Informatics

Advances in Search Techniques for Combinatorial Optimization: New Anytime A* Search and Decision Diagram Based Approaches

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Matthias Horn, BSc
Registration Number 01027929

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther R. Raidl
Second advisor: Prof. Christian Blum, PhD

The dissertation has been reviewed by:

Luca Di Gaspero   Willem-Jan van Hoeve

Vienna, 2nd June, 2021

Matthias Horn

Informatics

Advances in Search Techniques for Combinatorial Optimization: New Anytime A* Search and Decision Diagram Based Approaches

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Matthias Horn, BSc
Matrikelnummer 01027929

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther R. Raidl
Zweitbetreuung: Prof. Christian Blum, PhD

Diese Dissertation haben begutachtet:

Luca Di Gaspero

Willem-Jan van Hoeve

Wien, 2. Juni 2021

Matthias Horn

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Matthias Horn, BSc

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

Wien, 2. Juni 2021

Matthias Horn

Acknowledgements

First and foremost, I am deeply grateful to my supervisor Günther Raidl for his comprehensive support during my PhD studies. I personally as well as this work had greatly benefited from his insightful constructive comments and warm encouragement. He always gave me advice when I needed it and showed me new interesting research directions. Besides my supervisor, I want also to thank my co-supervisor Christian Blum for his appreciated advises and comments. Especially, I want to thank him for his warm welcome and support during my research stay in Barcelona.

Furthermore, I would like to express my gratitude to former and current coworkers of the Algorithms and Complexity group for providing a lovely work environment. Discussing different topics was always helpful and illuminating. In particular, I want to thank Marko Djukanovic, Johannes Maschler, and Nikolaus Frohner with whom I coauthored articles and work closely together.

I want to thank the Doctoral Program “Vienna Graduate School on Computational Optimization (VGSCO)”, Austrian Science Foundation (FWF) Project No. W1260-N35, which funded my research work, conference trips, my research stay in Barcelona, and provided many excellent courses and workshops in the field of optimization.

A very special thank goes to my former coworkers at DESTION and close friends Simeon Kuran and Andreas Chwatal, who encouraged me to accept this PhD position.

Last but not least, my deepest heartfelt appreciation goes to my close family. Without their support and encouragement in the past years this work would not be possible.

Abstract

Graph search strategies are important methodologies in order to solve combinatorial optimization problems (COPs). Thereby a search tree or search graph is usually considered during the search that covers certain parts of a COP's solution space. In this thesis the search graphs will be mainly based on a state-space representation, where a node of the search graph corresponds to a state that represents a set of partial solutions of the considered COP. A transition from one state to another state, indicated by an arc in the search graph, represents a feasible extension of the partial solutions. We will apply different search techniques to obtain (proven optimal) solutions as well as dual bounds for different COPs. Most considered approaches in this thesis are based on the informed search algorithm A* search, which uses a heuristic function to guide the search towards the solution space and, under certain conditions, is able to terminate with a proven optimal solution.

The first part of this thesis focuses on turning A* search into an anytime A* search such that the algorithm is able to find a feasible heuristic solution shortly after the start and then continuously updates it until a proven optimal solution is finally found. To find heuristic solutions, the approach switches in regular intervals from best-first search to an advanced diving mechanism based on beam search (BS). The novel anytime A* approach is tested on the job sequencing problem with one common and multiple secondary resources (JSOCMSR), which consists of a set of jobs that must be feasible scheduled, a common resource, and a set of secondary resources. Each job needs during its execution the common resource and one of the secondary resources. The common resource acts as a bottleneck resource. The objective is to minimize the makespan. One main application of the JSOCMSR is in the field of scheduling treatments for cancer patients who are to receive particle therapies. Experimental evaluation will show an excellent anytime behavior of our novel anytime A* search. Furthermore, the anytime A* approach is compared to other anytime algorithms as well as to different exact approaches.

The second part of this thesis considers decision diagrams (DDs), which are a rather new methodology for solving COPs that has a strong connection to state-space representations as well. Decision diagrams provide graphical representations of a COP's solution space. In particular relaxed DDs represent compact discrete relaxations of the solution space. Thus, relaxed DDs have the potential to provide strong dual bounds on problems where traditional relaxations, e.g. linear programming relaxations, may be rather weak.

Restricted DDs are another important kind of DDs, which encode a compact subset of a COP's solution space. Hence, they are able to provide heuristic solutions. This thesis will propose a novel construction algorithm of relaxed DDs that is based on the principles of A* search. The construction algorithm is tested by creating relaxed DDs for two NP-hard problems. The first problem is a prize-collecting variant of the JSOCMSR, where each job is equipped with a prize and a set of time windows such that a job can only be feasibly scheduled within one of its time windows. The task is to find a subset of jobs that can be feasible scheduled and that maximizes the total prize over the selected jobs. The second problem is the well known classical longest common subsequence (LCS) problem, which consists of multiple input strings over a finite alphabet. The task is to find a longest subsequence that is common to all input strings. The LCS problem has its application, for instance, in bioinformatics, where strings often represent RNA or DNA segments. For both problems we are able to compile in a shorter time relaxed DDs with the novel A*-based compilation method that are smaller and yield stronger dual bounds than relaxed DDs compiled with traditional methods from the literature.

