKSTRA and DIJKSTRA-PRUNING. Firstly, fewer queue operations may be performed because of the edges being pruned. Secondly, edge pruning might start after a few iterations only, potentially before having found any path to a target node and finally, queue operations may take less time because the size of the priority queue remains smaller.

3.3.1 Detailed Description of DIJKSTRA-PREDICTION

We elaborate on our algorithm DIJKSTRA-PREDICTION (Algorithm 5) in more detail. The algorithm builds upon DIJKSTRA-PRUNING, see Section 3.2. The three new input parameters i_0, α and β will become clear below. During the first i_0 iterations, a vector $\mathbf{x} = (x_1, x_2, \ldots, x_{2i_0})$ is maintained for storing the *trace* (as we term it) of the algorithm. In iteration i_0, the constructed trace $\mathbf{x}$ is then used to compute an initial prediction by calling the PREDICTION procedure, for which several alternatives are given in Section 3.4. The algorithm keeps track of both the bound B on the smallest distance to a node in T encountered so far and the current prediction P. A scanned edge (u, v) is not inserted into the priority queue, we call this *pruning*, whenever its tentative distance $d(u)+w(u, v)$ exceeds B or P. There is a somewhat subtle point in the algorithm: Note that during the first i_0 iterations the prediction P remains at ∞ as the trace is just being built. As a consequence, throughout this stage it could happen that nodes

are inserted into the priority queue PQ, whose tentative distances are larger than the first prediction P (determined in iteration i_0). After this stage, this is impossible due to the pruning. It is because of these nodes that we have to add the second condition to the while loop, which checks whether the minimum distance of a node in PQ is less than the current prediction P. If not, the while loop has terminated without encountering a target node. In that case, the Update-Prediction procedure has to be initiated to increase the prediction and add all newly relevant nodes to PQ, as explained below.

Ideally, we would like to come up with a Prediction procedure that provides a prediction P which comes close to the actual shortest path distance D. In fact, both over- and underestimations of D can be harmful, though in different ways: If P overestimates D then edges that are irrelevant for the shortest path might not be pruned and the algorithm might perform redundant operations—which is undesirable. If P underestimates D then edges which are essential for the shortest path might be pruned and an incorrect solution might be returned—which is unacceptable. To remedy the latter, we equip our algorithm with an Update-Prediction procedure (Algorithm 7): If the prediction P turns out to be too small, it is increased by a factor $\beta > 1$ and the algorithm continues. Clearly, such Update-Prediction procedures should not happen too often as this might reduce the efficiency of the approach, therefore, the initial prediction is slightly inflated by a factor $\alpha \geq 1$. By inflating the prediction in an Update-Prediction routine, nodes that were previously considered irrelevant could potentially become relevant. Here a node is considered *relevant* if its tentative distance is smaller than the updated prediction. We need to insert the nodes that have become relevant during the Update-Prediction routine into the priority queue, we call this a *batch insertion*. During a batch insertion, we only insert a node if its tentative distance does not exceed the current upper bound B.

We are able to efficiently execute a batch insertion by maintaining a set of *reserve nodes*, R, during the algorithm. R will contain all nodes that have a finite tentative distance, but have not been added to the priority queue because their tentative distance exceeds the prediction in the current trial. Maintaining set R is done by using a different relax routine than Dijkstra-Pruning, namely Relax-Prediction and by using a hand-tailored data structure, both on which we elaborate below.

The Relax-Prediction routine is similar to the standard Relax routine (see Algorithm 6 and Algorithm 4). The main difference is that the node v is only inserted into the priority queue if its tentative distance is smaller than the prediction; otherwise, it is inserted into the reserve set R.

By using a tailored data structure for the reserve set, we can quickly execute the batch insertions. In this data structure, we store nodes based on their tentative distance, like in the priority queue. However, unlike in the priority queue, the nodes are not *sorted* based on this tentative distance. Instead, nodes are stored in several (doubly) linked lists, which we call *buckets*. Each bucket has a bucket

Algorithm 6 RELAX-PREDICTION$(u, v, tent, P)$

```
1: if d(v) > tent then                                   ▷ lower tentative distance
2:     if d(v) = ∞ then                                  ▷ v reached for the first time
3:         if tent ≤ P then PQ.INSERT(v, tent)           ▷ add v to PQ
4:         else R.INSERT(v, tent)                        ▷ add v to R
5:     else
6:         if v ∉ R then PQ.DECREASE-PRIO(v, tent)
7:         else                                          ▷ v is in R already
8:             if tent > P then                          ▷ v remains in R
9:                 R.DECREASE-PRIO(v, tent)
10:                continue
11:            R.REMOVE(v)                               ▷ move v from R to PQ
12:            PQ.INSERT(v, tent)
13:    d(v) = tent                                       ▷ update distance of v
```

Algorithm 7 UPDATE-PREDICTION

```
1: P = β · P                                             ▷ inflate prediction
2: for each v ∈ R do   ▷ iterate over all relevant nodes in R and do batch insertion
3:     if d(v) ≤ B then
4:         R.REMOVE(v)                                   ▷ move v from R to PQ
5:         PQ.INSERT(v, d(v))
```

number $j \in \mathbb{N}$, and a node will be stored in bucket $j = 1, 2, \ldots$ if and only if it has a tentative distance *tent* such that $\beta^{j-1}P < tent \leq \beta^j P$. Then, during the j'th batch insertion, all the nodes from bucket j can simply be moved from R into PQ.

3.3.2 Correctness Proof

As mentioned above, we insist that our algorithm is *correct* in the sense that it

1. terminates in polynomial time, and

2. computes an optimal solution to the SSMTSP problem.

The following theorem is proven by relating the runs of DIJKSTRA-PREDICTION and DIJKSTRA-PRUNING.

Theorem 3.3.1. DIJKSTRA-PREDICTION *is correct and has a worst-case running time of* $O(m + n \log n)$.

The proof of Theorem 3.3.1 follows directly from the following invariant, which establishes a connection between DIJKSTRA-PREDICTION with UPDATE-PREDICTION and DIJKSTRA-PRUNING, and from Lemma 3.3.3, which establishes that the reserve set operations do not increase the worst-case running time.

Invariant 3.3.2. *Consider the runs of* DIJKSTRA-PREDICTION *and* DIJKSTRA-PRUNING *on the same input instance. We use* d, PQ *and* R *to refer to the respective data structures in* DIJKSTRA-PREDICTION, *and* d', PQ' *to refer to the respective data structures in* DIJKSTRA-PRUNING. *The following properties are satisfied in each iteration:*

(P1) Both algorithms remove the same node u *from* PQ *and* PQ', *respectively.*

(P2) The set of nodes in PQ' *can be partitioned into the set of nodes in* PQ *and the set of nodes in* R *with* $d(v) > P$ *for all* $v \in R$.

(P3) The tentative distances are equal in both algorithms, i.e., $d(v) = d'(v)$ *for all* $v \in V$.

Proof:
We assume that both algorithms employ a consistent tie-breaking rule for nodes with similar distances. It is easy to see that the invariant holds for the first iteration: the prediction is initialized to $P = \infty$ and thus the algorithms do exactly the same, and R remains empty. Now, suppose by induction that the invariant holds at the beginning of iteration $i \geq 1$. We argue that the invariant holds at the end of iteration i:

(P1) Suppose that node u is deleted from PQ in iteration i. By the condition in the while-loop in Algorithm 5 it holds that $d(u) \leq P$, which together with (P2) gives that $d(u) < d(v)$ for all $v \in R$. By the REMOVE-MIN operation, $d(u) < d(v)$ for all $v \in \text{PQ}$, so $d(u) < d(v)$ for all $v \in \text{PQ} \cup R$. Since $\text{PQ} \cup R = \text{PQ}'$ (because of (P2)), we have $d(u) < d(v)$ for all $v \in \text{PQ}'$. From (P3), it follows that $d'(u) < d'(v)$ for all $v \in \text{PQ}'$, which proves that the same node is deleted in DIJKSTRA-PRUNING.

(P2) We consider each queue operation executed by the algorithms in this iteration separately and argue that the claim remains true. Firstly, if the claim holds at the beginning of an iteration, then it still holds after the REMOVE-MIN operation because the same node u is deleted from PQ and PQ'. Secondly, suppose that in iteration i node v is inserted into PQ' because edge (u, v) is relaxed. Then, before the insertion, $d'(v) = \infty$ and $d'(u) + w(u, v) < \infty$. Edge (u, v) will also be relaxed in the DIJKSTRA-PREDICTION algorithm. By (P3), we have $d(v) = \infty$ and $d(u) + w(u, v) < \infty$. This means that node v is either inserted into PQ or R. A node v is only added to R if $d(u) + w(u, v) > P$. So the claim still holds after

an insertion when $d(v)$ is set to $d(u) + w(u, v)$. Thirdly, suppose that in iteration i the tentative distance of node v in PQ′ is decreased because edge (u, v) is relaxed. Then, before the tentative distance is decreased, $d'(v) < \infty$ and $d'(u) + w(u, v) < d'(v)$. Again, edge (u, v) will also be relaxed in the DIJKSTRA-PREDICTION algorithm, and by (P3) we have $d(v) < \infty$ and $d(u) + w(u, v) < d(v)$. This means that node v will remain in PQ if it was already there, and will be moved from R to PQ if $d(u) + w(u, v) \leq P$. In both cases, the claim will remain true after the DECREASE-PRIO operation when $d(v)$ is set to $d(u) + w(u, v)$. Lastly, the property also remains true when P is inflated in the UPDATE-PREDICTION, procedure, since all $v \in R$ with $d(v) \leq P$ are moved to PQ and removed from R.

(P3) Both algorithms remove the same node u, update the tentative distance of u's neighbors based on the same condition and update it to the same value. So if the claim holds before the iteration, it will also hold at the end of the iteration.

□

[WF5]: Added this to make sure that running time remains linear

Lemma 3.3.3. *For fixed prediction P, [WF5] the worst-case running time of all operations on the reserve set R is $O(m + n)$.*

Proof:
If we let $w_{\max}$ be the maximum weight of an edge in the graph, then the maximum number of buckets we need is at most

$$\log_\beta \left(\frac{nw_{\max}}{P} \right) = \log \left(\frac{nw_{\max}}{P} \right) / \log(\beta).$$

We conclude that the UPDATE-PREDICTION procedure is called at most $\log((nw_{\max})/P)/\log(\beta)$ many times.

We analyze the complexity of three operations on the reserve set. Firstly, we can insert a node into the reserve set in constant time by calculating the bucket number with the given tentative distance, after which we can insert it into the correct linked list. Secondly, a decrease priority operation is done by deleting the node first, after which it is inserted with the updated priority. A decrease priority can therefore also be performed in constant time. Lastly, in the j'th UPDATE-PREDICTION procedure, all the nodes of the j'th bucket are deleted from the reserve list and inserted into the standard priority queue. Furthermore, we note that the number of insertions into R is bounded by the number of nodes, and the total number of decrease priorities is bounded by the number of edges in the graph. This proves that all operations on the reserve set can be done in $O(m + n + \log(w_{\max}))$ time. □

3.4 Prediction Methods

The PREDICTION algorithm used in our algorithm DIJKSTRA-PREDICTION can be obtained in numerous ways. Below, we explain how we obtain a prediction algorithm based on a machine learning approach. We elaborate on two different machine learning models and compare them to a benchmark prediction. Moreover, two alternative prediction methods based on breadth-first search (BFS) are given.

3.4.1 ML-Based Predictions

In order to make a prediction after i_0 iterations, we need to be able to describe the current optimization run by means of some characteristic features. One of the challenges here is to come up with features that capture the essence of the current run such that they can be used by the machine learning model to make a good prediction of the shortest path distance. We do this by keeping track of a lower and upper bound on the shortest path distance in each iteration. More precisely, in iteration $i \leq i_0$, the distance $d(u)$ of the node u extracted from the priority queue serves as the lower bound d_i and the current value of the pruning bound B is used as the upper bound B_i. The resulting sequence $\mathbf{x} = (d_1, B_1, d_2, B_2, \ldots, d_{i_0}, B_{i_0})$ of these lower and upper bounds for the first i_0 iterations then constitutes what we call the *trace* of the algorithm.

A *training sample* and *target* for the machine learning algorithm then consists of the trace $\mathbf{x}$ and the corresponding shortest path distance D, respectively. The set of samples for the machine learning models can be created by executing a run of DIJKSTRA-PRUNING on each problem instance of the training set. During this run, both the trace $\mathbf{x}$ and the final shortest path distance D need to be stored. Before the traces are used to train the machine learning models, we normalize each feature by subtracting the mean and dividing by the standard deviation. To prevent blowing up the mean value of the upper bound feature, all bounds B_i which are equal to the initial value of $B = \infty$ are set to 0.

We implemented and compared two standard machine learning models, namely a neural network model and a linear regression model, see Section 2.3 for a brief explanation of these models. The neural network model that we use is a straightforward *multilayer perceptron network* consisting of two hidden layers, for which we optimize the number of nodes per layer by a *k-fold cross validation* (see, e.g., [Refaeilzadeh et al., 2009] for more details). To verify whether anything has been learned by these models at all, the results for these models are compared with a straightforward benchmark prediction. This benchmark prediction, which is independent of the instance, is computed by taking the average of the shortest path distance for each instance in the training set. The results for this validation can be found in Section 3.5.2.

3.4.2 BFS-Based Predictions

As an alternative to the machine learning models given in Section 3.4, we implement two prediction methods that are based on breadth-first search (BFS). For each instance, we can simply run a BFS from the source node s to determine a path P^{BFS} to any of the target nodes having the smallest number of edges. We also call P^{BFS} a *BFS-path* and use L^{BFS} to denote its length (i.e., number of edges). Note that, equivalently, L^{BFS} is the shortest path distance to any of the target nodes if all edge weights are set to 1. We use this BFS-path to derive two different predictions of the actual shortest path distance D: (i) BFS: We define the prediction P as $L^{BFS} \cdot \mu_w$, where μ_w is the expected edge weight. (ii) w-BFS: We define the prediction P as the sum of the *actual* weights on P^{BFS}, i.e., $P = \sum_{e \in P^{BFS}} w(e)$. Note that the latter prediction might overestimate but never underestimate the actual distance D.

3.5 Experimental Findings

In this section, we present our experimental findings. We first introduce our experimental setup and then discuss the results and insights we obtained from the experiments.

3.5.1 Experimental Setup

We generated 100,000 instances of the SSMTSP problem using the random model described in Section 3.2 with $n = 1000$ nodes, edge probability $p = c/n$ with $c = 8$, and target probability $q = f/n$ with $f = 20$. The edge weights were chosen independently uniformly at random from $[0, 1]$. Further, we fixed the length of the constructed traces to $i_0 = 10$. We only accepted an instance if Dijkstra-Pruning executed more than i_0 iterations to ensure that Dijkstra-Prediction reaches the point where a prediction is made. This set of 100,000 instances was split into a *training set* of 80,000 instances used for building the machine learning models, a *validation set* of 10,000 instances used for parameter tuning and a *test set* of 10,000 instances used for the final experiments.

To get an idea of a few parameters related to the shortest path distance in the generated instances, we provide some statistical data for the validation and test set in Table 3.1. We computed the average number of edges on a shortest path, the average, minimum and maximum cost of a shortest path, and the average weight of an edge on a shortest path. Note that the first row refers to these parameters with respect to the actual random weights, while the second row refers to the case when all edge weights are set to 1.

WEIGHTS	EDGES	MEAN	MIN	MAX	w-MEAN
RANDOM	4.363	0.553	0.065	1.848	0.127
UNIT	2.225	1.154	0.129	3.217	1.000

Table 3.1: Statistics on different shortest path parameters for 20,000 instances (validation and test set).

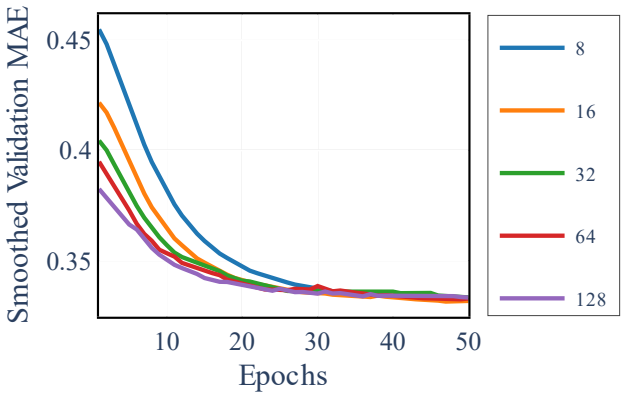


Figure 3.2: Normalised smoothed mean absolute error of the validation set for different hidden layer sizes.

3.5.2 Machine Learning Results

For each of the 80,000 graphs in the training set, we executed a run of Dijkstra-Pruning, during which we stored both the features $\mathbf{x}$ and the final returned shortest path distance y. This process yielded a training set of 80,000 samples, where each sample was of shape $(\mathbf{x}, y)$. We used these 80,000 samples to built both a neural network and a linear regression model.

The number of nodes in the two hidden layers of the neural network was optimized by minimizing the mean absolute error (MAE) in a k-fold cross-validation with $k = 4$ and a batch size of 256. We tested layer sizes $8, 16, 32, 64$ and 128; the results for the smoothed validation Mean Absolute Error can be found in Figure 3.2. We decided to use 16 nodes per layer and train for 47 epochs. We did not use the validation set of 10,000 instances in the k-fold cross-validation, since it was used to tune parameters α and β. We also tested a linear regression model and the averaging benchmark prediction, but the neural network performed best in both the training and the test set.

The results for the performance of the neural network, linear regression and the benchmark are given in Table 3.2. Both machine learning models perform better than the benchmark prediction, in both the training and test set. Moreover, the neural network outperforms the linear regression model on both the training and

Algorithm	NN	LR	AB
Train MAE	0.0619	0.0883	0.1469
Train MAPE	0.1228	0.1844	0.3150
Test MAE	0.0617	0.0880	0.1477
Test MAPE	0.1217	0.1837	0.3160

Table 3.2: Mean absolute error (MAE) and mean absolute percentage error (MAPE) for neural network (NN), linear regression (LR) and averaging benchmark (AB).

the test set.

3.5.3 Benchmark Algorithm Oracle

In order to assess the performance of the different algorithms, we decided to use the following (idealized) benchmark algorithm to compare against: We run Dijkstra-Pruning with the pruning bound B being *initialized* with the actual shortest path distance D. We refer to this algorithm as Oracle.

Note that this algorithm only inserts nodes into the priority queue which are necessary for finding the shortest path distance D. Said differently, the algorithm spends the minimum possible amount of work to provide a certificate of optimality for the shortest path distance D; no other algorithm could spend less work (as long as we insist that the shortest path distance is always computed correctly).

We want to stress that the Oracle algorithm is a hypothetical algorithm, in the sense that it uses information, namely the shortest path distance, that would usually be unknown during solving the SSMTSP. The oracle can therefore only be used to test the potential of the Dijkstra-Pruning, but it should not be considered a fair competitor of the algorithms that use actual predictions.

3.5.4 Parameter Tuning

There are two parameters in the Update-Prediction procedure that decide how to handle the machine learning prediction. The first one is α, which specifies the amount by which the initial prediction is inflated. The second one is β, the amount by which the prediction is inflated in the Update-Prediction algorithm. We analyzed the impact of these parameters on the queue size; see Figure 3.3. Both figures depict the queue size for the same instance, but with different values for α and β. On the left, we fixed $\beta = 1.3$ and varied α; on the right, we fixed $\alpha = 1.3$ and varied β. As is visible from these plots, a larger α means that the first call of Update-Prediction occurs later. Also, a larger β leads to a larger number of nodes inserted during Update-Prediction.

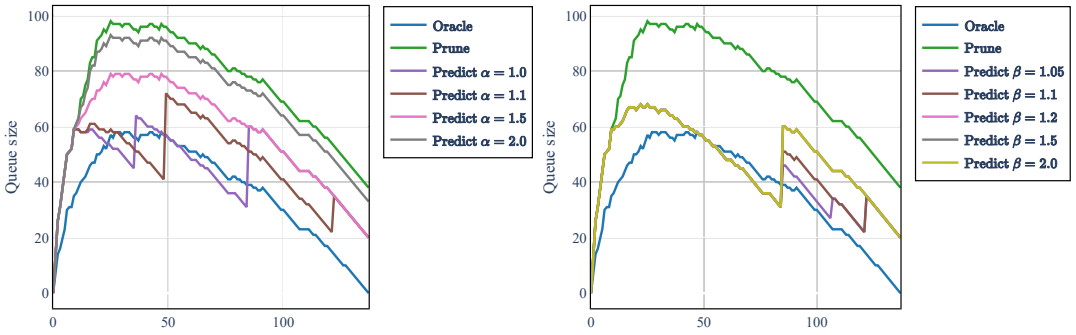


Figure 3.3: The queue size for fixed $\beta = 1.3$ and varying α (left) and fixed $\alpha = 1.3$ and varying β (right) for Dijkstra-Prediction. The x-axis refers to the number of iterations.

$\alpha \backslash \beta$	1.05	1.10	1.20	1.50	2.00
1.00	**153.08**	153.59	154.90	158.55	160.20
1.05	154.37	154.80	155.83	158.08	158.84
1.10	156.06	156.37	157.06	158.39	158.74
1.20	160.38	160.50	160.82	161.20	161.30
1.50	172.57	172.58	172.58	172.61	172.61
2.00	182.27	182.27	182.27	182.27	182.27

Table 3.3: Average number of queue operations (Q) for Dijkstra-Prediction for different values of α and β on the validation set.

In the results for parameter tuning, and also in the next results section, we have used the cumulative queue size to compare the outcomes of different algorithms. It is defined as the sum of the number of elements in the queue over all the iterations and denoted with C. Pruning more edges will result in fewer nodes in the queue, resulting in a smaller queue size.

We tested several configurations for α and β on the instances in the validation set. Table 3.3 and Table 3.4 state the respective number of queue operations Q and the cumulative queue size C for various choices. As it turns out, for both these performance indicators it is best to choose α and β small.

3.5.5 Discussion of Results

After tuning the parameters α and β, we ran our algorithms on the generated instances. For four of these instances, the size of the queue throughout the algorithm is illustrated in Figure 3.4. We start by considering how the queue sizes differ for the different algorithms; see the top two rows in Figure 3.4. As to be expected, the queue size of Dijkstra-Prediction never exceeds the one

$\alpha\backslash\beta$	1.05	1.10	1.20	1.50	2.00
1.00	**2446.0**	2561.9	2740.4	3115.7	3274.6
1.05	2573.5	2669.2	2815.5	3067.5	3150.1
1.10	2732.6	2805.3	2914.2	3065.1	3104.4
1.20	3112.6	3148.4	3197.7	3251.9	3266.3
1.50	4104.3	4106.6	4109.5	4114.2	4114.2
2.00	4809.9	4809.9	4809.9	4809.9	4809.9

Table 3.4: Average cumulative queue size (C) for Dijkstra-Prediction for different values of α and β on the validation set.

of Dijkstra-Pruning and is closer to the one of Oracle. The improvement of our Dijkstra-Prediction with respect to Dijkstra-Pruning varies and depends on the instance. The top-left figure shows that Dijkstra-Pruning was a great improvement compared to Dijkstra, but Dijkstra-Prediction, did not decrease the queue size much more. This figure shows that there was an Update-Prediction around iteration 42 and around iteration 55, which can be seen from the sudden increase in the queue size. In the top-right figure, the benefit of Dijkstra-Prediction is much larger, and the queue size of Dijkstra-Prediction comes close to the oracle. Next, we comment on the Update-Prediction procedure; see the bottom row of Figure 3.4. On the left, no restart was necessary since the prediction was not an overestimation of D. This plot also indicates the interplay of the prediction P and the pruning bound B; first the former (iterations 10 to 14) and later the latter (iterations 15 and onwards) providing the smaller upper bound. On the right, we needed to do several Update-Prediction because the initial prediction turned out to be too small. Update-Prediction adds some nodes from the reserve list to the priority queue (queue size increases) and continues. After several inflations of the prediction with β, the prediction was sufficiently large to find the shortest path distance.

If we zoom in to obtain a more fine-grained picture of the different queue operations executed by the algorithms, the results are as specified in Table 3.5. The respective rows state the number of Remove-Min (RM), Insert (IS) and Decrease-Prio (DP) operations, the total number of queue operations (Q), the number of trials (T), the cumulative queue size C (over all iterations), and the cumulative queue size relative to the Oracle $\bar{C}$. The cumulative queue size relative to the Oracle gives a quick overview on how the algorithm performs in relation to the Oracle i.e., the algorithm in which the predictions are perfect. The table shows the averages for all 10,000 graphs in the test set.

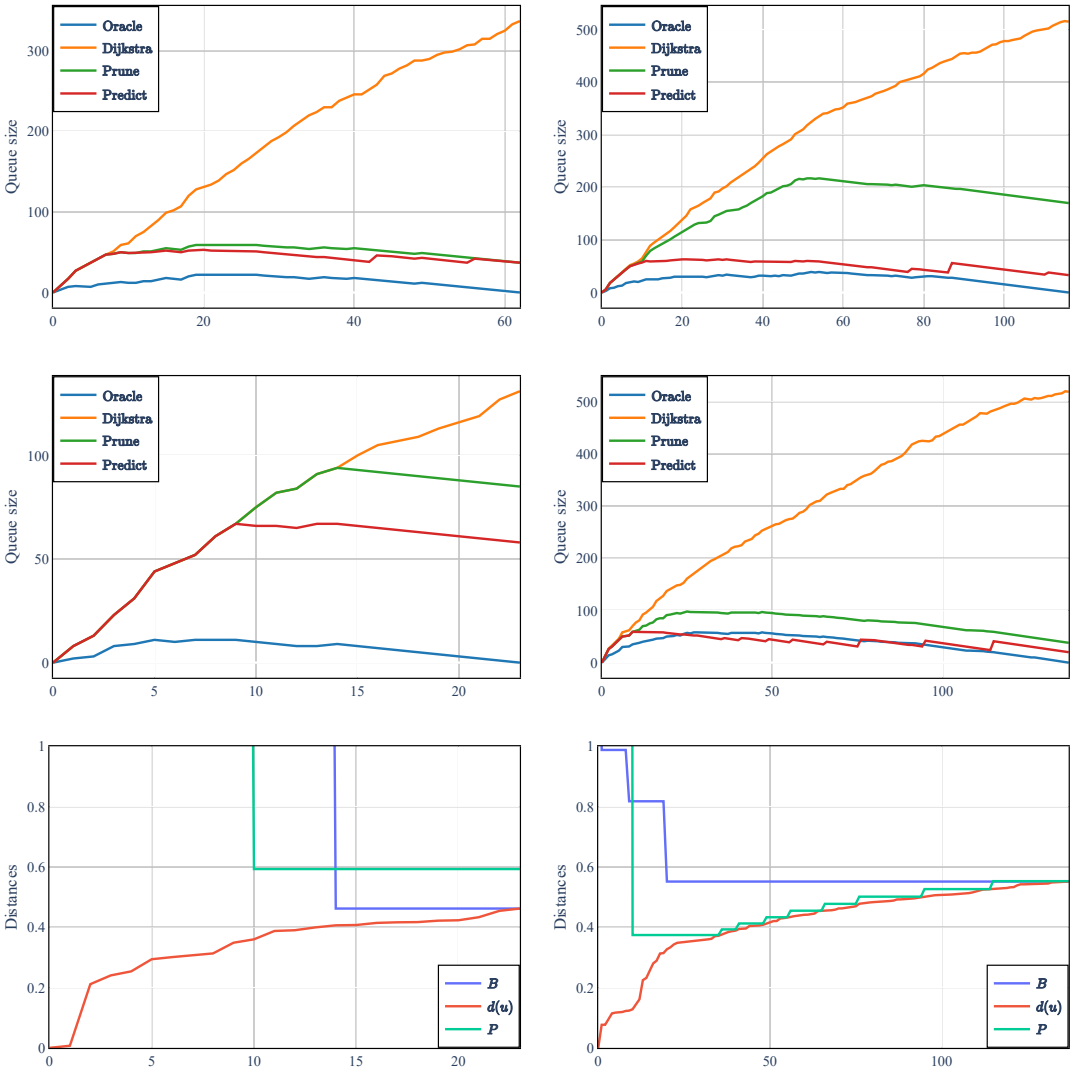


Figure 3.4: (In all plots the x-axis refers to the number of iterations.) Top two rows (four distinctive instances): queue size of the algorithms. Bottom row (same instances as in the second row): distance $d(u)$ together with B and P for the Dijkstra-Prediction.

As to be expected, Oracle inserts and removes the minimum possible number of nodes only. As can also be inferred from Invariant 3.3.2, Dijkstra, Dijkstra-Pruning and Dijkstra-Prediction have the same number of Remove-Min operations. Observe that the results show that our algorithm Dijkstra-Prediction outperforms all other algorithms, both in terms of the total number of queue operations and cumulative queue size. Dijkstra-Prediction outperforms Dijkstra-Pruning mostly on the number of insertions, as expected from the analysis in Section 3.6. In terms of cumulative queue size, our algorithm

	Oracle	Dijks	Prune	Prediction	bfs	w-bfs
RM	59.39	59.39	59.39	59.39	59.39	59.39
IS	59.39	335.50	122.91	91.73	117.66	118.86
DP	0.78	43.96	5.87	2.89	5.29	5.46
Q	119.55	438.85	188.17	154.01	182.33	183.70
T	1.00	1.00	1.00	2.28	1.23	1.00
C	1456.16	13949.37	5245.96	2476.35	4825.09	4980.69
$\bar{C}$	1.00	9.58	3.60	1.70	3.31	3.42

Table 3.5: Number of Remove-Min (RM), Insert (IS) and Decrease-Prio (DP) operations, the total number of queue operations (Q), the number of trials (T), the cumulative queue size C (over all iterations), and the cumulative queue size relative to the Oracle $\bar{C}$ for Dijkstra, Dijkstra-Pruning, Dijkstra-Prediction and the oracle algorithm. Averaged over all graphs in the test set.

	Oracle	Dijks	Prune	Prediction
run time	2.91	14.51	5.29	4.80

Table 3.6: Run time in seconds for 10.000 graphs in test set for Dijkstra, Dijkstra-Pruning, Dijkstra-Prediction and the oracle algorithm. Summed over all graphs in test set.

Dijkstra-Prediction even comes close to the benchmark Oracle, the average cumulative queue size being only 1.7 times larger than the one of Oracle; Dijkstra-Pruning perform much worse, being off by a factor 3.6.

Next to counting the queue operations, which is a reliable measure in the sense that it is machine-independent, we also measured the run time of the different algorithms. The sum of the run times for the graphs in the test set can be seen in Table 3.6. The Dijkstra-Prediction outperforms Dijkstra-Pruning, which on itself is already a great improvement over Dijkstra. As expected, the hypothetical Oracle algorithm beats them all, and shows the queue size required in the case of perfect predictions. Even though the Dijkstra-Prediction algorithm needs time to store the trace, make the prediction and keep track of the reserve set, the benefit that the prediction provides is sufficient to improve Dijkstra-Pruning.

3.5.6 Results for Different Graph Parameters

In all results so far in this chapter, we used a fixed set of parameters for the random graph model, namely an average degree c of 8, and a target probability q of 0.02. A natural question that might arise is how our algorithm performs on

	$q = 0.02$			0.06			0.18		
	DIJK	PRUN	PRED	DIJK	PRUN	PRED	DIJK	PRUN	PRED
$c = 2$	2.20	1.57	0.40	2.55	1.61	0.90	2.84	1.66	1.49
5	6.10	2.62	1.01	7.71	2.92	2.19	8.86	2.95	2.82
8	9.58	3.60	1.82	12.97	4.13	3.35	15.02	3.64	3.50
16	16.04	4.76	2.84	24.78	5.26	4.51	31.20	5.07	4.92
32	22.55	6.05	4.15	43.21	6.76	5.95	57.54	6.03	5.89

Table 3.7: Cumulative queue sizes relative to the the cumulative queue size of ORACLE ($\bar{C}$) for DIJKSTRA, DIJKSTRA-PRUNING and DIJKSTRA-PREDICTION algorithm, for different random graph parameters c and q.

a less specific random graph structure. To answer this question, we built a new ML model, which is based on graphs with various random graph parameters, as opposed to the single setting it relied on before. This new ML model showed us that, even though it is based on graphs with various input parameters, DIJKSTRA-PREDICTION is still able to reduce the cumulative queue size.

By taking c from $\{2, 5, 8, 16, 32\}$ and q from $\{0.02, 0.06, 0.18\}$, we created 15 pairs of random graph parameters. For each of these pairs, we constructed 80,000 graphs, which together formed a large training set of 1.2 million instances. We created a machine learning prediction model based on this training set, as explained before in Section 3.5.2.

For each of the pairs of random graph parameters, we performed the ORACLE, DIJKSTRA, DIJKSTRA-PRUNING and DIJKSTRA-PREDICTION algorithm and compared the cumulative queue size of each algorithm to that of the ORACLE. Table 3.7 shows the average relative cumulative queue size of $1,000$ instances. These results reveal two things. Firstly and crucially, for each pair of random graph parameters, DIJKSTRA-PREDICTION is able to reduce the cumulative queue size compared to DIJKSTRA-PRUNING. This means that our algorithm does not lose its power to decrease the cumulative queue size, even when it is used on less specific random graph structures. For lower values of q, which means there are fewer target nodes in the instances, DIJKSTRA-PREDICTION has a larger improvement over DIJKSTRA-PRUNING than for larger values of q. A second observation is that the relative cumulative queue size can be lower than 1.0. For example, when $c = 2$ and $q = 0.02$, DIJKSTRA-PREDICTION has a smaller cumulative queue size than ORACLE. This seems unexpected at first, but can be explained by the fact that on average 4.83 UPDATE-PREDICTION routines are executed for these graphs. This fairly large number of UPDATE-PREDICTION routines shows that the prediction was significantly too low and had to be increased several times. The low prediction caused the queue size to be low as well, which explains the small $\bar{C}$.

3.6 Lower Bounds on Savings

In this section, we derive our lower bounds on the savings of Dijkstra-Prediction. We assume without loss of generality that all edge weights are normalized such that $w(u, v) \in [0, 1]$ for all $(u, v) \in E$. Further, for the sake of the analysis, we assume that Dijkstra-Prediction starts with a prediction $P = D + \varepsilon$, where $\varepsilon \geq 0$ is the additive error of the prediction. In particular, we assume that the algorithm starts with this prediction from the beginning (while it actually only becomes available after i_0 many iterations); but given that i_0 is small, this assumption is negligible. Note that this clearly captures the case when P is an overestimation of the actual distance D. However, our analysis also provides bounds on the number of priority queue operations when P is an underestimation of D. Let P_{under} be such an underestimation. Running Dijkstra-Prediction with $i_0 = 0$ and $P = P_{under}$ will result in consecutive calls of Update-Prediction until P exceeds D for the first time. Let P_{over} be the value of P at this point in the algorithm. Now, we can compare the number of queue operations between a run of Dijkstra-Prediction that makes a prediction $P = P_{under}$ and a run of Dijkstra-Prediction that makes a prediction $P = P_{over}$. It holds that in the run with $P = P_{under}$ each node can only be inserted at a later stage into the priority queue than in the run with $P = P_{over}$. It follows that the number of queue operations in the run starting with the underestimation is at most the number of queue operations in the run starting with the overestimation.

3.6.1 Worst-Case Instances

We first show that Dijkstra-Prediction might not prune a single edge, even if the prediction is perfect (i.e., $\varepsilon = 0$). To see this, suppose that an adversary can fix the entire input instance. Consider the instance depicted in Figure 3.5 (t being the only target node), parameterized by $\varepsilon > 0$. A moment of thought reveals that, in the general case, a necessary condition for an edge $(u_1, v_i) \in E$ to be pruned is that it belongs to the set L, defined as:

$$L := \{(u, v) \in E \ : \ d(u) \leq D \text{ and } d(u) + w(u, v) > D\}.$$

In Figure 3.5, L is indicated in grey. However, none of these edges will be pruned, neither because of B nor P, since the tentative distance of each v_i is $1 + \varepsilon/2$, $B = \infty$ and $P = 1 + \varepsilon$. This holds even if the prediction is perfect (i.e., $\varepsilon = 0$). The point here is that the distance $d(u_1) = \varepsilon$ of the start node u_1 is small, and thus the tentative distance of v_i cannot exceed P.

With this insight, we can strengthen the necessary condition for an $(u, v) \in E$ to be pruned. Not only must (u, v) be in L, it must also hold that $d(u) > D-1+\varepsilon$. Based on a fixed threshold for $d(u)$, namely $\theta \in (D - 1 + \varepsilon, D]$, we define the set

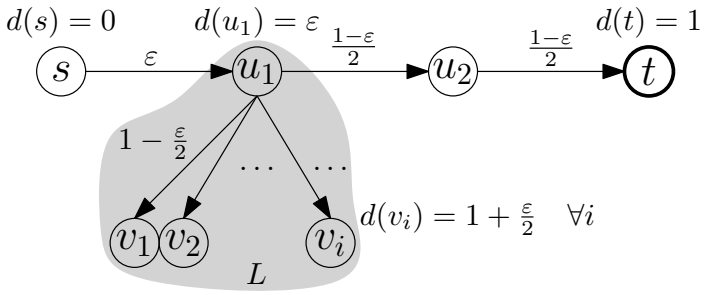


Figure 3.5: Instance in which no savings can be expected from our algorithm. There is no pruning, even if predictions are perfect ($\varepsilon = 0$).

of *relevant* edges as the set for which this necessary condition holds:

$$L_\theta := \{(u, v) \in E \ : \ \theta \leq d(u) \leq D \text{ and } d(u) + w(u, v) > D\}. \tag{3.2}$$

Suppose that an adversary can fix the instance as before, but now we enforce it to have many relevant edges. Even then, none of these relevant edges might be pruned. To see this, consider the partial instance depicted in Figure 3.6. Here, the nodes u_i removed from the priority queue are sorted by increasing distances $d(u_i)$ (from left to right); only the u_i's are shown for which $d(u_i) > D - 1 + \varepsilon$. Also, only the relevant edges in L_θ are shown (indicated in grey). Without further restrictions, the adversary can still fix the weights of the relevant edges in L_θ to be $D + \varepsilon/2 - d(u_i)$. Like in the previous example, none of these edges will be pruned, neither because of B nor P, since $d(v_i) = D + \varepsilon/2$, B can be as large as ∞ and $P = D + \varepsilon$. Note that this holds even for perfect predictions and θ being arbitrarily close to D (i.e., $\varepsilon = 0$ and $\theta \to D$).

The conclusion to draw from these examples is that our algorithm might not save on priority queue operations at all in the worst case. In essence, the crux here is that even though we have perfect information about the shortest path distance, this is not enough to speed-up the construction of the optimality certificate. Note that for both instances the distances of all nodes need to be determined correctly to obtain such a certificate. Given that DIJKSTRA already uses the minimum number of priority queue operations to compute such a certificate, we cannot hope to improve on this.

3.6.2 Partial Random Instances

Based on these examples, it is clear that we need to further restrict the power of the adversary. We therefore introduce randomness in the instances to obtain a more fine-grained understanding of the savings achieved by our algorithm. Generally speaking, we will do this by enforcing randomness on some of the edges, while allowing the adversary to still control the rest of the input instance.

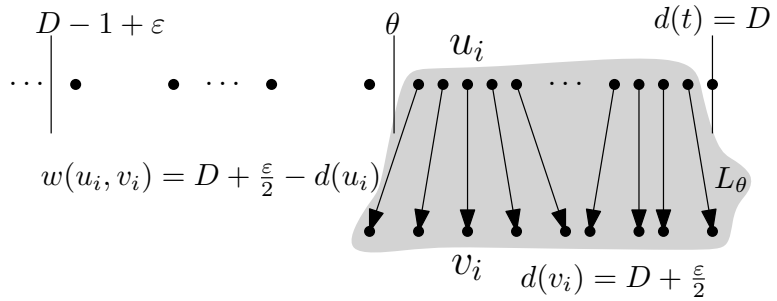


Figure 3.6: Instance in which no savings can be expected from our algorithm. There is no pruning, even if many relevant edges $(u_i, v_i) \in L_\theta$.

The setup is as follows: We suppose that the adversary can fix the set $Q = \{u_1, \dots, u_l\}$ of nodes that are removed from the priority queue, where u_i is the node removed in iteration i, and the corresponding distances $d(u_1) \leq \cdots \leq d(u_l)$ of these nodes. [WF6] Further, the adversary can fix all outgoing edges of the nodes in Q. Note that by doing so we implicitly allow that the adversary can fix the weights of certain edges to enforce this configuration (because these weights determine the order in which the nodes are removed from the priority queue). Crucially, however, we do not allow the adversary to fix the weights of the relevant edges in L_θ: the weight of each relevant edge in L_θ is random. In particular, in this partially random setting, we assume that the weights for all edges $(u, v) \in L_\theta$ are independent and the distance labels $d(u)+w(u, v)$ are uniform on $[D, d(u)+1]$. [WF7]

[WF6]: what does this mean for D

[WF7]: This means for the edge weight that:

Given this partially random adversarial setting, we can lower bound the probability of pruning an edge in L_θ. Like in the instances in Figure 3.5 and Figure 3.6, the adversary still has enough power to set the instance such that none of the edges in L_θ are pruned based on the bound B. In contrast, each of the edges in L_θ is pruned with positive probability due to the prediction P, due to the randomness assumption. By definition of L_θ and θ, we know that for the edges in L_θ the necessary condition for pruning, $d(u) > D + 1 - \varepsilon$, holds.

Lemma 3.6.1. *Suppose L_θ is defined as above, with $\theta = D + \gamma\varepsilon - 1$, for some $\gamma \in (1, \frac{1}{\varepsilon}]$. Let X_e be a random variable which equals 1 if edge $e \in L_\theta$ is pruned and zero otherwise. Then $\mathbf{P}(X_e = 1) \geq 1 - \frac{1}{\gamma}$.*

Proof:
Let $e = (u, v) \in L_\theta$ be a relevant edge and let $d(v)$ be the distance of v at the end of the algorithm. Then e is pruned whenever the tentative distance $tent := d(u) + w(u, v)$ exceeds the prediction P. As argued before, $e \in L_\theta$ implies that $d(u) + w(u, v)$ is uniformly distributed on $[D, d(u) + 1]$. Furthermore, note that $1 - (D - d(u)) \geq \gamma\varepsilon$ follows from $d(u) \geq \theta$ and the definition of θ and note that $D + \varepsilon$ is contained in the interval $[D, d(u) + 1]$, since $D + \varepsilon \leq D + \gamma\varepsilon \leq$

$d(u)+1$. We can simply apply the cumulative distribution function for the uniform distribution:

$$\begin{aligned}\mathbf{P}(X_e = 1 \mid e \in L_\theta) &\geq \mathbf{P}(tent \geq P \mid d(v) \geq D) \\ &= \mathbf{P}(d(u_i) + w(e) \geq D + \varepsilon \mid d(v) \geq D) \\ &= \mathbf{P}(w(e) \geq D - d(u_i) + \varepsilon \mid w(e) \geq D - d(u_i)) \\ &= \frac{1 - (D - d(u_i) + \varepsilon)}{1 - (D - d(u_i))} = 1 - \frac{\varepsilon}{1 - (D - d(u_i))} \\ &\geq 1 - \frac{\varepsilon}{\gamma\varepsilon} = 1 - \frac{1}{\gamma}.\end{aligned}$$

$$\begin{aligned}\mathbf{P}(X_e = 1) = \mathbf{P}(tent \geq P) &= \mathbf{P}(d(u) + w(u,v) \geq D + \varepsilon) \\ &= \frac{d(u) + 1 - (D + \varepsilon)}{d(u) + 1 - D} \geq 1 - \frac{\varepsilon}{\gamma\varepsilon} = 1 - \frac{1}{\gamma}.\end{aligned}$$

□

We give some intuition for the lemma: the number of edges in L_θ is largest when γ is close to 1, and the lemma shows there is a small positive probability for each of those to be pruned by the prediction. As γ increases to $\frac{1}{\varepsilon}$, the threshold θ approaches D and the size of L_θ decreases. So the number of relevant edges decreases, while the probability that each such edge is pruned increases.

Define X as the total number of pruned edges in L_θ, i.e., $X = \sum_{e \in L_\theta} X_e$. We can lower bound both the expectation of X and X with high probability, which is formalized in the following theorem.

Theorem 3.6.2. *Suppose L_θ is defined as above, with $\theta = D + \gamma\varepsilon - 1$, for some $\gamma \in (1, \frac{1}{\varepsilon}]$. Then the expected number of pruned edges in L_θ is*

$$\mathbf{E}[X] \geq \left(1 - \frac{1}{\gamma}\right) |L_\theta|.$$

Further, if $(1 - \frac{1}{\gamma})|L_\theta| \geq 8 \ln n$, then

$$X \geq \frac{1}{2}(1 - \frac{1}{\gamma})|L_\theta|$$

with high probability, i.e.,

$$\mathbf{P}\left(X \geq \frac{1}{2}\left(1 - \frac{1}{\gamma}\right)|L_\theta|\right) \geq 1 - \frac{1}{n}.$$

Proof:
By linearity of expectation, it follows from Lemma 3.6.1 that:

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{e \in L_\theta} X_e\right] = \sum_{e \in L_\theta} \mathbf{E}[X_e] = \sum_{e \in L_\theta} \mathbf{P}(X_e = 1).$$

Note that $X = \sum_{e \in L_\theta} X_e$ is a sum of $|L_\theta|$ independent random variables. Let $\mu := \mathbf{E}[X]$ be the expected value of X. The following (standard) Chernoff bound holds for every $\delta \in (0, 1)$:

$$\mathbf{P}(X \leq (1-\delta)\mu) \leq e^{-\mu\delta^2/2}.$$

By choosing $\delta = \frac{1}{2}$, we obtain

$$\mathbf{P}\left(X \leq \frac{1}{2}\mu\right) \leq e^{-\mu/8} \leq \frac{1}{n},$$

where the second inequality holds because $\mu = \mathbf{E}[X] \geq (1 - \frac{1}{\gamma})|L_\theta| \geq 8 \ln n$ by our assumption.

Using this, we conclude that

$$\mathbf{P}\left(X \geq \frac{1}{2}\left(1 - \frac{1}{\gamma}\right)|L_\theta|\right) \geq \mathbf{P}\left(X \geq \frac{1}{2}\mu\right) \geq 1 - \frac{1}{n}.$$

□

3.6.3 Random Model

We consider random instances constructed according to the model introduced in Section 3.2. For these instances, we show that our Dijkstra-Prediction algorithm saves a significant number of queue operations compared to Dijkstra-Pruning.

In the random model, it is not straightforward to compute the size of L_θ. Therefore, we will consider a specific subset of L_θ for which we *are* able to compute the size. More specifically, we only consider the edges (u, v) from L_θ for which the *final* distance $d(v)$ is larger than D, i.e., edges from L_θ that lead to a node v which is not in Q. Note that L_θ may contain edges (u, v) with tentative distance $d(u) + w(u, v) > D$, but whose final distance $d(v) \leq D$. These are relevant edges having both endpoints in Q that might be pruned. However, we do not account for these savings in our analysis here. Also, note that there could be multiple edges in L_θ that lead to such a node $v \notin Q$. In that case, we only consider the (unique) edge in L_θ which has led to an insertion of v into the priority queue in the standard Dijkstra algorithm (disregarding edges that have led to a decrease priority operation). We use L'_θ to denote this subset of L_θ:

$$L'_\theta := \{(u, v) \in L_\theta \ : \ v \notin Q, \ (u, v) \text{ leads to insertion of } v \text{ in PQ of Dijkstra}\}.$$

In the Dijkstra algorithm, all the end nodes of edges in L'_θ are inserted in the priority queue, but they are never removed. Dijkstra-Prediction can actually save a number of these insert operations by pruning the edges in L'_θ. We will

lower bound these savings by computing an upper bound for the number of these nodes which are still inserted in our DIJKSTRA-PREDICTION algorithm. Consequently, all the edges which lead to nodes which are not inserted in DIJKSTRA-PREDICTION are pruned.

First, we will prove the following Key Lemma which will help us to upper bound the probability of inserting an endpoint of an edge in L'_θ below.

Lemma 3.6.3: (Key Lemma). *Let $X_j, j = 1, \ldots, k+1$, be $k+1$ uniform random variables, with X_j uniform on $[a, b_j]$ and $b_1 \leq \cdots \leq b_{k+1}$. Let $P > 0$ be a real number, which is contained in all intervals, i.e., $a < P < b_1$. Let $\mathcal{E}_1$ be the event $\{X_{k+1} \leq X_j, \forall j \in [k]\}$ and let $\mathcal{E}_2$ be the event $\{X_{k+1} \leq P\}$. Then:*

$$\mathbf{P}(\mathcal{E}_1 \wedge \mathcal{E}_2) \leq \frac{1}{k+1}\left(1 - \left(1 - \frac{P-a}{b_k - a}\right)^{k+1}\right).$$

Proof:
We will upper bound the probability by conditioning on values of X_{k+1}, using the law of total probability and applying the density function of X_{k+1}: $f_{X_{k+1}}(s) = \frac{1}{b_{k+1}-a}$.

$$\begin{aligned}\mathbf{P}(\mathcal{E}_1 \wedge \mathcal{E}_2) &= \int_a^{b_{k+1}} \mathbf{P}(\mathcal{E}_1 \wedge \mathcal{E}_2 \mid X_{k+1} = x) f_{X_{k+1}}(x)dx \\ &= \frac{1}{b_{k+1}-a}\int_a^{b_{k+1}} \mathbf{P}(\mathcal{E}_1 \wedge \mathcal{E}_2 \mid X_{k+1} = x)dx\end{aligned}$$

Since $\mathbf{P}(\mathcal{E}_1 \wedge \mathcal{E}_2 \mid X_{k+1} = x) = 0$ if $x > P$, we can write:

$$\begin{aligned}\mathbf{P}(\mathcal{E}_1 \wedge \mathcal{E}_2 \mid X_{k+1} = x) &= \mathbf{P}(\mathcal{E}_1 \wedge \mathcal{E}_2 \mid \{X_{k+1} = x\} \wedge \{x \leq P\}) \\ &= \mathbf{P}(\{X_{k+1} \leq X_j, \forall j\} \wedge \{X_{k+1} \leq P\} \mid \\ &\qquad \{X_{k+1} = x\} \wedge \{x \leq P\}) \\ &= \mathbf{P}(x \leq X_j, j = 1, \ldots, k \mid \{X_{k+1} = x\} \wedge \{x \leq P\}) \\ &= \mathbf{P}(x \leq X_j, j = 1, \ldots, k \mid x \leq P) \\ &= \prod_{j=1}^{k} \mathbf{P}(x \leq X_j \mid x \leq P) \\ &= \prod_{j=1}^{k}\left(1 - \frac{x-a}{b_j - a}\right) \leq \left(1 - \frac{x-a}{b_k - a}\right)^k.\end{aligned}$$

The third equality holds because the conditioning already implies that $X_{k+1} \leq P$. The fourth equality holds since the value of $X_j, j = 1, \ldots, k$ is independent of the value of X_{k+1}. Thereafter we use that all the X_j's are identically distributed, and we use the cumulative distribution function of the uniform distribution. In the

last inequality, we exploit that $b_j - a \leq b_k - a$ for all j. We can use this, together with $\frac{b_k - a}{b_{k+1} - a} \leq 1$, to upper bound the expectation of $\mathcal{E}_1 \wedge \mathcal{E}_2$:

$$\begin{aligned}
\mathbf{P}(\mathcal{E}_1 \wedge \mathcal{E}_2) &\leq \frac{1}{b_{k+1} - a} \int_a^P \left(1 - \frac{x - a}{b_k - a}\right)^k dx \\
&= \frac{1}{b_{k+1} - a} \left[-\frac{b_k - a}{k+1} \left(1 - \frac{x - a}{b_k - a}\right)^{k+1} \right]_a^P \\
&= \frac{1}{b_{k+1} - a} \left[-\frac{b_k - a}{k+1} \left(1 - \frac{P - a}{b_k - a}\right)^{k+1} + \frac{b_k - a}{k+1} \right] \\
&= \frac{b_k - a}{b_{k+1} - a} \frac{1}{k+1} \left[1 - \left(1 - \frac{P - a}{b_k - a}\right)^{k+1} \right] \\
&\leq \frac{1}{k+1} \left[1 - \left(1 - \frac{P - a}{b_k - a}\right)^{k+1} \right].
\end{aligned}$$

□

[WF8]

[WF8]: Add some intutiion about

We will continue to lower bound the expected number of end nodes of edges in L'_θ which are inserted in the DIJKSTRA-PREDICTION algorithm, despite the prunings. We call this quantity INRP_θ. We condition on the event E_l. Similar to the adversarial setting, the event E_l fixes the set Q of nodes that are removed from the priority queue and the corresponding distances of these nodes. Further, E_l fixes all outgoing edges of the nodes in Q, and it fixes the number of edges in L'_θ, which is denoted with l. Again, we do not allow the conditioning to fix the weights of the relevant edges in L_θ (and therefore L'_θ): the weight of each relevant edge in L_θ remains random.

It holds that the weights for all edges $(u, v) \in L'_\theta$ are independent and the distance labels $d(u) + w(u, v)$ are uniform on $[D, d(u) + 1]$. This follows from the random model, and because $(u, v) \in L'_\theta$ implies $d(u) + w(u, v) \geq D$ and $d(u) + w(u, v) \leq d(u) + 1$, but nothing else. Note that the distribution of the distance labels is the same as in the partial random instances that we considered in Section 3.6.2, however, in that section, it was an assumption, while here it follows from the random model in combination with the conditioning on E_l.

Theorem 3.6.4. *Suppose L'_θ is defined as above, with $\theta = D + \gamma\varepsilon - 1$, for some $\gamma \in (1, \frac{1}{\varepsilon}]$. Let $INRP_\theta$ be the number of end nodes of edges in L'_θ which are inserted but never removed in the* DIJKSTRA-PREDICTION *algorithm. Under the conditioning of the event E_l, i.e., $|L'_\theta| = l$ and the adversarial setting, we have*

that:

$$\mathbf{E}[INRP_\theta \mid E_l] \leq \begin{cases} \frac{1}{q}\left(1 + \ln\left(\frac{lq}{\gamma}\right)\right) & \text{if } l \geq \frac{\gamma}{q}, \\ \frac{l}{\gamma} & \text{if } l < \frac{\gamma}{q}. \end{cases}$$

Proof:
Let l be the size of L'_θ and let $e_1 = (u_1, v_1), e_2 = (u_2, v_2), \ldots, e_l = (u_l, v_l)$ be all the edges in L'_θ. Note that there might be repetitions in the u_i's, but all the v_i's are distinct. For $i = 1, \ldots, l$, define $X_i = d(u_i) + w(e_i)$. We observed in the previous section that for all edges e_i in L'_θ it holds that X_i is randomly uniform on $[D, d(u_i) + 1]$.

In Dijkstra-Pruning e_i leads to an insertion only if X_i is smaller than X_j for every free v_j, with $j < i$. Suppose that there are k of these free v_j's preceding v_i in the endpoints of L'_θ. In Dijkstra-Prediction, an extra condition must be met, namely that X_i does not exceed the prediction. To lower bound the expectation of INRP_θ, we partition over k, the number of free v_j preceding v_i:

$$\mathbf{E}[\text{INRP}_\theta \mid E_l] \leq \sum_{1 \leq i \leq l} \sum_{0 \leq k < i} \binom{i-1}{k} q^k (1-q)^{i-1-k} \mathbf{P}(\{X_i \leq X_j, \forall j \in [k]\} \wedge \{X_i \leq P\})$$

To be able to apply our Key Lemma (Lemma 3.6.3), we need the variables X_j to be randomly uniform. We have already shown that they are randomly uniform on $[D, d(u_j) + 1]$, for which the upper bounds increase as j increases. Moreover, we need that $P < d(u_1) + 1$, which holds since for all edges (u_i, v) in L'_θ we have $d(u_i) > D - 1 + \varepsilon$ and thus $d(u_i) + 1 > P$. Therefore, we can apply our Key Lemma to upper bound the probability in the sum above by

$$\frac{1}{k+1}\left[1 - \left(1 - \frac{P - D}{d(u_k) + 1 - D}\right)^{k+1}\right] \leq \frac{1}{k+1}\left[1 - \left(1 - \frac{1}{\gamma}\right)^{k+1}\right],$$

which gives

$$\mathbf{E}[\text{INRP}_\theta \mid E_l] \leq \sum_{1 \leq i \leq l} \sum_{0 \leq k < i} \binom{i-1}{k} q^k (1-q)^{i-1-k} \frac{1}{k+1} \tag{3.3}$$

$$- \sum_{1 \leq i \leq l} \sum_{0 \leq k < i} \binom{i-1}{k} q^k (1-q)^{i-1-k} \frac{1}{k+1}\left(1 - \frac{1}{\gamma}\right)^{k+1}. \tag{3.4}$$

From Bast et al. [2003] it follows that (3.3) is equal to $\sum_{1 \leq i \leq l} \frac{1}{iq}(1 - (1-q)^i)$. We will use similar techniques to obtain such an expression for (3.4). We use

$\binom{i-1}{k}\frac{1}{k+1} = \frac{1}{i}\binom{i}{k+1}$, and then use the binomial theorem of Newton to rewrite the sum:

$$\begin{aligned}(3.4) &= \sum_{1\le i\le l}\frac{1}{iq}\sum_{0\le k<i}\binom{i}{k+1}\left(q\left(1-\frac{1}{\gamma}\right)\right)^{k+1}(1-q)^{i-(k+1)}\\ &= \sum_{1\le i\le l}\frac{1}{iq}\left[\sum_{0\le k\le i}\binom{i}{k}q^k\left(1-\frac{1}{\gamma}\right)^k(1-q)^{i-k}-(1-q)^i\right]\\ &= \sum_{1\le i\le l}\frac{1}{iq}\left[\left(1-\frac{q}{\gamma}\right)^i-(1-q)^i\right].\end{aligned}$$

Combining these two bounds, we obtain

$$\begin{aligned}\mathbf{E}[\mathrm{INRP}_\theta \mid E_l] &\le \sum_{1\le i\le l}\frac{1}{iq}\left[1-(1-q)^i-\left(1-\frac{q}{\gamma}\right)^i+(1-q)^i\right]\\ &= \sum_{1\le i\le l}\frac{1}{iq}\left[1-\left(1-\frac{q}{\gamma}\right)^i\right].\end{aligned}$$

For each i such that $1 \le i \le l$ holds that $(1-(1-\frac{q}{\gamma})^i) \le \frac{iq}{\gamma}$. Moreover, for each i such that $\frac{\gamma}{q} \le i \le l$ we can use the stronger bound $(1-(1-\frac{q}{\gamma})^i) \le 1$. Note that we do not use the latter bound in case $\frac{\gamma}{q} > l$. This gives:

$$\mathbf{E}[\mathrm{INRP}_\theta \mid E_l] \le \begin{cases}\frac{1}{q}+\frac{1}{q}\sum_{\lceil\frac{\gamma}{q}\rceil\le i\le l}\frac{1}{i} & \text{if } l \ge \frac{\gamma}{q},\\ \frac{l}{\gamma} & \text{if } l < \frac{\gamma}{q}.\end{cases}$$

If $l \ge \gamma/q$, it holds that $\sum_{\lceil\frac{\gamma}{q}\rceil\le i\le l}\frac{1}{i} \approx \ln(l/\lceil\gamma/q\rceil) \le \ln(lq/\gamma)$, and therefore:

$$\mathbf{E}[\mathrm{INRP}_\theta \mid E_l] \le \begin{cases}\frac{1}{q}\left(1+\ln\left(\frac{lq}{\gamma}\right)\right) & \text{if } l \ge \frac{\gamma}{q},\\ \frac{l}{\gamma} & \text{if } l < \frac{\gamma}{q}.\end{cases}$$

□

The event E_l fixes the distances of all of the nodes that are removed from the priority queue, including the last node. It follows that D is fixed by the event E_l. Assuming that the value of D equals d allows us to choose γ based on d, as follows (note that this does require that $d \in [0, 1-\varepsilon)$): $\gamma(d) = (1-d)/\varepsilon$, which makes θ equal to 0. This means that all the edges that lead to nodes that are inserted but not removed by Dijkstra are in the set L'_θ. Said differently, the size of $|L'_\theta|$ is equal to the number of nodes that are inserted but never removed in the priority queue by the Dijkstra algorithm. Bast et al. [2003] estimate the

expected value of this quantity, conditional on that many nodes are reachable from s, which is denoted with "R is large" and explained in more detail below.

The number of nodes reachable from s is studied in Section 10.5 of Alon and Spencer [2016]. Let $\alpha > 0$ be such that $\alpha = 1 - \exp(-c\alpha)$, and let R be the number of nodes reachable from s. Intuitively, R is either bounded by a constant with probability about $1 - \alpha$, or it is approximately αn with probability about α. Formally, for every $\varepsilon > 0$ and $\delta > 0$, there is a t_0 such that for all sufficiently large n, it holds that

$$1 - \alpha - \varepsilon \leq \mathbf{P}(R \leq t_0) \leq 1 - \alpha + \varepsilon,$$

and

$$\alpha - \varepsilon \leq \mathbf{P}((1-\delta)\alpha n < R < (1+\delta)\alpha n) \leq \alpha + \varepsilon.$$

Like in Bast et al. [2003], we set $\delta = 0.01$ and we restrict ourselves to the set of graphs with more than $(1-\delta)\alpha n$ nodes reachable from s.

We want the probability of a target node being reachable from s to be large, which we accomplish by upper bounding the probability that all reachable nodes are not target nodes. This probability is

$$(1-q)^{\alpha n} \leq \exp(-\alpha n q) = \exp(-\alpha f).$$

This probability is at most $1/n^2$ whenever $c \geq 2$ and $f \geq 4 \ln n$, since $c \geq 2$ implies $\alpha > \frac{1}{2}$. We therefore require that $c \geq 2$, $f \geq 4 \ln n$ and assume that $R > (1-\delta)\alpha n$. We refer to these assumption as "R is large".

Bast et al. [2003] denote the number of nodes reachable from s by INRS and estimate its expected value. We summarise their findings in the following proposition.

Proposition 3.6.5: (Bast et al. [2003]). *Consider an instance from the random model introduced in Section 3.2. Let R be the number of reachable nodes from s in the random graph. Then the expected number of nodes that are inserted but never removed in the priority queue by the* Dijkstra *algorithm, given that R is large, is approximately:*

$$\mathbf{E}[\mathit{INRS} \mid R \text{ is large}] \approx \frac{c-1}{q}.$$

[WF9] By exploiting this proposition, we can drop the dependency on the size of L'_θ and we obtain the following theorem.

[WF9]: paragraph about other paper edge weights indpenednt. and say expectation is equal to depnedent

Theorem 3.6.6. ~~*Suppose D lies in $[0, 1-\varepsilon)$.*~~ [WF] *Let INRP be the number of nodes that are inserted but never removed by* Dijkstra-Prediction. *For each $d \in [0, 1-\varepsilon)$ holds:*

$$\mathbf{E}[\mathit{INRP} \mid R \text{ is large and } D = d] \leq \frac{1}{q}\left(1 + \max\left\{0,\ \ln(c-1) - \ln\left(\frac{1-d}{\varepsilon}\right)\right\}\right).$$

Proof:
We can upper bound the expected value of INRP_θ given in Theorem 3.6.4 as follows.

Since the upper bound for the expectation of INRP_θ only depends on l and γ (where we choose γ based on d as described above) the following holds:

$$\mathbf{E}[\text{INRP} \mid \text{INRS}= l,\ D = d \text{ and } R \text{ is large}] \leq \begin{cases} \frac{1}{q}\left(1 + \ln\left(\frac{lq}{\gamma(d)}\right)\right) & \text{if } l \geq \frac{\gamma(d)}{q}, \\ \frac{l}{\gamma(d)} & \text{if } l < \frac{\gamma(d)}{q}. \end{cases}$$

Note that this expectation is differentiable in l, and also concave in l since the derivative is monotonically non-increasing. The concavity allows us to drop the conditioning on l, and we can replace l with the expectation of INRS given that R is large (using Proposition 3.6.5). Note here that the expectation of INRS given that R is large is independent of the value of D. As described above, we choose $\gamma(d) = (1-d)/\varepsilon$.

$$\begin{aligned} \mathbf{E}[\text{INRP} \mid R \text{ is large and } D = d] &\leq \begin{cases} \frac{1}{q}\left(1 + \ln\left(\frac{(c-1)q\varepsilon}{q(1-d)}\right)\right) & \text{if } c-1 \geq \frac{1-d}{\varepsilon}, \\ \frac{(c-1)\varepsilon}{q(1-d)} & \text{if } c-1 < \frac{1-d}{\varepsilon} \end{cases} \\ &\leq \begin{cases} \frac{1}{q}\left(1 + \ln(c-1) - \ln\left(\frac{1-d}{\varepsilon}\right)\right) & \text{if } c-1 \geq \frac{1-d}{\varepsilon}, \\ \frac{1}{q} & \text{if } c-1 < \frac{1-d}{\varepsilon}. \end{cases} \\ &= \frac{1}{q}\left(1 + \max\left\{0,\ \ln(c-1) - \ln\left(\frac{1-d}{\varepsilon}\right)\right\}\right) \end{aligned}$$

□

Let INRR denote the number of nodes that are inserted but never removed by Dijkstra-Pruning. It is shown by Bast et al. [2003] that (independently of the value of D):

$$\mathbf{E}[\text{INRR} \mid R \text{ is large}] \leq \frac{1}{q}\left(1 + \ln(c-1)\right).$$

That is, our Dijkstra-Prediction algorithm either saves $\ln(1-d)/\varepsilon)/q$ (in case that $(c-1)\varepsilon \geq 1-d$), or $1/q(\ln(c-1) + (1-(c-1)\varepsilon/(1-d)$ (in case that $(c-1)\varepsilon \geq 1-d$) insertions of such nodes compared to the Dijkstra-Pruning algorithm. So even though Dijkstra-Pruning already saves a significant number of insertions, Dijkstra-Prediction is able to further improve on this. Naturally, these savings grow whenever the prediction becomes more accurate and ε decreases.

[WF10]: Check these numbers...

[WF10] In the experiments in the previous section, we considered random instances with $n = 1000$, $c = 8$ and $q = 0.02$. For these instances, D is approximately 0.55 (see Table 3.1. With a prediction $P = D + \varepsilon$ which overestimates D

by at most $\varepsilon = 0.1$ (which seems reasonable from the experiments), the expected number of INRP of Dijkstra-Prediction is at most 63. In comparison, the expected number of INRS of Dijkstra is 350; so our algorithm saves at least 287 of these insertions. The expected number of INRR of Dijkstra-Pruning is at most 137; our algorithm significantly improves upon this by exploiting the prediction.

3.7 Conclusion and Discussion

The goal of the research in this chapter was to investigate whether Dijkstra's algorithm for the SSMTSP problem can be improved by using a single prediction value while insisting on a certificate of optimality even if the prediction is arbitrarily bad. Our experiments show that our algorithm provides a clear improvement in terms of the number of queue operations compared to the (already sophisticated) Dijkstra-Pruning algorithm. Also, the running time of our algorithm improves over Dijkstra-Pruning.

From a practical point of view, there is a straightforward way to improve the running time of our algorithm further: The bottleneck of our implementation is that we insist on constructing a certificate of optimality even if the prediction is arbitrarily bad. This forces us to maintain a partition of all visited nodes into the priority queue and the reserve list. The latter is used to store all nodes that could potentially become relevant at a later stage, which might cause an overhead. If, however, we had some confidence that the predicted shortest path distance P is not too far from the actual shortest path distance D (e.g., say that we need to inflate P at most z times), then we could prune many more edges from the beginning (e.g., by setting $B = \beta^z P$).

We conducted empirical experiments on random graphs. A natural follow-up study would be to test the workings of Dijkstra-Pruning on real-world networks. Especially real-world networks with a known well-working distance approximation algorithm would be interesting candidates, like in Rizi et al. [2018]. The distance approximation algorithm predicts the distance between two given nodes in the network, and can be consulted for the source node together with each of the target nodes. The minimum of these predictions can then serve as prediction P in Dijkstra-Pruning from the first iteration. We expect that having such a prediction from the start can significantly reduce the algorithm's running time.

Chapter 4

Solving the Casting Problem

4.1 Introduction

The Generalized Assignment Problem (GAP) is a well-known and well-studied problem (see e.g. [Ross and Soland, 1975, Fisher et al., 1986, Savelsbergh, 1997, Martello and Toth, 1990, Cattrysse and Van Wassenhove, 1992, Yagiura and Ibaraki, 2007, Öncan, 2007, Maniezzo et al., 2021]). An instance of GAP consists of a tuple $(M, N, (b_i)_{i\in M}, (w_{ij})_{i\in M, j\in N}, (v_{ij})_{i\in M, j\in N})$ where M is a set of m knapsacks and N is a set of n items. Each knapsack $i \in M$ has a capacity $b_i > 0$, and for each item j and knapsack i, a weight $w_{ij} > 0$ and a value $v_{ij} > 0$ are given. A feasible solution for GAP is an assignment of all items to knapsacks such that the total weight of items assigned to a knapsack is at most the knapsack's capacity. The goal of GAP is to find a feasible solution that maximizes the value.

We consider a special case of GAP, referred to as the Casting Problem, in which each weight w_{ij} for an item j equals w_j, independently of the knapsack i, and each value v_{ij} equals w_j/b_i. An instance of the casting problem is denoted with $(M, N, (b_i)_{i\in M}, (w_j)_{j\in N})$, but for the sake of brevity we sometimes use (M, N). The Casting Problem can naturally be formulated as an integer linear program (ILP).To this aim, we introduce a decision variable $x_{ij} \in \{0, 1\}$ that indicates if item $j \in N$ is packed in knapsack $i \in M$ or not. The following ILP describes our Casting Problem; subsequently, we use (CP) to refer to this

formulation:

$$\text{maximize} \quad c(\boldsymbol{x}) = \sum_{i \in M} \frac{1}{b_i} \sum_{j \in N} w_j x_{ij} \tag{4.1a}$$

$$\text{subject to} \quad \sum_{j \in N} w_j x_{ij} \leq b_i \qquad \forall i \in M \tag{4.1b}$$

$$\sum_{i \in M} x_{ij} = 1 \qquad \forall j \in N \tag{4.1c}$$

$$x_{ij} \in \{0, 1\} \quad \forall i \in M,\ \forall j \in N \tag{4.1d}$$

The Casting Problem was introduced by Deb and Myburgh [2017] and is inspired by a practical problem present in metal foundries. In a foundry, heats of metal are molten and cast into objects of different sizes. The sizes of the objects that are cast from one heat may not add up to the size of the heat, and some molten metal remains. The ratio of used metal to the total size of the heat is called the metal utilization ratio. Formally, the utilization ratio of a heat is defined as the total weight of objects that are cast from the heat divided by the size of the heat. The goal of this practical problem is to create a casting assignment of products over heats such that the utilization ratio, averaged over the heats, is maximized. By regarding the heats as knapsacks, the Casting Problem exactly solves this problem. To be precise, the goal of the Casting Problem is to find an assignment that maximizes the sum of utilization ratios over the heats, which is equivalent to finding an assignment that maximizes the average utilization ratio.

It is not hard to see that the casting problem is closely related to the *related machine scheduling problem* (also referred to as the *uniform machine scheduling problem*) [Hochbaum and Shmoys, 1988]. In related machine scheduling, a set of jobs with processing time p_j for job j needs to be planned on machines with speed s_i for machine i. Executing a job j on machine i takes p_j/s_i time units. In the decision version of related machine scheduling, the goal is to decide if a schedule exists that plans each job on a machine and finishes within a given time limit. A moment of thought reveals that the problem of finding a feasible solution to the casting problem is equivalent to the decision version of related machine scheduling with a time limit of 1, see also Section 4.2. The decision version of related machine scheduling is NP-complete [Garey and Johnson, 1979, Hochbaum and Shmoys, 1988]. By exploiting this equivalence, we conclude that the problem of finding a feasible solution to the casting problem is NP-complete as well. In particular, this means that we cannot expect to find an algorithm that finds an optimal solution to the casting problem in polynomial time; in fact, this even rules out that a polynomial-time approximation algorithm exists, unless P = NP.

Even though the Casting Problem is NP-complete, the aim of this chapter is to find efficient methods of solving a specific set of instances. We focus on

the instances given in the paper that introduces the CASTING PROBLEM, Deb and Myburgh [2017]. A straightforward way to solve the given instances would be to input the CPformulation to a mathematical programming solver like the Gurobi optimizer [Gurobi Optimization, LLC, 2023]. However, as also pointed out in Deb and Myburgh [2017], these mathematical programming solvers are not capable of solving the larger instances in the instance set, i.e., with more than 10.000 variables, within a time limit of an hour.

The first method that we study is based on an alternative formulation of the CASTING PROBLEM in which the assignment Constraint (4.1c) is relaxed and lifted to the objective function. In this alternative formulation, a solution where not every item is assigned to a knapsack becomes feasible, allowing mathematical programming solvers to more easily find feasible solutions. A penalty in the objective function ensures that solutions in which every item is assigned are favored over other solutions.

The second method that we propose in this chapter is based on a disaggregated formulation for the CASTING PROBLEM, inspired by a branch-and-price method used for solving the GAP [Savelsbergh, 1997]. We present a disaggregated ILP formulation of the CASTING PROBLEM, with an exponential number of variables. Savelsbergh [1997] solve the GAP by considering the disaggregated formulation with a restricted set of variables, where the variables are created using iterative column generation techniques. We present an alternative for using column generation by exploiting the small number of distinct item sizes and knapsack capacities in the given instances. By doing so, we create a set of variables for which we can prove that any feasible solution of the disaggregated ILP formulation with these variables is optimal. This allows us to solve all the instances introduced in Deb and Myburgh [2017], also the larger ones, to optimality, taking at most $\sim$75 minutes. As a comparison, the algorithm in Deb and Myburgh [2017] needs more than 6 days to solve the largest instance, with no optimality guarantee.

4.1.1 Contributions

- We show that the problem of finding a feasible solution to the casting problem is equivalent to the decision version of related machine scheduling and therefore NP-complete. Therefore we cannot expect to find a polynomial-time approximation algorithm for the casting problem.
- We present a new formulation of the CASTING PROBLEM in which we relax the item assignment constraint and lift it to the objective. This aids mathematical programming solvers since it becomes easier to find feasible solutions. We show empirically that the optimization time decreases by using the alternative formulation.
- We present a disaggregated ILP formulation for the CASTING PROBLEM with an exponential number of varbiables. We prove that it suffices to con-

sider a specific subset of these variables, and show that any feasible solution with these variables is an optimal solution for the CASTING PROBLEM. We solve all considered instances to optimality within ∼75 minutes.

4.1.2 Related Work

The casting problem was introduced by Deb and Myburgh [2017]. The formulation of the casting problem in that paper differs slightly from our definition: the objective is scaled with the constant $1/m$ and an extra integer d_j is introduced for each item j, which indicates how many times item j needs to be assigned. Integer variables are used instead of binary variables, which indicate how many copies of item j are assigned to heat i. A moment of thought reveals that the two formulations are equivalent, since the demand d_j is bounded by $1/w_j \sum_{i \in M} b_i$ in every instance that has a feasible solution.

In [Deb and Myburgh, 2017], the set of knapsacks and their capacities is not given, but it is computed based on a given shortlist of knapsack capacities of length k, denoted with $l_0, l_1, \ldots, l_{k-1}$. The computed set of knapsacks consists of a certain number of repetitions of this shortlist. The number of repetitions, denoted with r, is computed based on a given parameter η, for which holds that $0 \leq \eta \leq 1$. η is the desired utilization ratio. Then the total number of repetitions is computed as follows: it is the largest integer such that the sum of all item weights divided by the total knapsack capacity exceeds η, i.e., the largest $r \in \mathbb{N}$ such that

$$\frac{\sum_{j \in N} w_j}{r \sum_{i=0}^{k-1} l_i} \leq \eta.$$

It follows that m equals the number of repetitions times the number of knapsacks in the shortlist, i.e., $m = r \cdot k$. Moreover, it follows that $b_i = l_{i'}$ for $i' = i \mod k$.

After computing m based on η, a genetic algorithm is used to search for a solution with a score that is at least η. In most experiments, η is set to 0.997. It is claimed in the paper that it suffices to stop searching for a better solution whenever a solution is found with score η, since it is implied that no solution can exist with a score that exceeds η. We believe otherwise for general η. For $\eta = 2/3$, we give a counterexample in Figure 4.1, where a feasible solution is given with a score strictly larger than η. The counterexample in Figure 4.1 generalizes to more values of η, by adding extra knapsacks with capacity 1 and the same number of items with weight 1.

Since it holds that solutions might exist that have a solution value that exceeds η, the genetic algorithm produces an approximate solution without any proof of optimality. Deb and Myburgh [2017] show that this genetic algorithm is able to find these approximate solutions for very large instances, up to 100.000 knapsacks, on a specially equipped computer. Since a straightforward implementation of the MIP formulation into CPLEX is not able to find solutions for these large

(a) Solution 1 (b) Solution 2

Figure 4.1: Two possible solutions for an instance with two knapsacks and six items, each with weight 1, for $\eta = 2/3$. The left solution has an average utilization ratio of $2/3$, exactly equal to η. The right solution has an average utilization ratio of $3/4$ and therefore shows that an average utilization ratio strictly larger than η is possible.

instances, they claim that their study 'directly compares with state-of-the-art popular commercial and public-domain software in establishing the superiority of population-based methods in solving large-scale problems'. However, we show in our empirical results section that using a disaggregated formulation and a predefined computation of specific variables solves even the large instances in reasonable computing time, *with* a certificate of optimality.

Moreover, they claim that their study 'clearly portrays the fact that if a near-optimal solution is desired, it is possible to develop a polynomial-time algorithm for addressing NP-hard problems'. We show that even without considering the objective, the problem of finding a feasible solution to the casting problem remains NP-complete. Stating that it is possible to develop a polynomial-time algorithm would imply that P = NP. Lastly, Deb and Myburgh [2017] draw a parallel between the casting problem and the multiple knapsack problem. We want to point out the important difference that in the multiple knapsack problem it is not necessary to assign all items to knapsacks, while this is required in the casting problem.

4.1.3 Organization of Chapter

In the next section, we show that the Casting Problem is NP-complete and elaborate more on the column generation approach for GAP. We present the auxiliary variable formulation in Section 4.3, after which we give the disaggregated formulation in Section 4.4. The results for running the different solving methods on the empirical instances are reported in Section 4.5.

4.2 Preliminaries

4.2.1 Casting Problem is NP-Complete

As explained in the introduction of this chapter, the Casting Problem problem is closely related to the related machine scheduling problem. The decision version of the related machine scheduling problem is defined by a set of jobs J, a set of machines I, and a time limit $T > 0$. Each job $j \in J$ has a given processing time $p_j > 0$ and each machine $i \in I$ has a given speed $s_i > 0$. Executing a job j on machine i takes p_j/s_i time units. The goal of the problem is to decide if there exists an assignment of jobs to machines such that all jobs are executed within the time limit T.

By exploiting the relation with the related machine scheduling problem, it is not hard to prove that the Casting Problem is an NP-complete problem. In fact, the feasibility problem is already NP-complete.

Theorem 4.2.1. *The problem of finding a feasible solution to the* Casting Problem *is strongly NP-complete.*

Proof:
We will prove that the feasibility problem of the Casting Problem is equivalent to the decision version of related machine scheduling. Garey and Johnson [1979] show that the decision version of related machine scheduling is strongly NP-complete, even if all machines are identical.[1] Given an instance of the related machine scheduling problem $(I, J, (s_i)_{i\in I}, (p_j)_{j\in J})$ we define an instance of the Casting Problem, $(M, N, (b_i)_{i\in M}, (w_j)_{j\in N})$, as follows. Let $M = I, N = J, b_i = s_i$ for $i \in M$ and $w_j = p_j/T$ for $j \in N$. Let x_{ij} denote a feasible solution of the related machine scheduling problem, where $x_{ij} = 1$ if and only if job j is assigned to machine i. We can compute a solution to the Casting Problem by assigning item j to knapsack i if job j was assigned to machine i. Since each feasible solution of the related machine scheduling problem assigns each job to exactly one machine, it holds that each item is assigned to exactly one knapsack in the casting solution. Moreover, for each feasible solution x_{ij} of the related machine scheduling problem it holds that for every $i \in I$:

$$\sum_j \frac{p_j}{s_i} x_{ij} \leq T,$$

which is equivalent to, for every $i \in M$:

$$\sum_j w_j x_{ij} \leq b_i.$$

[1] If the number of machines is bounded, the problem is no longer strongly NP-complete and can be solved in pseudo-polynomial time.

This proves that the capacity constraint for the CASTING PROBLEM holds. This proves that finding a feasible solution to the CASTING PROBLEM in the new instance is equivalent to deciding if an assignment of jobs to machines exists that can be executed within time limit T. □

4.2.2 Column Generation for GAP

In Savelsbergh [1997], a formulation with an exponential number of variables is presented for the GAP. The linear relaxation of this problem can be solved with column generation, while an integer solution is found using a branch-and-price framework. Later in this chapter, we present a formulation for the CASTING PROBLEM with an exponential number of variables, inspired by the formulation by Savelsbergh [1997]. Therefore we elaborate on this column generation approach for GAP here.

Let $(M, N, (b_i)_{i\in M}, (w_{ij})_{i\in M, j\in N}, (v_{ij})_{i\in M, j\in N})$ be an instance of GAP, as explained in the introduction of this chapter. Let $P_i = \{\boldsymbol{x}_1^i, \boldsymbol{x}_2^i, \ldots, \boldsymbol{x}_{p_i}^i\}$ be the set of all feasible assignments of items in N to a knapsack $i \in M$, i.e., each $\boldsymbol{x}_p^i = (x_{1p}^i, x_{2p}^i, \ldots, x_{np}^i)$, for $i \in M$ and $p \in \{1, \ldots, p_i\}$, is a feasible solution to the following set of constraints:

$$\sum_{j\in N} w_{ij} x_{jp}^i \leq b_i$$
$$x_{jp}^i \in \{0,1\} \qquad j \in N.$$

For each $i \in M$, this set of constraints is equivalent to the feasibility problem of a knapsack problem with capacity b_i.

For $i \in M$ and $p \in \{1, \ldots, p_i\}$, introduce binary variable y_p^i, which indicates if i gets assigned the items specified by the feasible assignment $\boldsymbol{x}_p^i$. The formulation for GAP with an exponential number of variables, referred to as the *disaggregated formulation for GAP* is then as follows:

$$\text{maximize} \quad \sum_{i\in M} \sum_{p=1}^{p_i} \left(\sum_{j\in N} v_{ij} x_{jp}^i \right) y_p^i \tag{4.2a}$$

$$\text{subject to} \quad \sum_{i\in M} \sum_{p=1}^{p_i} x_{jp}^i y_p^i = 1 \qquad \forall j \in N \tag{4.2b}$$

$$\sum_{p=1}^{p_i} y_p^i \leq 1 \qquad \forall i \in M \tag{4.2c}$$

$$y_p^i \in \{0,1\} \quad \forall i \in M, \quad p = 1, \ldots, p_i \tag{4.2d}$$

The first constraint ensures that each item is assigned exactly once over all knapsacks. The second constraint ensures that at most one feasible assignment of items is selected for each knapsack.

Savelsbergh [1997] observe that the disaggregated formulation for GAP is essentially obtained by applying Dantzig-Wolfe decomposition to the standard formulation of GAP, where the knapsack constraints have been placed in the subproblem. We refer to Bertsimas and Tsitsiklis [1997] for a detailed explanation of Dantzig-Wolfe decompositions.

We refer to the linear relaxation of the aggregated formulation of GAP as the *master problem* (MP). It is not possible to solve the MP directly since it contains an exponential number of variables. Instead, we consider the *restricted master problem* (RMP), which only considers a subset of the variables. We will add the necessary variables to the RMP using a technique called column generation. *Column generation* is a pricing scheme in which a column is created with the largest reduced cost. Note that a column in the matrix form corresponds with a variable in the disaggregated formulation, that is why the terms column and variable are used interchangeably in the column generation context. The technique is introduced by Gilmore and Gomory [1961] in the context of cutting stock problems.

We elaborate more on the workings of column generation for the GAP. The RMP can be solved with a standard technique like the simplex method. Suppose we have found an optimal solution to the RMP, then let (π_i, λ_j) be an optimal solution to the dual problem, where π_i is the dual variable belonging to Constraint (4.2b) and λ_j is the dual variable belonging to Constraint (4.2c). Additional columns to the RMP can then be generated by solving the following *pricing problem*:

$$\max_{i \in M} \{z(KP_i) - \pi_i\}, \tag{4.3}$$

where for each $i \in M$, $z(KP_i)$ is the value of the optimal solution to the following knapsack problem:

$$\begin{aligned} z(KP_i) = \text{ maximize } & \sum_{j \in N} (v_{ij} - \lambda_j)\, x_j^i \\ \text{subject to } & \sum_{j \in N} w_{ij} x_j^i \leq b_i \\ & x_j^i \in \{0,1\} \qquad \forall j \in N \end{aligned} \tag{4.4}$$

If the solution to the pricing problem (4.3) is positive, a column is found with positive reduced cost and it can be added to the RMP. If the solution to the pricing problem is less than or equal to zero, it follows that the reduced cost is less than or equal to zero for all $i \in M$. This must mean that the optimal solution for the RMP is also optimal for the MP, and we no longer need to search for new columns.

The column generation approach described above solves the MP, i.e., the linear relaxation of the disaggregated formulation for GAP. To solve the original, integer, disaggregated formulation for GAP, column generation must be applied

in a so-called *branch-and-price* framework. Since we do not use this branch-and-price framework for the Casting Problem, we omit the details here and refer to Savelsbergh [1997] for the framework and compatible branching rules.

4.3 Auxiliary Variable Formulation

In this section, we introduce an alternative formulation of the Casting Problem, in which we allow that an item is not assigned to a knapsack, at the cost of a penalty. We accomplish this by lifting Constraint (4.1c) to the objective function, i.e., we deal with it in a soft manner. By doing so, we increase the number of feasible solutions and therefore make it easier to find a feasible solution, which can aid mathematical programming solvers and potentially speed-up the optimization time.

To accomplish this, we introduce decision variable z_j for each item $j \in N$, which equals 1 if an item is not assigned to any knapsack, and 0 otherwise. We can then update Constraint (4.1c) and introduce a penalty in the objective function for any positive z_j. In order to prioritize decreasing the penalty over increasing the original objective value, we divide the original objective by m, similarly as in the original paper by Deb and Myburgh [2017] (see Section 4.1.2). The formulation, which we refer to as the Auxiliary Variable Formulation (AVF), is then as follows:

$$\begin{aligned} \text{maximize} \quad & \frac{1}{m}\sum_{i\in M}\frac{1}{b_i}\sum_{j\in N} w_j x_{ij} - \sum_{j\in N} z_j \\ \text{subject to} \quad & \sum_{j\in N} w_j x_{ij} \leq b_i, \quad i \in M \\ & \sum_{i\in M} x_{ij} + z_j = 1, \quad j \in N \\ & x_{ij} \in \{0,1\} \\ & z_j \geq 0 \end{aligned} \tag{4.5}$$

Note that it is not necessary to enforce variable z_j to be binary, this is implied by the constraints and integrality of x_{ij}. We prove that this formulation indeed solves feasible instances of the Casting Problem:

Theorem 4.3.1. *If the* Casting Problem *has a feasible solution, then the optimal solution value of the* Casting Problem *is equal to the optimal solution value of* (4.5)*.*

Proof:
We assume that the Casting Problem has a feasible solution. It is easy to see that $\boldsymbol{x}$ is a feasible solution to the Casting Problem if and only if $(\boldsymbol{x}, \boldsymbol{z} = \boldsymbol{0})$

is a feasible solution to (4.5). Moreover, the objective value of $\boldsymbol{x}$ in the CASTING PROBLEM equals the objective value of $(\boldsymbol{x}, \mathbf{0})$ in (4.5), multiplied with m. This means that (4.5) has a feasible solution with a positive objective value. Any solution for (4.5) for which $\boldsymbol{z}$ is unequal to $\mathbf{0}$ has a negative solution value, therefore, $\boldsymbol{z}$ equals $\mathbf{0}$ in the optimal solution for (4.5), and it is therefore equal to the solution value of the CASTING PROBLEM. □

4.4 Disaggregated Formulation

In this section we present a disaggregated formulation of the CASTING PROBLEM, based on the disaggregated formulation of the GAP as presented by Savelsbergh [1997] (see Section 4.2). Before giving the formulation, we introduce an item aggregation that is also mentioned in Section 4.1.2, and a knapsack aggregation, both are explained below. We also give assumptions on the instances that we solve with the disaggregated formulation and present a general theorem which is essentially an optimality condition for the CASTING PROBLEM.

Item Aggregation

Let $\bar{N}$ be the original item set. Then the aggregated item set, for the remainder of this section denoted with N, is the largest subset of $\bar{N}$, such that no two items have the same weight. Intuitively, N is obtained from $\bar{N}$ by traversing through the items and deleting each item that has a weight that has already been encountered. For each $j \in N$, let integer d_j indicate the number of times an item with weight w_j appears in the original item set $\bar{N}$. We refer to d_j as the *demand* of item j. The item constraint (4.1c) needs to be adjusted to the new demand parameter d_j as follows:

$$\sum_{i \in M} x_{ij} = d_j \quad j \in N. \tag{4.6}$$

For the creation of columns, explained later in this section, we make the following assumption. Whenever the instances grow in size, we assume that the number of unique item weights in N remains constant.

Knapsack Aggregation

We assume without loss of generality that the knapsacks are ordered based on non-decreasing capacity, so $b_1 \leq \ldots \leq b_m$. Suppose there are $\hat{m}$ distinct knapsack capacities. We partition the knapsacks into sets $M(1), \ldots, M(\hat{m})$, such that all knapsacks in a set have the same capacity, i.e., $i, i' \in M(k) \iff b_i = b_{i'}$. The partition preserves the order in the knapsacks, i.e., if $i \in M(k)$ and $i' \in M(k')$,

then $k \leq k' \iff b_i \leq b_{i'}$. By this notation, $M(\hat{m})$ contains all knapsacks that have the largest knapsack capacity, b_m.

For the creation of columns, explained later in this section, we make an assumption again. Namely, we assume that the number of unique knapsack capacities, $\hat{m}$, is at most 2.

Problem analysis

The aggregation of knapsacks and items allows us to prove the following theorem about the optimality of a solution to the CASTING PROBLEM.

Theorem 4.4.1. *Any feasible solution to the* CASTING PROBLEM *which maximizes $\hat{c}$ is optimal, where $\hat{c}$ equals:*

$$\hat{c} = \sum_{i \in M \setminus M(\hat{m})} \left(\frac{1}{b_i} - \frac{1}{b_m} \right) \sum_{j \in N} w_j x_{ij}$$

Proof:
It suffices to prove that $\hat{c}$ equals the objective function of the CASTING PROBLEM minus a constant value λ, defined as:

$$\lambda = \frac{1}{b_m} \sum_{j \in N} w_j d_j.$$

The objective function of the CASTING PROBLEM minus λ is equal to:

$$\begin{aligned} c(\boldsymbol{x}) - \lambda &= \sum_{i \in M} \frac{1}{b_i} \sum_{j \in N} w_j x_{ij} - \frac{1}{b_m} \sum_{j \in N} w_j d_j \\ &= \sum_{i \in M} \frac{1}{b_i} \sum_{j \in N} w_j x_{ij} - \sum_{i \in M} \frac{1}{b_m} \sum_{j \in N} w_j x_{ij} \\ &= \sum_{i \in M} \left(\frac{1}{b_i} - \frac{1}{b_m} \right) \sum_{j \in N} w_j x_{ij} \\ &= \sum_{i \in M \setminus M(\hat{m})} \left(\frac{1}{b_i} - \frac{1}{b_m} \right) \sum_{j \in N} w_j x_{ij} = \hat{c} \end{aligned}$$

The second equality holds by (4.6) and the last equality holds since $b_i = b_m$ for all $i \in M(\hat{m})$.

□

If there are only one or two different knapsack capacities, as we assume, Theorem 4.4.1 can be simplified to one of the following corollaries.

Corollary 4.4.2. *If all knapsacks have equal capacity, any feasible solution is optimal.*

Proof:
Follows directly from Theorem 4.4.1 with $\hat{m} = 1$. Then $M = M(\hat{m})$ and $\hat{c} = 0$, so any feasible solution is optimal. □

Corollary 4.4.3. *If there are two distinct knapsack capacities, any feasible solution that maximizes the total weight assigned to knapsacks in $M(1)$ is optimal.*

Proof:
Follows directly from Theorem 4.4.1 with $\hat{m} = 2$. Then $M = M(1) \cup M(2)$ and also $M \setminus M(\hat{m}) = M(1)$, and therefore $\hat{c}$ equals:

$$\hat{c} = \sum_{i \in M(1)} \left(\frac{1}{b_1} - \frac{1}{b_m} \right) \sum_{j \in N} w_j x_{ij}$$

It follows that maximizing the weight on knapsacks in $M(1)$ is equivalent to maximizing $\bar{c}$. □

Disaggregated Formulation for Casting Problem

For the remainder of this section, we allow a slight abuse of notation and write b_k if we mean b_i for an $i \in M(k)$, this is possible since all the knapsacks in $M(k)$ have the same capacity, by definition. For each of these subsets of knapsacks, we will define an exponential number (denoted by p_k) of decision variables. We do this as follows: for $k = 1, \ldots, \hat{m}$, let $P_k = \{\boldsymbol{x}_1^k, \boldsymbol{x}_2^k, \ldots, \boldsymbol{x}_{p_k}^k\}$ be the set of all feasible assignments of items in N to a knapsack with capacity b_k, i.e., each $\boldsymbol{x}_p^k = (x_{1p}^k, x_{2p}^k, \ldots, x_{np}^k)$ is a feasible solution to the following set of constraints:

$$\begin{aligned} \sum_{j \in N} w_j x_{jp}^k &\leq b_k \\ x_{jp}^k &\leq d_j \\ x_{jp}^k &\in \mathbb{N}_0, \qquad j \in N. \end{aligned}$$

This set of constraints can be regarded as finding a feasible solution to a knapsack problem in which at most d_j copies of item j can be assigned to a knapsack with capacity b_k.

For $k = 1, \ldots, \hat{m}$ and $p \in \{1, \ldots, p_k\}$, we introduce integer variable y_p^k, which indicates the number of knapsacks of $M(k)$ that get assigned the items specified by the feasible assignment $\boldsymbol{x}_p^k$. This allows us to give the following ILP, which we

refer to as the *disaggregated formulation of the* Casting Problem.

$$\begin{aligned}
\text{maximize} \quad & \sum_{k=1}^{\hat{m}} \frac{1}{b_k} \sum_{p=1}^{p_k} \left(\sum_{j \in N} w_j x_{jp}^k \right) y_p^k \\
\text{subject to} \quad & \sum_{k=1}^{\hat{m}} \sum_{p=1}^{p_k} x_{jp}^k y_p^k = d_j \qquad \forall j \in N \\
& \sum_{p=1}^{p_k} y_p^k = |M(k)| \quad k = 1, \ldots, \hat{m} \\
& y_p^k \in \mathbb{N}_0 \qquad k = 1, \ldots, \hat{m}, \quad p = 1, \ldots, p_k
\end{aligned} \tag{4.7}$$

The first constraint ensures that over all knapsacks, enough items are assigned. The second constraint ensures that exactly enough assignments for knapsacks in each $M(k)$ are chosen.

The disaggregated formulation of the Casting Problem has an exponential number of variables. Therefore, we will consider a so-called *restricted master problem* (RMP) that considers only a subset of these variables. Usually, in column generation approaches, new variables for the RMP are computed using column generation repeatedly until an optimal solution has been found, as explained in Section 4.2. The assumptions that we made on the instances that we consider allow us to circumvent the process of column generation. We assumed that the number of unique knapsack capacities and the number of unique item weights are constant. This allows us to solve the RMP using a pre-computed set of high-quality variables. The meaning of 'high quality' will be clarified below.

In fact, the quality of these pre-computed variables is as good as possible, meaning that if we can find any feasible solution to the master problem with these variables, it is automatically optimal. By Corollary 4.4.3, we know that for instances with only two unique knapsack capacities, a feasible solution is optimal if the total weight assigned to knapsacks with the small capacity is maximized. Therefore, for the small knapsacks $M(1)$, we let P_1 be the set of all assignments that consume all capacity on the knapsacks, i.e., variables $\boldsymbol{x}_p^1$ such that

$$\sum_{j \in N} w_j x_{jp}^1 = b_1.$$

The utilization ratio for these variables is exactly equal to one. For the larger knapsacks, we let P_2 be the set of all variables that have a utilization ratio that is at least a predetermined threshold. We are able to create these sets of variables in tractable time since the number of unique item weights remains constant. By constructing P_1 and P_2 as explained, it follows by Corollary 4.4.3 that any feasible solution of the RMP with these variables is automatically optimal.

However, it is not guaranteed that a feasible solution with variables from P_1 and P_2 exists. Suppose there is an instance with two distinct knapsack capacities,

for which all feasible solutions have some remaining capacity in a small knapsack. I.e., there does not exist a feasible solution for which all of the capacity on small knapsacks is assigned to items. Then, our method will not be able to find a feasible solution, since P_1 does not contain the right variables for finding a feasible solution to RMP. There are two options to remedy this problem. The first would be to increase the size of P_1 and P_2 by adding columns that correspond to knapsacks with a smaller utilization ratio. A second approach would be to fall back to standard column generation to generate new columns for the linear relaxation and use a branch-and-price framework to obtain an integral solution.

4.5 Emprical Results

Deb and Myburgh [2017] give practical instances of the Casting Problem. In this section, we elaborate on the various methods that we used for solving these instances and also give the running times for each of these, which we compare with the running times reported in the original paper.

Using various solving methods, we attempted to solve the given instances to optimality. First, we simply fed the formulation of the Casting Problem into a mathematical programming solver. We fed both the CP formulation (see Section 4.1) and the formulation using auxiliary variables (AVF) (see Section 4.3). The next two methods are based on the disaggregated formulation of the Casting Problem (see Section 4.4). The first of these methods is an off-the-shelf programming package specifically aimed at solving column-generation problems. The other method involves the disaggregated formulation and the specific set of variables as explained at the end of Section 4.4. We give more details about the instances that we use in Section 4.5.1 and finally present the running times in Section 4.5.2.

4.5.1 Datasets

We consider three different sets of instances, originally presented by Deb and Myburgh [2017], for which the numbers and sizes are given in Table 4.1, Table 4.2 and Table 4.3. All instances contain only 10 unique item weights, the number of items per weight differs per instance. The first set contains 3 small instances, in which the number of knapsacks ranges from 31 to 200, and all the knapsacks have the same capacity. The second set contains only one instance which is significantly larger and contains 100,000 knapsacks, with only 2 different knapsack capacities. The last set contains 10 instances, in which the number of knapsacks ranges from 5,000 to 100,000,000. As for the second instance set, the number of unique knapsack capacities in each of these instances is only 2.

	1	2	3	4	5	6	7	8	9	10	# knapsacks
w_j	175	145	65	55	95	75	195	20	125	50	$b_1 = 650$
d_j for 1a	20	20	20	20	20	20	20	20	20	20	31
d_j for 1b	63	65	65	65	65	65	65	65	65	65	100
d_j for 1c	127	130	130	130	130	130	130	130	130	130	200

Table 4.1: Instance 1a, 1b and 1c.

	1	2	3	4	5	6	7	8	9	10	# knapsacks	# knapsacks
w_j	175	145	65	55	95	75	195	20	125	50	$b_1 = 650$	$b_2 = 500$
d_j for 2	59227	58329	53327	53229	53429	53526	57022	52322	58229	52026	43480	56520

Table 4.2: Instance 2.

	w_j	3a	3b	3c	3d	3e	3f	3g	3h	3i	3j
1	79	6240	12560	62000	125600	620025	1255980	6200270	12559745	61649750	123649750
2	66	6262	12562	62545	125620	625450	1256200	6254500	12562000	61706480	122647195
3	31	6217	12517	62517	125170	625170	1251700	6251700	12517000	61609500	123609500
4	26	6267	12567	62567	125670	625670	1256700	6256700	12567000	61752675	123752675
5	44	6262	12562	62562	125620	625620	1256200	6256200	12562000	61654900	123654900
6	35	6172	12172	62172	121720	621720	1217200	6217200	12172000	61680400	123680400
7	88	6076	12076	60576	120760	605760	1207600	6057600	12076000	61621160	122621160
8	9	6052	12052	60552	120520	605520	1205200	6055200	12052000	61621600	123621600
9	57	6017	12017	60517	120170	605170	1201700	6051700	12017000	61621600	123621600
10	22	6012	12012	60512	120120	605120	1201200	6051200	12012000	61652160	123652160
$\|M(2)\|$	$b_1 = 650$	2174	4348	21740	43480	217400	434800	2174000	4348000	21740000	43480000
$\|M(1)\|$	$b_2 = 500$	2826	5652	28260	56520	282600	565200	2826000	5652000	28260000	56520000

Table 4.3: Instance 3a, 3b, 3c, 3d, 3e, 3f, 3g, 3h, 3i and 3j. Note that compared to Table 4.1 and Table 4.2, this table is transposed to optimize space.

4.5.2 Running Times

The first two methods involved solving the CP formulation and the Auxiliary Variable Formulation (see Section 4.3) with Gurobi, respectively, using version 10.0.3 [Gurobi Optimization, LLC, 2023]. As mentioned in Section 4.1.2, Deb and Myburgh [2017] terminate the optimization whenever a solution is found with a utilization ratio of $\eta = 0.997$. To be able to compare with the results by Deb and Myburgh [2017], we have also solved the AVF with this termination criterion and refer to it as the *objective limit* (OL) method. We show in section 4.4 that the casting problem is easily decomposed into smaller knapsack problems and can be solved with column generation. Therefore the fourth solving technique used the GCG (Generic Column Generation) package (PyGCGOpt version 0.1.4) [Gamrath and Lübbecke, 2010], which is part of the SCIP Optimization Suite (PySCIPOpt version 4.2.0) [Bestuzheva et al., 2021] and is specifically aimed at

solving problems with column generation. For the fifth solving technique, we used the RMP of the disaggregated formulation, as explained in Section 4.4, we let P_1 be the set of all assignments that consume all capacity on the small knapsack, and we let P_2 be the set of all variables that have a utilization ratio larger than a threshold of 0.98. When solving the instances with this method, we have always been able to find a feasible solution with the created variables in P_1 and P_2, which, as explained in Section 4.4, means we always found an optimal solution.

The optimization running times for all solving techniques can be seen in Table 4.4. We compare our running times with the results reported by Deb and Myburgh [2017], denoted with DM. For the first four solving methods, we terminated the optimization run after 1 hour, and did not try to solve the instances with more than 50.000 knapsacks. Both the CP and the AVF formulation solved with Gurobi do not terminate within this hour for the instances with more than 200 knapsacks. For the instance 1b and 1c, we observe that adding the auxiliary variables significantly decreases the optimization time. The objective limit method shows that it is possible to find a solution for the larger instances 3a and 3b within an hour if we settle for a solution that has a score above the objective limit of 0.997. Note that the times reported by Deb and Myburgh [2017] also have this objective limit, but their running times are much smaller for these instances. Even though GCG is built specifically to solve problems with column generation, this method was not able to beat any of the other methods in running time. For the instances with more than 200 knapsacks, GCG did not terminate since it ran into memory issues (MI). From the table, we can see that whenever the instances grow in size, the optimization time reported by Deb and Myburgh [2017] grows accordingly. In contrast, the optimization time for solving the RMP does not increase with the problem size, which is probably caused by the aggregation of both the items and the knapsacks in this formulation. For the five largest instances, this means we have a significantly smaller optimization time than the times reported by Deb and Myburgh [2017], even though our method guarantees that the returned solution is optimal, while the original paper does not have this guarantee.

4.6 Conclusion and Discussion

In this chapter, we proposed two new formulations for the Casting Problem problem. The aim was to solve given instances of the problem to optimality. We found that the auxiliary variable formulation succeeded in solving the smaller instances faster than the standard formulation. However, just like the standard formulation, no solutions were found for the larger instances.

The second formulation that we used was the disaggregated formulation, for which we created specific columns, guaranteeing that any feasible solution with these columns is optimal. We achieved this by exploiting the characteristics of the

Technique		CP	AVF	OL	GCG	RMP	DM
Optimal?		Yes	Yes	No	Yes	Yes	No
Inst.	m	time (s)	time (s)	time(s)	time (s)	time (s)	time (s)
1a	31	0.01	0.06	0.06	22.29	0.23	0.04
1b	100	1.50	0.75	0.75	2.60	0.22	0.05
1c	200	6.65	1.28	1.28	34.31	0.11	0.19
2	100k	>3,600	>3,600	>3,600	MI	0.09	250
3a	5k	>3,600	>3,600	350	MI	88	7
3b	10k	>3,600	>3,600	1,213	MI	83	26
3c	50k	>3,600	>3,600	>3,600	MI	119	143
3d	100k	-	-	-		122	308
3e	500k	-	-	-		4,404	1,749
3f	1M	-	-	-		3,656	4,207
3g	5M	-	-	-		3,503	24,000
3h	10M	-	-	-		584	47,593
3i	50M	-	-	-		1,151	261,951
3j	100M	-	-	-		139	535,503

Table 4.4: Running time in seconds for the given instances for solving CP and AVF with Gurobi, solving AVF with Gurobi with an objective limit, using GCG, solving the RMP with P_1 and P_2 as indicated and the running times reported by Deb and Myburgh [2017]. Moreover, the table indicates which solving method guarantees that the solution it returns is optimal.

instances, namely a small number of unique item weights and knapsack capacities. With this method, we found optimal solutions within ∼75 minutes, even for the largest instance containing 100 million knapsacks. This contrasts with earlier known methods, which only found approximate solutions in more than 6 days on specially equipped computers.

We acknowledge that our method of solving the disaggregated formulation also has limitations. We assume that the number of unique item weights and unique knapsack capacities remains constant. However, as we also explain at the end of Section 4.4, this assumption does not guarantee that our proposed method will find a feasible solution. If no feasible solution is found, one could either increase the number of columns in P_1 and P_2, or use standard column generation, together with a branch-and-price framework as explained in Section 4.2. Fortunately, we found feasible, and therefore optimal, solutions for all considered instances within reasonable time.

Chapter 5

Polynomial Time Bicriteria Approximation Scheme for Casting Problem

5.1 Introduction

In this chapter, we consider the CASTING PROBLEM again, as defined in Section 4.1 of the previous chapter. In Section 4.2 of the previous chapter, we have shown that the CASTING PROBLEM is NP-complete. This means that we cannot expect to find a polynomial-time algorithm that finds an optimal solution for the casting problem; in fact, this even rules out that a polynomial-time approximation algorithm exists, unless P = NP. In light of such intractability results, a common approach used in the literature is to consider *bicriteria approximation algorithms* instead [Lin and Vitter, 1992, Shmoys and Tardos, 1993]. Informally, an (α, β)-*bicriteria approximation algorithm* is an algorithm that finds a solution with an objective value of at least α times the value of an optimal feasible solution, with $0 \leq \alpha \leq 1$, and violates the knapsack capacities by at most a factor $\beta \geq 1$. The formal definition follows in Section 5.2.

As a warm-up, we first derive a $(1, 3/2)$-bicriteria approximation algorithm for the casting problem in Section 5.4. The underlying idea of our bicriteria approximation algorithm is as follows: for each knapsack, we define which items are *large* and *small* with respect to this knapsack. We create a solution by first greedily adding large items to the knapsacks; these items are assigned integrally. We then extend this solution by greedily adding small items to the knapsacks, which can be assigned fractionally. Finally, we round the obtained fractional solution to an integral one by using a procedure inspired by Shmoys and Tardos [1993]. We prove that the obtained solution has a value greater than or equal to the optimal objective value (i.e., $\alpha = 1$) and violates each capacity constraint by

at most a factor $3/2$ (i.e., $\beta = 3/2$).

We use a novel technique to compute both bicriteria approximation algorithms presented in this chapter. The basis of this technique is a partial relaxation of (CP), which we refer to as the Small Item Relaxation. The Small Item Relaxation is a mixed integer program (MIP) in which the integrality constraint (4.1d) for the decision variables of a predefined set of small items is relaxed. By finding an optimal solution to the Small Item Relaxation and rounding it to an integral solution, we obtain an approximate solution to the Casting Problem.

Before presenting the main result of this chapter, we want to highlight that if we generalize the Casting Problem problem only slightly, we cannot expect to find anything better than the warm-up algorithm. We exploit an inapproximability result on the *unrelated machine scheduling problem*, of which the related machine scheduling problem is a special case. Unrelated machine scheduling is like related machine scheduling, except for each job j and each machine i, a processing time p_{ij} is given. Lenstra et al. [1990] show that for unrelated machine scheduling no polynomial-time algorithm exists that finds a solution within β times the deadline, for $\beta < 3/2$, unless P = NP. This means that if we would generalize the Casting Problem to a problem in which the weights are defined as $(w_{ij})_{i \in M j \in N}$, no $(1, \beta)$-bicriteria approximation algorithm could exist for $\beta < 3/2$, unless P = NP.

As the main result of this chapter, we derive a $(1/(1+\epsilon), (1+\epsilon)(1+\epsilon+\epsilon^3))$-bicriteria approximation algorithm for any $0 < \epsilon \leq 1$ in Section 5.5. Here, we define for each knapsack which items are *large*, *medium* or *small*. Similarly to the warm-up algorithm, we compute a partially integral solution to the Small Item Relaxation such that all large and medium items are assigned integrally, while the small items can be assigned fractionally. To obtain this partially integral solution, we construct a weighted and directed acyclic graph with a start node s and an end node t. By construction of our graph, each path from s to t corresponds to a fractional packing whose length is equal to the value of the packing. We can efficiently find a maximum-length path as the underlying graph is directed acyclic. In short, this works by relaxing the outgoing arcs of the vertices in topological order, we refer to Cormen et al. [2022] for a detailed explanation. We show that a maximum-length path corresponds to a partially integral solution of the largest value. We use the same rounding procedure as in the warm-up algorithm to obtain an integer solution. We prove that the obtained solution has an objective value that is at least $1/(1+\epsilon)$ times the optimal value (i.e., $\alpha = 1/(1+\epsilon)$) and violates each capacity constraint by at most a factor $(1+\epsilon)(1+\epsilon+\epsilon^3)$ (i.e., $\beta = (1+\epsilon)(1+\epsilon+\epsilon^3)$).

5.1.1 Our Contributions

- We define a relaxation of the Casting Problem referred to as the Small Item Relaxation, which relaxes some of the integrality constraints in CP. We present a rounding procedure to obtain an integral solution for the Casting Problem from a, possibly fractional, solution to the Small Item Relaxation. Finding a solution to the Small Item Relaxation is the main component of the two bicriteria approximation algorithms that we propose in this chapter.
- First, we give a $(1, 3/2)$-bicriteria approximation algorithm. This algorithm computes a solution that violates the capacity constraint with at most a factor $3/2$ and has a value at least as large as the optimal feasible solution.
- Our main result is a $(1/(1+\epsilon), (1+\epsilon)(1+\epsilon+\epsilon^3))$-bicriteria approximation algorithm. This algorithm builds a layered directed acyclic graph, in which the path with maximum length corresponds to an optimal solution to a relaxation of the casting problem. Applying a rounding procedure yields an integer solution that has a value that is at least $1/(1+\epsilon)$ times the optimal solution value and violates the capacity constraints with a factor of at most $(1+\epsilon)(1+\epsilon+\epsilon^3)$. This algorithm can be extended to a *polynomial-time bicriteria approximation scheme* (PTBAS) for the casting problem, which is a family of algorithms that produces an (α, β)-bicriteria approximation algorithm for the casting problem for every value of α and β such that $0 \leq \alpha \leq 1$ and $\beta \geq 1$.

5.1.2 Related Work

As explained in the previous chapter, the casting problem is a special case of GAP. GAP was first introduced by Ross and Soland [1975]. After that, many variants of GAP appeared in the literature, of which we name a few (see e.g. [Ross and Soland, 1975, Martello and Toth, 1990, Wolsey, 2020]). A first variation to GAP is when the problem is defined as a minimization problem, with costs instead of profits. This variation is equivalent to the maximization version of GAP, a proof for the equivalence is given in the chapter about GAP by Martello and Toth [1990]. Another variation, one that is not equivalent, but also appears under the name GAP in the literature, is when the packing may consist of a subset of the items, i.e., it is not required to assign all items to knapsacks. The assignment constraint (4.1c) in the MIP for that formulation is then as follows:

$$\sum_{i \in M} x_{ij} \leq 1 \quad j \in N.$$

By allowing only a subset of the items being assigned, the feasibility problem is no longer an NP-complete problem, which makes it possible to derive approximation

algorithms for this variation of GAP. To distinguish it from GAP in which all items need to be assigned, we denote this variation of GAP as *subset-GAP*. Subset-GAP is mentioned in [Martello and Toth, 1990, Kellerer et al., 2004, Chekuri and Khanna, 2005, Nutov et al., 2006, Wolsey, 2020].

GAP is NP-hard. In fact, as for the Casting Problem, even the problem of finding a feasible solution for GAP is already NP-complete, see e.g. [Chekuri and Khanna, 2005]. This has motivated the study of bicriteria approximations. A $(1+\epsilon, 2+1/\epsilon)$-bicriteria approximation algorithm for GAP was given by Lin and Vitter [1992]. A rounding procedure introduced by Shmoys and Tardos [1993] yields a $(1, 2)$-bicriteria approximation. The same rounding procedure is used in the bicriteria algorithms presented in this chapter. Moreover, the result by Lenstra et al. [1990] implies that for GAP, no polynomial-time $(1, \beta)$-bicriteria approximation can exist for $\beta \leq 3/2$, unless P = NP. As opposed to the feasibility problem of GAP, the feasibility problem of subset-GAP is not NP-complete. The $(1, 2)$-bicriteria approximation by Shmoys and Tardos [1993] is used by Chekuri and Khanna [2005] to give a 2 approximation for subset-GAP. For knapsack-independent profits, an improvement for subset-GAP is given by Nutov et al. [2006], who present a $(1 - 1/\epsilon)$-approximation algorithm.

As mentioned in the previous chapter, the problem of finding a feasible solution to the casting problem is equivalent to the decision version of related machine scheduling. In the decision version of related machine scheduling, the question is if it is possible to schedule all jobs before some given timeline T. Hochbaum and Shmoys [1988] give an algorithm that produces a machine schedule that violates the timeline constraint by a factor at most β (we call this *β-feasible*), if it exists. The algorithm works by constructing a large layered graph based on the instance including a source and a sink node. They show that any path leading from the source to the end node corresponds to a β-feasible solution for the uniform machine scheduling problem, and any feasible solution has a corresponding path in the graph. The decision problem of related machine scheduling is therefore equivalent to the problem of finding any path from the source to the sink node in the constructed graph.

We exploit the relation between the casting problem and the uniform scheduling problem to create our bicriteria approximation algorithm. Like in Hochbaum and Shmoys [1988], we construct a large layered graph to construct solutions that are almost feasible. This graph has the property that each s, t-path through it corresponds with a casting solution in which each knapsack has almost enough capacity for items planned on it. However, the challenge in our case is to create an almost feasible solution with a large objective value. We deal with the objective function of the casting problem by not just considering *any* path in the graph like in Hochbaum and Shmoys [1988], but instead finding the path of *maximum length*. We need to define the edge lengths in such a way that the length of the path exactly corresponds with the objective value of the corresponding casting solution. It is challenging to define these edge lengths, since, even though a path

in the graph by Hochbaum and Shmoys [1988] ensures that all items assigned to a knapsack fit on that knapsack, the path does not specify the total weight of items assigned to the knapsack. The total weight of items assigned to a knapsack is exactly what we need to compute the edge lengths; therefore we need to increase the complexity of the graph such that a path does specify the total weight of items assigned to a knapsack.

A generalization of Hochbaum and Shmoys [1988] is given in Epstein and Sgall [2004], where a PTAS is given for related machine scheduling problems with several very general objective functions. This generalization seems not to be useful for the casting problem, since it does not give the possibility to limit the makespan to some time limit and deal with the casting objective function simultaneously.

5.1.3 Organization of Chapter

In the next section, we present definitions that we use throughout this chapter, namely of the bicriteria approximation algorithm and of the polynomial-time bicriteria approximation scheme. After that, we present the Small Item Relaxation in Section 5.5.2. The following two sections both introduce a bicriteria approximation algorithm: the $(1, 3/2)$-bicriteria approximation algorithm in Section 5.4 and the $(1/(1+\epsilon), (1+\epsilon)(1+\epsilon+\epsilon^3))$-bicriteria approximation algorithm in Section 5.5.

5.2 Preliminaries

Since the feasibility problem of the Casting Problem is NP-complete, we cannot expect to find a polynomial-time approximation algorithm. Therefore, we will consider bicriteria approximation algorithms, which allow a violation of the capacity constraints by a (small) multiplicative factor.

Definition 5.2.1. For an instance $(M, N, (b_i)_{i\in M}, (w_j)_{j\in N})$ of the Casting Problem, a solution $\boldsymbol{x}$ is *β-feasible* for $\beta \geq 1$ if constraints (4.1c) and (4.1d) hold for $\boldsymbol{x}$ and

$$\sum_{j\in N} w_j x_{ij} \leq \beta \cdot b_i \quad \forall i \in M.$$

Let $\boldsymbol{x}^*$ denote the optimal solution for the Casting Problem. This allows us to define a bicriteria approximation algorithm as follows.

Definition 5.2.2. An algorithm is an *(α, β)-bicriteria approximation algorithm* for the Casting Problem with $0 \leq \alpha \leq 1$ and $\beta \geq 1$ if for every instance $(M, N, (b_i)_{i\in M}, (w_j)_{j\in N})$ it computes a β-feasible solution $\boldsymbol{x}$ of cost $c(\boldsymbol{x}) \geq \alpha \cdot$

$c(\boldsymbol{x}^*)$ in time that is polynomially bounded in the input size of the casting instance. An $(\alpha, 1)$-bicriteria approximation is also simply called an *α-approximation algorithm.*

Note that the input size of an instance of the CASTING PROBLEM is $\mathcal{O}(n \log(\max_j w_j) + m \log(\max_i b_i))$. If we can find an (α, β)-bicriteria approximation algorithm for every possible value of α and β, we have found a *polynimal-time bicriteria approximation scheme* (PTBAS).

Definition 5.2.3. A *polynomial-time bicriteria approximation scheme* (PTBAS) for the CASTING PROBLEM is an algorithm that, for any given constants α and β with $0 \leq \alpha \leq 1$ and $\beta \geq 1$, produces a β-feasible solution $\boldsymbol{x}$ of cost $c(s) \geq \alpha \cdot c(\boldsymbol{x}^*)$. The running time of a PTBAS must be polynomial in the input size of the casting instance for any fixed α and β, but may depend exponentially on $1/\alpha$ and $1/\beta$.

In the proof of Lemma 5.3.2 about the rounding procedure, the notion of totally unimodularity is used. A matrix A is called *totally unimodular* if each square submatrix of A has determinant equal to $0, +1,$ or -1 (see, e.g., [Schrijver et al., 2003]). The following lemma directly follows from the definition of totally unimodularity:

Lemma 5.2.4. *If a matrix A is totally unimodular, any matrix B defined by a subset of the rows of A is also totally unimodular.*

Proof:
Every square submatrix of B is also a square submatrix of A. Since A is totally unimodular, the square submatrix has determinant equal to $0, +1$ or -1. □

5.3 Small Item Relaxation

5.3.1 Mixed Integer Program

In this section, we define a relaxation of the CASTING PROBLEM, which we term the SMALL ITEM RELAXATION, which relaxes some of the integrality constraints based on a given knapsack-dependent classification of small and large items. By finding an optimal solution to this relaxation and rounding it to an integral solution, we obtain an approximate solution to the CASTING PROBLEM. In the next two sections, we will give two different bicriteria approximation algorithms for the CASTING PROBLEM. For both of them, we define a knapsack-dependent item classification and give an algorithm that solves the SMALL ITEM RELAXATION.

We say that an item $j \in N$ *fits* on a knapsack $i \in M$ if $w_j \leq b_i$. This allows us to define the notion of a *small item collection*, which we will use to define the Small Item Relaxation.

Definition 5.3.1. For each knapsack $i \in M$, let $N^S(i) \subseteq N$ be a subset of items. We say that the collection of $\{N^S(i)\}$ for $i \in M$ is a *small item collection* if for each $i \in M$ such that $N^S(i) \neq \emptyset$ it holds that there is a so-called *split item* $j(i) \in N$ that fits on i such that $N^S(i)$ contains exactly those items with weight at most $w_{j(i)}$, i.e., $N^S(i) = \{j \in M \ : \ w_j \leq w_{j(i)}\}$.

Observe that each $j(i)$ splits N into a set of items $N^S(i)$ with a weight smaller than or equal to $j(i)$, and a set of items $N \setminus N^S(i)$ with a weight larger than $j(i)$.

We can now introduce the following mixed integer program, referred to as the Small Item Relaxation (SIR). SIR relaxes the ILP (CP) in two ways: (1) it allows each small item $j \in N^S(i)$ to be assigned fractionally to i, and (2) it increases the capacity constraint for each knapsack $i \in M$ to a value $\hat{b}_i \geq b_i$, but the extra capacity $\hat{b}_i - b_i$ may only be used by items in $N^S(i)$. Formally, given any small item collection $N^S(i)$ and an inflated capacity $\hat{b}_i \geq b_i$ for each $i \in M$, the Small Item Relaxation is defined as follows:

$$\text{maximize} \quad c(\boldsymbol{x}) = \sum_{i \in M} \frac{1}{b_i} \sum_{j \in N} w_j x_{ij} \tag{5.1a}$$

$$\text{subject to} \quad \sum_{j \in N \setminus N^S(i)} w_j x_{ij} \leq b_i \qquad \forall i \in M \tag{5.1b}$$

$$\sum_{j \in N} w_j x_{ij} \leq \hat{b}_i \qquad \forall i \in M \tag{5.1c}$$

$$\sum_{i \in M} x_{ij} = 1 \qquad \forall j \in N \tag{5.1d}$$

$$x_{ij} \in \{0,1\} \quad \forall i \in M, \ \forall j \in N \setminus N^S(i) \tag{5.1e}$$

$$x_{ij} \geq 0 \qquad \forall i \in M, \ \forall j \in N^S(i) \tag{5.1f}$$

Note that $N \setminus N^S(i)$ might contain items that do not fit on knapsack i. However, constraints (5.1b) and (5.1e) ensure that for each item j that does not fit into i, x_{ij} is set to zero. We give some intuition on the effect of $N^S(i)$. As $N^S(i)$ varies from all items ($N^S(i) = N$) to nothing ($N^S(i) = \emptyset$), the Small Item Relaxation interpolates between the LP relaxation of (CP) with respect to the inflated capacities $\hat{b}_i$ and the ILP (CP). If the integrality constraint is relaxed for many items, it becomes easier to find an optimal solution to the relaxation, but the resulting approximation guarantees for the bicriteria approximation algorithm become worse. This is also what our two applications in Section 5.4 and Section 5.5 demonstrate: In Section 5.4, the small item collection contains many items and we derive a simple algorithm to compute an optimal solution to the Small Item Relaxation. This yields a $(1, 3/2)$-bicriteria approximation algorithm. On

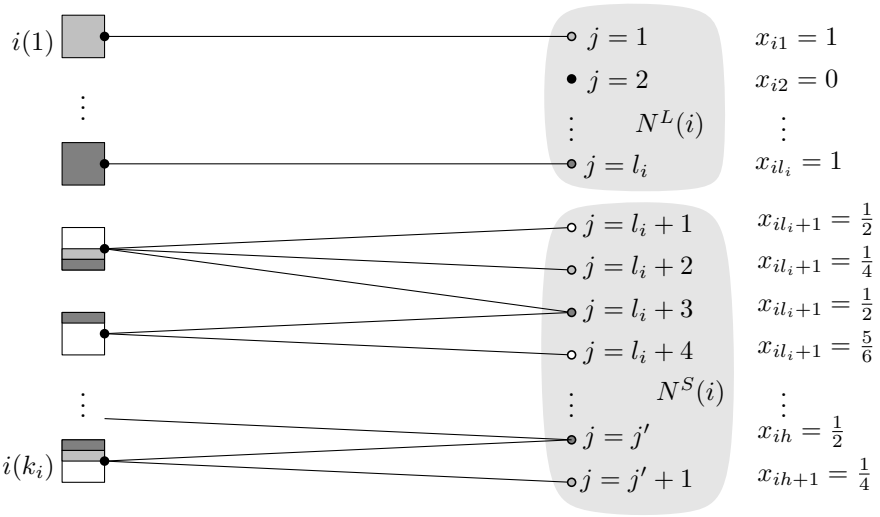


Figure 5.1: Example of graph G in LP-rounding procedure.

the other hand, in Section 5.5, the small item collection contains fewer items and we need to develop a more involved algorithm to compute an optimal solution to the respective Small Item Relaxation. In return, we can use this algorithm to obtain a polynomial-time bicriteria approximation scheme for the Casting Problem.

5.3.2 LP-Rounding Procedure

A crucial procedure in our bicriteria approximation algorithms is to round an optimal solution to SIR to an integral solution in order to obtain an approximate solution to the Casting Problem. To accomplish this, we apply a procedure introduced by Shmoys and Tardos [1993].

Let $\boldsymbol{x}$ be a feasible solution to SIR. In the rounding procedure, a bipartite graph $G = (L \cup R, E)$ is created based on $\boldsymbol{x}$ where the set of items N corresponds to the set R, i.e., $R = N$, and each knapsack $i \in M$ has $k_i = \lceil \sum_{j \in N} x_{ij} \rceil$ slots $i(1), \ldots, i(k_i)$ in L. We assume without loss of generality that the items are ordered by non-increasing weights, i.e., $w_1 \geq w_2 \geq \cdots \geq w_n$.

We construct the bipartite graph G by iteratively adding edges for each knapsack $i \in M$ as explained below. See Figure 5.1 for an example of such a graph G. Simultaneously, we define a fractional matching $y \in \mathbb{R}^E$ of G. The construction proceeds as follows:

1. If $\sum_{j \in N} x_{ij} \leq 1$, then k_i is 1 and there is only one slot for knapsack i. In that case we add an edge $(i(1), j)$ to E for each item j with $x_{ij} > 0$, and for each of these edges we set $y_{(i(1),j)} = x_{ij}$.

2. Otherwise, we find the minimum index h such that $\sum_{j=1}^{h} x_{ij} \geq 1$. We add

an edge $(i(1), j)$ to E for each item $j = 1, \ldots, h-1$ with $x_{ij} > 0$, and for each of these edges we set $y_{(i(1),j)} = x_{ij}$. Furthermore, we add the edge $(i(1), h)$ to E and set $y_{(i(1),h)} = 1 - \sum_{j=1}^{h-1} x_{ij}$. This ensures that the sum of y_e for all edges e adjacent to $i(1)$ is exactly 1.

3. If $\sum_{j=1}^{h} x_{ij} > 1$, a fraction of h is still unassigned, and we add an edge $(i(2), h)$ to E and set $y_{(i(2),h)} = x_{ih} - y_{(i(1),h)} = \sum_{j=1}^{h} x_{ij} - 1$. Likewise, if $k_i > 2$, we continue adding edges between item $j > h$ and slot $i(2)$ until the sum of y_e for all edges e adjacent to $i(2)$ is exactly 1.

4. We repeat this for each slot up to and including slot $i(k_i - 1)$. Now, let j' be the last (smallest) item that is assigned to $i(k_i - 1)$ (and possibly to $i(k_i)$). We add an edge $(i(k_i), j)$ to E for each $j > j'$ that has $x_{ij} > 0$ and set $y_{(i(k_i),j)} = x_{ij}$.

By proceeding in this way, there will be exactly one or two edges in G for each positive coordinate of $\boldsymbol{x}$. We set the weight of each edge $(i(s), j)$ in G equal to w_j/b_i.

We say that a (possibly fractional) matching y of G is *R-perfect* if the sum of the fractional values of the edges in the matching that are incident to each item $j \in R$ equals one, i.e.,

$$\sum_{i:\{i,j\}\in G} y_{(i,j)} = 1.$$

By construction, it follows that the constructed y indeed defines a fractional matching in G with weight exactly equal to $c(\boldsymbol{x})$ that is R-perfect.

Finally, the LP-rounding procedure is then as follows:

1. Based on a feasible, possibly fractional, solution $\boldsymbol{x}$ for SIR, create G as described above.

2. Out of all the integral matchings in G that are R-perfect, let $\bar{y}$ be the one with maximum weight. This can be computed in polynomial time in bipartite graphs.[1]

3. Let $\boldsymbol{x}^{\text{round}}$ be the assignment of items to knapsacks that we obtain from $\bar{y}$ by assigning item j to knapsack i if and only if there is a slot $i(s)$ such that $\bar{y}_{(i(s),j)} = 1$.

In the following lemma, we show that $\bar{y}$ always exists, since y is an R-perfect fractional matching.

[1] The Hungarian method (as explained in Section 17.2 of [Schrijver et al., 2003], but introduced by Kuhn [1955], and improvements given by Munkres [1957], Iri [1960], Edmonds and Karp [1972], Tomizawa [1971]) can be used to find an integral R-perfect matching in the graph G. Theorem 17.2 in [Schrijver et al., 2003] proves that this matching is the integral R-perfect matching with maximum weight.

Lemma 5.3.2. *Let y be an R-perfect fractional matching in a bipartite weighted graph $G = (L \cup R, E)$. Then there exists an R-perfect integer matching $\bar{y}$, with a weight at least as large as the weight of y.*

Proof:
Since the graph G is bipartite, by Corollary 18.1 in [Schrijver et al., 2003], the perfect matching polytope of G (defined as the convex hull of the incidence vectors of perfect matchings in G) is equal to the vectors $z \in \mathbb{R}^E$ for which it holds that $z \geq \mathbf{0}$ and

$$\sum_{e \ni v} z^e = 1 \quad \forall v \in L \cup R.$$

These constraints can be split into the following equivalent inequalities:

$$\sum_{e \ni v} z^e \leq 1 \quad \forall v \in L \cup R, \qquad -\sum_{e \ni v} z^e \leq -1 \quad \forall v \in L \cup R.$$

Since G is bipartite, it follows from Theorem 18.2 in [Schrijver et al., 2003] that the matrix corresponding to these constraints is totally unimodular. Then by Lemma 5.2.4, the matrix corresponding to the following subset of these constraints is also totally unimodular:

$$\sum_{e \ni v} z^e \leq 1 \quad \forall v \in L \cup R, \qquad -\sum_{e \ni v} z^e \leq -1 \quad \forall v \in R.$$

This set of constraints is equivalent to the following set of constraints, for which the corresponding matrix is therefore also totally unimodular:

$$\sum_{e \ni v} z^e \leq 1 \quad \forall v \in L, \qquad \sum_{e \ni v} z^e = 1 \quad \forall v \in R.$$

These constraints precisely describe an R-perfect matching. Since the corresponding matrix is totally unimodular, it follows that the extreme points of the corresponding polytope are integral, by Theorem 5.20 in [Schrijver et al., 2003]. This must mean that $\bar{y}$ exists and has weight at least as large as the weight of y. □

In Theorem 5.3.3, we present the result of applying the rounding procedure to solutions of SIR. Let $w(i, \boldsymbol{x})$ denote the total weight that is assigned to knapsack i with respect to $\boldsymbol{x}$, i.e.,

$$w(i, \boldsymbol{x}) = \sum_{j \in N} w_j x_{ij}.$$

Theorem 5.3.3. *Let $\boldsymbol{x}$ be a solution to SIR. Let $\boldsymbol{x}^{round}$ be the result of applying the above LP-rounding procedure to $\boldsymbol{x}$. Then it holds that $\boldsymbol{x}^{round}$ satisfies Constraint* (4.1c) *and for each $i \in M$:*

$$w(i, \boldsymbol{x}^{round}) \leq w(i, \boldsymbol{x}) + \max_{j \in N^S(i)} w_j.$$

Moreover, it holds that $c(\boldsymbol{x}^{round}) \geq c(\boldsymbol{x})$.

Proof:
As shown above, the fractional matching created in the rounding procedure, y, is R-perfect. By Lemma 5.3.2, this proves the existence of $\bar{y}$, where $\bar{y}$ is also R-perfect. This proves that $\boldsymbol{x}^{\text{round}}$ assigns each item to a knapsack, and therefore Constraint (4.1c) holds.

To prove the second part of the theorem, fix knapsack i. Let $m(i(s))$ be the maximum weight of an item j having an edge to $i(s)$ in G. We use $N^L(i)$, referred to as the *large items*, to denote all the items that are not small for knapsack i, i.e.: $N^L(i) := N \setminus N^S(i)$. Let l_i be the number of items from $N^L(i)$ assigned to i. Since $x_{ij} \in \{0,1\}$ for these large items, and since these large items are connected to slots before small items, it must hold that each slot $i(s)$ for $s = 1, \ldots, l_i$ is connected to exactly one large item in G, and we have that

$$\sum_{s=1}^{l_i} m(i(s)) = \sum_{j \in N^L(i)} w_j x_{ij}. \tag{5.2}$$

If $\sum_{j \in N^S(i)} x_{ij} \leq 1$, at most one item of $N^S(i)$ is assigned to i in $\boldsymbol{x}^{\text{round}}$ and the second part of the theorem holds. Otherwise $k_i \geq l_i + 2$ and the slots $i(l_i+1)$ up to $i(k_i)$ are all connected to items in $N^S(i)$ only. It holds that:

$$\begin{aligned} w(i, \boldsymbol{x}^{\text{round}}) = \sum_{j \in N} w_j x_{ij}^{\text{round}} &\leq \sum_{s=1}^{k_i} m(i(s)) \\ &= \sum_{s=1}^{l_i} m(i(s)) + m(i(l_i+1)) + \sum_{s=l_i+2}^{k_i} m(i(s)) \\ &\leq \sum_{j \in N^L(i)} w_j x_{ij} + \max_{j \in N^S(i)} w_j + \sum_{s=l_i+2}^{k_i} m(i(s)). \end{aligned}$$

The first inequality holds by definition of $m(i(s))$. The second inequality holds by (5.2) and since slot $l_i + 1$ is connected to items in $N^S(i)$ only. We focus on the third term and rewrite it as follows:

$$\sum_{s=l_i+2}^{k_i} m(i(s)) = \sum_{s=l_i+1}^{k_i-1} m(i(s+1)).$$

For $s = l_i+1, \ldots, k_i-1$ it holds that $\sum_{j \in N^S(i)} x_{i(s)j} = 1$ and therefore $m(i(s+1)) = \sum_{j \in N^S(i)} m(i(s+1)) x_{i(s)j} \leq \sum_{j \in N^S(i)} w_{ij} x_{i(s)j}$. Here, the last inequality holds since $m(i(s+1)) \leq w_{ij}$ for all items j that have an edge to $i(s)$, since the items

are added to slots in decreasing order. Therefore it holds that

$$\begin{aligned}\sum_{s=l_i+1}^{k_i-1} m(i(s+1)) &\leq \sum_{s=l_i+1}^{k_i-1} \sum_{j\in N^S(i)} w_{ij}x_{i(s)j} \\ &\leq \sum_{j\in N^S(i)} \sum_{s=l_i+1}^{k_i} w_{ij}x_{i(s)j} \\ &= \sum_{j\in N^S(i)} w_{ij}x_{ij}.\end{aligned}$$

The last equality holds by construction of the bipartite graph. Together, this proves the second part of the theorem:

$$w(i,\boldsymbol{x}^{\text{round}}) \leq \sum_{j\in N^L(i)} w_j x_{ij} + \max_{j\in N^S(i)} w_j + \sum_{j\in N^S(i)} w_{ij}x_{ij} = w(i,\boldsymbol{x}) + \max_{j\in N^S(i)} w_j.$$

Lemma 5.3.2 shows that the weight of $\bar{y}$ is at least the weight of y, which proves the last part of the theorem:

$$c(\boldsymbol{x}) = \sum_{i\in M}\sum_{j\in N} \frac{w_j}{b_i} y_{(i,j)} \leq \sum_{i\in M}\sum_{j\in N} \frac{w_j}{b_i} \bar{y}_{(i,j)} = c\left(\boldsymbol{x}^{\text{round}}\right).$$

□

5.4 Warm-up: $(1, 3/2)$-Bicriteria Approximation

In this section, we derive a bicriteria approximation algorithm for the Casting Problem. The main result of this section is given in the following theorem:

Theorem 5.4.1. *There exists a $(1, 3/2)$-bicriteria approximation algorithm for the* Casting Problem.

To prove the theorem, we first define a small item collection to obtain a Small Item Relaxation that we use in this section. After that, we show that a simple greedy algorithm produces an optimal solution to this relaxation. Combining this algorithm with the rounding scheme described in the previous section yields the respective bicriteria approximation algorithm.

Throughout this section, we assume without loss of generality that the knapsacks are ordered by non-decreasing capacities, i.e., $b_1 \leq b_2 \leq \cdots \leq b_m$. To define the small item collection used in this section, we partition the set of items that fit into knapsack $i \in M$ into a set of *small* items $N^S(i)$ and a set of *large* items $N^L(i)$ as follows:

$$N^S(i) = \left\{ j \in N \;\middle|\; w_j \leq \frac{1}{2} b_i \right\} \text{ and } N^L(i) = \left\{ j \in N \;\middle|\; \frac{1}{2} b_i < w_j \leq b_i \right\}. \quad (5.3)$$

Algorithm 8 LargeItemsFirst(M, N)

Order the knapsacks such that $b_1 \leq b_2 \leq \cdots \leq b_m$
$x_{ij}^{\text{MIP}} = 0 \quad \forall i \in M, \forall j \in N$ ▷ *Initialize all x_{ij}^{MIP}'s to zero*
for $i = 1, \ldots, m$ **do** ▷ *In order of non-decreasing capacities*
  if $\left\{j \in N^L(i) \;\middle|\; x_{ij}^{\text{MIP}} = 0\right\} \neq \emptyset$ **then**
    $j = \arg\max \left\{w_j \;\middle|\; j \in N^L(i),\ x_{ij}^{\text{MIP}} = 0\right\}$
    $x_{ij}^{\text{MIP}} = 1$ ▷ *Assign max-weight unassigned item of $N^L(i)$ to i*
  $R = \left\{j \in N^S(i) \;\middle|\; \sum_{i' \in M} x_{i'j}^{\text{MIP}} < 1\right\}$ ▷ *R is the set of remaining small items*
  while $\sum_{j' \in N} w_{j'} x_{ij'}^{\text{MIP}} < b_i$ **and** $R \neq \emptyset$ **do** ▷ *i not full and small item remains*
    $j = \arg\min \{w_j \mid j \in R\}$ ▷ *j is the min-weight item in R*
    $x_{ij}^{\text{MIP}} = 1 - \sum_{i' \in M} x_{i'j}^{\text{MIP}}$ ▷ *Assign j possibly fractionally, while ...*
    $x_{ij}^{\text{MIP}} = \min\left\{x_{ij}^{\text{MIP}}, \frac{1}{w_j}\left(b_i - \sum_{j' \in N} w_{j'} x_{ij'}^{\text{MIP}}\right)\right\}$ ▷ *... adhering capacity*
    $R = R \setminus \{j\}$
return $\boldsymbol{x}^{\text{MIP}}$

Note that $\{N^S(i)\}$ for $i \in M$ is indeed a small item collection according to Definition 5.3.1, and that $N^S(\cdot)$ is monotone: $N^S(1) \subseteq \cdots \subseteq N^S(m)$, while $N^L(\cdot)$ is not. For SIR considered in this section, we set $\hat{b}_i = b_i$ for all knapsacks $i \in M$. Note that by constraints (5.1b) and (5.1e) in SIR, x_{ij} is forced to be zero for each knapsack i and item $j \in N \setminus \left(N^S(i) \cup N^L(i)\right)$; in particular, these are exactly the items that do not fit on knapsack i.

As we will show below, the following simple greedy algorithm, referred to as LargeItemsFirst, computes an optimal solution to SIR. LargeItemsFirst greedily constructs a solution $\boldsymbol{x}^{\text{MIP}}$ to SIR by iterating over the knapsacks in non-decreasing order of their capacities. For each knapsack i, the algorithm first integrally assigns an unassigned large item (if any) of maximum weight to i. Crucially, at most one such item fits on knapsack i, by definition of $N^L(i)$. Then, it greedily assigns small items (possibly fractionally) to knapsack i until either its capacity is reached or no small unassigned items remain. A more detailed description LargeItemsFirst is given in Algorithm 8.

We show that the algorithm can be used to compute an optimal solution to SIR.

Theorem 5.4.2. LargeItemsFirst *computes an optimal solution to SIR with $N^S(i)$ as defined in* (5.3) *and $\hat{b}_i = b_i$ in running time $\mathcal{O}(mn + n\log(n))$.*

In the remainder of this section, when we refer to SIR, we mean SIR with $N^S(i)$ as defined in (5.3) and $\hat{b}_i = b_i$. We first prove an auxiliary lemma that will turn out to be useful for the proof of Theorem 5.4.2 below.

As defined in the previous section, we let $w(i, \boldsymbol{x})$ denote the total weight that is assigned to knapsack i with respect to $\boldsymbol{x}$. Given a feasible solution $\boldsymbol{x}^{\text{MIP}}$ of

SIR, we say that $\boldsymbol{x}^{\text{MIP}}$ *maximizes the weight on the small knapsacks* if for every feasible solution $\boldsymbol{x}'$ of SIR it holds that for every $i \in [m]$:

$$\sum_{k=1}^{i} w(k, \boldsymbol{x}^{\text{MIP}}) \geq \sum_{k=1}^{i} w(k, \boldsymbol{x}').$$

Lemma 5.4.3. *Let $\boldsymbol{x}^{MIP}$ be a feasible solution for SIR that maximizes the total weight on the small knapsacks. Then $\boldsymbol{x}^{MIP}$ is an optimal solution for SIR.*

Proof:
The solution value of $\boldsymbol{x}^{\text{MIP}}$ can be written as follows:

$$c(\boldsymbol{x}^{\text{MIP}}) = \sum_{i \in M} \frac{1}{b_i} \sum_{j \in N} w_j x_{ij}^{\text{MIP}} = \sum_{i \in M} \frac{1}{b_i} w(i, \boldsymbol{x}^{\text{MIP}}).$$

Let $\boldsymbol{x}'$ be any feasible solution for SIR. We will prove that the solution value of $\boldsymbol{x}^{\text{MIP}}$ is at least as large as the solution value of $\boldsymbol{x}'$.

For each $i \in M$, divide $w(i, \boldsymbol{x}^{\text{MIP}})$ into two parts, $w_1(i, \boldsymbol{x}^{\text{MIP}})$ and $w_2(i, \boldsymbol{x}^{\text{MIP}})$, defined as follows:

$$w_1(i, \boldsymbol{x}^{\text{MIP}}) := \sum_{k=1}^{i} w(k, \boldsymbol{x}') - \sum_{k=1}^{i-1} w(k, \boldsymbol{x}^{\text{MIP}})$$
$$w_2(i, \boldsymbol{x}^{\text{MIP}}) := \sum_{k=1}^{i} w(k, \boldsymbol{x}^{\text{MIP}}) - \sum_{k=1}^{i} w(k, \boldsymbol{x}').$$

Then the following properties hold:

1. $w(i, \boldsymbol{x}^{\text{MIP}}) = w_1(i, \boldsymbol{x}^{\text{MIP}}) + w_2(i, \boldsymbol{x}^{\text{MIP}})$.

2. $w(i, \boldsymbol{x}') = w_1(i, \boldsymbol{x}^{\text{MIP}}) + w_2(i-1, \boldsymbol{x}^{\text{MIP}})$, where $w_2(0, \boldsymbol{x}^{\text{MIP}})$ is defined as 0.

3. $w_2(i, \boldsymbol{x}^{\text{MIP}}) \geq 0$, since $\boldsymbol{x}^{\text{MIP}}$ maximizes the weight on the small knapsacks.

By using these properties, we obtain:

$$\begin{aligned}
c(\boldsymbol{x}^{\mathrm{MIP}}) = \sum_{i=1}^{m} \frac{1}{b_i} w(i, \boldsymbol{x}^{\mathrm{MIP}}) &= \sum_{i=1}^{m} \frac{1}{b_i} w_1(i, \boldsymbol{x}^{\mathrm{MIP}}) + \sum_{i=1}^{m} \frac{1}{b_i} w_2(i, \boldsymbol{x}^{\mathrm{MIP}}) \\
&\geq \sum_{i=1}^{m} \frac{1}{b_i} w_1(i, \boldsymbol{x}^{\mathrm{MIP}}) + \sum_{i=1}^{m-1} \frac{1}{b_i} w_2(i, \boldsymbol{x}^{\mathrm{MIP}}) \\
&\geq \sum_{i=1}^{m} \frac{1}{b_i} w_1(i, \boldsymbol{x}^{\mathrm{MIP}}) + \sum_{i=1}^{m-1} \frac{1}{b_{i+1}} w_2(i, \boldsymbol{x}^{\mathrm{MIP}}) \\
&= \sum_{i=1}^{m} \frac{1}{b_i} w_1(i, \boldsymbol{x}^{\mathrm{MIP}}) + \sum_{i=2}^{m} \frac{1}{b_i} w_2(i-1, \boldsymbol{x}^{\mathrm{MIP}}) \\
&= \sum_{i=1}^{m} \frac{1}{b_i} w_1(i, \boldsymbol{x}^{\mathrm{MIP}}) + \sum_{i=1}^{m} \frac{1}{b_i} w_2(i-1, \boldsymbol{x}^{\mathrm{MIP}}) \\
&= \sum_{i=1}^{m} \frac{1}{b_i} w(i, \boldsymbol{x}') = c(\boldsymbol{x}').
\end{aligned}$$

The first inequality holds by the third property. The second inequality holds by both the third property and since knapsacks are sorted in non-decreasing order of capacities. Moreover, we use for the third last equality that $w_2(0, \boldsymbol{x}^{\mathrm{MIP}}) = 0$ by definition. The second last equality holds by the second property again. This concludes the proof. □

Proof of Theorem 5.4.2:
Let $\boldsymbol{x}^{\mathrm{MIP}}$ be a solution returned by Algorithm 8. We prove that $\boldsymbol{x}^{\mathrm{MIP}}$ maximizes the weight on the small knapsacks. By Lemma 5.4.3, $\boldsymbol{x}^{\mathrm{MIP}}$ is then an optimal solution.

Suppose for contradiction, that there is a feasible solution $\boldsymbol{x}'$ for SIR such that for some knapsack i it holds that

$$\sum_{k=1}^{i} w(k, \boldsymbol{x}') > \sum_{k=1}^{i} w(k, \boldsymbol{x}^{\mathrm{MIP}}). \tag{5.4}$$

Without loss of generality, let i be the knapsack with smallest index for which this holds. Consider the number of items from $N^L(i)$ that $\boldsymbol{x}'$ assigned to knapsacks $1, \ldots, i$.

We prove the intermediate claim that $\boldsymbol{x}^{\mathrm{MIP}}$ assigned at least as many items from $N^L(i)$ to knapsacks $1, \ldots, i$ as $\boldsymbol{x}'$. Note that at most one item from $N^L(i)$ fits on each knapsack $1, \ldots, i$. Now, suppose for contradiction of the intermediate claim, that there is a knapsack $i' \leq i$ such that $\boldsymbol{x}'$ assigned more items from $N^L(i)$ to $1, \ldots, i'$ than $\boldsymbol{x}^{\mathrm{MIP}}$ did, and suppose w.l.o.g. that i' is the first knapsack for which this holds. This means $\boldsymbol{x}^{\mathrm{MIP}}$ did not assign any item from $N^L(i)$ to

Algorithm 9 LargeItemsRounded(M, N)

1: $\boldsymbol{x}^{\text{MIP}} = \text{LargeItemsFirst}(M, N)$ ▷ *Compute optimal solution to SIR*
2: Apply LP-rounding procedure to $\boldsymbol{x}^{\text{MIP}}$ to obtain $\boldsymbol{x}^{\text{round}}$
3: **return** $\boldsymbol{x}^{\text{round}}$

i', but only items from $N^S(i)$, even though there is an unassigned item from $N^L(i)$ that fits on i'. This contradicts Line 6 in Algorithm 8 and therefore proves the intermediate claim that $\boldsymbol{x}^{\text{MIP}}$ assigned at least as many items from $N^L(i)$ to knapsacks $1, \ldots, i$.

Moreover, $\boldsymbol{x}^{\text{MIP}}$ always assigned the maximum-weight unassigned item of $N^L(i)$, which together proves that

$$\sum_{k=1}^{i} \sum_{j \in N^L(i)} w_j x_{kj}^{\text{MIP}} \geq \sum_{k=1}^{i} \sum_{j \in N^L(i)} w_j x'_{kj}.$$

Since i is the first knapsack for which (5.4) holds, it must hold that $w(i, \boldsymbol{x}') > w(i, \boldsymbol{x}^{\text{MIP}})$. This must mean that $w(i, \boldsymbol{x}^{\text{MIP}}) < b_i$, so the while loop for knapsack i in Line 8 in Algorithm 8 terminated because R was empty, from which it follows that $\boldsymbol{x}^{\text{MIP}}$ assigned all items from $N^S(i)$ to knapsacks $1, \ldots, i$. We can conclude that

$$\begin{aligned}\sum_{k=1}^{i} w(k, \boldsymbol{x}^{\text{MIP}}) &= \sum_{k=1}^{i} \sum_{j \in N^S(i)} w_j x_{kj}^{\text{MIP}} + \sum_{k=1}^{i} \sum_{j \in N^L(i)} w_j x_{kj}^{\text{MIP}} \\ &\geq \sum_{k=1}^{i} \sum_{j \in N^S(i)} w_j x'_{kj} + \sum_{k=1}^{i} \sum_{j \in N^L(i)} w_j x'_{kj} = \sum_{k=1}^{i} w(k, \boldsymbol{x}'),\end{aligned}$$

which is the contradiction that proves the theorem. The running time follows since after sorting both the knapsacks and the items, each item is considered at most once for each knapsack. □

The LargeItemsRounded algorithm, for which the pseudocode is in Algorithm 9 combines the LargeItemsFirst algorithm and the rounding procedure explained in the previous section. This algorithm is used in the proof of the main result of this section, Theorem 5.4.1.

Proof of Theorem 5.4.1:
Consider an instance of the Casting Problem. Let $\boldsymbol{x}^{\text{round}}$ be the output of the LargeItemsRounded algorithm and let $\boldsymbol{x}^*$ be an optimal solution. Since $\boldsymbol{x}^*$ is feasible for SIR, but $\boldsymbol{x}^{\text{MIP}}$ is optimal for SIR, it holds that $c\left(\boldsymbol{x}^{\text{MIP}}\right) \geq c\left(\boldsymbol{x}^*\right)$. Moreover, Theorem 5.3.3 gives that $c\left(\boldsymbol{x}^{\text{round}}\right) \geq c\left(\boldsymbol{x}^{\text{MIP}}\right)$. Together this gives

that

$$c\left(\boldsymbol{x}^{\text{round}}\right) \geq c\left(\boldsymbol{x}^{*}\right),$$

which proves that $\alpha = 1$. Moreover, Theorem 5.3.3 shows that the weight assigned to a knapsack i is upper bounded by $w(i, \boldsymbol{x}^{\text{MIP}}) + \max_{j \in N^S(i)} w_j$. For the $N^S(i)$ defined in this section, it holds that

$$w(i, \boldsymbol{x}^{\text{MIP}}) + \max_{j \in N^S(i)} w_j \leq b_i + \frac{1}{2} b_i,$$

by the knapsack constraint in SIR and the definition of $N^S(i)$. This proves that $\boldsymbol{x}^{\text{round}}$ is β-feasible for $\beta = 3/2$. □

5.5 PTBAS

In this section, we present a second bicriteria approximation, which will be used to prove the main result of this chapter, as given in the following theorem:

Theorem 5.5.1. *There exists a polynomial-time bicriteria approximation scheme for the* Casting Problem.

To prove this theorem, we first define a small item collection and $\hat{b}_i$ for $i \in M$ to obtain a second version of SIR. In order to solve this SIR to optimality, we need an involved algorithm that boils down to finding a path in a large directed acyclic graph. In Section 5.5.1, we introduce the necessary notation that is used in the PTBAS. In Section 5.5.2, we give the algorithm that takes the path in the graph, converts it to an optimal solution of SIR and rounds it, to eventually obtain a solution to the Casting Problem. We then prove that this algorithm is a bicriteria approximation algorithm and prove Theorem 5.5.1. In Section 5.5.3, we show how to construct the directed acyclic graph and prove that a path of maximum length corresponds to an optimal solution of SIR.

5.5.1 Rounding and Classification

We start by making the following assumptions. We can assume without loss of generality that $1/\epsilon$ is a positive integer.[2] Throughout this section, we assume without loss of generality that the knapsacks are ordered by non-increasing capacities, i.e., $b_1 \geq b_2 \geq \cdots \geq b_m$, note that this differs from the previous section. Without loss of generality, we assume that $b_m \geq \min_j w_j$ and $b_1 \geq \max_j w_j$. Also

[2]We can do this without loss of generality, since we can replace ϵ with an $\bar{\epsilon} < \epsilon$ such that $1/\bar{\epsilon}$ is integer (for example, choosing $\bar{\epsilon} = 1/\lceil 1/\epsilon \rceil$). The resulting bicriteria approximation for $\bar{\epsilon}$ then implies a bicriteria approximation for ϵ.

without loss of generality, we assume that $b_i \in (0,1]$ for all $i \in M$ and $w_j \in (0,1]$ for all $j \in N$. If this is not the case, this can be easily accomplished by dividing the capacities and weights with the largest knapsack capacity b_1.

We split the interval $(0,1]$ into segments based on ϵ as follows: for $k = 0, 1, \ldots,$ *segment* k is defined as the interval $(\epsilon^{k+1}, \epsilon^k]$. These segments will be used to partition both the items and the knapsacks. After that, we use these partitions to classify the items from a knapsack perspective and to classify the knapsacks from an item perspective.

Item Partitioning and Rounding. First, we partition the items in N according to the segment k in which an item weight lies:

$$N(k) := \left\{ j \in N \;\middle|\; w_j \in (\epsilon^{k+1}, \epsilon^k] \right\}, \quad k = 0, 1, \ldots, \hat{n},$$

where $\hat{n}$ is such that $\epsilon^{\hat{n}+1} < \min_j w_j \leq \epsilon^{\hat{n}}$. See also Figure 5.2 for an illustration. Now, for each $k = 0, \ldots, \hat{n}$ we partition the k-th segment further into $\omega := \frac{1-\epsilon}{\epsilon^2}$ subsegments of size ϵ^{k+2}, see the zoomed-in part of Figure 5.2. Note that the number of subsegments per segment is equal for each k, since both the segment size and the subsegment size decrease in size for larger k. For each item falling into a subsegment, we round the item weight down to the smaller endpoint of the subsegment, i.e., to the nearest multiple of ϵ^{k+2}; more formally:

$$\bar{w}_j = \lfloor w_j / \epsilon^{k+2} \rfloor \epsilon^{k+2}, \quad j \in N(k),\ k = 0, 1, \ldots, \hat{n}. \tag{5.5}$$

To distinguish what weights we use, from now on we will write $c(\boldsymbol{x}, \boldsymbol{w})$ for the original objective function and $c(\boldsymbol{x}, \bar{\boldsymbol{w}})$ for the objective function with rounded weights. We denote the lower end of the j-th subsegment of segment k by $\bar{w}_j^{(k)}$, or formally:

$$\bar{w}_j^{(k)} := \epsilon^{k+1} + (j-1)\epsilon^{k+2}, \quad j = 1, \ldots, \omega.$$

We denote the number of items with weight $\bar{w}_j^{(k)}$ with $y_j^{(k)}$, and use $\boldsymbol{y}^{(k)}$ to denote $(y_1^{(k)}, \ldots, y_\omega^{(k)})$.

[WF11]: scale and position of figure

[WF11]The following property holds by the rounding:

Property 5.5.2. *By rounding, it holds for each $j \in N$ that $w_j \leq (1+\epsilon)\bar{w}_j$.*

Proof:
For every $j \in N(k)$ it holds that w_j is rounded down to the nearest multiple of ϵ^{k+2} and therefore:

$$w_j \leq \bar{w}_j + \epsilon^{k+2} \leq \bar{w}_j + \epsilon \bar{w}_j = (1+\epsilon)\bar{w}_j.$$

□

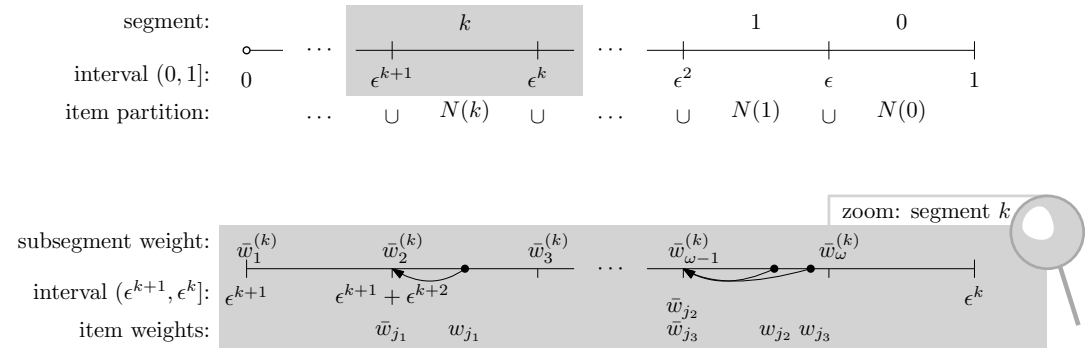


Figure 5.2: The top figure depicts the interval $(0, 1]$, split into segments based on ϵ and the items partitioned accordingly. The bottom figure is a zoomed-in figure of segment k which depicts the (rounded) item weights. An item $j \in N(k)$ is rounded down to the nearest multiple of ϵ^{k+2}. So for $j_1, j_2, j_3 \in N(k)$ it holds that: $\bar{w}_{j_1} = \bar{w}_2^{(k)}$ and $\bar{w}_{j_2} = \bar{w}_{j_3} = \bar{w}_{\omega-1}^{(k)}$.

Knapsack Partitioning. As for the items, we partition the knapsacks based on the segments as follows:

$$M(k) := \left\{ i \in M \;\middle|\; b_i \in (\epsilon^{k+1}, \epsilon^k] \right\}, \quad k = 0, 1, \ldots, \hat{m},$$

where $\hat{m}$ is such that $\epsilon^{\hat{m}+1} < b_m \leq \epsilon^{\hat{m}}$.[3] Note that the assumption $b_m \geq \min_j w_j$ implies that $\hat{m} \leq \hat{n}$.

Items from Knapsack Perspective. Like in the warm-up algorithm, we define for each knapsack i which set of items is large with respect to i, denoted as $N^L(i)$, and a set of items that is small with respect to i, denoted as $N^S(i)$. Moreover, we define a set of items that is medium to i, denoted as $N^M(i)$. See also Figure 5.3 for an illustration. For $i \in M(k)$ it holds that:

$$\begin{aligned} N^L(i) &:= N(k) \\ N^M(i) &:= N(k+1) \\ N^S(i) &:= N(k+2) \cup \cdots \cup N(\hat{n}). \end{aligned} \tag{5.6}$$

For a given segment k, the sets $N^L(i)$ are equal for all $i \in M(k)$. The intuition here is that the classification into large items is not based on the knapsack capacity directly, but solely on the segment in which the knapsack falls. This allows us to

[3]Note that we use a different definition of $\hat{m}$ in this chapter as in Chapter 4.

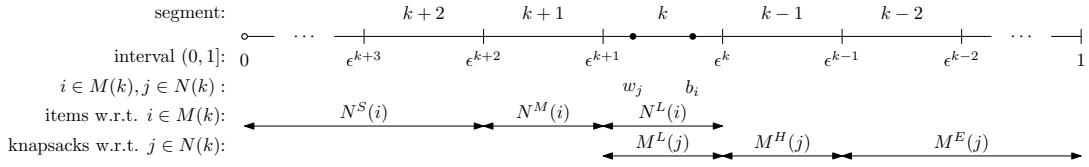


Figure 5.3: Knapsack i is in $M(k)$ since b_i lies in the k-th segment. The items that fit on i are partitioned into $N^S(i)$, $N^M(i)$ and $N^L(i)$. Item j is in $N(k)$ since w_j lies in the k-th segment. The knapsacks on which j fits are partitioned into $M^L(j)$, $M^H(j)$ and $M^E(j)$.[4]

slightly abuse notation and use $N^L(k)$ to refer to $N^L(i)$ for an $i \in M(k)$. The same holds for medium and small items. Items from $N(0)$ to $N(k-1)$ do not fit on a knapsack $i \in M(k)$ and are therefore not classified. Note that $\{N^S(i)\}$ for $i \in M$, is a small item collection, see Definition 5.3.1. This small item collection will be used for SIR considered in this section.

The following property of small items directly follows from the definitions:

Property 5.5.3. *For every knapsack $i \in M$ and every item $j \in N^S(i)$ it holds that $\bar{w}_j \leq w_j \leq \epsilon b_i$.*

Knapsacks from Item Perspective. Lastly, we will classify knapsacks from the item perspective. We denote the set of knapsacks that is *large* with respect to an item j by $M^L(j)$, the set of knapsacks that is *huge* with respect to an item j by $M^H(j)$ and the set of knapsacks that is *enormous* with respect to an item j by $M^E(j)$. See also Figure 5.3 for an illustration. For $j \in N(k)$ it holds that:

$$\begin{aligned} M^L(j) &:= M(k) \\ M^H(j) &:= M(k-1) \\ M^E(j) &:= M(0) \cup \cdots \cup M(k-2). \end{aligned}$$

An item $j \in N(k)$ does not fit on knapsacks from $M(k+1)$ to $M(\hat{m})$, and these knapsacks are therefore not classified with respect to j. As for the items, we sometimes abuse notation slightly by denoting $M^H(k)$ if we mean $M^H(j)$ for a $j \in N(k)$. The same holds for large and enormous knapsacks. Lastly, we want to notice that an item $j \in N(k)$ does not necessarily fit on a knapsack $i \in M(k)$, e.g. if $b_i < w_j < \epsilon^k$.

[WF12] [WF13]

[WF12]: When almost done: make sure that footnote comes on same page as figure.

Algorithm 10 RoundedLongestPath(ϵ)

1: Construct weighted directed acyclic graph based on ϵ ▷ *Explained in next section*
2: Let P be a path in G of maximum length from start node s to end node t
3: Construct optimal solution $\boldsymbol{x}^{\text{MIP}}$ to SIR based on P
4: Apply LP-rounding procedure to $\boldsymbol{x}^{\text{MIP}}$ to obtain $\boldsymbol{x}^{\text{round}}$
5: **return** $\boldsymbol{x}^{\text{round}}$

[WF13]: scale of figure

5.5.2 Small Item Relaxation

The SIR that is used to obtain the PTBAS in this section is based on a different small item collection and different $\hat{b}_i$'s than the SIR in Section 5.4. The small item collection $\{N^S(i)\}$ for $i \in M$ that we use for the PTBAS is defined in (5.6). Furthermore, we round up b_i to obtain the values of $\hat{b}_i$ as follows: for $i \in M(k)$, for $k = 1, \ldots, \hat{m}$:

$$\hat{b}_i := \lceil b_i/\epsilon^{k+4} \rceil \epsilon^{k+4}. \tag{5.7}$$

We consider the instance of SIR with rounded weights $\bar{w}_j$ instead of w_j, as defined in (5.5). To summarize, whenever we refer to SIR in the remainder of this section, we mean SIR with $N^S(i)$ as in (5.6), $\hat{b}_i$ as in (5.7) and weights $w_j = \bar{w}_j$ as in (5.5). [WF14] In the next section, we will describe how to obtain an optimal solution for SIR. As for the warm-up algorithm, a solution for the Casting Problem is found by rounding the solution of SIR. Algorithm 10 summarizes this procedure.

[WF14]: Need to compare macro's that changed, e.g. s-t-path

The next theorem shows that Algorithm 10 is a bicriteria approximation algorithm for the Casting Problem. This algorithm forms the basis for the PTBAS of the Casting Problem.

Theorem 5.5.4. *Algorithm* RoundedLongestPath *is a* $(1/(1+\epsilon), (1+\epsilon)(1+\epsilon+\epsilon^3))$*-bicriteria approximation algorithm to the* Casting Problem.

We first give a corollary and prove two auxiliary lemmas that will turn out to be useful for the proof of Theorem 5.5.4 below. The following corollary shows that LP-rounding does not decrease the objective value and might cause a bounded increase of used capacity in a knapsack. It follows directly from Theorem 5.3.3 and Property 5.5.3.

[4] Actually, since $N^L(i)$ contains all items in $N(k)$, it can be that there is an item $j' \in N^L(i)$ with weight $b_i < w_{j'} < \epsilon^k$ that does not fit on b_i. Moreover, since $M^L(j)$ contains all knapsacks in $M(k)$, it can be that there is a knapsack $i' \in M^L(j)$ with capacity $\epsilon^{k+1} < b_{i'} < w_j$ on which j does not fit.

Corollary 5.5.5. *Let $\boldsymbol{x}^{MIP}$ be a solution to SIR, and let $\boldsymbol{x}^{round}$ be the result of applying the LP-rounding procedure to $\boldsymbol{x}^{MIP}$. Then it holds that $c\left(\boldsymbol{x}^{round}, \bar{\boldsymbol{w}}\right) \geq c\left(\boldsymbol{x}^{MIP}, \bar{\boldsymbol{w}}\right)$ and for every knapsack $i \in M$ it holds that*

$$\sum_{j \in N} \bar{w}_j x_{ij}^{round} \leq \sum_{j \in N} \bar{w}_j x_{ij}^{MIP} + \max_{j \in N^S(i)} \bar{w}_j \leq \sum_{j \in N} \bar{w}_j x_{ij}^{MIP} + \epsilon b_i.$$

The following lemma shows that the violation of Constraint (4.1b) by $\boldsymbol{x}^{\text{MIP}}$ because of the relaxation of the right-hand side of Constraint (5.1c) from b_i to $\hat{b}_i$ is bounded.

Lemma 5.5.6. *Let $\boldsymbol{x}^{MIP}$ be any feasible solution to SIR. Then for every knapsack i it holds that:*

$$\sum_{j \in N} \bar{w}_j x_{ij}^{MIP} \leq (1 + \epsilon^3) b_i.$$

Proof:
Suppose i falls in segment k, then by Constraint (5.1c), we have:

$$\sum_{j \in N} \bar{w}_j x_{ij}^{\text{MIP}} \leq \lceil b_i / \epsilon^{k+4} \rceil \epsilon^{k+4} \leq b_i + \epsilon^{k+4}.$$

Since i falls in segment k, we know that $\epsilon^{k+1} < b_i$ and therefore:

$$b_i + \epsilon^{k+4} \leq b_i + \epsilon^3 b_i = (1 + \epsilon^3) b_i.$$

□

The following lemma shows that objective value of SIR with rounded weights is at least a factor $1/(1+\epsilon)$ times the objective value of the CP with the original weights.

Lemma 5.5.7. *Let $\boldsymbol{x}^*$ and $\boldsymbol{x}^{MIP}$ be the optimal solutions to CP and SIR respectively. Then:*

$$c\left(\boldsymbol{x}^{MIP}, \bar{\boldsymbol{w}}\right) \geq \frac{1}{1+\epsilon} c\left(\boldsymbol{x}^*, \boldsymbol{w}\right).$$

Proof:
It holds that:

$$\begin{aligned} c\left(\boldsymbol{x}^*, \boldsymbol{w}\right) = \sum_{i \in M} \frac{1}{b_i} \sum_{j \in N} w_j x_{ij}^* &\leq (1+\epsilon) \sum_{i \in M} \frac{1}{b_i} \sum_{j \in N} \bar{w}_j x_{ij}^* \\ &\leq (1+\epsilon) \sum_{i \in M} \frac{1}{b_i} \sum_{j \in N} \bar{w}_j x_{ij}^{\text{MIP}} \\ &= (1+\epsilon) c\left(\boldsymbol{x}^{\text{MIP}}, \bar{\boldsymbol{w}}\right). \end{aligned}$$

where the first inequality holds because of Property 5.5.2 and the second inequality holds because $\boldsymbol{x}^{\text{MIP}}$ and $\boldsymbol{x}^*$ are both feasible solutions to SIR but $\boldsymbol{x}^{\text{MIP}}$ is the optimal one. □

Proof of Theorem 5.5.4:
We start by showing that $\boldsymbol{x}^{\text{round}}$ is β-feasible, for $\beta = (1+\epsilon)(1+\epsilon+\epsilon^3)$. $\boldsymbol{x}^{\text{round}}$ satisfies Constraint (4.1c) (every item is packed), by Theorem 5.3.3. Secondly, the LP-rounding ensures that each variable in $\boldsymbol{x}^{\text{round}}$ becomes binary. So, to show that $\boldsymbol{x}^{\text{round}}$ is β-feasible, we only need to check that it satisfies the capacity constraints:

$$\begin{aligned}\sum_{j\in N} w_j x_{ij}^{\text{round}} &\leq (1+\epsilon)\sum_{j\in N} \bar{w}_j x_{ij}^{\text{round}} \\ &\leq (1+\epsilon)\left[\sum_{j\in N} \bar{w}_j x_{ij}^{\text{MIP}} + \epsilon b_i\right] \\ &\leq (1+\epsilon)\left[\left(1+\epsilon^3\right) b_i + \epsilon b_i\right] = \beta \cdot b_i,\end{aligned}$$

where the first inequality holds by Property 5.5.2, the second by Corollary 5.5.5 and the third by Lemma 5.5.6. This proves the required feasibility of the solution. The required objective value follows from Property 5.5.2, Corollary 5.5.5 and Lemma 5.5.7:

$$c\left(\boldsymbol{x}^{\text{round}}, \boldsymbol{w}\right) \geq c\left(\boldsymbol{x}^{\text{round}}, \bar{\boldsymbol{w}}\right) \geq c\left(\boldsymbol{x}^{\text{MIP}}, \bar{\boldsymbol{w}}\right) \geq \frac{1}{1+\epsilon} c\left(\boldsymbol{x}^*, \boldsymbol{w}\right).$$

□

We can extend the result of Theorem 5.5.4 further to obtain a PTBAS and hence prove the main result of this section, Theorem 5.5.1:
Proof of Theorem 5.5.1:
For any α and β such that $0 \leq \alpha \leq 1$ and $\beta \geq 1$, one can always choose an ϵ such that $1/(1+\epsilon) \geq \alpha$ and $(1+\epsilon)\left(1+\epsilon+\epsilon^3\right) \leq \beta$. Then it follows from Theorem 5.5.4 that RoundedLongestPath(ϵ) is an (α, β)-bicriteria approximation algorithm. □

5.5.3 Solution to SIR

We will reduce the problem of computing the solution to SIR to the problem of computing a maximum-length s-t-path in a specifically constructed directed acyclic graph, in which each arc has an associated length.

General Graph Structure. The graph is a directed graph consisting of $\hat{m}+1$ stages. Each stage consists of multiple layers, and each layer consists of multiple

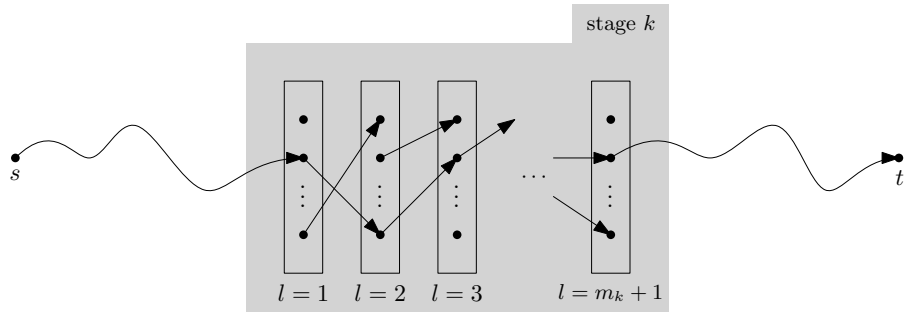


Figure 5.4: General graph structure.

nodes. Stage k corresponds to a segment k. Let m_k be the number of knapsacks in the k-th segment, i.e., $m_k = |M(k)|$. Then for every stage k there will be $m_k + 1$ layers in the graph. Figure 5.4 shows an example of the structure of the graph. The knapsacks within a stage are sorted by non-increasing capacities. Since each stage corresponds to a segment, the stages are automatically sorted by order of non-increasing capacities as well. For a segment k that does not contain any knapsack, i.e., $m_k = 0$, the stage will consist of one so-called *dummy layer*. For the other stages, an arc between layer l and layer $l+1$ in stage k corresponds to packing the l-th knapsack in stage k. The packing of a knapsack i in stage k exactly specifies the configuration of large (in $N^L(k)$) and medium items (in $N^M(k)$) that will be assigned to i. The configuration is a vector of length 2ω that specifies how many items of each subsegment of segment k and $k+1$ are planned on knapsack i. The packing also specifies what remaining part of the capacity will be reserved for small items (in $N^S(k)$). When making such a reservation, it is uncertain which small items will make use of the reserved capacity. Crucially, for a solution to be feasible, the total reserved capacity must correspond with the total weight of all items that are planned as small, which will become clear below.

Consider all the items in $N(k)$. In stage $k-1$, some of these items might be assigned to knapsacks in $M(k-1)$, as medium items. In the next stage, stage k, another subset of $N(k)$ might be assigned to knapsacks in $M(k)$, as large items. The remaining items in $N(k)$ are still unassigned at the end of stage k. We call those items the *remaining unassigned items of stage k*. These are exactly the items that must be assigned to knapsacks in $M^E(k)$, as small items. This means that at the end of stage k, we need to check if enough reservations for small items were made on knapsacks in $M^E(k)$.

To summarize, for each stage $k = 0, 1, \ldots, \hat{m}$:

Step 1. For each knapsack i in stage k (in order of non-increasing capacities):

- Specify the configuration of large items in $N^L(k)$ and medium items in $N^M(k)$ that is assigned to knapsack i.
- Specify how much remaining capacity of knapsack i is reserved for

small items in $N^S(k)$.

Step 2. After executing step 1 for each knapsack $i \in M(k)$, the remaining unassigned items of stage k still need to be assigned. These are exactly the items that are not assigned as medium items to knapsacks in $M^H(k)$ or as large items to knapsacks in $M^L(k)$. They must be assigned as small items to knapsacks in $M^E(k)$. We need to check if enough capacity was reserved on the knapsacks in $M^E(k)$. If yes, then the reservations on the knapsacks in $M^E(k)$ are reduced accordingly by the total weight of the remaining unassigned items of stage k.

State Vectors and Layers. In order to execute these steps, each node in the graph is labeled with a state vector. A state vector contains the necessary information about unassigned items. More specifically: in stage k, the state vector indicates how many of each large and medium item ($N^L(k)$ and $N^M(k)$) still need to be assigned. Additionally, the state vector indicates how much capacity is reserved for small items ($N^S(k)$).

A state vector is of the form $(\boldsymbol{y}^L, \boldsymbol{y}^M, \rho_E, \rho_H, \rho_L, \iota)$. For a node in stage k, the elements in the state vector are as follows:

- $\boldsymbol{y}^L$ is a vector of length ω: $(y_1^L, \ldots, y_\omega^L)$, in which an entry y_j^L is an integer indicating how many items with weight $\bar{w}_j^{(k)}$ from $N^L(k)$ still need to be assigned. Recall that for knapsacks in $M(k)$, large items ($N^L(k)$) are items from the k-th segment and are therefore multiples of ϵ^{k+2}. Recall that ω is the number of subsegments in each segment.

- $\boldsymbol{y}^M$ is a vector of length ω: $(y_1^M, \ldots, y_\omega^M)$, in which an entry y_j^M is an integer indicating how many items with weight $\bar{w}_j^{(k+1)}$ from $N^M(k)$ still need to be assigned. Recall that for knapsacks in $M(k)$, medium items ($N^M(k)$) are items from the $(k+1)$-th segment and are therefore multiples of ϵ^{k+3}.

- ρ_E indicates how much of the reserved capacity on enormous knapsacks, $M^E(k)$, remains for items in $N(k) \cup \cdots \cup N(\hat{n})$. The quantity ρ_E is used, as explained in Step 2 above, to check if the remaining items at the end of stage k fit on enormous knapsacks $M^E(k)$. Formally, ρ_E is defined as the sum of the reservations made on knapsacks in $M(0)$ to $M(k-2)$, subtracted by the weight of all remaining unassigned items of previous stages $0, \ldots, k-1$. Since the items in stage k are of granularity ϵ^{k+2}, it suffices to store ρ_E in granularity ϵ^{k+2}.

- ρ_H indicates the reserved capacity on knapsacks in $M^H(k)$ for items in $N(k+1) \cup \cdots \cup N(\hat{n})$. ρ_H is stored in order to be able to compute ρ_E in later stages. I.e., when going from stage k to stage $k+1$, the huge knapsacks

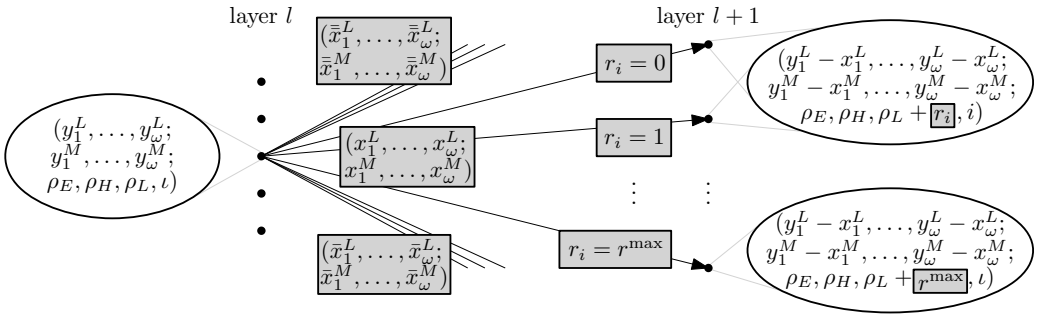


Figure 5.5: Outgoing arcs for a node in layer l of stage k.

become enormous, and ρ_H is added to ρ_E. Since the items in stage $k+1$ are of granularity ϵ^{k+3}, it suffices to store ρ_H in granularity ϵ^{k+3}.

- ρ_L indicates the reserved capacity on knapsacks in $M^L(k)$ for items in $N(k+2) \cup \cdots \cup N(\hat{n})$. Like ρ_H, it is stored in order to be able to compute ρ_E in later stages. I.e., when going from stage k to stage $k+1$, the large knapsacks become huge, and ρ_L is added to ρ_H, which in turn will be added to ρ_E at the end of the next stage, as explained above. During stage k, ρ_L is incremented with the reservations made on the knapsacks in $M(k)$. Since the items in stage $k+2$ are of granularity ϵ^{k+4}, it suffices to store ρ_L in granularity ϵ^{k+4}. Next to the reserved capacity in stage k, ρ_L might contain a *refined reservation*, which we will explain below.

- ι indicates the index of the knapsack with the smallest capacity encountered so far for which the second capacity constraint (5.1c) is not tight. Intuitively, ι is the smallest knapsack encountered that has some residual capacity. It is used for calculating the length of update arcs, which is explained later. ι is initialized to a value 'init', indicating that no knapsack with residual capacity has been encountered yet.

The first 2ω entries in a state vector, in $\boldsymbol{y}^L$ and $\boldsymbol{y}^M$, are non-negative integers of value at most n. The next three values, ρ_E, ρ_L and ρ_H, are non-negative integers of size at most n/ϵ^2; we show in the proof of Lemma 5.5.8 why it suffices to store at most n/ϵ^2 values for each reservation. The last entry in a state vector is either 'init', or a positive integer of size at most m. Each layer in the graph, also the dummy layers, consists of all possible state vectors with a structure as described above.

Arcs Within a Stage. Next, we define what the graph looks like within a stage k, if $m_k > 0$. Recall, if there are m_k knapsacks in segment k, then there will be $m_k + 1$ layers in stage k, from layer 1 to layer $m_k + 1$. If $m_k = 0$, there is only a dummy layer in stage k and there are no arcs within this stage. Otherwise, there can be arcs between different layers in a stage. An arc between the l-th and

$(l+1)$-th layer of stage k indicates the packing of the l-th knapsack in stage k, or equivalently, of the i-th knapsack overall, where $i = \sum_{k'=0}^{k-1} m_{k'} + l$. See Figure 5.5 for an example of the arcs between layer l and layer $l+1$ in stage k. There is an arc between node $(y_1^L, \ldots, y_\omega^L; y_1^M, \ldots, y_\omega^M; \rho_E, \rho_H, \rho_L, \iota)$ in the l-th layer of stage k and node $(\hat{y}_1^L, \ldots, \hat{y}_\omega^L; \hat{y}_1^M, \ldots, \hat{y}_\omega^M; \rho_E, \rho_H, \hat{\rho}_L, \hat{\iota})$ in the $l+1$ layer of stage k if:

- There is a configuration $(x_1^L, \ldots, x_\omega^L; x_1^M, \ldots, x_\omega^M)$ such that for $j = 1, \ldots, \omega$: $x_j^L \geq 0$, $x_j^M \geq 0$, $\hat{y}_j^L = y_j^L - x_j^L$ and $\hat{y}_j^M = y_j^M - x_j^M$. Note that since both $\hat{y}_j^L$ and y_j^L are non-negative integers, it follows that x_j^L is an integer for which holds that $x_j^L \leq y_j^L$. The same holds for the medium items.
- The configuration is feasible with respect to b_i, i.e.,

$$\sum_{j=1}^{\omega} \bar{w}_j^{(k)} x_j^L + \sum_{j=1}^{\omega} \bar{w}_j^{(k+1)} x_j^M \leq b_i.$$

- There is a non-negative integer reservation r_i such that $\hat{\rho}_L = \rho_L + r_i$. The reservation is at most $r^{\max}$, which is defined as:

$$r^{\max} = \lceil b_i/\epsilon^{k+4} \rceil - \left(\sum_{j=1}^{\omega} \bar{w}_j^{(k)} x_j^L + \sum_{j=1}^{\omega} \bar{w}_j^{(k+1)} x_j^M \right) / \epsilon^{k+4}.$$

- The pointer to the last knapsack that has free capacity, $\hat{\iota}$, remains ι if the reservation r_i is equal to $r^{\max}$. Otherwise, $\hat{\iota}$ is updated to i: the index of the knapsack corresponding to the l-th layer of stage k.

The reservation r_i is a reservation of capacity on knapsack i for small items in $N^S(k)$. We make a reservation on knapsack i in granularity ϵ^{k+4} of every multiple from 0 to $r^{\max}$. A reservation of 0 means that the knapsack will only contain large items (in $N^L(k)$) and medium items (in $N^M(k)$) and no small items (in $N^S(k)$). A reservation equal to $r^{\max}$ means that all the remaining capacity (after assigning large and medium items) will contain small items.

The length of such an arc is precisely the weight that is assigned to the arc (coming from both large items in $N^L(k)$, medium items in $N^M(k)$ and the reservation for small items in $N^S(k)$), divided by the knapsack capacity. So again, let i be the index of the l-th knapsack in stage k, then the length of an arc between layer l and layer $l+1$ in stage k is

$$\frac{1}{b_i} \left[\sum_{j=1}^{\omega} \bar{w}_j^{(k)} x_j^L + \sum_{j=1}^{\omega} \bar{w}_j^{(k+1)} x_j^M + r_i \epsilon^{k+4} \right].$$

In the first layer of stage 0, there is only the initial node s, labeled with the state vector corresponding to the number of large and medium items of stage 0.

ρ_L, ρ_E and ρ_H are initialized to zero and ι is set to an initialization value 'init', indicating that there was no knapsack with spare capacity yet.

The intuition of the graph for stage k is as follows: a node v in layer $l+1$ of stage k is reachable from a node v_0 in the first layer of stage k if there is a packing of large items (in $N^L(k)$) and medium items (in $N^M(k)$) on the first l knapsacks in $M(k)$ using feasible configurations, where the remaining unassigned large and medium items are exactly indicated by $\boldsymbol{y}^L$ and $\boldsymbol{y}^M$ in the state vector of node v, and the total reservation for small items (in $N^S(k)$) on the first l knapsacks of stage k is given by the difference in ρ_L for v and v_0, in granularity ϵ^{k+4}. The length of this path is exactly the objective that is gained on the first l knapsacks of stage k, coming from assigned large and medium items and the capacity reserved for small items.

Arcs Between Stages. The state vectors in the last (possibly dummy) layer of a stage k need to be connected to the state vectors in the first (possibly dummy) layer of the next stage $k+1$. We do this with so-called *update arcs*. Intuitively the following happens at such an update arc: (see also Step 2 in the general graph structure)

- The remaining unassigned items of stage k will be assigned to enormous knapsacks ($M^E(k)$). We thus check if ρ_E, i.e., the total amount of reservations on $M^E(k)$, is large enough to accommodate all the items and then subtract the total weight of the remaining unassigned items of stage k from ρ_E. Recall that we store ρ_E in granularity ϵ^{k+2}, so before subtracting the weight of unassigned items, we multiply ρ_E with ϵ^{k+2} to obtain the actual size of the reservation. No update arc will be created if ρ_E is not large enough to accommodate the remaining items, intuitively this means that not enough capacity was reserved.

- We update the reservations for the next stage, $k+1$. This means we need to express ρ_E in granularity ϵ^{k+3}, which we do by dividing it by ϵ^{k+3}. The multiplication of ρ_E with ϵ^{k+2} in the previous step possibly caused ρ_E to be non-integer, but the division with ϵ^{k+3} in this step restores this property again.

  Moreover, we need to add ρ_H to ρ_E since the knapsacks that are huge in stage k will become enormous in stage $k+1$. Similarly, ρ_H is updated to the value of ρ_L.

- If at least one knapsack has been encountered with some residual capacity, i.e., if ι is not 'init' anymore, we are going to introduce the possibility of making a *refined reservation*, denoted with u. We need these refined reservations because the reservations that are made on the arcs within a stage k are made in granularity ϵ^{k+4}. However, we want to accommodate

the possibility of making a more fine-grained reservation, in granularity ϵ^{k+5}. We make a refined reservation for every size u, such that u is a multiple of ϵ^{k+5}, and smaller than ϵ^{k+4}. Since the granularity of the refined reservation is ϵ^{k+5}, this means we make a refined reservation for every positive integer u that is smaller than $1/\epsilon$.

Since the granularity of ρ_L in the next stage $k+1$ is also ϵ^{k+5}, we can add the refined reservation to ρ_L. This means that in stage $k+1$, ρ_L does not only contain the reservations made on knapsacks in stage $k+1$, but also a refined reservation on ι.

In Property 5.5.12 we prove that for the optimal solution $\boldsymbol{x}^{\text{MIP}}$ there is at most one knapsack in $M(0) \cup \cdots \cup M(k)$ that has such a fine-grained weight, namely ι. Therefore it suffices to create the possibility to make a refined reservation in each granularity once.

Algorithm 11 defines which update arcs will be created. The input to the algorithm consists of the stage k and a state vector in the last layer of stage k. Whenever the last stage is reached, i.e., $k = \hat{m}$, we need to check if there is an arc to the target node t. For the nodes in the last layer of stage k, we therefore do not use Algorithm 11 to compute the outgoing arcs, but Algorithm 12, which is elaborated on below. An example for the update arcs between the last layer in stage k and the first layer in stage $k+1$ is given in Figure 5.6.

We elaborate on Algorithm 11 in more detail. The algorithm starts with decreasing the reservation for small items, ρ_E, with the total weight of all remaining items in $N(k)$ in Line 1, after which the algorithm checks if enough reservations were made to accommodate these items in Line 2, otherwise it returns 'failure'. In Lines 3 and 4, the algorithm initializes ρ_E and ρ_H for the next stage. The output of Algorithm 11 is a set of nodes in the first layer of stage $k+1$. An update arc will be created from the input node to all of these nodes in the output set. Recall that we use $y_j^{(k)}$ to denote the number of items with weight $\bar{w}_j^{(k)}$. If a knapsack with free capacity exists, an arc is created with every possible refined reservation value. The value of the refined reservation is a non-negative integer strictly smaller than $1/\epsilon$. Otherwise, no refined reservation is made and only one arc will be created.

The length of an update arc originating in the last layer of k to the first layer of stage $k+1$ is the value of the refined reservation divided by the capacity of ι, defined as follows:

$$\frac{1}{b_\iota} u \epsilon^{k+5}.$$

It follows that the length of an update arc on which no refined reservation was made because ι was equal to 'init' is 0.

Arcs to the Target Node. As we explained above, for the nodes in the last layer of the last stage, i.e., when $k = \hat{m}$, we need an algorithm to check if there is

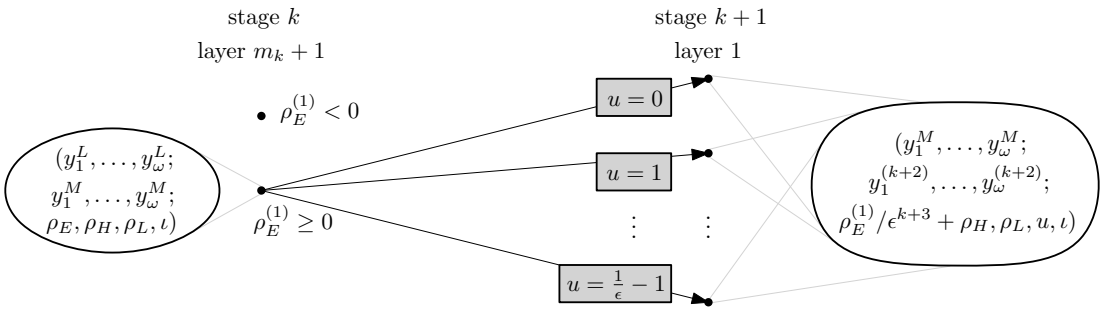


Figure 5.6: Outgoing arcs for two nodes in the last layer of stage k to the first layer of stage $k+1$. The upper node has no outgoing arcs since $\rho_E^{(1)} < 0$: the made reservations are not large enough to accommodate the remaining items in stage k.

Algorithm 11 UPDATE$\left(k, \left(y_1^L, \ldots, y_\omega^L; y_1^M, \ldots, y_\omega^M; \rho_E, \rho_H, \rho_L, \iota\right)\right)$

1: $\rho_E^{(1)} = \rho_E \epsilon^{k+2} - \sum_{j=1}^{\omega} \bar{w}_j^{(k)} y_j^L$ ▷ *Remaining items of $N(k)$ are assigned as small*
2: **if** $\rho_E^{(1)} < 0$ **then return** failure ▷ *Check if enough reservations are made*
3: $\rho_E = \rho_E^{(1)}/\epsilon^{k+3} + \rho_H$ ▷ *Update values for next stage*
4: $\rho_H = \rho_L$
5: **if** $\iota \neq$'init' **then**
6: **return** $\left\{ \left(y_1^M, ..., y_\omega^M; y_1^{(k+2)}, ..., y_\omega^{(k+2)}; \rho_E, \rho_H, u, \iota \right) \middle| u = 0, 1, ..., 1/\epsilon - 1 \right\}$
7: **else**
8: **return** $\left\{ \left(y_1^M, ..., y_\omega^M; y_1^{(k+2)}, ..., y_\omega^{(k+2)}; \rho_E, \rho_H, 0, \iota \right) \right\}$

an arc to the target node t. Algorithm 12 does exactly this; the input is a node in the last layer of stage $\hat{m}$ and the output is either t or 'failure'. Intuitively, the algorithm checks if the made reservations are enough to accommodate the remaining items in stage $\hat{m}$ and all subsequent stages (recall that possibly, $\hat{m} \leq \hat{n}$). Conversely, the algorithm also checks if not too much capacity was reserved, i.e., it checks if all reservations are justified. If this is the case, the algorithm returns the target node t and an arc to t is created, otherwise, the algorithm returns 'failure', and no outgoing arc is created.

We elaborate on Algorithm 12 in more detail. At the end of stage $\hat{m}$, it can be that there are items in $N(\hat{m})$ that were not assigned to knapsacks in $M^L(\hat{m})$ or $M^H(\hat{m})$ and therefore must be assigned as small items to knapsacks in $M^E(\hat{m})$, these are exactly the remaining unassigned items of stage $\hat{m}$. Consequently, in the first line of the algorithm, we assign these remaining items of $N(\hat{m})$ to $M^E(\hat{m})$, and check if enough capacity was reserved to assign them in Line 2. We continue with the remaining unassigned items in stage $\hat{m}+1$. Some items in $N(\hat{m}+1)$ might have been assigned as medium items to knapsacks in $M^H(\hat{m}+1) =$

Algorithm 12 RESERVATIONVALIDATION$\left(y_1^L, \ldots, y_\omega^L; y_1^M, \ldots, y_\omega^M; \rho_E, \rho_H, \rho_L, \iota\right)$

1: $\rho_E^{(1)} = \rho_E \epsilon^{\hat{m}+2} - \sum_{j=1}^{\omega} \bar{w}_j^{(\hat{m})} y_j^L$ ▷ *Remaining items of $N(\hat{m})$ are assigned as small*
2: **if** $\rho_E^{(1)} < 0$ **then return** failure ▷ *Check if enough reservations are made*
3: $\rho_E^{(2)} = \rho_E^{(1)} + \rho_H \epsilon^{\hat{m}+3} - \sum_{j=1}^{\omega} \bar{w}_j^{(\hat{m}+1)} y_j^M$ ▷ *Rem. items of $N(\hat{m}+1)$ assigned as small*
4: **if** $\rho_E^{(2)} < 0$ **then return** failure ▷ *Check if enough reservations are made*
5: $\rho_E^{(3)} = \rho_E^{(2)} + \rho_L \epsilon^{\hat{m}+4}$ ▷ *All remaining unused reservations*
6: $\theta = \sum_{q=\hat{m}+2}^{\hat{n}} \sum_{j=1}^{\omega} \bar{w}_j^{(q)} y_j^{(q)}$ ▷ *Total weight of unassigned items*
7: **if** $\rho_E^{(3)} \leq \theta < \rho_E^{(3)} + \epsilon^{\hat{m}+4}$ **and** $\iota \neq$'init' **then**
8: | **return** t
9: **else if** $\rho_E^{(3)} = \theta$ **then**
10: | **return** t
11: **else**
12: | **return** failure

$M(\hat{m})$, but the remaining remaining items of $N(\hat{m}+1)$ need to be assigned to knapsacks in $M^E(\hat{m}+1)$. In Line 4, the algorithm checks if enough capacity was reserved to do so. The algorithm then computes $\rho_E^{(3)}$, which is the remaining reserved capacity in all knapsacks. Moreover, it computes θ: the total weight of all remaining unassigned items, which is equal to the total weight from all items in $N(\hat{m}+2) \cup \cdots \cup N(\hat{n})$ since none of the items in these segments were assigned yet. Since the reservations were made as a multiple of $\epsilon^{\hat{m}+4}$ at the finest, the algorithm allows that the remaining weight exceeds the remaining reserved capacity with strictly less than $\epsilon^{\hat{m}+4}$. This difference is the last refined reservation that must be made on ι. If ι is 'init', no refined reservation can be made and the remaining reserved capacity must be equal to the remaining weight.

The length of an update arc from the last layer to t is equal to the difference between the remaining weight and the remaining reserved capacity, divided by the capacity of ι (if ι is 'init', the length is 0):

$$\frac{1}{b_\iota}\left(\theta - \rho_E^{(3)}\right).$$

Proofs. In the remainder of this section, we prove three lemmas that together show that an optimal solution to SIR can be found by finding a maximum-length path in the created graph. First, we will prove that the size of the graph is polynomial in Lemma 5.5.8. After that, Lemma 5.5.9 and Lemma 5.5.10 show the correspondence between a maximum-length path in the graph and the optimal solution to SIR.

Lemma 5.5.8. *The number of nodes in the graph is polynomial.*

Proof:
There are exactly $\hat{m}+1$ stages in the graph. Since $b_m \leq \epsilon^{\hat{m}}$, it follows that

$$\hat{m} \leq \frac{\log b_m}{\log \epsilon}.$$

Each stage has at least one layer and there is an extra layer in a stage k for each knapsack in $M(k)$. This means there are exactly $\hat{m}+1+m$ layers in the graph. A layer may contain all possible state vectors. The first two entries of the state vectors represent the number of large and medium items that need to be assigned. They are vectors of length $\omega = (1-\epsilon)/\epsilon^2$, where each entry is at most n. Since $2\omega \leq 2/\epsilon^2$, this gives at most n^{2/ϵ^2} possible vectors. Next, we derive an upper bound on the weight that ρ_E must be able to represent. During a stage k, ρ_E needs to represent the weight of at most n items. A small item to an enormous knapsack comes from stage k (or higher), so is at most of size ϵ^k. Therefore, $n\epsilon^k$ is an upper bound on the weight that ρ_E might represent. Since ρ_E is represented in granularity ϵ^{k+2}, there only need to be n/ϵ^2 different values for ρ_E in the state vector. Similarly, we only need n/ϵ^2 different values for ρ_H and ρ_L. Together with m values for ι, this gives that the number of nodes in the graph is

$$\mathcal{O}\left(\frac{1}{\epsilon^6}\left(\frac{\log b_m}{\log \epsilon}+m\right)n^{2/\epsilon^2+3}\right).$$

□

Lemma 5.5.9. *If there is a path in the graph from s to t with a total length of c, then there is a feasible solution to SIR with value c.*

Proof:
We can construct a feasible solution to SIR denoted by $\boldsymbol{x}$, by tracing the given path in the graph. For each stage k and each knapsack $i \in M(k)$, there is an arc along the path that specifies: (1) the feasible configuration, i.e., how many large items of each size $\bar{w}_j^{(k)}, j = 1, \ldots, \omega$ and how many medium items of each size $\bar{w}_j^{(k+1)}, j = 1, \ldots, \omega$ must be assigned to knapsack i and (2) r_i: the remaining capacity in i that will contain small items in granularity ϵ^{k+4}. Since the arc indicates r_i in granularity ϵ^{k+4}, we multiply them to ensure we have the actual reservation: $\bar{r}_i = r_i\epsilon^{k+4}$. Since the configuration only specifies the number of large (medium) items of each size according to the rounded weight, $\bar{w}$, it remains to choose which exact items we assign to i. Any subset of unassigned items with rounded weights specified by the configuration suffices and we can set $x_{ij} = 1$ for the items j in this subset.

After traversing the path through the layers of stage k, and assigning large and medium items to knapsacks accordingly, we end up at an update arc at the

end of stage k. We still need to assign the remaining unassigned items in stage k. While there remains an item $j \in N(k)$ for which $\sum_{i \in M} x_{ij} < 1$, assign it, possibly fractionally, on the largest encountered knapsack i' for which the reservation $\bar{r}_{i'}$ is positive: let $i' = \arg\max \{b_i \mid i \in M(0) \cup \cdots M(k),\ \bar{r}_i > 0\}$, set

$$x_{i'j} = \min \left\{ \frac{\bar{r}_{i'}}{\bar{w}_j}, 1 - \sum_{i \in M} x_{ij} \right\},$$

and decrease $\bar{r}_{i'}$ with $\bar{w}_j x_{i'j}$. We know that we can always find such an i' since Algorithm 11 only outputs outgoing arcs if sufficient reservations are made for these remaining items. The update arc also specifies the so-called refined reservation, u on ι in granularity ϵ^{k+5}. We increase the reservation $\bar{r}_\iota$ of knapsack ι with this amount: $\bar{r}_\iota = \bar{r}_\iota + u\epsilon^{k+5}$. We continue traversing the path and constructing $\boldsymbol{x}$ until a node in the last layer of stage $\hat{m}$ is reached.

Similarly as to an update arc, we assign the remaining unassigned items of stage $\hat{m}$, possibly fractionally, to knapsacks in $M^E(\hat{m})$, and subsequently we plan the remaining unassigned items of stage $\hat{m}+1$, possibly fractionally, to knapsacks in $M^H(\hat{m})$. Before we plan the remaining items of stage $\hat{m}+2$, we increase $\bar{r}_\iota$ with the value of the last refined reservation, i.e., $\bar{r}_\iota = \bar{r}_\iota + \theta - \rho_E^{(3)}$, in the case that ι does not equal 'init'.

By construction, $\boldsymbol{x}$ satisfies integrality constraints (5.1e) and non-negativity constraints (5.1f). The first capacity constraint (5.1b) holds by the feasibility of the large and medium configuration, checked during construction of the graph. Constraint (5.1d) holds since the construction of $\boldsymbol{x}$ ensures that every item is assigned to a knapsack. In case the item is assigned fractionally, the construction ensures that the values for the item add up to one. In order to show that $\boldsymbol{x}$ is a feasible solution to SIR, we only need to show that $\boldsymbol{x}$ satisfies the second capacity constraint (5.1c). Observe that (5.1c) holds by construction of the graph if the refined reservations are not taken into account. We need to prove that the constraint is still satisfied after adding the value of refined reservations. When an arc is encountered that specifies the large and medium items that must be assigned to a knapsack i, we either have that Constraint (5.1c) is tight, which means that $r_i = r^{\max} = 0$ and by construction of the graph, ι is not updated in that case. Subsequently, no refined reservation will be made on this knapsack and it holds that Constraint (5.1c) remains valid. Otherwise, it holds that (5.1c) is not tight, and the arc leads to a node where ι is updated to i. We prove that Constraint (5.1c) remains valid for ι after a refined reservation, by proving the following invariant: at each point of the path in stage k, the slack for Constraint (5.1c) of knapsack ι is at least ϵ^{k+4}. After updating ι to i in stage k, the invariant holds by construction of the graph, since ι is only updated if there is slack and the slack cannot be smaller than ϵ^{k+4}. We prove that the invariant remains true after encountering an update arc between stage k and stage $k+1$. The refined reservation is maximum whenever $u = 1/\epsilon - 1$. Since the refined reservation is

stored in granularity ϵ^{k+5}, we multiply them to get:

$$u = (1/\epsilon - 1)\epsilon^{k+5} = \epsilon^{k+4} - \epsilon^{k+5}.$$

This means that the refined reservation fits on ι, since there is ϵ^{k+4} slack in the constraint. For the next stage, $k+1$, the slack in the constraint for ι is at least $\epsilon^{k+4} - (\epsilon^{k+4} - \epsilon^{k+5}) = \epsilon^{k+5}$. This proves the invariant and therefore proves that Constraint (5.1c) also holds for ι.

Lastly, it holds that the path's length is equal to the value of $c(\boldsymbol{x})$ since each arc has a length that corresponds exactly with the total assigned weight induced by following that arc, divided by the size of the associated knapsack. To see this, observe that only an arc to t is created if all reservations are justified. □

Lemma 5.5.10. *If SIR has a feasible solution and $\boldsymbol{x}^{MIP}$ is the optimal solution with value $c\left(\boldsymbol{x}^{MIP}, \bar{\boldsymbol{w}}\right)$, then there is a path in the graph from s to t with a total length of $c\left(\boldsymbol{x}^{MIP}, \bar{\boldsymbol{w}}\right)$.*

Before we prove the lemma, we introduce some extra notation and prove two properties that hold for the solution $\boldsymbol{x}^{\text{MIP}}$. Let $\boldsymbol{z}_{kl}$ be the vector of length ω indicating the number of items from each subsegment of stage l that $\boldsymbol{x}^{\text{MIP}}$ has assigned to knapsacks in $M(k)$, or formally,

[WF15]: Think we need to choose different notation here

$$\boldsymbol{z}_{kl} = (z_{1kl}, z_{2kl}, \ldots, z_{\omega kl}), \quad \text{where } z_{jkl} = \sum_{i \in M(k)} \sum_{j' : \bar{w}_{j'} = \bar{w}_j^{(l)}} x_{ij'}^{\text{MIP}}.$$

Furthermore, let $w(k, l, \boldsymbol{x}^{\text{MIP}})$ denote the total weight of items in $N(l)$ that are assigned to knapsacks in $M(k)$ with respect to $\boldsymbol{x}^{\text{MIP}}$, i.e.:

$$w(k, l, \boldsymbol{x}^{\text{MIP}}) = \sum_{i \in M(k)} \sum_{j \in N(l)} \bar{w}_j x_{ij}^{\text{MIP}}.$$

For simplicity of notation, we will often write w_{kl}, and sometimes $w_{k,l}$ to distinguish what belongs to the first and the second subscript, when we mean $w(k, l, \boldsymbol{x}^{\text{MIP}})$.

In the construction of the s-t-path, we will need to choose one of the update arcs, based on the refined reservation. For the update arc between stage $k-1$ and stage k, we choose the following quantity for the refined reservations:

$$\hat{u}_k := \left\lfloor \left(\sum_{k'=0}^{k-1} \sum_{l' \geq 0} w_{k'l'} \right) / \epsilon^{k+4} \right\rfloor - \left\lfloor \left(\sum_{k'=0}^{k-1} \sum_{l' \geq 0} w_{k'l'} \right) / \epsilon^{k+3} \right\rfloor \frac{1}{\epsilon}.$$

Intuitively, $\hat{u}_k \epsilon^{k+4}$ is the difference between rounding down the total weight on knapsacks of the first $k-1$ stages to the nearest multiple of ϵ^{k+4} and ϵ^{k+3}. For every k it holds that $0 \leq \hat{u}_k < 1/\epsilon$ and $\hat{u}_k \in \mathbb{N}_0$. Moreover, the following property holds for $\hat{u}_k$:

Property 5.5.11.

$$\hat{u}_k = \left\lfloor \left(\sum_{k'=0}^{k-1} \sum_{l' \geq k+2} w_{k'l'} \right) / \epsilon^{k+4} \right\rfloor - \left\lfloor \left(\sum_{k'=0}^{k-1} \sum_{l' \geq k+2} w_{k'l'} \right) / \epsilon^{k+3} \right\rfloor \frac{1}{\epsilon}.$$

Proof:
The property holds since both w_{kl}/ϵ and w_{kl} are multiples of ϵ^{k+3} for every l : $0 \leq l \leq k+1$. □

To deal with knapsacks that have the same size, we make the following tie-breaking assumption for $\boldsymbol{x}^{\text{MIP}}$ without loss of generality: if $\exists i, i' \in M$ such that $i \leq i'$ and $b_i = b_{i'}$, and if Constraint (5.1c) is not tight for i', then $x_{ij}^{\text{MIP}} = 0$ for every $j \in N^S(i)$. Intuitively, this means that $\boldsymbol{x}^{\text{MIP}}$ is the solution such that if there are multiple knapsacks with equal size, small items are always assigned to the knapsack with the larger index first. This assumption about $\boldsymbol{x}^{\text{MIP}}$ is used to prove the following property for $\boldsymbol{x}^{\text{MIP}}$.

Property 5.5.12. *For every k, there is at most one knapsack in $M(0) \cup \cdots \cup M(k)$ in the optimal solution to SIR, $\boldsymbol{x}^{MIP}$, that has a total weight assigned to it with granularity ϵ^{k+5} or smaller. This knapsack is exactly the smallest knapsack, i.e., the knapsack with the largest index, in stages 0 to k for which* (5.1c) *is not tight. The rest of the knapsacks in $M(0) \cup \cdots \cup M(k)$ have a weight that is a multiple of ϵ^{k+4}.*

Proof:
Fix a k. Suppose that in $\boldsymbol{x}^{\text{MIP}}$, there are two knapsacks in $M(0) \cup \cdots \cup M(k)$ that have a total weight assigned to them which has granularity ϵ^{k+5} or smaller. That must mean that both knapsacks have items in $N(k+3) \cup \cdots \cup N(\hat{n})$ assigned to them, i.e., both knapsacks have items assigned to them that are small for them. Moreover, it must hold that Constraint (5.1c) is not tight for both of these knapsacks. By our tie-breaking assumption, the knapsacks cannot be of the same size. However, this means $\boldsymbol{x}^{\text{MIP}}$ can be improved by moving weight (coming from small items) from the larger to the smaller knapsack. This can be done until at most one knapsack has granularity ϵ^{k+5}. By moving weight from a larger to a smaller knapsack, the objective value of $\boldsymbol{x}^{\text{MIP}}$ improves, which is a contradiction. This proves that there is at most one knapsack in stages 0 to k with granularity ϵ^{k+5} or smaller.

Now suppose that in $\boldsymbol{x}^{\text{MIP}}$ there is a knapsack with granularity ϵ^{k+5} and there is another knapsack, with a larger index, for which Constraint (5.1c) in SIR is not tight. By the tie-breaking assumptions, these knapsacks cannot be of the same size. Then again, by moving the weight coming from small items to the smaller knapsack, we can improve the solution. This gives a contradiction that proves the latter part of the property. □

Proof of Lemma 5.5.10:
We assume that SIR has a feasible solution, and therefore an optimal solution, which we denote by $\boldsymbol{x}^{\text{MIP}}$. We first prove by induction that a path from s to t exists, without regarding the length of the path. Recall that we use $\boldsymbol{y}^{(k)}$ to denote $(y_1^{(k)}, \ldots, y_\omega^{(k)})$, where $y_j^{(k)}$ indicates the total number of items with rounded weight $\bar{w}_j^{(k)}$.

Inductive Hypothesis 1. *For $k = 0, \ldots, \hat{m}$, there is a path P from s to some node in the first layer of stage k, where the state vector of that node has*

- *item distribution* $\left(\boldsymbol{y}^{(k)} - \boldsymbol{z}_{k-1,k}; \boldsymbol{y}^{(k+1)}\right)$;
- $\rho_E = \left\lfloor \left(\sum_{k'=0}^{k-2} \sum_{l' \geq k} w_{k'l'} \right) / \epsilon^{k+2} \right\rfloor$;
- $\rho_H = \hat{u}_{k-1} + \left\lfloor \left(\sum_{l' \geq k+1} w_{k-1,l'} \right) / \epsilon^{k+3} \right\rfloor$;
- $\rho_L = \hat{u}_k$;
- ι *is the smallest knapsack of segment 0 to $k-1$ for which* (5.1c) *in SIR is not tight.*

For $k = 0$, $\rho_E = \rho_H = \rho_L = 0$ and ι is 'init', and there is a trivial path from s to itself. Inductively, we can now assume that at the beginning of stage k (so that $M(k) \neq \emptyset$) there is a path from s to a state vector in the first layer of k. We need to prove that there is a path to a node in the first layer of stage $k+1$.

To construct the path in stage k, we follow the optimal solution $\boldsymbol{x}^{\text{MIP}}$ to decide which large and medium items to pack on the knapsacks, and how much weight we must reserve for small items. We can do this easily, since for every feasible configuration of large and medium items and every possible reservation of small items that is a multiple of ϵ^{k+4}, there is an arc. For each knapsack i, we take the arc that has exactly the large and medium configuration as in $\boldsymbol{x}^{\text{MIP}}$, and which reserves r_i, where r_i is as much as the weight from small items that $\boldsymbol{x}^{\text{MIP}}$assigns to i, but rounded down:

$$r_i = \left\lfloor \frac{\sum_{j \in N^S(i)} \bar{w}_j x_{ij}^{\text{MIP}}}{\epsilon^{k+4}} \right\rfloor.$$

As explained in the construction of the graph, this arc increases ρ_L with the amount that we reserve, i.e., $\rho_L = \rho_L + r_i$.

After following these arcs for every knapsack in stage k, we end up in the last layer of stage k with item distribution $\left(\boldsymbol{y}^{(k)} - \boldsymbol{z}_{k-1,k} - \boldsymbol{z}_{k,k}; \boldsymbol{y}^{(k+1)} - \boldsymbol{z}_{k,k+1}\right)$. ρ_E and ρ_H have not changed since the beginning of the stage. ρ_L was updated every

time a reservation was made and is therefore equal to:

$$\begin{aligned}\rho_L &= \hat{u}_k + \sum_{i \in M(k)} \left\lfloor \frac{\sum_{j \in N^S(i)} \bar{w}_j x_{ij}^{\text{MIP}}}{\epsilon^{k+4}} \right\rfloor \\ &= \hat{u}_k + \left\lfloor \frac{\sum_{i \in M(k)} \sum_{j \in N^S(i)} \bar{w}_j x_{ij}^{\text{MIP}}}{\epsilon^{k+4}} \right\rfloor \\ &= \hat{u}_k + \left\lfloor \frac{\sum_{l' \geq k+2} w_{kl'}}{\epsilon^{k+4}} \right\rfloor. \end{aligned} \tag{5.8}$$

The second equality holds by Property 5.5.12 and the third equality by the definition of w_{kl}. To prove Inductive Hyothesis (IH) 1, we need to show there is an update arc to a node in the first layer of the next stage for which IH 1 holds.

First, we check if the UPDATE function, Algorithm 11, does not return failure, by checking if $\rho_E^{(1)} \geq 0$. The input of the algorithm, denoted with $(k, (y_1^L, \ldots, y_\omega^L; y_1^M, \ldots, y_\omega^M; \rho_E, \rho_H, \rho_L, \iota))$, is given by $(k, (\boldsymbol{y}^{(k)} - \boldsymbol{z}_{k-1,k} - \boldsymbol{z}_{k,k}; \boldsymbol{y}^{(k+1)} - \boldsymbol{z}_{k,k+1}; \rho_E, \rho_H, \rho_L, \iota)))$, which gives:

$$\begin{aligned}\rho_E^{(1)} &= \rho_E \epsilon^{k+2} - \sum_{j=1}^{\omega} \bar{w}_j^{(k)} y_j^L \\ &= \left\lfloor \left(\sum_{k'=0}^{k-2} \sum_{l' \geq k} w_{k'l'} \right) / \epsilon^{k+2} \right\rfloor \epsilon^{k+2} - \sum_{k'=0}^{k-2} w_{k'k} \\ &= \left\lfloor \left(\sum_{k'=0}^{k-2} \sum_{l' \geq k} w_{k'l'} - \sum_{k'=0}^{k-2} w_{k'k} \right) / \epsilon^{k+2} \right\rfloor \epsilon^{k+2} \\ &= \left\lfloor \left(\sum_{k'=0}^{k-2} \sum_{l' \geq k+1} w_{k'l'} \right) / \epsilon^{k+2} \right\rfloor \epsilon^{k+2} \geq 0 \end{aligned}$$

Here we use IH 1 for ρ_E and that $\sum_{j=1}^{\omega} \bar{w}_j^{(k)} y_j^L$ is all the weight from items in stage k that were not assigned as large or medium in $\boldsymbol{x}^{\text{MIP}}$, so therefore it is equal to $\sum_{k'=0}^{k-2} w_{k'k}$. We use that $w_{k'k}$ is a multiple of ϵ^{k+2} for every k' in the third equality. It follows that $\rho_E^{(1)} \geq 0$, so we know that the UPDATE function does not return 'failure'. The UPDATE function might return many nodes in the first layer of stage $k+1$, corresponding to different values for the refined reservations u, from which we can choose a specific one. We choose the update arc which makes a refined reservation u equal to $\hat{u}_{k+1}$. Note that this arc exists because $0 \leq \hat{u}_{k+1} < 1/\epsilon$.

We check if IH 1 holds for the node that this update arc points to. The condition for the item distribution and ι are satisfied by construction of the graph, in particular the output of the UPDATE algorithm. We only need to check if IH 1 holds for the new ρ_L, ρ_H and ρ_E that are computed in the UPDATE function,

we denote them with ρ'_L, ρ'_H and ρ'_E to distinguish them from the current values. ρ'_L is updated to u, for which we have chosen the value $\hat{u}_{k+1}$. It follows that the condition for ρ_L in IH 1 holds. ρ'_H is updated to ρ_L. Recall that in the last layer of stage k, ρ_L is equal to

$$\hat{u}_k + \left\lfloor \frac{\sum_{l' \geq k+2} w_{kl'}}{\epsilon^{k+4}} \right\rfloor,$$

so the condition for ρ'_H in IH 1 holds as well. To compute ρ'_E, we can use IH 1 for ρ_H and the computation of $\rho_E^{(1)}$ above:

$$\begin{aligned}
\rho'_E &= \rho_E^{(1)}/\epsilon^{k+3} + \rho_H \\
&= \left\lfloor \left(\sum_{k'=0}^{k-2} \sum_{l' \geq k+1} w_{k'l'} \right) /\epsilon^{k+2} \right\rfloor \epsilon^{k+2}/\epsilon^{k+3} + \hat{u}_{k-1} + \left\lfloor \left(\sum_{l' \geq k+1} w_{k-1,l'} \right) /\epsilon^{k+3} \right\rfloor \\
&= \left\lfloor \left(\sum_{k'=0}^{k-2} \sum_{l' \geq k+1} w_{k'l'} \right) /\epsilon^{k+3} \right\rfloor + \left\lfloor \left(\sum_{l' \geq k+1} w_{k-1,l'} \right) /\epsilon^{k+3} \right\rfloor \\
&= \left\lfloor \left(\sum_{k'=0}^{k-1} \sum_{l' \geq k+1} w_{k'l'} \right) /\epsilon^{k+3} \right\rfloor \qquad (5.9)
\end{aligned}$$

which is exactly what ρ_E should be according to IH 1 for stage $k+1$. We used Property 5.5.11 for the third equality and Property 5.5.12 for the last equality. This proves that there is an update arc that leads to a state vector in the first layer of stage $k+1$ for which IH 1 holds. This proves IH 1.

It remains to show that there is a path from the first layer of stage $\hat{m}$ to t. We can follow the arcs in stage $\hat{m}$ as described above for all the previous stages. Similar arguments as described above can be used to prove that ρ_L gets updated as in (5.8), that $\rho_E \geq 0$ and that the input for Algorithm 12, denoted with $(y_1^L, \ldots, y_\omega^L; y_1^M, \ldots, y_\omega^M; \rho_E, \rho_H, \rho_L, \iota)$, is given by $(\boldsymbol{y}^{(\hat{m})} - \boldsymbol{z}_{\hat{m}-1,\hat{m}} - \boldsymbol{z}_{\hat{m},\hat{m}}; \boldsymbol{y}^{(\hat{m}+1)} - \boldsymbol{z}_{\hat{m},\hat{m}+1}; \rho_E, \rho_H, \rho_L, \iota)$. Then, in Algorithm 12 it holds that $\rho_E^{(1)}/\epsilon^{\hat{m}+3} + \rho_H = (5.9)$, with $k = \hat{m}$. It follows that Algorithm 12 does not return 'failure' in Line 2. We will show that Algorithm 12 also does not return 'failure' in Line 4 by showing that $\rho_E^{(2)} \geq 0$.

$$\begin{aligned}
\rho_E^{(2)} &= \rho_E^{(1)} + \rho_H \epsilon^{\hat{m}+3} - \sum_{j=1}^{\omega} \bar{w}_j^{(\hat{m}+1)} y_j^M \\
&= \left\lfloor \left(\sum_{k'=0}^{\hat{m}-1} \sum_{l' \geq \hat{m}+1} w_{k'l'} \right) /\epsilon^{\hat{m}+3} \right\rfloor \epsilon^{\hat{m}+3} - \sum_{k'=0}^{\hat{m}-1} w_{k'\hat{m}+1} \\
&= \left\lfloor \left(\sum_{k'=0}^{\hat{m}-1} \sum_{l' \geq \hat{m}+2} w_{k'l'} \right) /\epsilon^{\hat{m}+3} \right\rfloor \epsilon^{\hat{m}+3} \geq 0
\end{aligned}$$

Here we use that $\rho_E^{(1)}/\epsilon^{\hat{m}+3} + \rho_H = (5.9)$ and that $\sum_{j=1}^{\omega} \bar{w}_j^{(\hat{m}+1)} y_j^M$ is all the weight from items in $N(\hat{m}+1)$ that are not assigned as medium to $M^H(\hat{m}+1)(= M(\hat{m}))$, so therefore it is equal to $\sum_{k'=0}^{\hat{m}-1} w_{k'\hat{m}+1}$. The last equality holds since $\sum_{k'=0}^{\hat{m}-1} w_{k'\hat{m}+1}$ is a multiple of $\epsilon^{\hat{m}+3}$. Since $\rho_E^{(2)} \geq 0$, Algorithm 12 does not return 'failure' in Line 4.

To compute the value of $\rho_E^{(3)}$, we use the computation of $\rho_E^{(2)}$ and the updated value of ρ_L:

$$\begin{aligned}
\rho_E^{(3)} &= \rho_E^{(2)} + \rho_L \epsilon^{\hat{m}+4} \\
&= \left\lfloor \left(\sum_{k'=0}^{\hat{m}-1} \sum_{l' \geq \hat{m}+2} w_{k'l'} \right) / \epsilon^{\hat{m}+3} \right\rfloor \epsilon^{\hat{m}+3} \qquad (5.10) \\
&\qquad + \hat{u}_{\hat{m}} \epsilon^{\hat{m}+4} + \left\lfloor \left(\sum_{l' \geq \hat{m}+2} w_{\hat{m},l'} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4} \\
&= \left\lfloor \left(\sum_{k'=0}^{\hat{m}} \sum_{l' \geq \hat{m}+2} w_{k'l'} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4} \\
&= \left\lfloor \frac{\theta}{\epsilon^{\hat{m}+4}} \right\rfloor \epsilon^{\hat{m}+4} \qquad (5.11)
\end{aligned}$$

For the third equality, we used Property 5.5.11 and Property 5.5.12, similarly as in the computation of (5.9). The last equality holds by definition of θ in Algorithm 12.

This allows us to show that Algorithm 12 returns t. If ι equals 'init', it holds that θ is a multiple of $\epsilon^{\hat{m}+4}$ and therefore $\rho_E^{(3)} = \theta$, and an arc to t is created. Otherwise, the difference between $\rho_E^{(3)}$ and θ is at most $\epsilon^{\hat{m}+4}$ and an arc to t is created as well.

It remains to prove the following inductive hypothesis about the length of the path.

Inductive Hypothesis 2. *For $k = 0, \ldots, \hat{m}$, let P be the path from s to a node in the first layer of stage k, as constructed above. Then the length of path P equals:*

$$\sum_{k'=0}^{k-1} \sum_{i \in M(k')} \frac{1}{b_i} \left\lfloor \left(\sum_{j \in N} \bar{w}_j x_{ij}^{MIP} \right) / \epsilon^{k+4} \right\rfloor \epsilon^{k+4}$$

The length of the path from s to itself is 0, which proves the base case. Inductively, we can assume that IH 2 holds for the length of the path up to the first layer of

stage k. We can fill in the choice of r_i into the length of an arc within a stage k:

$$\begin{aligned}
&\frac{1}{b_i}\left[\sum_{j=1}^{\omega}\bar{w}_j^{(k)}x_j^L+\sum_{j=1}^{\omega}\bar{w}_j^{(k+1)}x_j^M+r_i\epsilon^{k+4}\right]\\
=&\frac{1}{b_i}\left[\sum_{j=1}^{\omega}\bar{w}_j^{(k)}x_j^L+\sum_{j=1}^{\omega}\bar{w}_j^{(k+1)}x_j^M+\left\lfloor\left(\sum_{j\in N^S(i)}\bar{w}_jx_{ij}^{\mathrm{MIP}}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}\right]\\
=&\frac{1}{b_i}\left[\sum_{j\in N^L(i)\cup N^M(i)}\bar{w}_jx_{ij}^{\mathrm{MIP}}+\left\lfloor\left(\sum_{j\in N^S(i)}\bar{w}_jx_{ij}^{\mathrm{MIP}}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}\right]\\
=&\frac{1}{b_i}\left[\left\lfloor\left(\sum_{j\in N}\bar{w}_jx_{ij}^{\mathrm{MIP}}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}\right].
\end{aligned}$$

The second equality holds since we follow $\boldsymbol{x}^{\mathrm{MIP}}$ to choose the configuration for the i'th knapsack. The last equality holds since all large and medium items with respect to stage k are multiples of ϵ^{k+4}

Accumulating the length of the arcs during stage k, the length of the path has increased as follows:

$$\begin{aligned}
&\sum_{k'=0}^{k-1}\sum_{i\in M(k')}\frac{1}{b_i}\left\lfloor\left(\sum_{j\in N}\bar{w}_jx_{ij}^{\mathrm{MIP}}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}\\
&\qquad+\sum_{i\in M(k)}\frac{1}{b_i}\left\lfloor\left(\sum_{j\in N}\bar{w}_jx_{ij}^{\mathrm{MIP}}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}\\
=&\sum_{k'=0}^{k}\sum_{i\in M(k')}\frac{1}{b_i}\left\lfloor\left(\sum_{j\in N}\bar{w}_jx_{ij}^{\mathrm{MIP}}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}\\
=&\sum_{k'=0}^{k}\sum_{i\in M(k'),i\neq\iota}\frac{1}{b_i}\left(\sum_{j\in N}\bar{w}_jx_{ij}^{\mathrm{MIP}}\right)+\frac{1}{b_\iota}\left\lfloor\left(\sum_{j\in N}\bar{w}_jx_{\iota j}^{\mathrm{MIP}}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}
\end{aligned}\tag{5.12}$$

The last equality is true since Property 5.5.12 proves that ι might be the only knapsack that has a weight that is not a multiple of ϵ^{k+4}. We will add the length of the chosen update arc and prove that the update arc points to a node for which IH 2 holds. The length of the chosen update arc (with $u=\hat{u}_{k+1}$) is as follows:

$$\begin{aligned}
\frac{1}{b_\iota}\hat{u}_{k+1}\epsilon^{k+5}=&\frac{1}{b_\iota}\left(\left\lfloor\left(\sum_{k'=0}^{k}\sum_{l'\geq 0}w_{k'l'}\right)/\epsilon^{k+5}\right\rfloor\epsilon^{k+5}-\left\lfloor\left(\sum_{k'=0}^{k}\sum_{l'\geq 0}w_{k'l'}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}\right)\\
=&\frac{1}{b_\iota}\left(\left\lfloor\left(\sum_{j\in N}\bar{w}_jx_{\iota j}^{\mathrm{MIP}}\right)/\epsilon^{k+5}\right\rfloor\epsilon^{k+5}-\left\lfloor\left(\sum_{j\in N}\bar{w}_jx_{\iota j}^{\mathrm{MIP}}\right)/\epsilon^{k+4}\right\rfloor\epsilon^{k+4}\right)
\end{aligned}\tag{5.13}$$

Here the last equality holds since by Property 5.5.12, ι is the only knapsack in $M(0) \cup \cdots \cup M(k)$ that might not be a multiple of ϵ^{k+4}.

So the total path length to the first layer of the next stage $k+1$ is then as follows:

$$\begin{aligned}(5.12) + (5.13) &= \sum_{k'=0}^{k} \sum_{i \in M(k'), i \neq \iota} \frac{1}{b_i} \left(\sum_{j \in N} \bar{w}_j x_{ij}^{\text{MIP}} \right) \\ &\quad + \frac{1}{b_\iota} \left\lfloor \left(\sum_{j \in N} \bar{w}_j x_{\iota j}^{\text{MIP}} \right) / \epsilon^{k+5} \right\rfloor \epsilon^{k+5} \\ &= \sum_{k'=0}^{k} \sum_{i \in M(k')} \frac{1}{b_i} \left\lfloor \left(\sum_{j \in N} \bar{w}_j x_{ij}^{\text{MIP}} \right) / \epsilon^{k+5} \right\rfloor \epsilon^{k+5}\end{aligned}$$

The second equality holds since $\sum_{j \in N} \bar{w}_j x_{ij}^{\text{MIP}}$ is a multiple of ϵ^{k+5} for every knapsack except ι. This shows that IH 2 holds for the node in the first layer of stage $k+1$ that the chosen update arc points to, which proves that IH 2 is true.

Inductive Hypothesis 2 with $k = \hat{m}$ gives the path length to the node in the first layer of stage $\hat{m}$. Following the same reasoning as for the earlier stages, we get that the path length to a node in the last layer of stage $\hat{m}$ equals:

$$\sum_{k'=0}^{\hat{m}} \sum_{i \in M(k'), i \neq \iota} \frac{1}{b_i} \left(\sum_{j \in N} \bar{w}_j x_{ij}^{\text{MIP}} \right) + \frac{1}{b_\iota} \left\lfloor \left(\sum_{j \in N} \bar{w}_j x_{\iota j}^{\text{MIP}} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4}.$$

What remains to prove is that adding the length of the last arc to t ensures that this path length adds up to $c\left(\boldsymbol{x}^{\text{MIP}}, \bar{\boldsymbol{w}}\right)$. The length of the last arc is defined as

$\frac{1}{b_\iota}\left(\theta - \rho_E^{(3)}\right)$. We rewrite $\theta - \rho_E^{(3)}$ as follows:

$$\begin{aligned}
\theta - \rho_E^{(3)} &= \sum_{k'=0}^{\hat{m}} \sum_{l' \geq \hat{m}+2} w_{k'l'} - \left\lfloor \left(\sum_{k'=0}^{\hat{m}} \sum_{l' \geq \hat{m}+2} w_{k'l'} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4} \\
&= \sum_{k'=0}^{\hat{m}} \sum_{l' \geq \hat{m}+2} w_{k'l'} + \sum_{k'=0}^{\hat{m}} \sum_{l' \leq \hat{m}+1} w_{k'l'} \\
&\quad - \left\lfloor \left(\sum_{k'=0}^{\hat{m}} \sum_{l' \geq \hat{m}+2} w_{k'l'} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4} - \sum_{k'=0}^{\hat{m}} \sum_{l' \leq \hat{m}+1} w_{k'l'} \\
&= \sum_{k'=0}^{\hat{m}} \sum_{l' \geq 0} w_{k'l'} - \left\lfloor \left(\sum_{k'=0}^{\hat{m}} \sum_{l' \geq 0} w_{k'l'} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4} \\
&= \sum_{i \in M} \sum_{j \in N} \bar{w}_j x_{ij}^{\mathrm{MIP}} - \left\lfloor \left(\sum_{i \in M} \sum_{j \in N} \bar{w}_j x_{ij}^{\mathrm{MIP}} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4} \\
&= \sum_{j \in N} \bar{w}_j x_{\iota j}^{\mathrm{MIP}} - \left\lfloor \left(\sum_{j \in N} \bar{w}_j x_{\iota j}^{\mathrm{MIP}} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4}
\end{aligned}$$

The third equality holds since $w_{k'l'}$ is a multiple of $\epsilon^{\hat{m}+4}$ for every $l' \leq \hat{m}+1$. For the fourth equality we use the definition of $w_{k'l'}$. The fifth equality holds since by Property 5.5.12, ι is the only knapsack for which the two terms differ.

Now we can add the length of the last arc to the length of the rest of the path to obtain:

$$\begin{aligned}
&\sum_{k'=0}^{\hat{m}} \sum_{i \in M(k'), i \neq \iota} \frac{1}{b_i} \left(\sum_{j \in N} \bar{w}_j x_{ij}^{\mathrm{MIP}} \right) + \frac{1}{b_\iota} \left\lfloor \left(\sum_{j \in N} \bar{w}_j x_{\iota j}^{\mathrm{MIP}} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4} \\
&+ \frac{1}{b_\iota} \sum_{j \in N} \bar{w}_j x_{\iota j}^{\mathrm{MIP}} - \frac{1}{b_\iota} \left\lfloor \left(\sum_{j \in N} \bar{w}_j x_{ij}^{\mathrm{MIP}} \right) / \epsilon^{\hat{m}+4} \right\rfloor \epsilon^{\hat{m}+4} \\
&= \sum_{k'=0}^{\hat{m}} \sum_{i \in M(k'), i \neq \iota} \frac{1}{b_i} \left(\sum_{j \in N} \bar{w}_j x_{ij}^{\mathrm{MIP}} \right) + \frac{1}{b_\iota} \sum_{j \in N} \bar{w}_j x_{\iota j}^{\mathrm{MIP}} \\
&= \sum_{k'=0}^{\hat{m}} \sum_{i \in M(k')} \frac{1}{b_i} \left(\sum_{j \in N} \bar{w}_j x_{ij}^{\mathrm{MIP}} \right) = c\left(\boldsymbol{x}^{\mathrm{MIP}}, \bar{\boldsymbol{w}} \right)
\end{aligned}$$

□

5.6 Conclusion and Discussion

In this chapter, we presented two bicriteria approximation algorithms. In essence, both algorithms are different methods for finding an optimal solution to the Small Item Relaxation, after which the presented LP-rounding procedure yields an approximate solution for the Casting Problem. In the first bicriteria approximation algorithm that we present, a greedy algorithm suffices to find the optimal solution. Even though the algorithm is fairly straightforward, it results in a $(1, 3/2)$-bicriteria approximation algorithm.

For the second bicriteria approximation algorithm, it takes significantly more work to find a feasible solution to the Small Item Relaxation. We acknowledge that a complex and large, although polynomial, graph must be constructed. Future research could explore to find out if the construction of the graph or the analysis of the algorithm can be simplified. On the other hand, even though the construction of the graph is complex, it pays off in the form of a PTBAS. Since it is not possible to find an algorithm that computes a feasible solution for the Casting Problem within polynomial time, unless P=NP, there is not much more that we can expect than a PTBAS.

Chapter 6

Machine Learning in Large Neighborhood Search for VRPTW: Neighborhood Selection

6.1 Introduction

Efficiency in the planning of large logistics companies is of major importance, both for reducing the environmental footprint and for reducing costs. To create efficient schedules, planners at those companies need to be able to obtain high-quality solutions quickly. In practice, such solutions are often computed by using an *iterative approach* that proceeds along the following lines: (1) create an initial feasible solution, (2) iteratively improve upon the current solution until a certain termination criterion is met (e.g., maximum number of iterations is reached, sufficient solution quality is achieved). *Large Neighborhood Search (LNS)* is one such approach that is broadly applicable and has proven to be highly efficient in practice (see, e.g., the survey by Mara et al. [2022]). Each iteration of LNS consists of two steps: the *destroy method* and the *repair method*. For the repair method, often a general-purpose solver like a mixed integer programming (MIP) solver or a constraint programming solver can be used [Pisinger and Ropke, 2010]. If available, one can also take advantage of heuristics that are known to be well-performing and embed them as a repair method [Mara et al., 2022]. However, for the destroy method the situation is different: often this asks for the development of specialized algorithms tailored towards the specific scheduling problem, which requires expert knowledge and needs time to be developed and implemented.

As also mentioned by Ropke and Pisinger [2006], LNS works particularly well if the considered problem can be easily partitioned into a number of subproblems where some constraints must be satisfied, covered by a master problem to control how the subproblems are combined. Many routing and scheduling problems con-

form to this structure and are therefore oftentimes successfully solved with LNS in practice. A recent survey paper by Mara et al. [2022] on a specific type of LNS, namely *Adaptive Large Neighborhood Search (ALNS)*, categorized 252 scientific publications. Most of these articles are on routing and scheduling, but there are also applications in, e.g., manufacturing and agriculture. Below, we highlight a few specific applications of LNS to showcase its broad applicability.

Rastani and Çatay [2021] use LNS to solve the *Electric Vehicle Routing Problem with Time Windows*, where a level of complexity is added to standard *Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)* since vehicle batteries need to be recharged during the day. Factors like temperature and speed are taken into account to calculate a battery's range, and in this paper specifically, the carried load of a vehicle is taken into account. Chen et al. [2021] solve another variant of CVRPTW, named *Vehicle Routing Problem with Time Windows and Delivery Robots*. In this variant, vehicles can carry multiple robots which can take over part of the deliveries. An LNS algorithm is used to solve this problem. In a study by Kuhn et al. [2021], the Vehicle Routing Problem is intertwined with the order batching problem, where multiple customers are grouped together. They solve it using *General Adaptive Large Neighborhood Search*, a newly introduced Adaptive LNS method inspired by *General Variable Neighborhood Search* (see [Hansen et al., 2019]).

Recently, much research has been done on combining combinatorial algorithms and Machine Learning (ML), classified into three paradigms by Bengio et al. [2021] (see also the introduction of this thesis, Chapter 1). In the first paradigm, combinatorial problems are solved with machine learning directly from the input instance, like in Kool et al. [2019] for example. The second and third paradigms use ML as a subroutine of a combinatorial solving method, either once (paradigm 2) or repetitively (paradigm 3). We believe that ML can be a great benefit as a subroutine of already proven and widely used methods like Large Neighborhood Search (LNS).

In this chapter, we propose to integrate ML into LNS to assist in deciding which parts of the schedule should be destroyed and repaired in each iteration. In particular, we investigate how to exploit machine learning techniques to amplify the workings of any possible destroy algorithm. Conceptually, our new approach can be applied to any LNS that makes some random choices (explained in more detail below). We refer to our new approach as *Learning-Enhanced Neighborhood Selection* (*LENS* for short).

This research was inspired by experimental findings that we obtained for a real-world application solving large-scale routing problems on a daily basis. When using our LENS approach to guide the destroy method of a (highly sophisticated) LNS for this application we observed that this leads to a significant speed-up of the optimization process. The LENS approach described in this chapter is based on similar ideas. However, our approach is implemented for and tested on publicly available and well-studied benchmark instances of the *Capicitated Vehicle Routing*

Problem with Time Windows (see also [Accorsi et al., 2022]). The algorithms and data sets that were used for the experiments reported in this chapter are publicly available from the following repository:

`https://github.com/w-feijen/ML4LNS`

6.1.1 Our Contributions

The main contributions presented in this chapter are as follows:

1. We introduce a general *Learning-Enhanced Neighborhood Selection approach, referred to as* LENS *for short*, that integrates ML in the destroy method of an LNS algorithm. This approach is based on supervised learning and can be used as an enhancement of the workings of any destroy method using some form of randomization (as explained below).

2. We create an LNS algorithm for the *Capacitated Vehicle Routing Problem with Time Windows* (CVRPTW). The algorithm consists of (1) a destroy heuristic that exploits a newly defined distance measure, based both on distance and time windows, and (2) an existing repair heuristic.

3. We apply our LENS approach to the LNS algorithm of CVRPTW. For this purpose, we first define features that describe the potential improvement of a set of routes. We then collect data on these features and build an ML model on top of it. Our ML model can predict whether or not an improvement can be found in a given set of routes.

4. We provide general guidelines on how to collect the right data sets if supervised learning-based ML is used in optimization algorithms. Based on our experiments, it seems crucial to perform multiple iterations of data collection, where a premature ML model is used to guide subsequent data collection iterations. Eventually, the final ML model is trained on all the collected data. By following these guidelines, we ensure that the relevant information is gathered on runs in which the optimization is guided by the ML approach.

5. We generate a training set of CVRPTW instances based on the *R1* and *R2 instances* of Homberger and Gehring [2005] consisting of 1000 customers. Our generated training instances can be used to collect data samples for future supervised learning studies on the R1 and R2 instances.

6.1.2 Related Work

Combining machine learning with optimization is a hot topic. There is a survey about reinforcement learning in combinatorial optimization by Mazyavkina et al.

[2021], about enhancing optimization algorithms with ML for Vehicle Routing Problems by Bai et al. [2023], about using ML techniques in meta-heuristics (the class in which LNS falls as well) by Karimi-Mamaghan et al. [2022] and an overview which distinguishes three different paradigms of combinations between ML and combinatorial optimization by Bengio et al. [2021]. The first of these paradigms is to leverage machine learning to solve combinatorial optimization problems directly from the input instance. As opposed to this first paradigm, in which the combinatorial algorithm is replaced by an ML algorithm, the second and third paradigms use ML next to combinatorial methods, either in a single place (paradigm 2), or repetitively (paradigm 3).

Accorsi et al. [2022] give very useful guidelines for testing ML approaches in Vehicle Routing Problems (VRP). We highlight some of their advice: first, they stress the importance of a clear problem description and the representativeness of the test instances. In particular, for CVRPTW they advise using the instances by Homberger and Gehring [2005]. Moreover, they stress including the best available algorithms to benchmark against, and they give examples of how to visualize comparisons between algorithms.

Moreover, Accorsi et al. [2022] and also Wu et al. [2022] state that most of the proposed approaches consider *construction heuristics*, in which ML is used to construct a feasible solution. An example of using ML to construct a feasible solution is a study by Kool et al. [2019], in which a solution to the Travelling Salesperson Problem (TSP) is constructed iteratively. A few others focus on using ML in *improvement heuristics*, to guide the exploration of the search space and iteratively improve an existing solution. We focus on these improvement heuristics, which follow the third paradigm by Bengio et al. [2021] of combining combinatorial optimization and ML and replacing some of the intermediate steps in the larger improvement heuristic framework. Some related works in which ML is used to enhance search heuristics are given by Wu et al. [2022], Hottung and Tierney [2020], Li et al. [2021], Sonnerat et al. [2021]. We elaborate on these examples below.

Wu et al. [2022] use reinforcement learning in a neighborhood search to solve TSP and VRP without time windows. In particular, RL decides which pair of nodes to feed to a pairwise improvement operator. Similarly to Wu et al. [2022], we use ML to choose what to destroy in the solution. On the contrary, we choose a substantially larger part to destroy. Hottung and Tierney [2020] augment an LNS algorithm with a neural network to solve capacitated VRP without time windows. However, the neural network model is not used in the destroy step but in the repair step. Li et al. [2021] augment a local search algorithm for VRP without time windows with a transformer network, that identifies which set of routes needs to be optimized. A black box solver is used in the improvement step. Finally, Sonnerat et al. [2021] use an LNS algorithm to solve Mixed Integer Programs and ML is used to decide which of the variables to destroy.

Another line of research worth mentioning is the *Adaptive Large Neighborhood*

Search (ALNS), introduced by Ropke and Pisinger [2006] which is an extension of LNS. In each iteration, ALNS makes a choice on which of several destroy or repair methods to use, based on weights that are adapted during the run of the algorithm. A meta-analysis on the impact of the additional adaptive layer is done by Turkeš et al. [2021]. It shows that the adaptive layer improves the objective function value by 0.14%. Since the adaptive layer does add extra complexity, they recommend it only in some specific situations. A very related method is the *Hyper-Heuristic* (HH), which also chooses out of a set of predetermined heuristics how to improve the solution in each iteration. Lagos and Pereira [2024] present a HH based on a Markov model in which the weights are trained with a Multi-Armed Bandit model.

Kerscher and Minner [2024] solve a VRP by using a decomposition technique based on ML. They introduce spatial, temporal, and demand-based features for the customers and cluster the customers based on these features. A separate VRP is solved for each cluster, after which the solutions to the subproblems are combined into a large solution. Local search techniques are used to improve the aggregated solution. Even though this paper falls into the second paradigm of Bengio et al. [2021], since ML is only used to determine the clustering of the customers, the idea of clustering customers based on their spatial and temporal properties is very much related to our algorithm. Our method differs crucially though, since we create clusters of customers in each iteration, instead of only once before optimization.

6.1.3 Organization of Chapter

In the next section, we give a brief overview of the LNS algorithm, which is described in more detail in the preliminaries of this thesis. The main result of this chapter, our novel destroy method, is presented in Section 6.3. How to apply this method to the CVRPTW problem is described in Section 6.4, after which the results for such an application are presented in Section 6.5. We end the chapter with conclusions and a discussion of the results.

6.2 Preliminaries

Our algorithm is based on an iterative local search approach, known as *Large Neighborhood Search (LNS)* (see, e.g., Pisinger and Ropke [2010] and Shaw [1998]). LNS is a universal approach that can be applied to any generic combinatorial optimization problem. The approach is described in more detail in the preliminaries of this thesis in Section 2.2. We give a concise overview of the algorithm here again, and repeat the pseudocode in this Section in Algorithm 1 (repeated).

Let $\Pi = (\mathcal{I}, F, c)$ be the optimization problem under consideration, and let $I \in \mathcal{I}$ be the considered instance. LNS starts with an arbitrary feasible initial

Algorithm 1 Large Neighborhood Search (repeated)

Input: feasible solution $s \in F$
$s^{\text{best}} = s$
repeat
    $\eta = \text{SelectNeighborhood}(s)$
    $s^{\text{temp}} = \text{Repair}(\text{Destroy}(s, \eta))$
    if $\text{Accept}(s^{\text{temp}}, s)$ **then** $s = s^{\text{temp}}$
    if $c(s^{\text{temp}}) < c(s^{\text{best}})$ **then** $s^{\text{best}} = s^{\text{temp}}$
until StoppingCriterion is met
return s^{best}

solution $s \in F(I)$ as input. In the remainder of this chapter, for ease of notation, we omit the problem instance when denoting the set of feasible solutions and write F if we mean $F(I)$, and write $c(s)$ if we mean $c(I, s)$. In each iteration, a (small) part of the solution s, called *neighborhood*, is selected, which is then destroyed and repaired (or rebuilt) again, by a so-called *destroy method* and *repair method*, respectively. Crucially, the repair method only rebuilds a small part of the (potentially very large) solution, and might therefore outperform an approach that re-optimizes the whole solution. The *accept method* is used to determine whether the improvement of the newly created solution is significant enough, the best-known solution s^{best} is updated if necessary, and the algorithm terminates when a predefined *stopping criterion* is met.

As a key concept in this chapter, we elaborate more on our notion of a *neighborhood*, denoted by η in Algorithm 1. Typically, a solution s is defined by a (possibly ordered) set of solution elements. For a given solution, we define a neighborhood as a subset of these elements that might be selected in order to be destroyed.[1] It remains problem-specific how these neighborhoods are defined precisely. For example, for a routing problem like CVRPTW the neighborhood could be a subset of routes or customers (see also below), and for a scheduling problem, it could be a subset of machines or jobs.

As explained in the preliminaries, many advanced and well-working procedures are used for the implementation of the repair method, e.g., integer linear programming solvers or advanced path-building algorithms. For the neighborhood selection method, however, often more hand-crafted solutions are required, demanding time and expert knowledge. Therefore, we propose a novel neighborhood selection subroutine in this chapter, leveraging the power of machine learning, resulting in a subroutine that is application-independent.

[1] We remark that our notion of a neighborhood differs slightly from the one typically used in the context of LNS, where it refers to the set of all solutions that can be obtained from a given solution by applying the destroy and repair methods.

Algorithm 13 Learning-Enhanced Neighborhood Select'n (LENS)

1: **Input:** feasible solution $s \in F$ and integer n_1
2: **for** $j = 1, \ldots, n_1$ **do**
3: $\quad \eta_j = \text{CreateNeighborhood}(s)$
4: $\quad p_j = \text{PredictPotential}(s, \eta_j)$
5: $j^* = \arg\max_j p_j$
6: **return** η_{j^*}

6.3 LENS

6.3.1 Algorithm

Pisinger and Ropke [2010] observe that a destroy method typically uses randomization to ensure that different parts of the solution (i.e., neighborhoods) are destroyed in each iteration. However, if neighborhoods are destroyed that were already of high quality, it might be hard to find any improvement by repairing the solution again subsequently. Instead, one would rather aim to destroy neighborhoods in which much improvement can be found.

Based on this observation, we propose to use information from past iterations to make a decision on which neighborhoods to destroy. We do this by using ML techniques to predict the improvement gained after destroying and repairing a certain neighborhood. Ideally, the prediction enables us to identify a low-quality part of the solution such that the objective value improves significantly when this part is destroyed and repaired. Algorithm 13 contains the pseudocode for our proposed *Learning-Enhanced Neighborhood Selection (LENS)* method, which can be used as a SelectNeighborhood routine in Line 4 of LNS (Algorithm 1).

LENS starts with creating n_1 candidate neighborhoods to destroy. Specifically, the CreateNeighborhood method identifies a candidate neighborhood η_j of the solution to destroy. After that, an ML procedure PredictPotential predicts the potential p_j (in terms of improvement) of each candidate neighborhood η_j. The neighborhood with the highest potential is selected and returned. Subsequently, the returned neighborhood will be destroyed and repaired in the LNS algorithm.

Ideally, we would want to generate a diverse pool set of n candidate neighborhoods to choose from. This will be guaranteed if the CreateNeighborhood method uses randomization to create a new neighborhood. Generally, our approach applies whenever there is a meaningful way to generate such a pool set of neighborhoods.

Algorithm 14 DATA COLLECTION

Input: feasible solution $s \in F$ and integer n_1
$i = 0$
repeat
  $i = i + 1$
  for $j = 1, \ldots, n_1$ **do**
    $\eta_j = \text{CREATENEIGHBORHOOD}(s)$
    $\mathbf{x}_{ij} = \text{COMPUTEFEATURES}(s, \eta_j)$
    $s_j^{\text{temp}} = \text{REPAIR}(\text{DESTROY}(s, \eta_j))$
    $y_{ij} = \max(c(s) - c(s_j^{\text{temp}}), 0)$
    store $(\mathbf{x}_{ij}, y_{ij})$
  if DCSTRATEGY is random **then**
    $j^* = \text{RANDOMINTEGER}(1, n_1)$
  else ▷ DCSTRATEGY *is ML*
    $j^* = \arg\max_j \text{PREDICTPOTENTIAL}(s, \eta_j)$
  if $\text{ACCEPT}(s_{j^*}^{\text{temp}}, s)$ **then** $s = s_{j^*}^{\text{temp}}$
until STOPPINGCRITERION is met

6.3.2 Machine Learning Model

We describe the ML model that we use to predict the potential in our LENS approach described above (see Algorithm 13). We propose to use a supervised classification model. To train this supervised classification model, we will create a history of previous iterations.

The first step for creating this history is to define features. These features describe the neighborhood and are used by the ML model to make a prediction. It is application-dependent what the right features are for describing a neighborhood, an example of features for the application to CVRPTW is given in Section 6.4.2 and Appendix A.1.

In general, a sample in the history consists of (1) a set of features describing the neighborhood, denoted by $\mathbf{x}$, and (2) the improvement that this neighborhood gives, denoted by y. In contrast with the notation in Section 2.3 in the preliminaries of this thesis, we index the notation for the features and labels as follows: for the i'th iteration, and the j'th neighborhood in that iteration ($j \in \{1, \ldots, n_1\}$), the sample is denoted by $(\mathbf{x}_{ij}, y_{ij})$.

Our algorithm DATA COLLECTION creates a history by collecting such samples (see Algorithm 14). We elaborate on Algorithm 14 in more detail.

The CREATENEIGHBORHOOD method creates n_1 neighborhoods in each iteration of the DATA COLLECTION algorithm. As opposed to Algorithm 13, the repair method is executed for *all* of these neighborhoods. However, none of the updated solutions given by the repair method are directly accepted. Instead, in the i'th iteration of the DATA COLLECTION algorithm, both the features that de-

scribe the j'th neighborhood and the improvement that is gained after destroying and repairing this neighborhood are stored in the sample $(\mathbf{x}_{ij}, y_{ij})$. After storing the n_1 samples for iteration i, one neighborhood is selected. s is updated if the repaired solution for this neighborhood is accepted and the data collection LNS continues with the next iteration.

Which neighborhood is selected is based on the *data collection strategy* (DCStrategy in Algorithm 14). We have found, as is also elaborated on in the results in Section 6.5.6, that it is crucial to have the right data collection strategy. We first give some intuition on what this means, after which we summarize our advice in the guidelines below. The data collection strategy can either be random or follow predictions from a given ML model. During the first run of data collection, we use a random strategy, meaning one of the computed neighborhoods will be chosen uniformly at random. For an ML model to generalize well, it is crucial to learn from data that comes from the same distribution as the data used for testing. An ML model based on the data from random data collection only will probably not work well, since it guides the LNS to solutions on which no samples were collected during data collection. Therefore, it is important to collect data on the search space encountered when neighborhood selection is guided by ML, as done in our LENS algorithm. This is done by executing subsequent runs of data collection, this time guided by an ML strategy. After each run of data collection, a new ML model is trained on all previously collected data, which is then used to guide the subsequent run. By doing so, the idea is that each new ML model is trained on data that has a better resemblance to the data that will be encountered during testing.

To summarize, we give the following guidelines:

Guidelines 6.3.1. *When using supervised learning for combinatorial algorithms, the data collection should be executed as follows:*

1. *Perform data collection with random data collection strategy.*
2. *Based on the collected data, create an ML model, MLk with $k = 1$.*
3. *Given MLk for some $k \geq 1$, perform another run of data collection with MLk as the new data collection strategy.*
4. *Create a new ML model ML$(k + 1)$, based on collected data in all previous runs of data collection.*
5. *Repeat steps 3 and 4 to create new ML models ML3, ML4 and ML5.*

The result of collecting the data following these guidelines is a dataset of features of the shape $(\mathbf{x}_{ij}, y_{ij})$, where y_{ij} denotes the improvement that a neighborhood gives. However, we use classification models to make predictions in the LENS algorithm, and therefore the samples are labeled with a binary label before they

are used for training. This is done by defining an improvement threshold, after which each data sample $(\mathbf{x}_{ij}, y_{ij})$ gets label 1 if its improvement y_{ij} is above the threshold; otherwise, it gets label 0. When the classification model is used in our LENS algorithm, neighborhoods are ranked based on the output of the ML model. Therefore we need to choose an ML model that can output probabilities, like a neural network or a random forest.

6.4 Application to CVRPTW

The inspiration for the LENS algorithm came from a proof-of-concept applied to a real-world application. After finishing the proof-of-concept, we made the decision to further implement and test the LENS algorithm on synthetic CVRPTW datasets. Doing so allowed us to elaborate in full detail on all the aspects of the used LNS algorithm, which would not have been possible if we would have tested on the real-world application, due to a non-disclosure agreement about parts of the software. We elaborate more on the application to the CVRPTW in the following sections.

6.4.1 CVRPTW Definition

The vehicle routing problem is defined on a complete (undirected) graph $G = (V, E)$ on $n+2$ nodes. Without loss of generality, we identify the node set V with $\{0, 1, \ldots, n+1\}$. There are two designated nodes, indicated by 0 and $n+1$, that are called the *starting* and *ending depot*, respectively. We assume that the starting and ending depot are the same (i.e., they are co-located), but conceptually it is more convenient to distinguish between them in the formulation (as will become clear below). The remaining nodes correspond to customers requiring service and we use $V^{CST} = \{1, \ldots, n\}$ to refer to this set (excluding the depots). Each edge $(i, j) \in E$ represents a possible trip between node i and j with associated *travel time* d_{ij}. Oftentimes, as is also the case for the test instances that we consider in this chapter, customer locations are defined by coordinates on a grid and travel times are represented by the Euclidean distance between the respective locations. Each customer needs to be visited by one of the m identical vehicles in the fleet. The vehicles all start and end at the starting and ending depot, respectively.

In the *Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)*, extra capacity and time window constraints are imposed on the solution. More specifically, each customer i has a non-negative *demand* q_i which needs to be fulfilled by the vehicle, whilst the maximum capacity of each vehicle is limited to Q. For the time window constraints, each customer is associated with a *service duration* τ_i and an interval $[e_i, l_i]$, called the *time window*, in which the service must start. Note that an arrival at a customer before e_i is allowed, but this will force the vehicle to wait until e_i before the customer can be served. Moreover,

the end of the time window specifies the latest starting time of the service; in particular, a service that starts before l_i but ends after l_i is allowed. For the depots, we define $q_0 = q_{n+1} = 0$ and $\tau_0 = \tau_{n+1} = 0$. The time window of the depots, $[e_0, l_0] = [e_{n+1}, l_{n+1}]$, defines the entire time horizon of the problem during which all customers have to be served.

There are different objective functions studied in the context of CVRPTW. Here, we consider the case in which we want to minimize the total travel distance of the vehicles over the set of all feasible solutions that satisfy all customer requests (i.e., demand and time window) by using at most m vehicles of capacity Q.

The following is a Mixed Integer Programming (MIP) formulation of the problem (see, e.g., the formulation by Munari et al. [2016]).

$$\text{minimize} \quad \sum_{i \in V} \sum_{j \in V} d_{ij} x_{ij} \tag{6.1a}$$

subject to

$$\sum_{j \in V \setminus \{0\}} x_{ij} = 1 \quad \forall i \in V^{CST} \tag{6.1b}$$

$$\sum_{i \in V \setminus \{n+1\}} x_{ij} = 1 \quad \forall j \in V^{CST} \tag{6.1c}$$

$$\sum_{j \in V \setminus \{0\}} x_{0j} \leq m \tag{6.1d}$$

$$y_i + q_j x_{ij} - Q(1 - x_{ij}) \leq y_j \quad \forall i, j \in V \tag{6.1e}$$

$$q_i \leq y_i \leq Q \quad \forall i \in V \tag{6.1f}$$

$$t_i + (d_{ij} + \tau_i) x_{ij} - (l_0 - e_0)(1 - x_{ij}) \leq t_j \quad \forall i, j \in V,\ i \neq n+1,\ j \neq 0 \tag{6.1g}$$

$$e_i \leq t_i \leq l_i \quad \forall i \in V \tag{6.1h}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in V \tag{6.1i}$$

$$t_i \geq 0 \quad \forall i \in V \tag{6.1j}$$

$$y_i \in \mathbb{R} \quad \forall i \in V \tag{6.1k}$$

There are three types of decision variables in this MIP formulation:

- x_{ij} indicates if customer j is visited immediately after i.
- t_i is the time that the service of customer i starts.
- y_i is the cumulative demand on the route that visits customer i up to and including this visit.

Constraints (6.1b) and (6.1c) make sure that each customer is visited exactly once. Constraint (6.1d) ensures that at most m vehicles leave the depot. The vehicle capacities are monitored by constraints (6.1e) and (6.1f). Constraint (6.1e)

ensures that the cumulative demand is increased from y_i to $y_i + q_j$, if customer j is visited immediately after i and Constraint (6.1f) makes sure that the maximum capacity of a vehicle is not violated. Constraint (6.1g) and (6.1h) deal with the time window restrictions. More specifically, Constraint (6.1g) ensures that if customer j is served after i, then the service time of j is at least the service time of i plus the distance from i to j plus the service duration at i. This constraint also eliminates sub-tours. Constraint (6.1h) ensures that the start of the service is within the time window.

Recall that for a given problem instance $I \in \mathcal{I}$, we denote the set of feasible solutions with F and the cost of a solution $s \in F$ with $c(s)$. For CVRPTW, we adopt the convention that a feasible solution is represented by a set $s := \{r_1, r_2, \ldots, r_m\}$ of at most m routes, where each r_i is an ordered set of customers from V^{CST}. The cost $c(s)$ then simply refers to the total distance of all routes in s.

6.4.2 Large Neighborhood Search for CVRPTW

In order to solve the CVRPTW problem with LNS we need to define the necessary subroutines. More specifically, we need to define how to obtain an initial feasible solution, the Destroy, Repair and Accept methods, and a StoppingCriterion. Below, we elaborate in more detail on our Destroy method (Section 6.4.2) and Repair method (Section 6.4.2). To obtain an initial solution, we use an open-source optimization engine for vehicle routing problems called VROOM (see [Coupey et al., 2023]). That is, we simply call VROOM to compute an initial solution for each of our CVRPTW test instances. For the Accept method, we use a simple *hill-climbing procedure*, i.e., we only accept a new solution if its distance is smaller than the current one. Our StoppingCriterion is such that the LNS algorithm stops after a fixed number of iterations.

Destroy Method

To use the LENS method, (see Algorithm 13 in Section 6.3) we need to define a CreateNeighborhood method and define features that will be used to make an ML prediction.

Our CreateNeighborhood method is given in Algorithm 15. First, we choose a so-called *anchor route* r_a by choosing one route uniformly at random from the set of routes s in the current solution. Then, the neighborhood will be created around this anchor route as follows. Let $s^o = s \setminus \{r_a\}$ denote the set of remaining routes. We sort the routes in s^o based on a specific distance measure $\tilde{d}(\cdot, \cdot)$ between routes (defined below). We relabel the routes in s^o such that after the sorting it holds that for $r_i, r_j \in s^o$:

$$i \leq j \iff \tilde{d}(r_\mathrm{a}, r_i) \leq \tilde{d}(r_\mathrm{a}, r_j).$$

Algorithm 15 CreateNeighborhood

1: **Input:** set of routes S and integers n_2, n_3
2: $j = 0$
3: **repeat**
4: Let $r_a = \text{RandomRoute}(s)$ ▷ *Choose anchor route uniformly at random*
5: $s^o = s \setminus r_a$ ▷ *All other routes*
6: $\text{relabel}(s^o, r_a)$ ▷ *Sort s^o according to distance to r_a*
7: $p = (p_1, \dots p_{m-1}) = \text{RBP}(s^o, D)$ ▷ *Compute rank-based probabilities*
8: $\eta = r_a \cup \text{Sample}(s^o, n_2, p)$ ▷ *Sample n_2 routes using p*
9: $j = j + 1$
10: **until** $(j = n_3)$ **or** $(i(\eta)$ not in TabuList$)$
11: **return** η

Based on this ordering, we define rank-based probabilities $\text{RBP}(\cdot, D)$, depending on a fixed parameter D, that ensure that routes with a smaller distance to the anchor route r_a have a higher probability to being added to the neighborhood. More specifically, $\text{RBP}(s^o, D)$ defines a probability p_i for each route r_i in the ordered set s^o as follows:

$$p_i = \frac{\bar{p}_i}{\sum_{i:r_i \in s^o} \bar{p}_i}, \text{ where } \bar{p}_i = (|s^o| - i)^D.$$

Here $D > 1$ is a parameter that controls how much the probabilities differ from each other. A large D results in probabilities that are far apart and, consequently, the random sample drawn in Line 8 of Algorithm 15 will almost surely return the first n_2 available routes. On the contrary, a small value for D will result in probabilities that are closer together, and therefore cause more variety in the outcome of the random sample.

We use this probability distribution $p = (p_1, \dots, p_{m-1})$ over the routes in s^o to sample n_2 additional routes without replacement. Those n_2 routes then form the neighborhood together with the anchor route.

We elaborate in more detail on the definition of our distance measure $\tilde{d}(r_i, r_j)$ used above. Ideally, the distance measure is not based on customer locations only but also takes differences in time windows into account. Especially tight time windows impact how interchangeable the customers are between routes. The distance from route r_i to r_j is defined as follows:

$$\tilde{d}(r_i, r_j) := \min_{u \in r_i} \{\text{dist}(u, r_j)\},$$

where $\text{dist}(u, r_j)$ is a distance measure between location u and route r_j. The

distance $\text{dist}(u, r_j)$ is calculated based on the tightness of the time window of u:

$$\text{dist}(u, r_j) = \begin{cases} d_{u,\text{suc}(r_j,u)} & \text{if time window of } u \text{ is tight,} \\ d_{u,\text{cent}(r_j)} & \text{else.} \end{cases}$$

In the instances that we considered, some customers have very tight time windows ($\sim 2\%$ of the length of the total planning horizon), and others have a time window equal to the full planning horizon. If customer u has a tight time window, we can make a well-educated guess before which customer it would be served in r_j, would it be added to r_j. We denote this successor node of u in r_j as $\text{suc}(r_j, u)$. It is computed by taking the first customer in r_j that has an arrival time that is later than the midpoint of the time window of u. We set $\text{dist}(u, r_j)$ equal to the distance from u to $\text{suc}(r_j, u)$. Otherwise, if u does not have a tight time window, we simply take the Euclidean distance from u to the centroid of r_j, denoted by $\text{cent}(r_j)$. The centroid of a route is defined as the arithmetic mean of its customer locations.

In the neighborhood creation method, we want to avoid creating neighborhoods that we have created before. Therefore we keep track of a list of neighborhoods that we created before, which we term the TabuList. We want to check if a neighborhood is in the TabuList efficiently and therefore introduce a new way of storing a neighborhood. If we assume that each vehicle in the solution has an index between 1 and m, then instead of denoting a neighborhood as an unordered set of ordered lists of customers (as η does), we use $i(\eta)$ to denote a neighborhood, where $i(\eta)$ indicates the indices of the vehicles in the neighborhood η. Note that referring to a neighborhood by the vehicle indices only works as long as the routes of the indicated vehicles do not change.

In Algorithm 15, we continue creating neighborhoods until we find a neighborhood for which the set of indices is not in the TabuList. To ensure that the neighborhood creation terminates, we accept any neighborhood, even if it is in the TabuList, after n_3 tries.

We need to make sure that the neighborhood that is selected in each iteration, say η, is added to the TabuList. Moreover, if destroying and repairing η caused the solution to change, we need to delete every neighborhood from the TabuList that has overlapping routes with η. We can adjust the hill-climbing accept method slightly to perform these tasks for the TabuList. The accept method presented in Algorithm 16 does exactly this. Each selected neighborhood is added to the TabuList in Line 4. Before that, in Line 3, every neighborhood η' in the TabuList for which $i(\eta) \cap i(\eta') \neq \emptyset$ is deleted from the TabuList, if η is an improving neighborhood.

Next to defining the CreateNeighborhood method, we need to define features that describe the potential of improvement of a neighborhood, as explained in Section 6.3. We define several properties that describe both the routes in the neighborhood and the customers on these routes. To aggregate these features,

Algorithm 16 TabuAccept

```
Input: neighborhood η, new solution s^temp and current solution s
if c(s^temp) < c(s) then
    TabuList.DeleteRoutesFrom(i(η))
TabuList.Add(i(η))
return c(s^temp) < c(s)
```

we take the average, maximum, minimum, sum and standard deviation over the routes, for the route properties, and over the customers, for the customer properties. We elaborate on the type of features briefly, but give an extensive list in A.1. The first feature is the number of customers in the neighborhood. Secondly, we define properties that describe the customers in the neighborhood, like waiting time, distance contribution and closeness to other routes in the neighborhood. Thirdly, we define route properties like route distance, route duration and free capacity. Lastly, we add the distance measure $\tilde{d}(r_i, r_j)$ between all the routes in the neighborhood.

Repair Method

The repair method considers the routes that are deleted by the destroy method and tries to create a new solution for them. Many repair and improvement methods are known, such as 2-opt, 3-opt, LKH-3 or exact solvers.

In our application, a neighborhood is defined by a set of routes. A moment of thought reveals that the sub-problem of finding an improvement in a neighborhood is therefore a CVRPTW problem in itself. In fact, the number of customers and routes in this sub-problem is much smaller than in the original CVRPTW. Namely, the number of vehicles in this sub-problem is equal to the number of routes that were destroyed, and the set of customers in the sub-problem is the set of customers that was destroyed. The task of the repair method is then to find a, hopefully improved, solution for this sub-problem. To speed up the repair method, we can use the current solution for the destroyed routes as a warm start.

We have seen that in practice often black-box solvers are used to repair a routing solution after it has been destroyed. Therefore, we have also chosen to use an external CVRPTW solver to solve the sub-problem defined by the deleted set of routes, namely VROOM [Coupey et al., 2023]. VROOM is an open-source vehicle routing problem solver that is tailored towards getting high-quality solutions quickly. By choosing such a black box solver as a repair method, we can exploit the power that this solver has on smaller instances and leverage it to solve larger instances.

6.5 Results

We tested our algorithm on the 20 R1 and R2 Homberger and Gehring [2005] instances with 1000 customers. To build our ML models, we needed to collect data on similar instances. Therefore we generated 200 training instances similar to the test instances. Below, we elaborate on the creation of these training instances (Section 6.5.1), the values used for the hyperparameters (Section 6.5.2), and the subsequent data collection (Section 6.5.3). Based on the collected data, we trained a random forest classification model which we validated on a validation dataset (Section 6.5.5). We tested our model with the LNS algorithm. Our initial findings revealed that we needed to change the data collection strategy in order for the ML model to be based on the correct data (Section 6.5.6). This insight was formalized in the guidelines presented above in Section 6.3.

6.5.1 Instance Generation

Following a recommendation by Accorsi et al. [2022], we used the R1 and R2 Homberger and Gehring [2005] instances for our experimental results. We call these instances the *test instances*. There are 20 test instances, which contain 1000 customers each.

We want to train our ML models based on different instances from the ones we test on. Therefore, we decided to generate new training instances. Following the distribution of the original 20 test instances by Homberger and Gehring [2005], we created 200 new training instances. For both test sets, these instances were created in 10 batches of 10 instances each, where each original test instance forms the basis for 10 new training instances. We call the 10 training instances that belong to one test instance its *similar training instances*. Just as in the original set of 20 test instances, for a given test set, the customer locations in all of the created instances are exactly the same. In other words, for a given test set, the only property that differs between the instances are the time windows.

For the first test set R1, the generation of the instances in a batch works as follows. Once for each batch, we compute for each customer what the middle of its time window will become. We do this by sampling uniformly at random over the instance horizon (to ensure feasibility of the instance, we take into consideration the time necessary to travel back from the customer to the depot when sampling the middle of the time windows) . The 10 instances in each batch differ by the number of customers that get a restrictive time window, and the length that this restrictive time window will be. All customers with a non-restrictive time window can be served during the whole horizon. The first 4 instances in a batch have a time window length of 10, and the fraction of customers with a restrictive time window is 100%, 75%, 50% and 25%, respectively. The same holds for the next 4 instances in the batch, but with a time window length of 30. In the last two

Test Instance Name	R1_10_1	R1_10_2	R1_10_3	R1_10_4	R1_10_5
TW Length	10	10	10	10	30
TW Fraction	100%	75%	50%	25%	100%
Test Instance Name	R1_10_6	R1_10_7	R1_10_8	R1_10_9	R1_10_10
TW Length	30	30	30	$\mathcal{N}(60, 20)$	$\mathcal{N}(120, 30)$
TW Fraction	75%	50%	25%	100%	100%

Table 6.1: Time window length (or distribution) and fraction of customers with time window for 10 R1 test instances used during instance generation.

Test Instance Name	R2_10_1	R2_10_2	R2_10_3	R2_10_4	R2_10_5
TW Length	$\mathcal{N}(60, 20)$	$\mathcal{N}(60, 20)$	$\mathcal{N}(60, 20)$	$\mathcal{N}(60, 20)$	240
TW Fraction	100%	75%	50%	25%	100%
Test Instance Name	R2_10_6	R2_10_7	R2_10_8	R2_10_9	R2_10_10
TW Length	240	240	240	$\mathcal{N}(185, 84)$	$\mathcal{N}(240, 30)$
TW Fraction	75%	50%	25%	100%	100%

Table 6.2: Time window length (or distribution) and fraction of customers with time window for 10 R2 test instances used during instance generation.

instances of a batch, all customers have a restrictive time window, for which the length is sampled from a normal distribution with a mean of 60 (120, resp.) and standard deviation of 20 (30, resp.). The generation for the second test set R2 works similarly. The details of this instance generation for both test sets are summarized in Table 6.1 and Table 6.2.

6.5.2 Hyperparameters

Table 6.3 shows the values for the hyperparameters that we use in our algorithms. The second and third column show the value for the randomness parameter D in Algorithm 15, for which we used a different parameter during data collection. The rest of the columns show the values for the parameters used in Algorithm 13, 14, and 15. For R1, n_3 is equal to 1, since we did not use a TabuList there and therefore returned the first created neighborhood in Algorithm 15.

For building the random forest models, we used the following hyperparameters. We set the number of features to consider when looking for the best split equal to the square root of the total number of features. We set the minimum number of samples required to be at a leaf node equal to 0.1% of the total number of samples and the minimum number of samples required to split an internal node equal to 0.2% of the total number of samples.

Instance Set	D	D for DC	n_1	n_2	n_3
R1	40	10	10	10	1
R2	5	10	10	2	5

Table 6.3: Hyperparameters used for two test sets.

6.5.3 Data Collection

In order to train an ML model, we collected data on the generated instances, as explained in Section 6.3. For each of the 200 training instances, we did 10 runs of data collection of 500 iterations each. We used the random data collection strategy to build an ML model, ML1, and followed our Guidelines for building the consecutive ML models, ML2, ML3, ML4 and ML5, based on more data. The sample that we saved for each neighborhood in an iteration, defined by $(\mathbf{x}_{ij}, y_{ij})$ (Section 6.3) consists of $\mathbf{x}_{ij}$, the features describing the neighborhood (Section 6.4.2), and y_{ij}, the improvement that VROOM is able to find for this neighborhood. This improvement is defined as the difference in total travel distance between the routes in the neighborhood before VROOM optimizes them and after optimization.

There are 10 *similar training instances* for each test instance. We did 10 training runs of 500 iterations, during which 10 samples were stored in each iteration. This gave $10 \cdot 10 \cdot 500 \cdot 10 = 500,000$ samples for each test instance, on which we based the random forest model for ML1. Each consecutive created ML model (ML2, ML3, ML4 and ML5) was based on 500,000 more samples.

6.5.4 Benchmarking Algorithms

To test our ML models, we define two algorithms to compare with: the oracle model and the random model. Both of these algorithms follow the Large Neighborhood Search structure presented in Algorithm 1, and like LENS, they create n_1 neighborhoods in each iteration. However, the benchmarking algorithms differ in how they select one of these neighborhoods, i.e., in their SelectNeighborhood subroutine.

The random model selects one of the created n_1 neighborhoods at random. In fact, the random model mimics the situation in which only one neighborhood is created per iteration, since the other nine neighborhoods may be regarded as never created. We hope our ML algorithm will be able to beat this random model.

The second benchmark is the oracle model, which selects the neighborhood that would yield the highest improvement if it was destroyed and repaired. The oracle model can be seen as an ML model with perfect predictions; or, alternatively, it represents the potential that the best, though fictitious, LENS algorithm could give. To acquire this full knowledge the oracle needs to compute the de-

stroy and repair step for each of the n_1 neighborhoods. As a consequence, it is a very slow and expensive procedure. In fact, for most practical applications it is even infeasible to compute it. However, it provides an insightful and strong benchmarking algorithm for our experiments.

6.5.5 Classification and Validation

The 500,000 collected samples per instance were partitioned into a training (60%) and validation (40%) set. After the collection, we labeled the samples with an improvement threshold of 0, meaning that a sample belongs to the positive class if it has a positive improvement, and to the negative class otherwise, as explained in Section 6.3. This resulted in a heavily unbalanced dataset with, e.g., only 11% of training samples being positive (for the first test instance). Therefore, after applying a standard scalar to normalize the data, we balanced the sample weights and trained a random forest model. For each of the test instances, we created a random forest like this. We call these models *ML1*.

For R1, we validated our ML1 model on two different sets of samples. The first set is the *validation samples*, set apart earlier, based on the similar training instances. Next to that, we created a set of samples, based on the test instance, which we call the *test samples*. We created the test samples by running the data collection with the random data collection strategy on the test instance and obtained 50,000 samples coming from 5,000 iterations.

The validation of an Algorithm ALG works as follows. Recall that a sample is denoted by $(\mathbf{x}_{ij}, y_{ij})$, where i denotes the iteration and $j \in \{1, \ldots, n_1\}$ is the index for the neighborhood in iteration i. In each iteration i, we let ALG decide which of the n_1 neighborhoods is the best, and denote its index with j^{ALG}. Then for each iteration i we save $y_{i,j^{ALG}}$ and compute the average over all iterations, this value indicates the average improvement of the neighborhoods that are considered best by ALG. Moreover, we check in how many of the iterations $y_{i,j^{ALG}}$ was a strictly positive improvement. This value indicates the fraction of iterations in which ALG would be able to find an improvement.

The results for the R1 validation samples are given in Table 6.4 and the results for the test samples are given in Table 6.5. For the results in this table, we only considered the iterations in which there was at least one improving neighborhood. Table 6.4 and Table 6.5 show the oracle model, the random model and the ML1 model.

For the validation samples, we see that the ML1 model is able to identify significantly more improving neighborhoods than the random model (44% against 18%). Also, the average improvement is about three times higher for the ML model. For the test samples, we see that the ML model is able to find an improvement in 28% of the iterations, while the random model only finds an improvement in 20% of the cases. Moreover, we see that the average improvement of the ML model (4.7) is about 46% higher than that of the random model (3.2). Even

though we lose some prediction power if we start predicting for samples from the test set, we are still able to outperform random.

Neighborhood Selection Model	Oracle	Random	ML1
Average true rank of best prediction	1	2.53	2.1
Average fraction of improving neighborhoods	100%	18%	44%
Average improvement	21.7	3.0	8.8

Table 6.4: Validation of ML1 for R1 on validation samples for three neighborhood selection models

Neighborhood Selection Model	Oracle	Random	ML1
Average true rank of best prediction	1	2.67	2.52
Average fraction of improving neighborhoods	100%	20%	28%
Average improvement	21.3	3.2	4.7

Table 6.5: Validation of ML1 for R1 on test samples for three neighborhood selection models

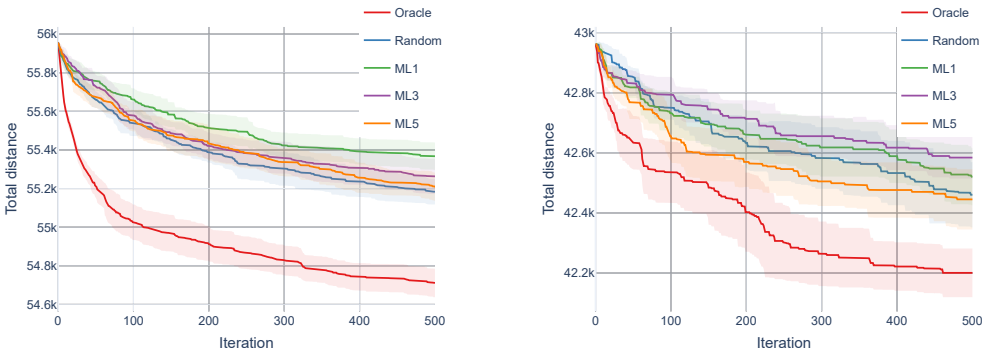
6.5.6 Algorithm Results

After the validations, we checked how our ML1 performed in the LNS algorithm introduced in Section 13. We tested this for the 20 test instances and started the optimization with an initial feasible solution created by VROOM. All of the tests ran for 500 iterations, and each test instance was run 10 times.

For two instances, Figure 6.1 shows the decrease of the total distance during the optimization runs of LNS with the ML1 model compared to LNS with the random model and LNS with the oracle model. Table 6.6 shows the average total distance after 500 and 200 iterations, for the two sets of 10 test instances. The extended results are given in B.1. Next to these average distances, the tables show the average best-known solution for these instances, as recorded by Sintef [2023].

We compare our ML algorithms with the oracle and the random model by using a gap measure, which indicates how far off the results with ML are from the best-case algorithm, which is the oracle. Let s_{O}, s_{R}, s_{ALG} be the (average) solution found by the oracle model, random model, and another algorithm ALG. Formally, the gap for the ALG solution is then defined as follows:

$$\mathrm{GAP}(s_{\mathrm{ALG}}) = \frac{c(s_{\mathrm{ALG}}) - c(s_{\mathrm{O}})}{c(s_{\mathrm{R}}) - c(s_{\mathrm{O}})}.$$



(a) Total distance for instance R1_10_1

(b) Total distance for instance R2_10_1

Figure 6.1: Total distance for 2 test instances for the oracle model, the random model, ML1 model, ML3 model and ML5 model

A gap of 100% means the model is as bad as the random model, a gap of 0% means the model is as good as the oracle model.

Figure 6.1 and Table 6.6 show that the quality of the solution increases during optimization since the total distance of the solution decreases. The oracle (which takes the most time and effort per iteration) has the lowest total distance after 500 iterations, as expected. For the R2 instances, we see that the ML1 model has a GAP of 97.69% and is therefore an improvement as compared to random. Unfortunately, we see for most of the R1 instances that the ML1 model is not able to improve over the random model. This means that for those instances, following the recommendation by the ML1 model is worse than following random choices. The average gap of 108.51% for R1 also shows that ML1 is not able to beat random for the R1 instances.

At first sight, this seems a contradiction to the validation that was shown in the previous section. However, a more in-depth analysis of our experiments reveals that, because we follow the choices based on ML, we enter a part of the solution space that was not encountered during the data collection. Since the ML model was not trained on this part of the solution space, it does not perform well here.

Therefore we believe that it is crucial to do multiple runs of data collection, as stated in our Guidelines (Section 6.3). In the second (third/fourth/fifth) run of data collection, instead of using the *random* data collection strategy, we used the ML1 (ML2/ML3/ML4) model during data collection. That is, the ML1 (ML2/ML3/ML4) model decides which of the neighborhoods suggested by VROOM should be implemented. This resulted in an expanded dataset of samples. On this expanded dataset, consisting of all data collected in previous training rounds, we trained a new random forest model, denoted as *ML2* (ML3/ML4/ML5).

			Oracle	Random	ML1		ML5	
Set	Iteration	BKS	Avg	Avg	Avg	Gap	Avg	Gap
R1	200	46901.3	48363.9	48880.2	48860.2	96.13%	48820.2	88.38%
R1	500	46901.3	48217.0	48641.0	48677.1	108.51%	48628.6	97.08%
R2	200	28842.2	30835.5	31043.4	31009.0	83.45%	31011.7	84.75%
R2	500	28842.2	30804.1	30912.3	30909.8	97.69%	30909.7	97.60%

Table 6.6: Total distance after 200 and 500 iterations of the oracle model, random model, ML1 model and ML5 model. Also the total distance of the best-known solution (BKS). Recall that the gaps of the oracle and random models are defined as 0% and 100%, respectively.

Table 6.6 shows the results for ML5. Again, for the extended results, including ML3, we refer to Table B.2 and Table B.1 in Section B.1 in the Appendix of this thesis. For R1, the gap improved to 97.08% for ML5. This significant improvement compared to ML1 for the R1 instances shows the importance of performing multiple rounds of data collection. For R2, the gap for ML5 was similar to the gap for ML1. After 200 iterations, we see a larger improvement for both R1 and R2 compared to the random algorithm, with an average gap of 88.38% and 84.75%. This shows that mainly in the first half of the optimization run, we have learned to do better than random.

6.6 Conclusion and Discussion

This research was inspired by experimental results that we obtained for a real-world application, where we used our LENS approach to enhance the neighborhood selection of an LNS algorithm.

The results described in this chapter were obtained through a similar approach using publicly available synthetic CVRPTW datasets. On the two sets of CVRPTW instances, our LENS technique gives a solution with a cost that is 11.62% and 15.25% smaller after 200 iterations, respectively, and 2.92% and 2.40% smaller after 500 iterations, respectively, compared to a solution obtained with random neighborhood selection. The results that we obtained on the real-world application were even more significant.

We believe that this is mainly caused by the underlying baseline algorithm to which the new destroy method is added. In the real-world application, the baseline algorithm is a sophisticated and very well-working algorithm, and there are many neighborhoods in an iteration that yield an improvement. In our baseline algorithm, however, this number is much lower. It seems that the destroy method with learning benefits from a large number of improving neighborhoods since it is easier to identify large improvements if there are many promising neighborhoods available in the candidate set. In our baseline algorithm, however, the number of promising neighborhoods per iteration is much smaller and therefore it is also

harder to select a good one. Nevertheless, we still obtain the earlier mentioned improvements.

Given that LNS is a universal approach that is broadly applicable and easy to implement, we believe that our LENS approach is a promising way to improve the quality of the computed solutions through the help of ML techniques. The actual magnitude of improvement might be application-specific though and, in particular, depends on the quality of the created neighborhoods.

Chapter 7

Machine Learning in Large Neighborhood Search for VRPTW: Neighborhood Creation

7.1 Introduction

In the previous chapter, we used supervised learning to select neighborhoods in the destroy operator of a large neighborhood search algorithm. The neighborhoods were selected from a given set of already constructed neighborhoods. In this chapter, we consider the same large neighborhood search algorithm again, but we propose a different method of applying learning techniques to the destroy step. By doing so, we address two possible challenges that the neighborhood selection in the previous chapter might face.

The first challenge in the previous chapter is the inability to self-adapt. In a supervised learning setting, the distribution of the training data must be the same as the distribution of the test data, otherwise the ML model does not generalize well. The ML models need to be retrained if the problem structure changes, for example when there is a shift in the service area of the delivery vehicles. If the ML model is trained on instances with urban deliveries, it will probably not perform well in rural instances.

The second challenge that the neighborhood selection method in the previous chapter faces is the limited freedom that the ML model is given. The ML model can only select a neighborhood based on a given set of created neighborhoods, limiting its power to the given options. In this chapter, we increase the freedom of the ML model by not only selecting neighborhoods but also creating them.

To summarize, to address the challenges that the method in the previous chapter might face, the goal in this chapter is to create a self-adapting neighborhood creation method that has the power to construct powerful neighborhoods.

We achieve a self-adapting method by training the ML models with reinforcement learning (RL), continuously during optimization. This results in robust ML models that have the ability to adapt to a changing problem environment. Lifting the selection of neighborhoods to the creation of neighborhoods is achieved by constructing neighborhoods iteratively, route by route. To construct these neighborhoods we adopt a graph-attention-based method for constructing combinatorial solutions.

The graph attention model that we adopt is introduced in Kool et al. [2019]. It is a versatile method for constructing combinatorial solutions for several routing problems. In order to use the method for an application, we need to define the context, a mask and a stopping criterion. The *context* is used to store information about the currently constructed solution necessary for making the next decision. The *mask* identifies which solution elements cannot be chosen. The *stopping criterion* defines when constructing the solution is finished.

The first application we consider is a pickup and delivery problem with time windows (PDPTW) and additional compatibility constraints, coming from a real-world instance (more details in Section 7.4.1). Instances of this problem are solved daily for many logistics clients by an existing tool of DELMIA Quintiq, called the *Dassault Systèmes DELMIA Quintiq Logistics Planner* (DSLP)[Dassault Systemes, 2023]. DSLP uses an LNS algorithm to solve these problems. It uses a proprietary solver as a repair step, which is expensive both in time and computing power. Using DSLP as a starting point enables us to make use of the existing routines necessary for LNS, while we replace the destroy operator with our proposed RL-based destroy operator. Unfortunately, due to a non-disclosure agreement, we are not able to share the code for the DSLP application. DSLP is the application that also served as the proof-of-concept for the neighborhood selection techniques that were studied in the previous chapter.

To showcase the versatility and applicability of our method, and to enhance reproducibility, we have therefore also applied our method to a second application: an LNS implementation for the classical Capacitated Vehicle Routing Problem with Time Windows (CVRPTW), like in the previous chapter. The algorithms and data sets that were used for the experiments on the CVRPTW application are publicly available from the following repository:

`https://github.com/w-feijen/ML4NC` .

7.1.1 Our Contributions

1. We define context, mask, and a stopping criterion to adapt the graph attention encoder-decoder model by Kool et al. [2019] to a smart neighborhood creation (SNC) model. Given a set of routes to choose from and a given starting route, this neighborhood creation model produces a probability distribution over possible neighborhoods that can be used in LNS to sample

a neighborhood. This neighborhood creation model is a robust method of creating high-quality neighborhoods that can subsequently be destroyed in LNS.

2. We enhance two LNS implementations with our novel Smart Neighborhood Creation: the DSLP algorithm that solves real-world instances and an LNS implementation for a set of classical CVRPTW instances. In both applications, the high-quality neighborhoods produced by SNC help the LNS algorithms to find more and better improvements. The neighborhood creation model is aided by a novel method to create a pre-selection of routes, before giving the options to the RL model.

3. We propose a framework on how to simultaneously optimize the Vehicle Routing Problem (VRP) and train the SNC with RL. Since the SNC model influences the optimization of the VRP, standard learning methods for RL do not suffice. We investigate two subroutines to enhance our framework: *experience replay* to recycle samples from previous iterations and *continuous learning* to deal with learning over multiple optimization runs.

4. We define three test phases in which we show our empirical results. In the first phase, the neighborhood creation problem instance, i.e., the routes to choose from and the starting route, remains equal. The second test phase allows more variability in neighborhood creation problem instances by allowing different starting routes and the third test phase is the actual optimization problem. In the first two phases for the DSLP application, our RL model finds higher improvements (3-4 times) and more frequent improvements (8-25% more) than the state-of-the-art optimizer used in daily practice. In the third test phase, we show that our model requires around 9% fewer iterations to reach the same objective value as the random benchmark. We verify these results on the CVRPTW application, where we also obtain results that clearly show that the SNC learns how to create high-quality neighborhoods.

7.1.2 Related Work

There are several papers in which RL or ML is used in an LNS algorithm. In Li et al. [2021], also briefly mentioned in the related work of the previous chapter, RL is used in the destroy step of an LNS algorithm to solve the VRP. They create a neighborhood for each route based on a k-nearest neighbors procedure. This means the creation of neighborhoods is a deterministic routine only based on the geographic locations of shipments in a route. RL is used after that to select one of these created neighborhoods. We expect more of our RL algorithm in two ways: instead of only choosing one neighborhood out of a set of created options, we want RL to iteratively construct a neighborhood. Next to that, since we need to

deal with time windows and other practical constraints, we cannot simply create neighborhoods solely based on geographical locations.

Three other examples of LNS algorithms that use RL are given in Kerscher and Minner [2024], **?**. In Kerscher and Minner [2024], customers are clustered using ML to decompose a VRP, after which the decomposed sub-VRPs are solved independently. In Wu et al. [2022], also mentioned in the related work of the previous chapter, RL decides which pair of nodes to improve with a pairwise improvement operator for the Traveling Salesperson Problem (TSP) and VRP. In the latter, RL is not used in the destroy step, but instead, it is used to decide how a destroyed solution can be quickly repaired to solve the capacitated VRP.

Adaptive Large Neighborhood Search is another method on which much research has been conducted. It is an extension of LNS that adaptively decides which subroutine to use based on weights [Ropke and Pisinger, 2006]. For extensive lists of other combinations of ML and combinatorial algorithms for the VRP, we refer to the surveys of Bai et al. [2023] and the section on hybrid method in Bogyrbayeva et al. [2022]. Lastly, we refer to Santini et al. [2023], which studies multiple decomposition techniques for VRPs and concludes that route-based decompositions work best.

7.1.3 Organization of Chapter

In the next section, we elaborate more on the original graph attention model that forms the basis of the RL models in SNC. In Section 7.3, we give the SNC algorithm, elaborate more on the underlying adapted graph attention model, and explain how to train it with reinforcement learning. Section 7.4 gives the results for the DSLP application and Section 7.5 gives the results for the CVRPTW application.

7.2 Preliminaries

We elaborate more on the graph attention model in Kool et al. [2019], since it is used as a basis for the RL model proposed in this chapter. In Kool et al. [2019], a graph attention model is used to construct solutions for routing problems, including TSP. The model is trained with REINFORCE and a greedy baseline [Williams, 1992]. For TSP, the problem instance I^{TSP} is defined as a graph with n nodes, where node $i \in [n]$ is represented by features $\mathbf{x}_i$. A solution s^{TSP} is defined as a permutation of the nodes: $(\pi_1, \ldots, \pi_n)$. An attention-based encoder-decoder model defines a stochastic policy $p(s^{\text{TSP}}|I^{\text{TSP}})$ for selecting solution s^{TSP} given instance I^{TSP}. This policy is parameterized by θ and is a factorization of

intermediate probabilities:

$$p_\theta(s^{\text{TSP}}|I^{\text{TSP}}) = \prod_{t=1}^{n} p_\theta(\pi_t|I^{\text{TSP}}, \pi_1, \ldots, \pi_{t-1}).$$

The encoder follows the transformer architecture by Vaswani et al. [2017] and produces embeddings $\mathbf{h}_i \in \mathbb{R}^{d_h}$ for all nodes in the graph. It takes the features of all nodes, $\mathbf{x}_1, \ldots, \mathbf{x}_n$, as input. For TSP these features are the coordinates of the nodes in the grid. The embedding dimension is set to $d_h = 128$. The decoder takes these node embeddings as input, together with a problem-specific mask and context, and produces s^{TSP} incrementally. The mask ensures that nodes are chosen only once. The context node contains information about the currently constructed solution necessary for making the next decision. At time step $t \in [n]$ the decoder outputs π_t, based on $\pi_1, \ldots \pi_{t-1}$, the embeddings from the encoder $\mathbf{h}_1, \ldots, \mathbf{h}_n$, the mask and the context node $\mathbf{h}_c$. For the TSP problem, the context $\mathbf{h}_c$ at time step t consists of the embedding of the first and last node in the current sub-tour, $\mathbf{h}_{\pi_1}$ and $\mathbf{h}_{\pi_{t-1}}$ and the *graph embedding* $\bar{\mathbf{h}}$. The graph embedding is computed by taking the average over all node embeddings. So the context node is as follows:

$$\mathbf{h}_c = \left[\bar{\mathbf{h}}, \mathbf{h}_{\pi_{t-1}}, \mathbf{h}_{\pi_1}\right], \quad \text{where } \bar{\mathbf{h}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{h}_i.$$

7.3 Method

7.3.1 LNS with Smart Neighborhood Creation

As described in Section 2.2 in the preliminaries of this thesis, and also Section 6.2 of the previous chapter, the main components of the LNS algorithm are the destroy and repair methods. The destroy method first creates a so-called *neighborhood*, defining which part of the solution may be improved in this iteration, after which this neighborhood is actually destroyed. In this chapter, we propose a new Smart Neighborhood Creation (SNC) method based on Reinforcement Learning. Our algorithm iteratively improves itself during optimization of the routing solution.

The pseudocode for our LNS algorithm with SNC is given in Algorithm 17. The neighborhood creation method we propose is novel, but it functions within an existing implementation of LNS. The existing routines are marked with an asterisk (*). Like the standard LNS algorithm given in Algorithm 1, LNS-SNC starts with the creation of a feasible solution, s. For the application in this chapter, a feasible solution is defined by an ordered set of shipments, each of which we call a *route*. Afterwards, s is iteratively improved by destroying and repairing a part of the

solution. This part of the solution is what we call a *neighborhood*, denoted by η, and is defined by a set of routes in s.

Since it is expensive to repair a neighborhood, it is important to consider neighborhoods that have the potential to yield large improvements. Therefore, we propose to use RL to create smart neighborhoods. The creation of a neighborhood is given in Lines 3-6 in Algorithm 17, with the crucial step in Line 6. Our RL model, on which we elaborate more later, outputs a probability over possible neighborhoods: $p_\theta(\eta|I^{NC})$. This probability distribution is used to sample a neighborhood, given a neighborhood creation problem instance I^{NC}. The problem instance I^{NC} is defined by an anchor route, r_1, and a set of routes from which the RL model can choose, $\mathcal{R}$. The creation starts with selecting this anchor route, around which the neighborhood is built. Before letting RL decide which routes to add to r_1 to create a neighborhood, we make a quick pre-selection of routes which we call the set of recommendations, $\mathcal{R}$. We perform this pre-selection in order to decrease the number of routes to choose from, which will speed up the training. Intuitively, the pre-selection discards routes that do not have proximity to the anchor roue, either in time or in location. We elaborate more on the computation of $\mathcal{R}$ in Section 7.3.2.

After creating the neighborhood with our proposed RL method, we continue with the routines that belong to the existing LNS implementation. This means we destroy and repair the created neighborhood, hopefully improving the solution s. After the repair step, the quantity $\bar{c}(\eta) = c(s) - c(s^{temp})$ can be computed, which indicates how much improvement the destroy and repair of η yielded. It can be considered as the quality of the neighborhood. Therefore we train the RL model with this information, which we explain more in Section 7.3.3. Finally, we update the solution if it improved, and we continue the algorithm until a time limit is reached. Other examples of stop criteria are an iteration limit or an objective value limit.

7.3.2 Recommendations

Since we want to learn fast what routes to add to the neighborhood, we make a pre-selection of routes from which the RL model can choose. We do this by discarding routes that are very different from the anchor route, either in geographical location or in time window. Intuitively, more improvements can be found if the routes in the neighborhood have high proximity, either in location or time. The pseudocode for creating the set of recommendations is given in Algorithm 18.

We elaborate on Algorithm 18 in more detail. First, we sort the routes geographically by partitioning them into shells around the anchor route. The anchor route is in the 0'th shell, the closest routes are in shell 1, et cetera. The width of a shell w is based on the anchor route and is calculated by dividing the distance of the anchor route, $d(r_1)$, by the number of shipments in the anchor route r_1. The distance between the anchor route and another route is approximated

Algorithm 17 LNS-SNC

$s = \textsc{FeasibleSolution}^*$ ▷ *Start with a feasible solution*
repeat
  $r_1 = \textsc{SelectAnchor}^*(x)$
  $\mathcal{R} = \textsc{Recommendations}(s, r_1)$ ▷ *Make pre-selection of routes*
  $I^{\text{NC}} = (\mathcal{R}, r_1)$ ▷ *Define problem instance for neighborhood creation*
  sample $\eta \sim p_\theta(\cdot|I^{\text{NC}})$ ▷ *Use RL to sample a n'hood from the recommendations*
  $s^{temp} = \textsc{Repair}^*(\textsc{Destroy}^*(s, \eta))$ ▷ *Improve by destroying and repairing η*
  $\bar{c}(\eta) = c(s) - c(s^{temp})$ ▷ *Compute the improvement of the neighborhood*
  $\theta = \textsc{Train}(\eta, \bar{c}(\eta), \theta)$ ▷ *Update the RL model with new improvement*
  if $c(s^{temp}) < c(s)$ **then** $s = s^{temp}$ ▷ *Update the solution if it improved*
until StoppingCriterion is met
return s

by sampling k_2 shipments from both routes, after which we take the minimum distance of all $(k_2)^2$ pairs of shipments.

Second, we want to penalize routes that serve customers with very different time windows than customers served in the anchor route. Therefore, we increase the shell number of routes without any time window overlap with the anchor route with probability δ. The size of the penalty, $\textsc{MaxShellNumber}(s, r_1)$, ensures that any route without time window overlap is chosen after all routes with time window overlap.

After partitioning all routes into shells based on their location and time window, we create the set of recommendations $\mathcal{R}$. Iterating from the first to the last shell, we add all routes from the shell to the $\mathcal{R}$ until $|\mathcal{R}|$ exceeds a predetermined threshold k_1.

7.3.3 Graph Attention Model

The model that we use to sample neighborhoods in Algorithm 17 is a graph attention model based on Kool et al. [2019]. We use RL to train this model. As explained in Section 7.3.1, the SNC problem instance I^{NC} is defined by the anchor route r_1 and the recommendations $\mathcal{R}$. In order to use the model by Kool et al. [2019], we need to define problem-specific input, mask, decoder context and a stopping criterion, as explained in Section 7.1.

- The *input* to our model are the feature vectors $\mathbf{x}_1, \ldots \mathbf{x}_i, \ldots$ that each describe one of the routes that can be added to the neighborhood. These features can be application-specific, but in general, they need to describe routes and their potential for improvement. The distance of a route, the

Algorithm 18 RECOMMENDATIONS(s, r_1)

input: minimum number of recommendations k_1, number of shipments per sample k_2, time window penalty probability δ
$\mathcal{S}$= SAMPLEUNIFORM(r_1, k_2) ▷ *Sample shipments from anchor route*
$w = \frac{d(r_1)}{|r_1|}$ ▷ *Shell width w is average distance for anchor route customers*
for $r_i \in s$ **do**
    $\mathcal{S}_o$= SAMPLEUNIFORM(r_i, k_2) ▷ *Sample shipments from other route*
    shellNumber$_i$ = $\lceil$MINDISTANCE($\mathcal{S}, \mathcal{S}_o$)$/w\rceil$
    if NOOVERLAP(r_i, r_1) and RAND() $< \delta$ **then**
        shellNumber$_i$ = shellNumber$_i$ + MAXSHELLNUMBER(s, r_1)
$\mathcal{R} = \emptyset$
for $j = 0, \ldots, 2 \cdot$ MAXSHELLNUMBER(s, r_1) **do**
    $\mathcal{R} = \mathcal{R} \cup \{r_i \in s | \text{shellNumber}_i = j\}$
    if $|\mathcal{R}| \geq k_1$ **then**
        break
return $\mathcal{R}$

distance between shipments in a route, and capacity utilization are typical features that can be used. Examples of features for the application to CVRPTW are given in Section 7.5.1 and Appendix A.2.

- The *mask* is used to ensure that chosen routes will not be chosen again. For our problem, the mask is the same as for TSP, since we exactly need to cancel out those routes that are already chosen in our neighborhood.

- The *context* node contains information about the currently constructed neighborhood which is necessary for making the next decision. The context for our problem, $\mathbf{h}_c$, consists of two elements: the *graph embedding*, $\bar{\mathbf{h}}$, and the *current neighborhood embedding*, $\widetilde{\mathbf{h}}$. The graph embedding $\bar{\mathbf{h}}$ is the same as for TSP as explained in Section 7.2. The current neighborhood embedding $\widetilde{\mathbf{h}}$ contains cumulative information of all the routes chosen so far, and is computed by taking the sum over the embeddings of the routes that are chosen during decoding in previous time steps. Formally,

$$\mathbf{h}_c := \left[\bar{\mathbf{h}}, \widetilde{\mathbf{h}}\right], \quad \text{where} \quad \bar{\mathbf{h}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{h}_i \quad \text{and} \quad \widetilde{\mathbf{h}} = \sum_{i=1}^{t-1} \mathbf{h}_{r_i}.$$

- As opposed to TSP, we want to stop adding routes before a full permutation is constructed. Therefore we also need to define the *stopping criterion*, i.e., when the model is finished with adding routes to the neighborhood η. This can for example be based on a minimum number of shipments or routes in the neighborhood.

7.3.4 Reinforcement Learning

As explained in the previous section, we use RL to train the proposed graph attention model. To train the model, we update the model with the quality of the neighborhood, $\bar{c}(\eta)$ The train call is given in Algorithm 19 and takes both η and $\bar{c}(\eta)$ as input. We elaborate on the algorithm in more detail.

Even though a train call happens in each iteration, we only update the RL model once every k_3 iterations. Otherwise, we just save the neighborhood and corresponding improvement for a later update in the memory of new neighborhoods M^{new}. We do not directly use the improvement to train our RL model, because we might value two neighborhoods with the same improvement differently. For example, we want to differentiate between a neighborhood with a small improvement in the beginning of the optimization run, and a neighborhood with an equally small improvement in the end. The first is bad, since we expect to find many good improvements in the beginning, while the latter is good, since it is harder to find any improvements, even of small size, during the end. Therefore we value a neighborhood by comparing it to a baseline B. The baseline B is defined by an exponential moving average over the improvement of recent iterations, parameterized by α. The RL model is trained with the negative improvement, relative to this baseline, denoted by $\textsc{Cost}(\eta, B)$. This cost is calculated once for each new neighborhood, and is defined as follows:[1]

$$\textsc{Cost}(\eta, B) = \begin{cases} -\frac{\bar{c}(\eta)+1}{B+1} & \text{if } \bar{c}(\eta) > B \\ 0 & \text{else.} \end{cases}$$

We have found better results when we focus only on neighborhoods that had a larger improvement than the baseline. We define the loss of the RL model as the expectation of the costs:

$$\mathcal{L}(\theta | I^{NC}) = \mathbb{E}_{p_\theta(\eta | I^{NC})} \left[\textsc{cost}(\eta) \right].$$

We optimize $\mathcal{L}$ by using gradient descent and the REINFORCE gradient estimator [Williams, 1992].

There are two optional procedures during the train call. The first option is continuous learning (ContLearn), which we explain in Section 7.3.5. If continuous learning is enabled, we store neighborhoods for later learning. The second option is to use experience replay (ExpReplay), which enables to learn from all the previous samples, which get stored in M, as opposed to only learning from the recent samples in M^{new}. The memory M has a maximum size, such that older neighborhoods are removed to make place for newer ones if necessary.

[1] We added $+1$ both in nominator and denominator of the fraction to ensure the cost will not blow up when the baseline happens to be close to zero.

Algorithm 19 TRAIN$(\eta, \bar{c}(\eta), \theta)$

input: baseline parameter α, recent sample memory M^{new}, experience replay memory M, train interval k_3 and experience replay sample size k_4.
Update M^{new} with $(\eta, \bar{c}(\eta))$
if Experience Replay **then** Update M with $(\eta, \bar{c}(\eta))$
if $|M^{\text{new}}| \geq k_3$ **then**
    $B = \alpha \cdot \left(\frac{1}{|M^{\text{new}}|} \sum_{\eta \in M^{\text{new}}} \bar{c}(\eta) \right) + (1 - \alpha) \cdot B$ ▷ *Update baseline*
    Compute COST(η, B) **for** $\eta \in M^{\text{new}}$ ▷ *Compute and store cost of n'hood*
    if ContLearn **then** STORESAMPLES(M^{new}) ▷ *Store for later training*
    if ExpReplay **then** $M^{\text{train}} :=$ SAMPLEUNIFORM$(\{\eta \in M | \bar{c}(\eta) > B\}, k_4)$
    else $M^{\text{train}} = M^{\text{new}}$
    $\nabla \mathcal{L} = \sum_{\eta \in M^{\text{train}}} \text{COST}(\eta, B) \nabla_\theta \log p_\theta(\eta)$ ▷ *Use stored cost to compute* $\nabla \mathcal{L}$
    $\theta = \text{Adam}(\theta, \nabla \mathcal{L})$
    $M^{\text{new}} = \emptyset$
return θ

7.3.5 Continuous Learning

If we start learning from scratch every time we start a new optimization run, it is very hard to perform well, since all learned knowledge must be gained during the current run. Instead, we propose to expose the model to the same instance multiple times after each other, say n times. In each next run, we start with the RL model that was trained in the previous runs.

We propose two methods for implementing this *continuous learning*, which are schematically shown in Figure 7.1. The first method is *naive continuous learning*. In naive continuous learning, we start the algorithm the $(i+1)$'th time with the model as it was at the end of run i. However, if we start the next run with the model that has just been adapted to the end of the previous optimization run, we risk the model being tailored to the end of the run end too much. Therefore, we introduce the *complete run update (continuous learning)*. In the complete run update, when we start run $i+1$, we take the model as it was at the beginning of run i and before starting run $i+1$, we train the model with all the (shuffled) data collected during run i. In this way, the model is less prone to being tailored to the previous run's end. Instead, the model should be able to capture the course of the whole previous optimization run.

7.4 Application to DSLP

The first existing LNS algorithm that we applied our SNC to, DSLP, is an application built with the DELMIA Quintiq system. It is used daily to solve many last-mile routing problems for large logistics clients [Dassault Systemes, 2023]. We

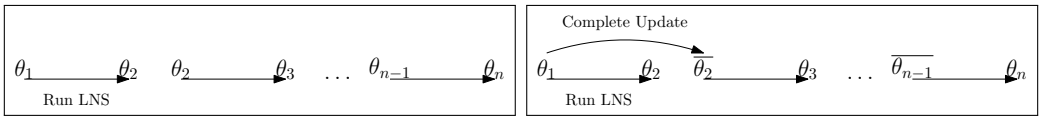


Figure 7.1: Two types of continuous learning. Left: naive continuous learning where the previous model is taken as the starting input for the next optimization run. Right: Complete model update continuous learning where we start with the model from the start of the previous run, updated with all the samples from the previous run.

describe the routing problems that DSLP solves below, after which we explain how we implemented SNC. We conclude this section with the results of using SNC in DSLP. Unfortunately, for this application, we are unable to share the detailed workings of the subroutines denoted with an asterisk (*) in Algorithm 17.

7.4.1 Problem Description

The problem we consider is based on a real-world last mile routing problem, and can be regarded a PDPTW, with a heterogeneous fleet and extra compatibility constraints [Dumas et al., 1991]. The input of the problem consists of a set of shipments that need to be picked up and delivered, and a set of vehicles that are used to perform these transportations. A *solution* to the problem, denoted by s, is a partition of these shipments into routes. A *route* is defined as an ordered set of pickups and deliveries, in which the delivery of a shipment must occur sometime after the pick up in that route. A *shipment* is defined by its pickup and delivery location, pickup and delivery time window and its capacity usage. A *vehicle* is defined by its capacity, its maximum operating duration and the following costs: one-time usage costs, mileage costs and an hourly cost for driving, waiting, working (i.e., all jobs which are not driving or waiting) and working overtime. Furthermore, compatibility matrices are given between shipments, indicating if shipments can go on any vehicle together, and between shipments and vehicles, indicating if shipments can go on a particular vehicle. The goal of the problem is to minimize the so-called *virtual costs* of a solution s, denoted by $c(s)$. The virtual costs of a solution consist of two components: the actual costs and the penalty costs. The actual costs are built up of the vehicles' hourly, mileage and usage costs. Penalty costs are incurred if any of the problem's soft constraints are violated. The only hard constraint in the problem is the feasibility of routes, meaning that the delivery of a shipment must occur sometime after the pick up in a route. All the other constraints in the problem, as stated below, are soft and dealt with using varying penalty parameters.

- All the shipments must be planned.
- The total load of shipments on a vehicle cannot exceed the vehicle capacity.

- A shipment must be picked up and delivered within its time window.
- The time of a route cannot exceed the maximum operating duration.
- The compatibilities between shipments (and vehicles) must be adhered to.

The instance used for the DSLP application is called 'Retail_u20210412', but cannot be shared in detail because of the earlier mentioned non-disclosure agreement.

7.4.2 SNC for DSLP

Algorithm 17 shows how we can solve the DSLP problem with smart neighborhood creation. An existing, undisclosed, DSLP routine creates a feasible solution and selects an anchor. As input of the RL model, we use an extensive list of features that describe routes and their potential for improvement. The features for a route are for example based on the actual costs and penalty costs for that route, distances between its shipments, and its capacity utilization. There are also shipment features, which are aggregated into route features, like location and time window. The repair step consists of a proprietary solver that is used daily in many DSLP applications and was formerly used to set several VRP world records [Sintef, 2023]. We stop adding routes to the neighborhood if both the number of shipments and the number of chosen routes exceed a certain threshold, which is obtained from the existing DSLP implementation.

7.4.3 Benchmarks

We benchmark our algorithm against two algorithms: a random algorithm and the existing DSLP neighborhood creation [Dassault Systemes, 2023], (the version used is 2022 Refresh1). The DSLP algorithm is the state-of-the-art and used in practice, in which handcrafted heuristics create smart neighborhoods that yield much improvement. Years of time and effort have gone into improving these heuristics, and we expect it is going to be hard to beat this tailor-made model. Next to the DSLP, we compare to an algorithm with random neighborhood creation. Like our algorithm, the random algorithm uses the recommendations to make a pre-selection of routes. However, instead of using RL to choose what routes to add to the neighborhood, the random algorithm just chooses one of the recommendations uniformly at random. In order to make a fair comparison with the DSLP and random algorithm, it is crucial that our SNC algorithm creates neighborhoods of similar sizes. This ensures that an improvement brought by the SNC algorithm can not be caused by an increased size of created neighborhoods. Therefore, all algorithms use the same stopping criterion in Algorithm 17.

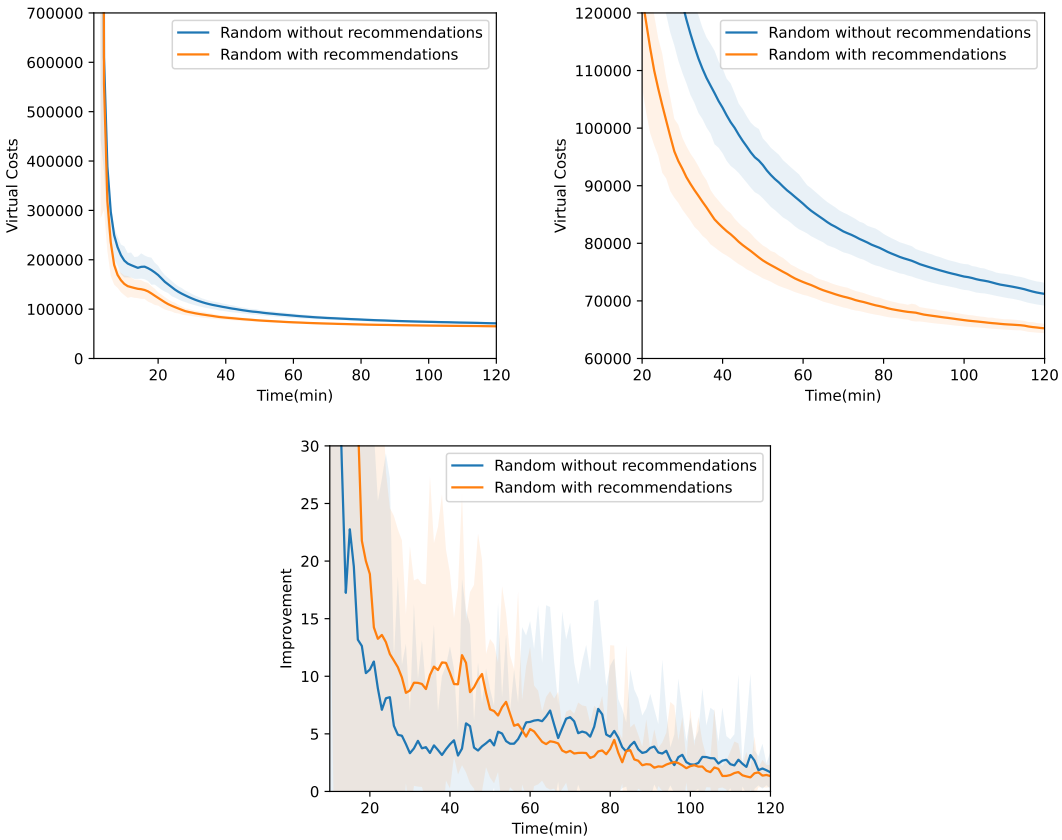


Figure 7.2: Top left: Virtual cost for random algorithm and random algorithm with recommendations, lower is better. Top right: Zoomed in of left figure. Bottom: Improvement found per iteration in the created neighborhood.

7.4.4 Recommendations Test

As explained in Section 7.3.1, we make a pre-selection of routes to make it easier for the RL model to create good neighborhoods. We tested the pre-selection on the random algorithm. In Figure 7.2 we show the virtual cost and the size of the improvement for both the random model without recommendations and the random model with recommendations. Even though the selection of routes is random in both algorithms, the algorithm with recommendations finishes with 7.5% less virtual costs. We see in the bottom picture that this is mainly because the size of the improvements that are found is larger in the beginning. However, it seems that from approximately 2500 iterations onwards, the algorithm with recommendations finds smaller improvements. This is probably because the better neighborhoods before that iteration have improved the solution more, which makes it harder to find new improvements after this point. Altogether we can conclude that limiting what routes we can add to our neighborhood by only

considering the recommendations does not harm us. Furthermore, learning will become easier because we reduce the number of routes that we can choose from.

7.4.5 Test Phases

After testing the recommendations, we defined three test phases to test if our algorithm learns. The difference per test phase is caused by fixing some parts of the neighborhood creation problem instance I^{NC}, defined as $I^{\text{NC}} = (\mathcal{R}, r_1)$. In the first test phase, we test if the RL model is able to learn how to create a neighborhood if it is given the same input in each iteration. This means we fix both the anchor route, r_1, and the options to choose from, $\mathcal{R}$, and let the RL model create a neighborhood for the same problem instance I^{NC} in each iteration. Since we want to solve the same neighborhood creation problem in each iteration, we only save how much improvement the created neighborhood would give, but we never implement the improvement in this neighborhood in our current solution.

In the second test phase, we maintain the fixed route options $\mathcal{R}$, but we allow variation in the anchor route r_1. As in the first phase, we do not implement the found improvement. In the last test phase we solve the real-world optimization problem, which means we consider different iterations with different sets of $\mathcal{R}$ and also implement the found improvement.

In phase 1 and phase 2, we test if our model learns by considering the average improvement and the fraction of iterations in which an improving neighborhood is found. In phase 3, we can no longer expect that the size of the improvement will remain equal over the optimization run. Therefore, it is not informative to consider the average improvement for this phase. Instead, we consider the decrease in virtual costs and compare this to the DSLP and random algorithm. All given results for DSLP are averages for 50 runs. The shaded areas in the plots show the confidence intervals.

7.4.6 Phase 1

In Figure 7.3 we show the average improvement that a neighborhood gives and the fraction of iterations in which a neighborhood is created that gives an improvement in phase 1. The figure shows that our model is able to beat the random model and the DSLP model, starting from the early iterations. After about an hour, our model is able to find an improvement in almost all iterations, and the average improvement found is 4 times larger than for the DSLP.

The DSLP and the random algorithm do not learn during optimization, and in phase 1, all models solve the exact same problem in each iteration. Therefore we do not expect to observe a change in the found improvement and it suffices to consider them for a shorter running time of 1 hour.

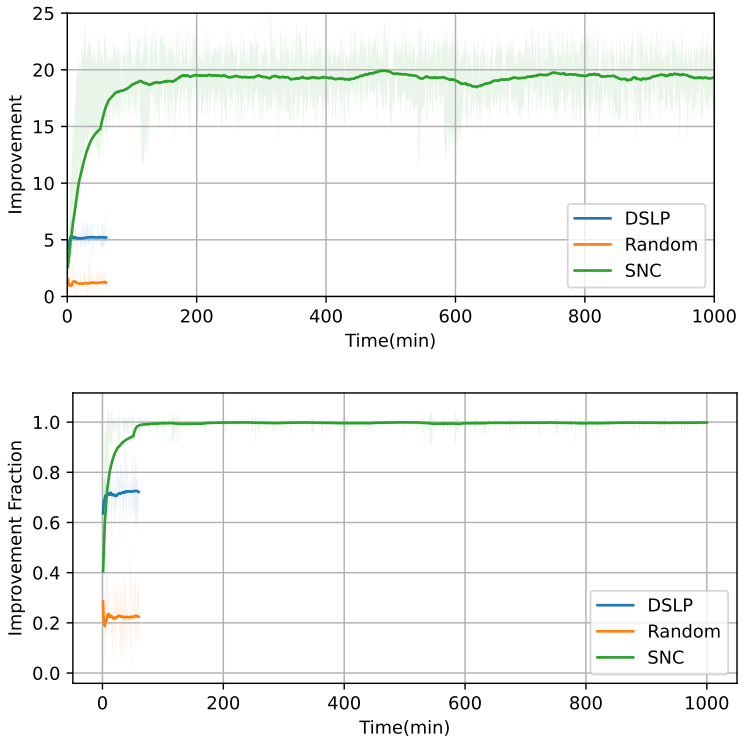


Figure 7.3: Size of the improvement and fraction of iterations in which an improvement is found for phase 1, for the DSLP, random, and SNC algorithm. In phase 1, both the iteration and the anchor route remain fixed.

7.4.7 Phase 2

Like in phase 1, we show in Figure 7.4 that in phase 2 we are able to beat the random and DSLP algorithm fairly quickly: both the average improvement and the fraction of improving neighborhoods quickly increase. If we compare the results to phase 1 we can clearly see the impact of varying the anchor route. Varying the anchor route causes the problem of creating a good neighborhood to change slightly per iteration. This makes it harder to find a neighborhood with an improvement, which can be seen in the fraction of improving neighborhoods, which is around 32% to the end of the algorithm, while it was at 100% in phase 1.

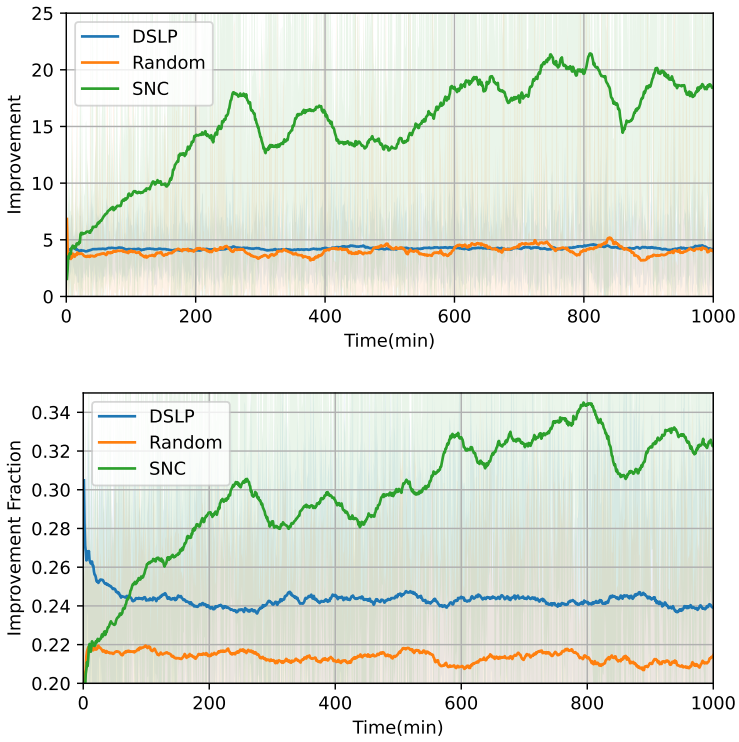


Figure 7.4: Size of the improvement and fraction of iterations in which an improvement is found for phase 2, for the DSLP, random and SNC algorithm. In phase 2, the iteration remains fixed, but the anchor route varies.

7.4.8 Phase 3

[WF16]

[WF16]: check what to do with this footnote, is it in right location?

In Figure 7.5 we show the virtual costs for several algorithms. Next to the DSLP and random algorithm, there is the SNC algorithm and the SNC algorithm with two versions of continuous learning, both with and without experience learning (see Section 7.3.1). Note that iterations are on the x-axis, instead of time. We think it suffices for now to consider virtual costs per iteration, since the expensive repair operator will be accountable for most of the running time after optimizing our code for efficiency.

The standard SNC model is not able to beat the random model. This is not very surprising, since the model needs to learn everything in a single optimization run. Therefore it is crucial in phase 3 to use the continuous learning methods. The naive continuous learning method, without experience replay, does not perform well. Like the model without continuous learning, it is not able to beat random, and it performs even worse. This is probably because the model is only updated with recent iterations, which causes the model to be tailored to these recent iterations. Usually this is good, but when we take the model at the end of an

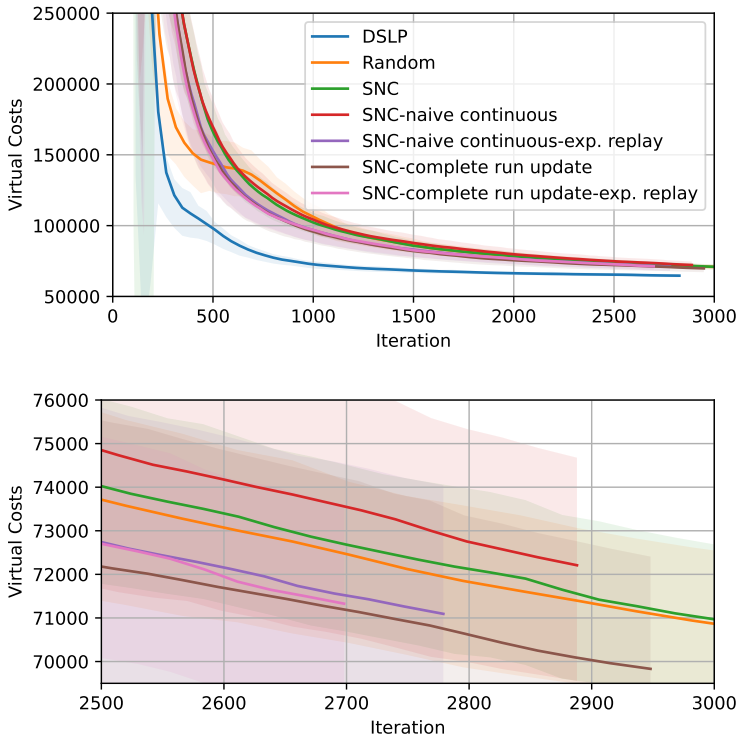


Figure 7.5: The virtual cost for phase 3, for the DSLP, random and multiple versions of the SNC algorithm, including different continuous learning and experience replay settings. The bottom picture is equal to the top picture but zoomed in on the end (legend remains the same). In phase 3 neither the iteration nor the anchor route is fixed, i.e., phase 3 reflects the real-world problem.[2]

optimization run, and reuse it at the start of the second run, it will not make the right choices at the start of this next run.

We tried to fix this in two different ways. The first method was to add experience replay, which assures that the model is less tailored to the recent iterations, but instead learns from less recent iterations. This model performed much better than continuous learning without experience replay and this model is able to beat the random model. This shows that naive continuous learning needs experience replay in order to work properly. Next to adding experience replay, we also tried the complete run update way of continuous learning. This model fixes the problem of naive continuous learning at the end and start of a run, but at the same time it allows the model to be tailored to recent iterations. This model, with the complete run update and without experience replay, therefore works best. Adding experience replay to this model does not help this model, and

[2]All optimization runs were executed for 120 minutes, which is why some algorithms performed more iterations than others.

even makes iterations slightly slower. The complete run update model without experience replay reached a virtual cost of 69768 after 2948 iterations, while the random model needed 3250 iterations to get to the same virtual cost, meaning that our model saves more than 9% of the iterations compared to the random model. Unfortunately, SNC does not beat the state-of-the-art DSLP model in the third test phase. DSLP is tailored to solving a specific problem structure, in which it excels. Our model on the other hand, has the great benefit of being more adaptive to changes in the problem structure, due to the reactive RL in the neighborhood creation.

7.5 Application to CVRPTW

The second application that we tested our SNC algorithm on is the Capacitated Vehicle Routing Problem with Time Windows (CVRPTW). In the CVRPTW, a given number of vehicles with a given capacity need to serve a set of customers with given locations and demand from a depot. The customers have time windows in which the service must happen. The goal of the problem is to find a solution that serves all customers, satisfies the capacity constraints, uses a given maximum number of vehicles, and minimizes the total traveled distance. For a more detailed explanation, we refer to Section 6.4 in the previous chapter. We tested our algorithm on the ten R1 Homberger-Gehring instances with 1000 customers [Homberger and Gehring, 2005].

7.5.1 SNC for CVRPTW

To test our SNC algorithm on the CVRPTW instances of Homberger-Gehring, we need an implementation of LNS that solves CVRPTW, in which we can embed our neighborhood creation, as explained in Algorithm 17. The implementation of LNS needs to define (1) a routine to find a feasible solution and (2) a repair method. Moreover, for the neighborhood creation, we need to define (3) how to select an anchor, (4) which features to use and (5) when to stop the neighborhood creation.

Since we also consider the CVRPTW application in the previous chapter, we can re-use several subroutines from the LNS algorithm explained there. Like in the previous chapter, we use VROOM in order to find a feasible solution (1) [Coupey et al., 2023]. In each iteration, an anchor route is selected (3) uniformly at random. Subsequently, an extensive list of features (4) that describe routes and their potential for improvement is fed to the neighborhood creation model. The features are similar to those of the previous chapter, but slightly different since they need to describe a route instead of a neighborhood. The features of a route are for example based on the route's distance, how far the route is from the anchor route, the distances between its shipments, and its capacity utilization.

There are also shipment features, which are aggregated into route features, which describe the characteristics of the shipments, like location and time window. The full list of features is given in Appendix A.2. The neighborhood creation stops (5) after adding ten routes, which ensures that each created neighborhood contains the same number of routes. After creating the neighborhood, the repair step (2) is executed by VROOM again. Using VROOM as a repair operator is also done in the LNS presented in the previous chapter. We can exploit that the sub-problem defined by the deleted set of routes is a CVRPTW in itself, albeit much smaller than the original. Therefore we can use VROOM's power on smaller instances and leverage the outcome to solve the large 1000 customer instances.

7.5.2 Benchmarks

As for the DSLP application, we compared our SNC algorithm with two other neighborhood creation methods. The first is the random route selection, also used as a benchmark in the DSLP application in Section 7.4.3. Moreover, we compared SNC to a heuristic neighborhood creation method. This is the neighborhood creation method presented in the previous chapter to create neighborhoods, Algorithm 15 in Section 7.4. In this algorithm, denoted with *Heuristic* in the rest of this chapter, routes are chosen based on a distance measure that also considers how well the time windows of routes match.

7.5.3 Results

In the next sections the highlights of the results on the CVRPTW instances are shown, the extensive results are given in Appendix B.2. All the results shown are averages of 10 optimization runs.

7.5.4 Recommendations Test

As for the DSLP application, we tested what effect the pre-selection has on the created neighborhoods, by testing the random neighborhood creation algorithm with and without the recommendations. Table 7.1 shows the average improvement over the 500 iterations for the test instances. From the table, we can conclude that using the recommendations and thus limiting the options that the neighborhood creation models can choose from, does not harm the optimization. In fact, we even benefit from using the recommendations, since the average improvement is higher for all test instances.

7.5.5 Phase 1

Figure 7.6 shows the improvement for the first test instance and Table 7.2 shows the average improvement at the end of the optimization run. Figure 7.6 shows

Table 7.1: Average improvement over 500 iterations for the random algorithm without and with recommendations, for ten test instances and averaged over the ten test instances.

R1_10_*	1	2	3	4	5	6	7	8	9	10	Avg.
No rec's	0.49	1.15	1.82	1.52	0.62	1.47	1.65	1.05	1.16	0.93	1.19
Rec's	0.85	1.48	2.02	1.71	1.01	1.84	2.08	1.30	1.75	1.48	1.55

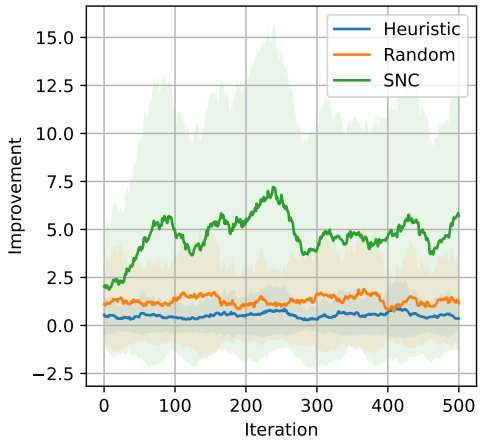


Figure 7.6: Improvement in phase 1 for instance R1_10_1 for the heuristic, random and SNC algorithm.

Table 7.2: Average improvement after 500 iterations in phase 1 for the heuristic, random and SNC algorithm.

Instance	Heuristic	Random	SNC
R1_10_1	0.37	1.15	5.70
R1_10_2	41.25	2.46	22.82
R1_10_3	1.57	3.48	12.78
R1_10_4	72.56	7.82	15.75
R1_10_5	1.66	0.93	20.58
R1_10_6	50.01	6.64	44.91
R1_10_7	3.26	4.66	21.84
R1_10_8	25.89	5.50	22.61
R1_10_9	0.00	1.40	9.27
R1_10_10	0.28	4.19	38.26
Average	19.69	3.82	21.45

that the size of the improvement increases for the SNC algorithm during the optimization, which shows that the model benefits from the learning. Clearly, the random and the heuristic algorithms do not show this behavior. In Table 7.2 we see that for phase 1, the heuristic algorithm performs really well for the second, fourth, sixth and eighth instance. For the other instances, the heuristic algorithm finds improvements that are almost always worse than the random algorithm. It seems that the heuristic algorithm either finds good improvements consistently or finds almost no improvements consistently. The SNC algorithm, however, performs much better than random for all instances. It does not beat the heuristic algorithm in the instances where the heuristic algorithm performs well, but overall, the average improvement shows that SNC algorithm learns how to create neighborhoods slightly better than the heuristic model. These results clearly show the potential of the SNC algorithm in phase 1.

7.5.6 Phase 2

To improve results for phase 2 (and also for phase 3), we have found it was necessary to start with a reinforcement learning model in the SNC algorithm that was already slightly informed about the task it was going to face. Therefore, before starting the first optimization run in which we use SNC, we trained the RL model using a pre-collected set of samples consisting of 10.000 samples per instance. The samples are collected before the SNC run, using both the heuristic neighborhood creation and the random neighborhood creation, resulting in heuristic samples and random samples. A heuristic sample consists of the route features from a certain iteration, the routes that the heuristic selected in that iteration, and the improvement that this choice yielded. A random sample is created similarly but with a random route selection instead. Now, before starting phase 2, we pre-trained the SNC model with all the heuristic samples in the pre-collected set for that instance which yielded an improvement. At the same time, we trained the model with just as many random samples. By feeding these samples before the optimization, we created an SNC model that is more informed, instead of having a model that needs to learn how to do its job from scratch at the start of the optimization.

Figure 7.7 shows the improvement for the first test instance and Table 7.3 shows the average improvement at the end of the optimization run for phase 2. The results show that both the heuristic algorithm and the SNC algorithm are able to outperform the random algorithm by far. The figures show that the SNC algorithm clearly benefits from having a pre-trained model, since iterations with sizes comparable to the heuristic are present from the start of the optimization. Compared to phase 1, we see less growth in the improvements of the SNC algorithm, it seems that less is learned during optimization than in phase 1. We can conclude for phase 2 that the SNC algorithm is able to be competitive with the heuristic algorithm but does not produce better solutions.

7.5.7 Phase 3

As for phase 2, we used an informed model at the start of the first optimization run by learning from heuristic and random samples that we collected beforehand. We used continuous learning with a complete run update and without experience replay, as this worked best for the DSLP application in phase 3. Figure 7.8 (left) shows how the total distance decreases during optimization for the first test instance and Table 7.4 shows the total distance of the solution after 500 iterations for phase 3. As opposed to the previous phases where the figures showed the improvement, here we show the total distance, and therefore a lower value means that the algorithms perform better. Phase 3 can be regarded as the hardest test phase, since the algorithms need to create neighborhoods in iterations where both the anchor route and the recommendations, $\mathcal{R}$, vary. The results reflect this: the

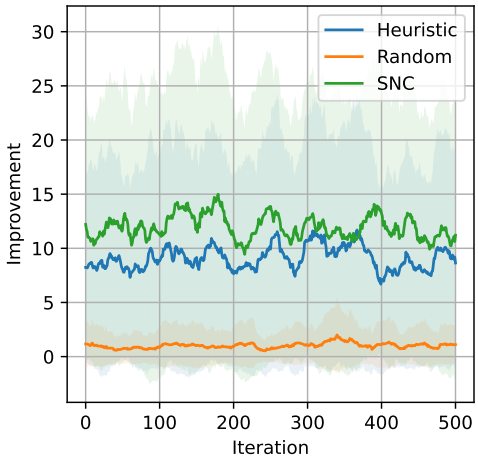


Figure 7.7: Improvement in phase 2 for instance R1_10_1 for the heuristic, random and SNC algorithm.

Table 7.3: Average improvement after 500 iterations in phase 2 for the heuristic, random and SNC algorithm.

Instance	Heuristic	Random	SNC
R1_10_1	8.64	1.11	11.21
R1_10_2	12.52	2.22	12.28
R1_10_3	35.58	6.66	33.70
R1_10_4	37.02	4.96	28.03
R1_10_5	17.31	1.81	13.37
R1_10_6	29.73	7.27	43.06
R1_10_7	39.20	5.48	31.25
R1_10_8	20.40	6.34	21.53
R1_10_9	19.65	3.65	26.68
R1_10_10	20.92	3.86	19.05
Average	24.10	4.34	24.01

quality of SNC compared to the heuristic model is worse in phase 3 than in the first two phases. The SNC algorithm still outperforms the random model slightly, but it is not competitive with the heuristic algorithm.

7.6 Conclusion and Discussion

We have enhanced both DSLP, a widely used tool to solve large vehicle routing problems, and an LNS implementation for CVRPTW with a destroy operator based on RL. We have found it is crucial to create neighborhoods that are able to yield large improvements. The first two test phases have shown that our RL model was able to learn how to create good neighborhoods, and performed better than or competitively with the state-of-the-art benchmark that we compared to. The results in test phase three also showed we can learn how to create good neighborhoods, and that it is good to have a model that is tailored to the most recent iterations. However, it is important to not simply use that model at the start of a new optimization run. Instead, one must follow the complete run update method for continuous learning. For both applications, we observed a drawback in the third test phase in the performance of our smart neighborhood creation. Phase 1 and 2 clearly show the potential of our SNC method, but it seems that the SNC model has not learned enough to face the challenges of phase 3. The running of the repair step imposes a serious limit on the amount of training data we can generate for our algorithm. It would be very interesting to see how SNC would behave in phase 3 if it was offered more training data.

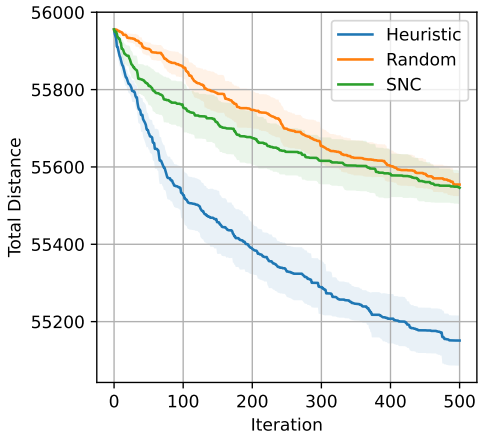


Figure 7.8: Total distance in phase 3 for instance R1_10_1 for the heuristic, random and SNC algorithm.

Table 7.4: Total distance after 500 iterations in phase 3 for the heuristic, random and SNC algorithm.

Instance	Heuristic	Random	SNC
R1_10_1	55151.08	55554.44	55546.16
R1_10_2	50204.44	50708.50	50840.71
R1_10_3	46343.85	46657.50	46850.69
R1_10_4	43699.00	43990.69	43973.72
R1_10_5	52468.82	53270.78	53092.27
R1_10_6	49092.80	49618.58	49479.06
R1_10_7	45619.97	45894.39	46049.32
R1_10_8	43514.53	43747.86	43718.19
R1_10_9	51151.49	51835.34	51687.12
R1_10_10	49095.27	49679.34	49614.42
Average	48634.13	49095.74	49085.16

Chapter 8

Final Remarks

This thesis aimed to explore the workings of algorithms for practical combinatorial optimization problems. We studied how we can improve algorithms in order to perform their job right, well, and fast. We took both a practical and a theoretical approach to addressing these questions.

We revise the main contributions presented in this thesis, following the three problem areas that were introduced in the introduction. For the SSMTSP, we implemented ML in an already well-working algorithm. Given a bounded additive error, we were able to show a lower bound on the number of queue operations saved. On random graph instances we demonstrated empirically that our method saves optimization time. We solved practical instances for the casting problem by using a disaggregated formulation for which we created high-quality variables. Subsequently, we were able to find an algorithm for the casting problem that finds solutions that are arbitrarily close to the optimal solution with constraint violations that are arbitrarily small. Lastly, for the CVRPTW instances and also the practical DSLP instances, we showed how we can use ML to improve the destroy operator, ultimately leading to a speed-up in optimization time.

Below, we highlight the significance of the contributions presented in this thesis. Following the relevant field of combining machine learning and combinatorial optimization, this thesis provides several variants for employing algorithms with learning techniques, following the second or third paradigm by Bengio et al. [2021]. The examples illustrate the hypothesis that we can accomplish a well-working interplay between ML and existing combinatorial algorithms. Secondly, we have shown that it pays off to analyze combinatorial problems further than determining the worst-case complexity. Even though we did not improve the worst-case complexity for the SSMTSP in Chapter 3, we were able to provably decrease the number of queue operations on the one hand and reduce the empirical optimization time on the other. For the casting problem, looking further than the worst-case complexity meant that we considered slightly infeasible solutions, which resulted in a polynomial-time bicriteria approximation scheme. Lastly,

the significance in our research lies in the frameworks and guidelines that we present for training machine learning models during optimization. In Chapter 6 we present guidelines for supervised learning, and in Chapter 7 we present options for when reinforcement learning is used during optimization.

Despite these contributions, our research also has limitations. We mention a few here and give recommendations for feature research to address these limitations. The experiments presented in Chapter 3 were conducted on synthetic random graphs only. It would be interesting to test if the methods have a similar impact on real-world road networks. Second, the empirical results for the casting problem are based on a method that relies on the structure of the instance. It is not possible to construct every feasible casting solution with the variables that are created for the disaggregated formulation. It would be interesting to see if the creation of the variables can be adapted such that the space of feasible solutions that can be constructed with them increases. Third, the PTBAS for the casting problem requires the construction of a large and involved graph. Moreover, the analysis of the algorithm contains many details. Future research would be necessary to check if it is possible to simplify the algorithm and the corresponding analysis. In the last two chapters, we have seen that it is hard to beat the existing state-of-the-art LNS algorithms for CVRPTW. Although we have been able to show the potential of LENS and SNC, future research would be necessary to come up with a method that is significantly stronger than the state-of-the-art in all test phases.

In conclusion, this research has provided novel, both practical and theoretical, insights into the study of algorithms for combinatorial optimization problems. We hope this work will inspire researchers to continue combining machine learning with combinatorial algorithms, to be curious about what is possible beyond worst-case complexity, and to be interested in theoretically proving why practical methods work.

Chapter A

Feature Lists

A.1 Feature List for Chapter 6

The first feature is the number of customers in the neighborhood. The following properties describe the customers in the neighborhood:

- Waiting time: The waiting time of a customer is zero if the arrival time is after the start of the time window, otherwise it is the positive difference between the arrival time and the start of the time window.

- Closeness: The closeness of customer c is the minimum of the distances between c and all customers in routes in the neighborhood that do not contain c.

- Temporal closeness: Defined exactly as the closeness, but defined with temporal distance instead of the Euclidean distance. Temporal distance is defined as the sum of the Euclidean distance and the time window difference. The time window difference indicates how compatible the time windows of two shipments are. The time window difference between two clients is the minimum waiting time caused by serving the clients directly after each other, in any order, if feasible. If the time windows make it infeasible to serve the clients after each other, the time window difference is set to a large penalty value.

- Centroid closeness: Given the distance between a customer c and the centroid of each route in the neighborhood that does not contain c, the centroid closeness is the minimum over these distances.

- Distance contribution: The difference between the length of the route in which the customer is planned and the length of the route if the customer would be removed from the route.

- Time window length: The length of the time window of the customer.
- Distance to the depot: The distance from the customer to the depot.
- Load: The demand of the customer.
- Minimum greedy addition cost: Given a customer c, for each route in the neighborhood that does not contain c, the greedy addition cost is how much the distance increases if the customer c is added to this route. The minimum greedy addition cost is the minimum of these increases over all the routes in the neighborhood that do not contain c. The greedy addition is calculated greedily, since only one possible location in the other route is tested: before the first customer that has a time window that starts later.
- Maximum gain: For a customer c, for each route r in the neighborhood that does not contain c, the gain is defined as the distance contribution of the customer minus the greedy addiction cost of adding c to r. The maximum gain is computed by taking the maximum of the gains over all the routes that do not contain customer c.
- Possible delay: The difference between the end of the time window and the current arrival time.

Moreover, we compute the following features which describe the routes in the neighborhood:

- Route distance: The distance of the route.
- Average route distance: The distance of the route, divided by the number of customers in the route.
- Empty distance: The distance between the last customer in the route and the depot.
- Worst case distance fraction: The distance of the route divided by the distance of a so-called worst-case solution. The worst-case solution is computed by traveling back to the depot after each customer in the route.
- Route duration: The travel time plus the waiting time plus the service time.
- Average route duration: The route duration divided by the number of customers in the route.
- Idle time: The total time the vehicle is not traveling nor servicing.
- Free capacity: The free capacity in the vehicle when leaving from the depot.

- Fitting candidates: The number of customers in the other routes in the neighborhood that have a demand that is smaller than the free capacity of the vehicle.
- Expected number fitting candidates: The free capacity of the vehicle divided by the average demand of the customers on the other routes in the neighborhood.

Lastly, we compute the following feature between each pair (r_i, r_j) of routes in the neighborhood:

- Distance between routes: The handcrafted distance measure $\tilde{d}(r_i, r_j)$, defined in Section 6.4.2.

A.2 Feature List for Chapter 7

The following features are the route features. All but the last route feature are also used in Chapter 6 and we refer to A.1 for the definition of the features.

- Distance.
- Average distance.
- Empty distance.
- Distance vs worst case fraction.
- Route duration.
- Average route duration.
- Free capacity.
- Approximate distance to anchor route.

The last feature is defined as the Euclidean distance between the centroid of the route and the centroid of the anchor route. The centroid of a route is defined as the element-wise mean of the location coordinates of the routes' customers.

The following are the features per customer. They are aggregated into route features by taking the maximum, minimum, average, standard deviation and sum over all the customers in a route. We refer to A.1 for the definition of the last two customer features.

- Time window start.
- Time window end.

- Time window length. Time window end minus time window start.
- Demand. Size of the load that this customer demands.
- Location x. x coordinate of this customer.
- Location y. y coordinate of this customer.
- Depot distance. Distance from this customer to the depot.
- Waiting time. Time between arrival at customer and time window start. Equals zero if arrival is after the start of the time window.
- Distance contribution.
- Possible delay.

Chapter B

Extended Results

B.1 Extended Results for Chapter 6

Note that in Table B.1 and Table B.2, the line with averages shows the GAP of the average values and not the average of the GAP values.

		Oracle		Random		ML1		ML3		ML5	
Instance	BKS	Avg	Gap	Avg	Gap	Avg	Gap	Avg	Gap	Avg	Gap
R1_10_1	53380.2	54911.8	0.00%	55388.8	100.00%	55512.3	125.89%	55421.2	106.79%	55431.8	109.01%
R1_10_2	48232.7	49798.1	0.00%	50541.5	100.00%	50420.8	83.77%	50468.7	90.22%	50491.1	93.22%
R1_10_3	44694.2	46007.4	0.00%	46623.0	100.00%	46634.7	101.90%	46553.7	88.75%	46518.7	83.06%
R1_10_4	42463.7	43529.9	0.00%	43827.8	100.00%	43824.0	98.75%	43798.6	90.20%	43737.1	69.55%
R1_10_5	50445.4	52166.3	0.00%	52729.2	100.00%	52765.2	106.40%	52705.1	95.71%	52696.7	94.23%
R1_10_6	46930.0	48834.9	0.00%	49335.0	100.00%	49361.6	105.32%	49255.3	84.06%	49324.4	97.88%
R1_10_7	43975.5	45357.2	0.00%	45797.3	100.00%	45770.6	93.94%	45763.9	92.40%	45724.7	83.50%
R1_10_8	42288.5	43245.9	0.00%	43591.1	100.00%	43582.6	97.53%	43531.4	82.70%	43576.2	95.66%
R1_10_9	49195.3	50935.6	0.00%	51571.6	100.00%	51372.7	68.72%	51344.0	64.22%	51390.6	71.54%
R1_10_10	47407.2	48852.1	0.00%	49396.4	100.00%	49357.5	92.86%	49372.8	95.66%	49311.2	84.35%
R1 Average	46901.3	48363.9	0.00%	48880.2	100.00%	48860.2	96.13%	48821.5	88.63%	48820.2	88.38%
R2_10_1	42182.6	42402.5	0.00%	42633.3	100.00%	42660.6	111.83%	42713.1	134.56%	42569.7	72.44%
R2_10_2	33411.2	34059.5	0.00%	34224.1	100.00%	34139.7	48.70%	34165.8	64.60%	34131.2	43.54%
R2_10_3	24916.9	26772.5	0.00%	26818.8	100.00%	26830.2	124.52%	26840.8	147.43%	26841.8	149.66%
R2_10_4	17852.0	20352.8	0.00%	20569.7	100.00%	20507.1	71.13%	20562.4	96.65%	20505.2	70.29%
R2_10_5	36216.1	38854.4	0.00%	38998.9	100.00%	38997.2	98.85%	38951.0	66.85%	38926.9	50.20%
R2_10_6	29978.0	32123.1	0.00%	32313.5	100.00%	32295.8	90.72%	32317.6	102.13%	32274.6	79.57%
R2_10_7	23219.6	25798.9	0.00%	25993.1	100.00%	26063.9	136.50%	26053.5	131.13%	26100.6	155.39%
R2_10_8	17442.3	19553.1	0.00%	20005.4	100.00%	19844.2	64.36%	19909.1	78.71%	19875.2	71.21%
R2_10_9	32995.7	35442.2	0.00%	35687.0	100.00%	35607.7	67.61%	35677.3	96.02%	35662.2	89.89%
R2_10_10	30207.5	32995.9	0.00%	33189.8	100.00%	33143.8	76.28%	33279.0	146.02%	33229.1	120.28%
R2 Average	28842.2	30835.5	0.00%	31043.4	100.00%	31009.0	83.45%	31047.0	101.73%	31011.7	84.75%

Table B.1: Total distance after 200 iterations of the oracle model, random model, ML1 model, ML3 model and ML5 model. Also the total distance of the best-known solution (BKS).

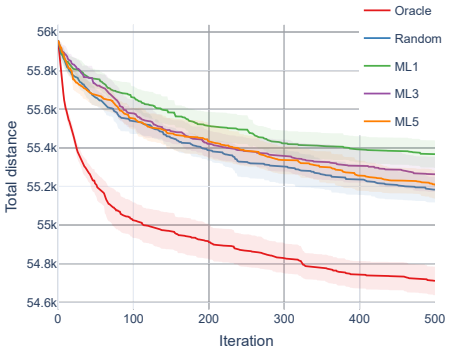
		Oracle		Random		ML1		ML3		ML5	
Instance	BKS	Avg	Gap	Avg	Gap	Avg	Gap	Avg	Gap	Avg	Gap
R1_10_1	53380.2	54711.2	0.00%	55183.8	100.00%	55367.5	138.86%	55263.2	116.80%	55206.9	104.88%
R1_10_2	48232.7	49589.3	0.00%	50203.6	100.00%	50172.5	94.94%	50172.0	94.85%	50221.0	102.84%
R1_10_3	44694.2	45881.8	0.00%	46320.1	100.00%	46467.1	133.55%	46341.3	104.85%	46305.6	96.71%
R1_10_4	42463.7	43426.6	0.00%	43668.2	100.00%	43725.2	123.59%	43676.8	103.55%	43648.6	91.87%
R1_10_5	50445.4	52007.1	0.00%	52541.1	100.00%	52496.4	91.62%	52432.3	79.62%	52513.9	94.90%
R1_10_6	46930.0	48682.6	0.00%	49126.6	100.00%	49191.3	114.57%	49139.6	102.93%	49181.0	112.26%
R1_10_7	43975.5	45238.5	0.00%	45589.6	100.00%	45628.4	111.07%	45608.2	105.30%	45537.9	85.29%
R1_10_8	42288.5	43165.2	0.00%	43450.1	100.00%	43485.5	112.44%	43403.8	83.77%	43388.8	78.47%
R1_10_9	49195.3	50743.9	0.00%	51187.7	100.00%	51098.2	79.82%	51057.0	70.54%	51133.3	87.75%
R1_10_10	47407.2	48724.0	0.00%	49139.5	100.00%	49139.4	99.98%	49175.8	108.74%	49149.4	102.40%
R1 Average	46901.3	48217.0	0.00%	48641.0	100.00%	48677.1	108.51%	48627.0	96.70%	48628.6	97.08%
R2_10_1	42182.6	42200.6	0.00%	42460.4	100.00%	42520.4	123.12%	42584.1	147.63%	42445.1	94.11%
R2_10_2	33411.2	34052.8	0.00%	34127.8	100.00%	34100.3	63.27%	34104.0	68.28%	34094.3	55.34%
R2_10_3	24916.9	26760.5	0.00%	26776.4	100.00%	26803.8	272.04%	26815.9	348.01%	26816.3	350.84%
R2_10_4	17852.0	20333.6	0.00%	20404.9	100.00%	20388.2	76.60%	20454.7	169.84%	20367.1	46.91%
R2_10_5	36216.1	38854.4	0.00%	38941.9	100.00%	38926.6	82.55%	38901.7	54.05%	38914.2	68.37%
R2_10_6	29978.0	32120.4	0.00%	32187.0	100.00%	32234.5	171.34%	32233.3	169.46%	32258.3	206.99%
R2_10_7	23219.6	25791.4	0.00%	25869.0	100.00%	26002.8	272.42%	25944.8	197.71%	26008.5	279.78%
R2_10_8	17442.3	19546.7	0.00%	19816.3	100.00%	19646.4	36.99%	19714.4	62.18%	19718.8	63.83%
R2_10_9	32995.7	35437.2	0.00%	35532.6	100.00%	35460.2	24.04%	35552.9	121.27%	35465.5	29.59%
R2_10_10	30207.5	32943.2	0.00%	33006.8	100.00%	33014.9	112.74%	33059.5	182.95%	33008.6	102.85%
R2 Average	28842.2	30804.1	0.00%	30912.3	100.00%	30909.8	97.69%	30936.5	122.37%	30909.7	97.60%

Table B.2: Total distance after 500 iterations of the oracle model, random model, ML1 model, ML3 model and ML5 model. Also the total distance of the best-known solution (BKS).

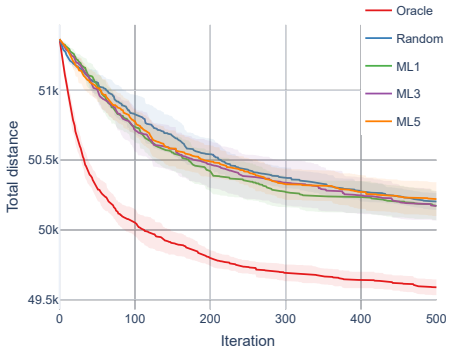
B.2 Extended Results for Chapter 7

[WF17]: Mention recommendations

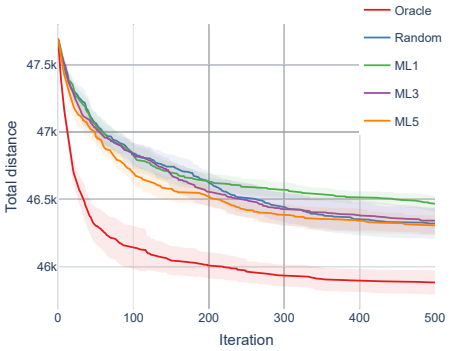
[WF17] Figure B.4, Figure B.5 and Figure B.6 show the average results for the three test phases for the CVRPTW application.



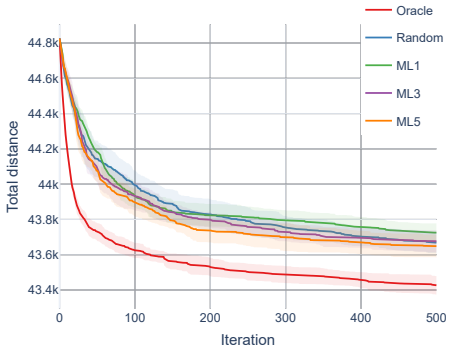
(a) Total Distance for instance R1_10_1



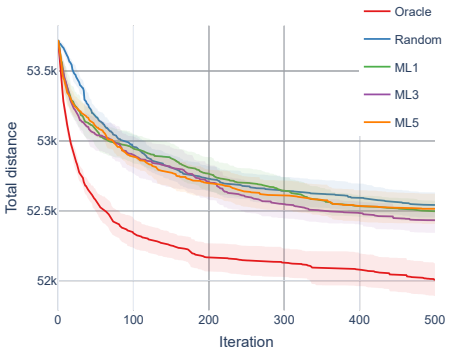
(b) Total Distance for instance R1_10_2



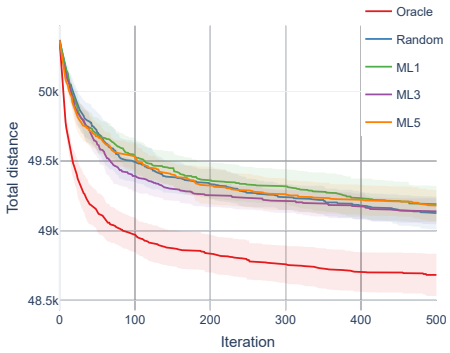
(c) Total Distance for instance R1_10_3



(d) Total Distance for instance R1_10_4

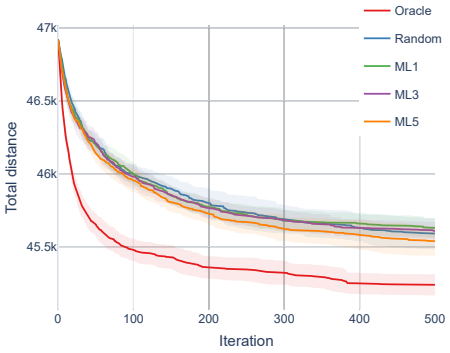


(e) Total Distance for instance R1_10_5

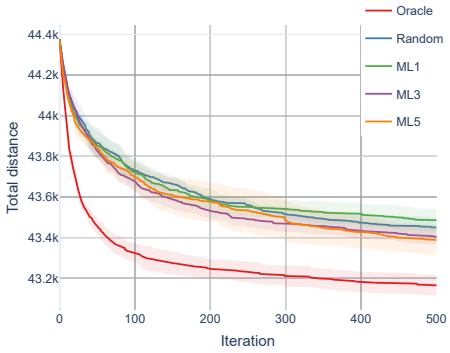


(f) Total Distance for instance R1_10_6

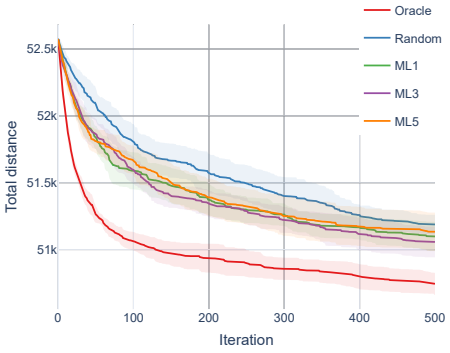
Figure B.1: Total distance for 10 R1 test instances for the oracle model, the random model, ML1 model, ML3 model and ML5 model



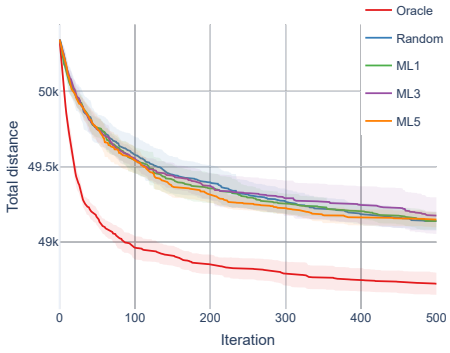
(g) Total Distance for instance R1_10_7



(h) Total Distance for instance R1_10_8

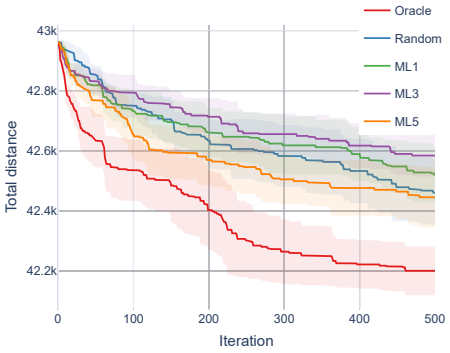


(i) Total Distance for instance R1_10_9

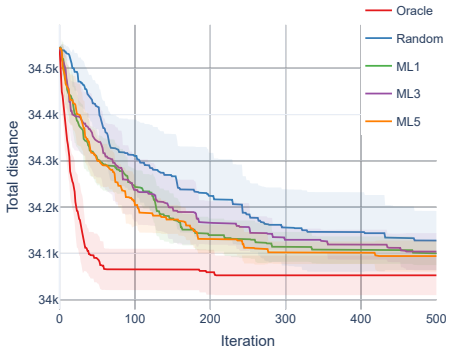


(j) Total Distance for instance R1_10_10

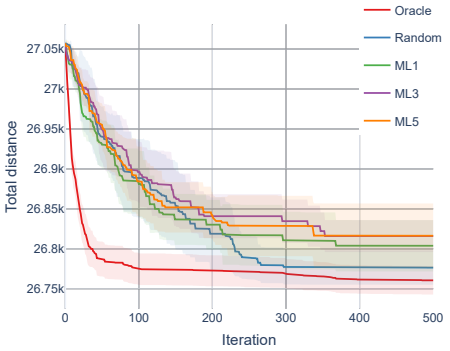
Figure B.1: Total distance for 10 R1 test instances for the oracle model, the random model, ML1 model, ML3 model and ML5 model (cont.)



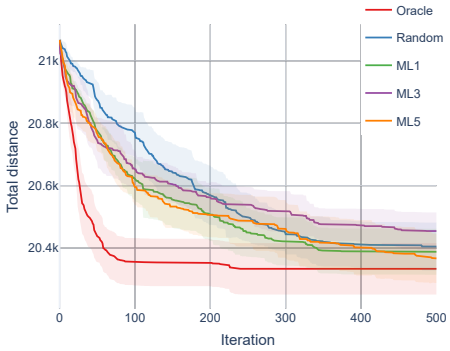
(a) Total Distance for instance R2_10_1



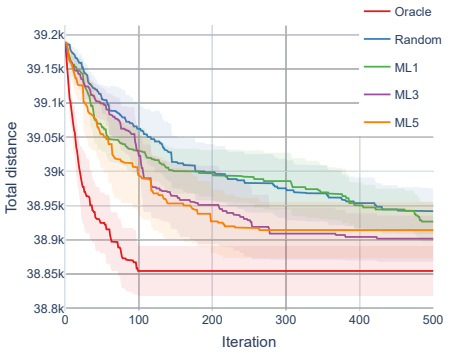
(b) Total Distance for instance R2_10_2



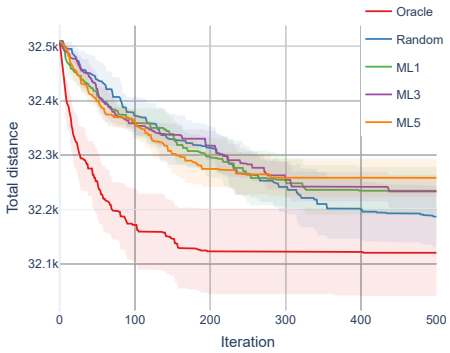
(c) Total Distance for instance R2_10_3



(d) Total Distance for instance R2_10_4

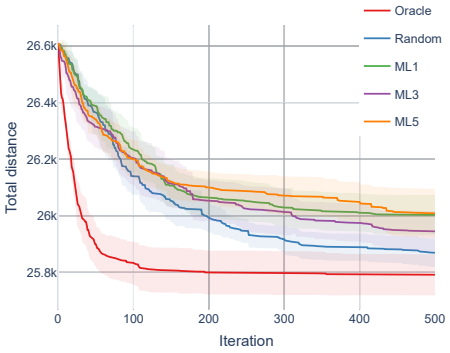


(e) Total Distance for instance R2_10_5

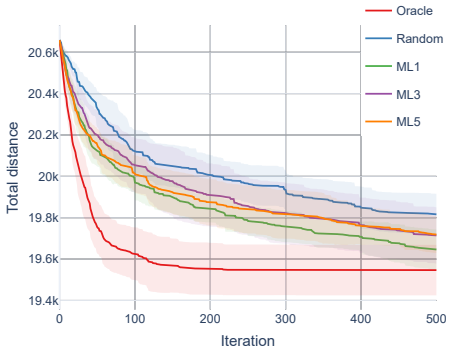


(f) Total Distance for instance R2_10_6

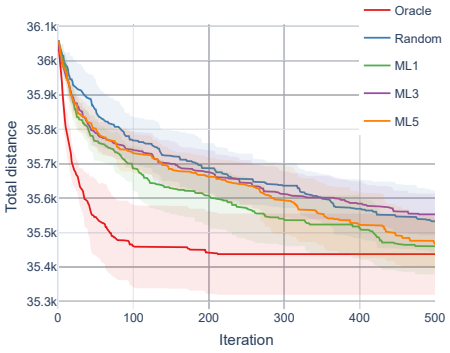
Figure B.2: Total distance for 10 R2 test instances for the oracle model, the random model, ML1 model, ML3 model and ML5 model



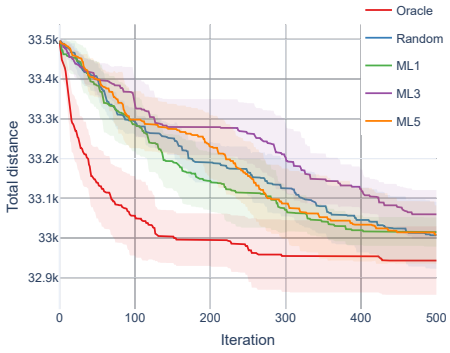
(g) Total Distance for instance R2_10_7



(h) Total Distance for instance R2_10_8



(i) Total Distance for instance R2_10_9



(j) Total Distance for instance R2_10_10

Figure B.2: Total distance for 10 R2 test instances for the oracle model, the random model, ML1 model, ML3 model and ML5 model (cont.)

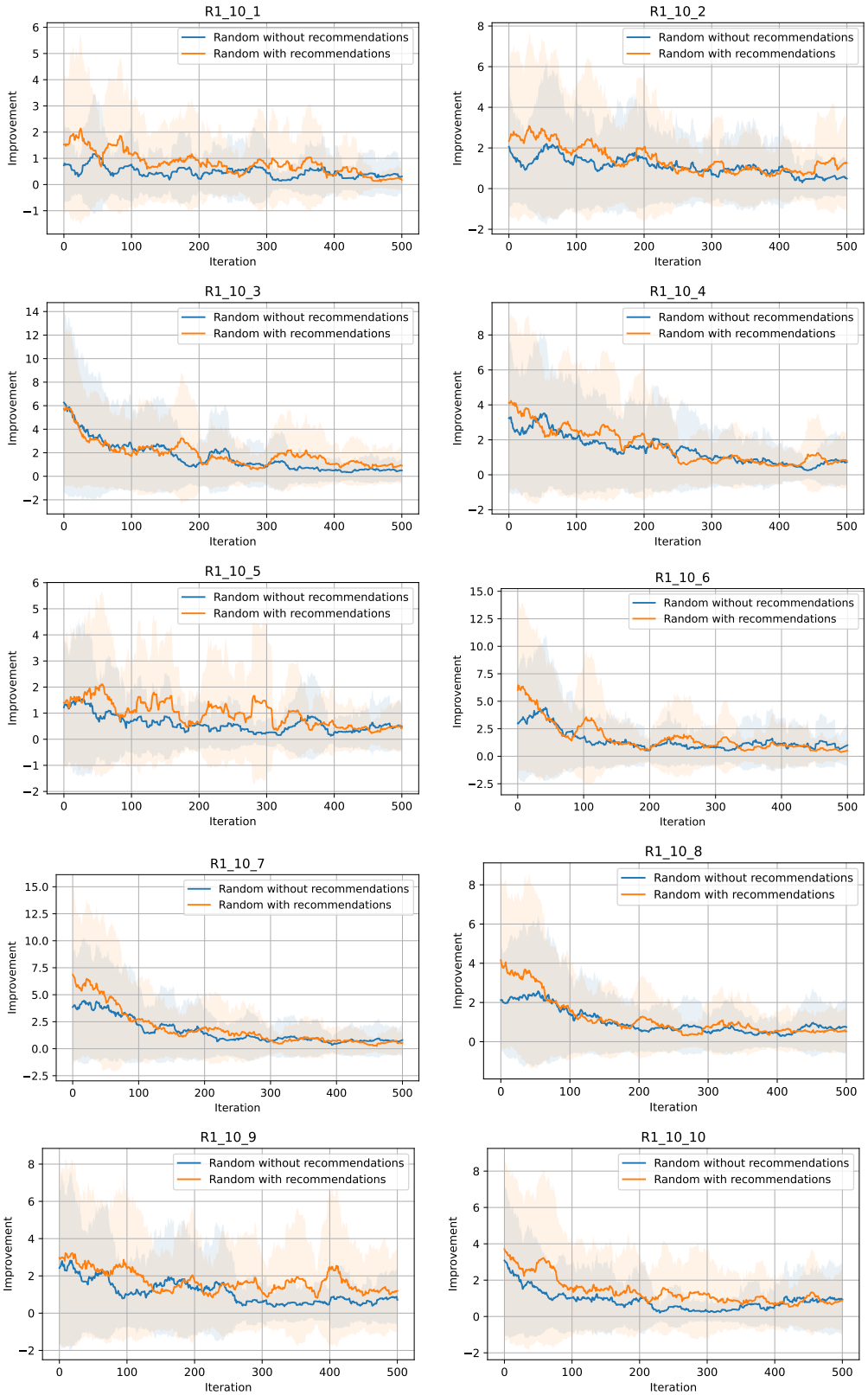


Figure B.3: Average improvement for 10 test instances for the random neighborhood creation with and without recommendations.

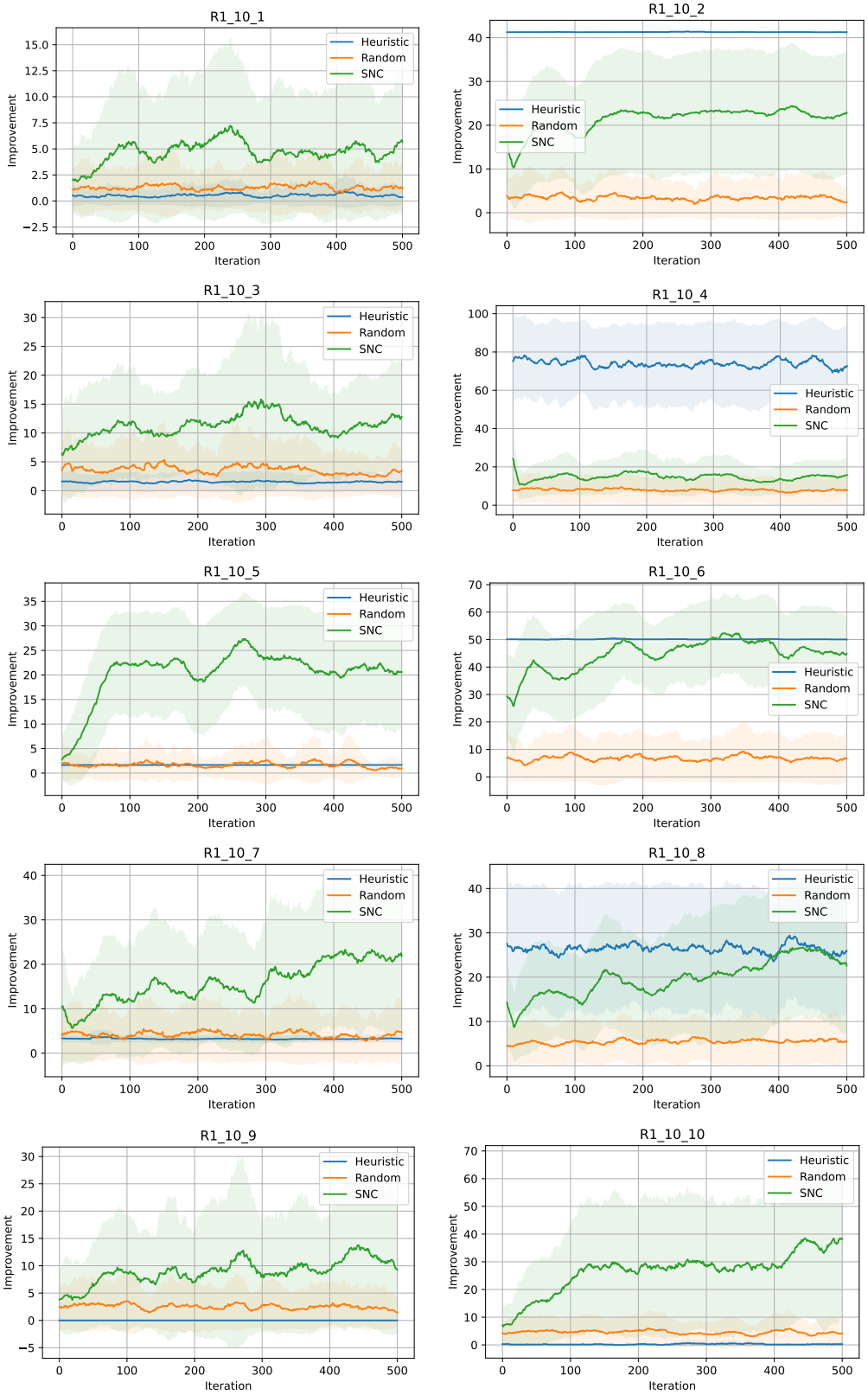


Figure B.4: Average improvement in phase 1 for 10 test instances for the heuristic, random and SNC algorithm.

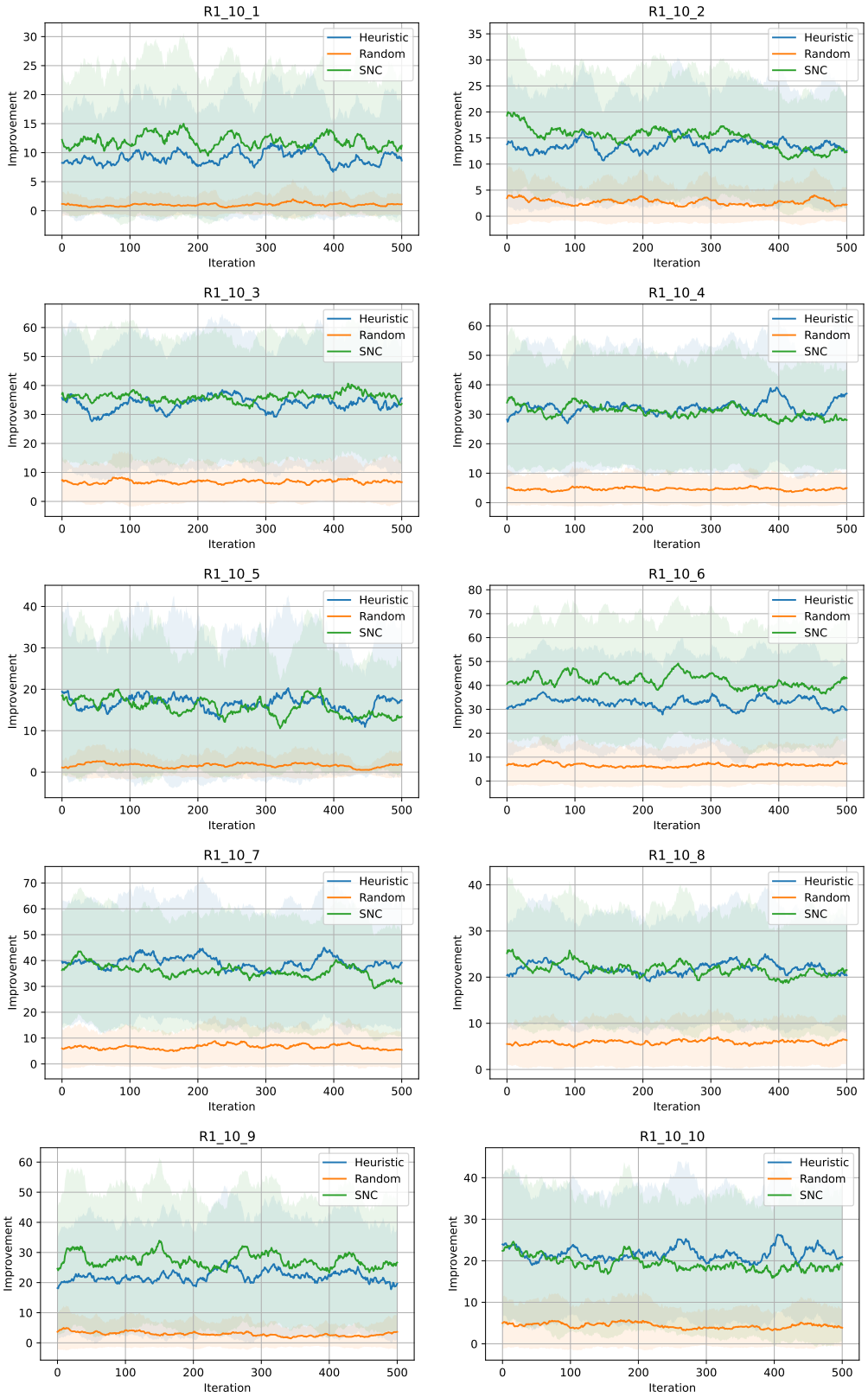


Figure B.5: Average improvement in phase 2 for 10 test instances for the heuristic, random and SNC algorithm.

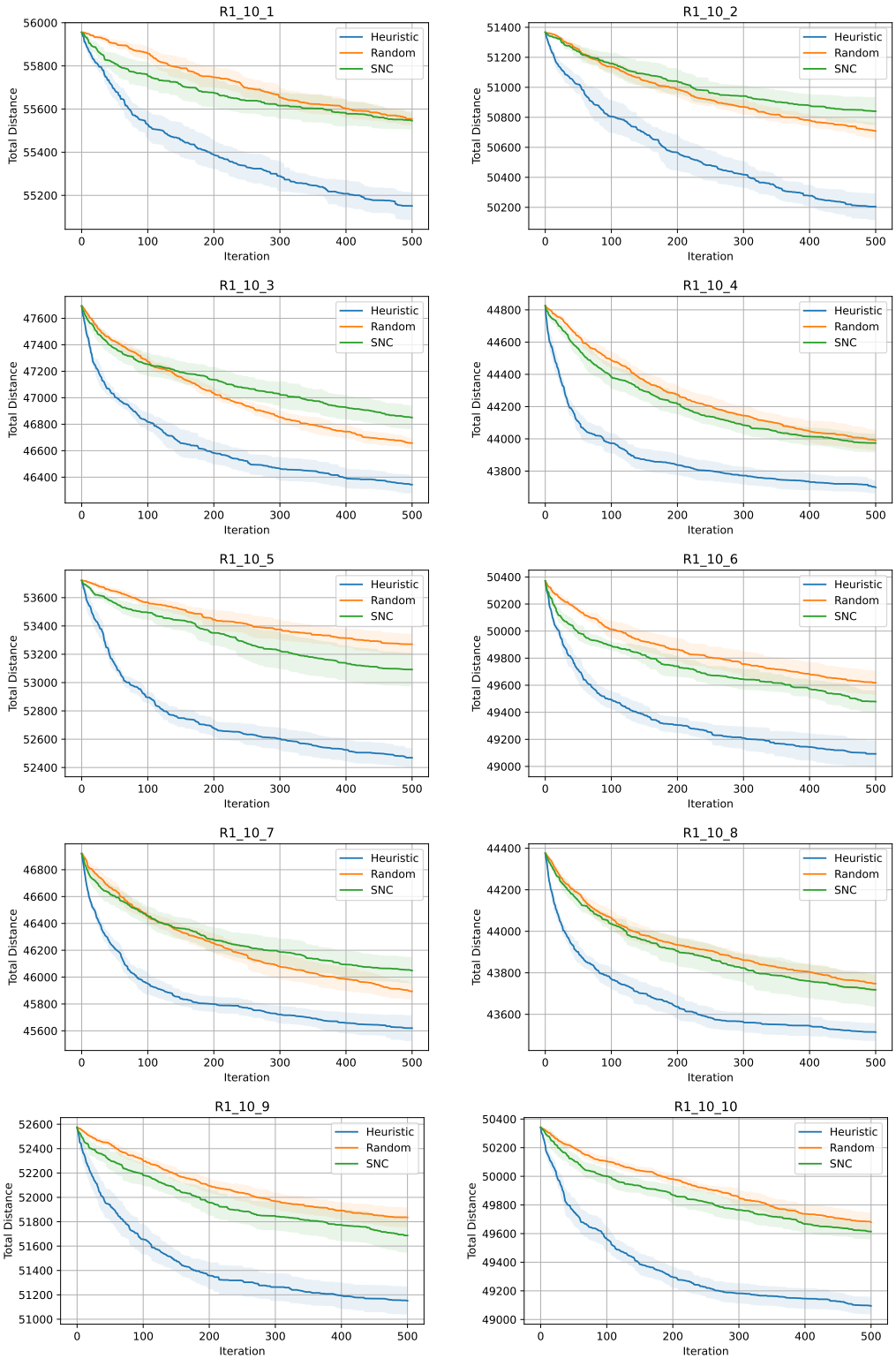


Figure B.6: Total distance in phase 3 for 10 test instances for the heuristic, random and SNC algorithm.

Bibliography

L. Accorsi, A. Lodi, and D. Vigo. Guidelines for the computational testing of machine learning approaches to vehicle routing problems. *Operations Research Letters*, 50(2):229–234, 2022.

N. Alon and J. H. Spencer. *The probabilistic method*. John Wiley & Sons, 2016.

A. Bagheri, M. R. Akbarzadeh Totonchi, et al. Finding shortest path with learning algorithms. *International Journal of Artificial Intelligence*, 1, 2008.

R. Bai, X. Chen, Z.-L. Chen, T. Cui, S. Gong, W. He, X. Jiang, H. Jin, J. Jin, G. Kendall, J. Li, Z. Lu, J. Ren, P. Weng, N. Xue, and H. Zhang. Analytics and machine learning in vehicle routing research. *International Journal of Production Research*, 61(1):4–30, 2023.

H. Bast, K. Mehlhorn, G. Schäfer, and H. Tamaki. A heuristic for Dijkstra's algorithm with many targets and its use in weighted matching algorithms. *Algorithmica*, 36(1):75–88, 2003.

R. Bauer and D. Delling. Sharc: Fast and robust unidirectional routing. *Journal of Experimental Algorithmics (JEA)*, 14:2–4, 2010.

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

D. Bertsimas and J. N. Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena Scientific Belmont, MA, 1997.

K. Bestuzheva, A. Chmiela, L. Eifler, A. Gleixner, K. Halbig, R. van der Hulst, S. J. Maher, B. Müller, S. Schlein, Y. Shinano, et al. The scip optimization suite 8.0, 2021.

C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

A. Bogyrbayeva, M. Meraliyev, T. Mustakhov, and B. Dauletbayev. Learning to solve vehicle routing problems: A survey. *arXiv preprint arXiv:2205.02453*, 2022.

D. G. Cattrysse and L. N. Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European journal of operational research*, 60 (3):260–272, 1992.

C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2005.

C. Chen, E. Demir, and Y. Huang. An adaptive large neighborhood search heuristic for the vehicle routing problem with time windows and delivery robots. *European journal of operational research*, 294(3):1164–1180, 2021.

J. Chen, S. Silwal, A. Vakilian, and F. Zhang. Faster fundamental graph algorithms via learned predictions. In *International Conference on Machine Learning*, pages 3583–3602. PMLR, 2022.

G. Chugh, S. Kumar, and N. Singh. Survey on machine learning and deep learning applications in breast cancer diagnosis. *Cognitive Computation*, pages 1–20, 2021.

S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

J. Coupey, J.-M. Nicod, and C. Varnier. *VROOM v1.13, Vehicle Routing Open-source Optimization Machine*. Verso (`https://verso-optim.com/`), Besançon, France, 2023. `http://vroom-project.org/`.

Dassault Systemes. DELMIA Quintiq Logistics Planning, 2023. URL `https://www.3ds.com/products/delmia/quintiq/logistics-planning`.

K. Deb and C. Myburgh. A population-based fast algorithm for a billion-dimensional resource allocation problem with integer variables. *European Journal of Operational Research*, 261(2):460–474, 2017.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

M. Dinitz, S. Im, T. Lavastida, B. Moseley, and S. Vassilvitskii. Faster matchings via learned duals. *Advances in neural information processing systems*, 34: 10393–10406, 2021.

J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

Y. Dumas, J. Desrosiers, and F. Soumis. The pickup and delivery problem with time windows. *European journal of operational research*, 54(1):7–22, 1991.

T. Eden, P. Indyk, and H. Xu. Embeddings and labeling schemes for A^*. In *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2), 1972.

M. Elkin and D. Peleg. (1+epsilon,β)-spanner constructions for general graphs. *SIAM Journal on Computing*, 33(3):608–631, 2004.

P. v. Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 75–84. IEEE, 1975.

L. Epstein and J. Sgall. Approximation schemes for scheduling on uniformly related and identical parallel machines. *Algorithmica*, 39:43–57, 2004.

J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5): 868–889, 2006.

M. L. Fisher, R. Jaikumar, and L. N. Van Wassenhove. A multiplier adjustment method for the generalized assignment problem. *Management Science*, 32(9), 1986.

M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7, 1990a.

M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 719–725. IEEE, 1990b.

M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.

G. Gamrath and M. E. Lübbecke. Experiments with a generic dantzig-wolfe decomposition for integer programs. In *International Symposium on Experimental Algorithms*, pages 239–252. Springer, 2010.

M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. Freeman San Francisco, 1979.

C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004.

R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms: 7th International Workshop, WEA 2008 Provincetown, MA, USA, May 30-June 1, 2008 Proceedings 7*, pages 319–333. Springer, 2008.

R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4): 1141–1144, 1959.

P. Gilmore and R. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

A. V. Goldberg and R. F. F. Werneck. Computing point-to-point shortest paths from external memory. In *ALENEX/ANALCO*, pages 26–40, 2005.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL `https://www.gurobi.com`.

R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. *ALENEX/ANALC*, 4:100–111, 2004.

T. Hagerup. Improved shortest paths on the word ram. In *Automata, Languages and Programming: 27th International Colloquium, ICALP 2000 Geneva, Switzerland, July 9–15, 2000 Proceedings 27*, pages 61–72. Springer, 2000.

Y. Han. Improved fast integer sorting in linear space. *Information and Computation*, 170(1):81–94, 2001.

P. Hansen, N. Mladenović, J. Brimberg, and J. A. M. Pérez. *Variable neighborhood search*. Springer, 2019.

S. Haykin. *Neural networks and learning machines, 3/E*. Pearson Education India, 2009.

M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:41–72, 2009.

D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM journal on computing*, 17(3):539–551, 1988.

J. Homberger and H. Gehring. A two-phase hybrid metaheuristic for the vehicle routing problem with time windows. *European journal of operational research*, 162(1):220–238, 2005.

A. Hottung and K. Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. In *ECAI 2020*, pages 443–450. IOS Press, 2020.

M. Iri. A new method of solving transportation-network problems. *Journal of the Operations Research Society of Japan*, 3(1), 1960.

M. Karimi-Mamaghan, M. Mohammadi, P. Meyer, A. M. Karimi-Mamaghan, and E.-G. Talbi. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research*, 296(2):393–422, 2022.

H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.

C. Kerscher and S. Minner. Spatial-temporal-demand clustering for solving large-scale vehicle routing problems with time windows. *arXiv preprint arXiv:2402.00041*, 2024.

E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005. Proceedings 4*, pages 126–138. Springer, 2005.

W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

H. Kuhn, D. Schubert, and A. Holzapfel. Integrated order batching and vehicle routing operations in grocery retail–a general adaptive large neighborhood search algorithm. *European journal of operational research*, 294(3):1003–1021, 2021.

H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics quarterly*, 2(1-2), 1955.

F. Lagos and J. Pereira. Multi-armed bandit-based hyper-heuristics for combinatorial optimization problems. *European Journal of Operational Research*, 312 (1):70–91, 2024.

S. Lattanzi, T. Lavastida, B. Moseley, and S. Vassilvitskii. Online scheduling via learned weights. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1859–1877. SIAM, 2020.

U. Lauther. An extremely fast, exact algorithm for finding shor test paths in static networks with geographical background. In *Münster GI-Forum*, 2006.

J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46:259–271, 1990.

S. Li, Z. Yan, and C. Wu. Learning to delegate for large-scale vehicle routing. *Advances in Neural Information Processing Systems*, 34, 2021.

J.-H. Lin and J. S. Vitter. e-approximations with minimum packing constraint violation. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 771–782, 1992.

D. Lu and Q. Weng. A survey of image classification methods and techniques for improving classification performance. *International journal of Remote sensing*, 28(5):823–870, 2007.

A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. Basalamah. A survey of shortest-path algorithms. *arXiv preprint arXiv:1705.02044*, 2017.

V. Maniezzo, M. A. Boschetti, and T. Stützle. The generalized assignment problem. In *Matheuristics: Algorithms and Implementations*, pages 3–33. Springer, 2021.

S. T. W. Mara, R. Norcahyo, P. Jodiawan, L. Lusiantoro, and A. P. Rifai. A survey of adaptive large neighborhood search algorithms and applications. *Computers & Operations Research*, 146:105903, 2022.

S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, 1990.

J. Maue, P. Sanders, and D. Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *Journal of Experimental Algorithmics (JEA)*, 14:3–2, 2010.

N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

M. Mitzenmacher and S. Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022.

R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 11:2–8, 2007.

P. Munari, T. Dollevoet, and R. Spliet. A generalized formulation for vehicle routing problems. *arXiv preprint arXiv:1606.01935*, 2016.

J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1), 1957.

Z. Nutov, I. Beniaminy, and R. Yuster. A (1–1/e)-approximation algorithm for the generalized assignment problem. *Operations Research Letters*, 34(3), 2006.

T. Öncan. A survey of the generalized assignment problem and its applications. *INFOR: Information Systems and Operational Research*, 45(3):123–141, 2007.

D. W. Otter, J. R. Medina, and J. K. Kalita. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(2):604–624, 2020.

Oxford English Dictionary. *Algorithm*. Oxford University Press, 2024. URL `https://languages.oup.com/research/oxford-english-dictionary/`. Accessed: 2024-06-18.

C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation, 1998.

D. Pisinger and S. Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 399–419. Springer, 2010.

A. Qayyum, J. Qadir, M. Bilal, and A. Al-Fuqaha. Secure and robust machine learning for healthcare: A survey. *IEEE Reviews in Biomedical Engineering*, 14:156–180, 2020.

S. Rastani and B. Çatay. A large neighborhood search-based matheuristic for the load-dependent electric vehicle routing problem with time windows. *Annals of Operations Research*, pages 1–33, 2021.

P. Refaeilzadeh, L. Tang, and H. Liu. *Cross-Validation*, pages 532–538. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. URL `https://doi.org/10.1007/978-0-387-39940-9_565`.

F. S. Rizi, J. Schloetterer, and M. Granitzer. Shortest path distance approximation using deep learning techniques. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 1007–1014. IEEE, 2018.

S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

G. T. Ross and R. M. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical Programming*, 8(1), 1975.

P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, volume 3669, pages 568–579. Springer, 2005.

P. Sanders and D. Schultes. Engineering highway hierarchies. In *ESA*, volume 6, pages 804–816. Springer, 2006.

A. Santini, M. Schneider, T. Vidal, and D. Vigo. Decomposition strategies for vehicle routing heuristics. *INFORMS Journal on Computing*, 35(3):543–559, 2023.

M. Savelsbergh. A branch-and-price algorithm for the generalized assignment problem. *Operations research*, 45(6):831–841, 1997.

A. Schrijver et al. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer, 2003.

F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *Journal of Experimental Algorithmics (JEA)*, 5:12–es, 2000.

P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.

D. B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical programming*, 62(1-3):461–474, 1993.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362 (6419):1140–1144, 2018.

Sintef. Sintef best known results for Gehring Homberger's 1000 customer instances. `https://www.sintef.no/projectweb/top/vrptw/1000-customers/`, 2023. Accessed: 2023-08-25.

C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4):1–31, 2014.

N. Sonnerat, P. Wang, I. Ktena, S. Bartunov, and V. Nair. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201*, 2021.

M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

N. Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1(2), 1971.

R. Turkeš, K. Sörensen, and L. M. Hvattum. Meta-analysis of metaheuristics: Quantifying the effect of adaptiveness in adaptive large neighborhood search. *European Journal of Operational Research*, 292(2):423–442, 2021.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.

L. A. Wolsey. *Integer programming*. John Wiley & Sons, 2020.

Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim. Learning improvement heuristics for solving routing problems. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

M. Yagiura and T. Ibaraki. Generalized assignment problem. *Handbook of Approximation Algorithms and Metaheuristics*, page 11, 2007.

X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao. Orion: shortest path estimation for large social graphs. *Networks*, 1:5, 2010.

X. Zhao, A. Sala, H. Zheng, and B. Y. Zhao. Efficient shortest paths on massive social graphs. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 77–86. IEEE, 2011.

Summary

Good algorithms have three qualities, they produce solutions *fast*, produce the *right* solutions and they aim to produce solutions with the *best* objective value. We study the interplay between these three criteria, by considering optimization problems that are all motivated by practical applications. However, this does not prevent us from considering these applications through a theoretical lens. I.e., next to studying well-working algorithms for practical problem instances, we aim to find algorithms with provable properties. Moreover, we give various examples of how the power of machine learning and combinatorial optimization can be combined, to create algorithms that exploit the best of both worlds.

The first problem we consider is a fundamental shortest path problem, known as the *single-source many-targets shortest path problem* (SSMTSP). Given a directed weighted graph, the goal is to compute a shortest path from a given source node to any of several designated target nodes. Basically, our idea is to equip an adapted version of Dijkstra's algorithm with machine learning predictions to solve this problem: Based on the trace of the algorithm, we design a neural network that predicts the shortest path distance after a few iterations. The prediction is then used to prune the search space explored by Dijkstra's algorithm, which may significantly reduce the number of operations on the underlying priority queue. We derive structural insights that allow us to lower bound the savings by our algorithm on partial random instances. In these instances, an adversary can fix the instance arbitrarily except for the weights of a subset of relevant edges, which are chosen randomly. Our bound shows that the number of relevant edges which are pruned increases as the prediction error decreases. We then use these insights to derive closed-form expressions of the expected number of saved queue operations on random instances. We also present extensive experimental results on random instances showing that the actual savings are oftentimes significantly larger.

In Chapter 4 and Chapter 5, we study a special case of the generalized assignment problem, the so-called *casting problem*. An instance of the casting problem is given by a set of knapsacks, each with a non-negative capacity, and a set of items, each with a non-negative weight. A feasible solution to the casting problem assigns each item to a knapsack such that the total weight of items assigned to a knapsack is at most the knapsack's capacity. The *utilization ratio* of a knapsack

is defined as the total weight of items assigned to the knapsack divided by the knapsack's capacity. The goal of the casting problem is to find a feasible solution that maximizes the sum over the utilization ratios. We show that the feasibility problem of the casting problem is NP-complete, and therefore we do not expect to find a polynomial-time algorithm that finds an optimal solution, or even a polynomial-time approximation algorithm, unless P=NP. We focus on bicriteria approximation algorithms instead, which are (informally) defined as follows: an (α, β)-*bicriteria approximation algorithm* is an algorithm that finds a solution with objective value at least α times the cost of an optimal feasible solution with $0 \leq \alpha \leq 1$ such that the knapsack capacities are violated by at most a factor $\beta \geq 1$. In Chapter 5, we first present a $(1, \frac{3}{2})$-bicriteria approximation algorithm, after which we present a $(1/(1+\epsilon), (1+\epsilon)(1+\epsilon+\epsilon^3))$-bicriteria approximation algorithm for any $0 < \epsilon \leq 1$. In Chapter 4, we show empirical results on some instances of the casting problem. We introduce a disaggregated formulation of the problem and show that we can create variables for the disaggregated formulation in such a way that any feasible solution with them is automatically optimal. This allows us to solve all the considered instances to optimality.

Large Neighborhood Search (LNS) is a universal approach that is broadly applicable and has proven to be highly efficient in practice for solving optimization problems. LNS iteratively improves a solution by destroying and repairing a part of the solution. Since the repair routine of LNS is usually expensive, it is crucial to destroy smartly. We have studied two different methods of how to integrate machine learning into LNS to assist in deciding which parts of the solution should be destroyed in each iteration.

The first approach, called *Learning-Enhanced Neighborhood Selection* (LENS), considers multiple sets of routes in each iteration, and predicts with machine learning which set of routes has the most potential to improve the solution. This set of routes is then selected, destroyed, and repaired, after which the solution has possibly improved. The idea is that LENS only selects sets of routes for which it is worth executing the expensive repair routine. We implement LENS into an LNS algorithm for the classical Capacitated Vehicle Routing Problem with Time Windows (CVRPTW).

Also in the second approach, called Neighborhood Creation, we use learning to decide what set of routes to destroy. However, instead of *selecting* the set of routes from a list of options of sets as LENS does, the set is *created* by adding routes one by one. Reinforcement learning is used to select the next route to add to the set, by using an adaptation of an existing graph attention encoder-decoder model. We implement our Neighborhood Creation approach into two applications: a state-of-the-art application used daily to solve many logistic problems and an implementation of LNS for the CVRPTW.