For the JSOCMSR variant we create further restricted DDs by using structural information of a previously compiled relaxed DD such that the compilation time of the restricted DDs is substantially accelerated. This idea of exploiting structural information of relaxed DDs is pursued further in this thesis by accelerating other search heuristics as well. In particular, we used a previously compiled relaxed DD to accelerate a hybrid approach of limited discrepancy search (LDS) and BS, which is used to heuristically solve the prize-collecting variant of the JSOCMSR with additional precedence constraints. Exhaustive experiments confirm that the LDS/BS approach exploiting a relaxed DD is substantially faster than a standalone approach. In this way it is possible to scan larger parts of the solution space in the same time as a standalone approach in order to obtain better heuristic solutions.

Finally, this thesis considers the repetition-free longest common susequence (RFLCS) problem, which consists of two input strings over a finite alphabet. The task is to find a LCS of both input strings that is repetition-free, i.e., each character in the subsequence appears no more than once. An instance of the problem is solved by transforming it into an instance of the maximum independent set (MIS) problem, which is then solved by an integer linear programming (ILP) approach. We contribute by using a relaxed DD to extensively reduce the size of the MIS problem's conflict graph. Numerical experiments confirm that, as a consequence of the reduction, the corresponding ILP model can be substantially faster solved.

Kurzfassung

Graph Suchstrategien sind wichtige Verfahren um kombinatorische Optimierungsprobleme (KOP) zu lösen. Dabei werden für gewöhnlich Suchbäume oder Suchgraphen während der Suche betrachtet, die bestimmte Bereiche des Lösungsraumes eines KOPs abdecken. In dieser Arbeit basieren die Suchgraphen hauptsächlich auf der Zustandsraumdarstellung, in der ein Knoten des Suchgraphen einem Zustand entspricht, der eine Teilmenge an unvollständigen Lösungen des betrachteten KOPs repräsentiert. Ein Übergang von einem Zustand zu einem anderen Zustand, dargestellt durch eine Kante im Suchgraphen, repräsentiert eine gültige Erweiterung der unvollständigen Lösungen. Wir werden unterschiedliche Suchtechniken anwenden um sowohl (bewiesene optimale) Lösungen als auch duale Schranken für verschiedene KOPs zu erhalten. Die meisten in dieser Arbeit betrachteten Ansätze basieren auf dem informierten Suchalgorithmus A*, der eine Schätzfunktion verwendet, um die Suche durch den Lösungsraum zu leiten und, unter bestimmten Umständen, mit einer bewiesen optimal Lösung terminieren kann.

Der erste Teil dieser Arbeit fokussiert sich auf die Adaptation der A*-Suche in einen "Anytime Algorithmus", sodass der Algorithmus i.A. eine gültige Lösung kurz nach dem Start findet und diese danach kontinuierlich verbessert, bis schließlich eine bewiesen optimale Lösung gefunden werden kann. Um heuristische Lösungen zu finden, wechselt dieses Verfahren in regelmäßigen Intervallen von einer Bestensuche zu einem fortgeschrittenen "Diving-Mechanismus" basierend auf Beam search (BS). Der neue anytime A* Ansatz wird auf einem Job Scheduling Problem mit einer gemeinsamen Ressource und mehreren Sekundärressourcen ("job sequencing problem with one common and multiple secondary resources (JSOCMSR)") getestet. Jeder Job benötigt während seiner Ausführung die gemeinsame Ressource und eine Sekundärressource. Die Nutzung der gemeinsamen Ressource stellt dabei einen Flaschenhals dar. Das Ziel ist die Bearbeitungsspanne über alle Jobs zu minimieren. Eine Hauptanwendung des JSOCMSR liegt im Bereich der Planung von Behandlungen von Krebspatienten, welche eine Strahlungstherapie erhalten. In Experimenten, wird die anytime A*-Suche mit anderen anytime Algorithmen sowie mit verschiedenen exakten Ansätzen verglichen. Die Auswertungen zeigen ein exzellentes "anytime" Verhalten der neuen anytime A*-Suche.

Der zweite Teil dieser Arbeit betrachtet Entscheidungsdiagramme (ED), eine eher neue Methodik um KOPs zu lösen, die ebenfalls eine starke Verbindung zur Zustandsraumdarstellung aufweist. Entscheidungsdiagramme sind Darstellungen des Lösungsraumes von KOPs in From von Graphen. Insbesondere relaxierte EDs liefern eine kompakte diskrete Relaxierung eines Lösungsraumes. Sie haben das Potential starke duale Schranken für

Probleme zur Verfügung zu stellen, bei denen traditionelle Relaxierungen, zum Beispiel lineare Programmierung, eher schwach sind. Eingeschränkte EDs sind eine weitere wichtige Art von EDs, die eine kompakte Untermenge eines KOP Lösungsraumes darstellen. Demzufolge liefern eingeschränkte EDs heuristische Lösungen. Diese Arbeit schlägt eine neue Konstruktionsmethode für relaxierte EDs vor, die auf den Prinzipien der A*-Suche basiert. Die Konstruktionsmethode wird getestet, indem für zwei NP-schwierige Probleme relaxierte EDs erzeugt werden. Das erste Problem ist eine Auswahl-Variante des JSOCMSR, bei der jeder Job mit einem Preis und einer Menge an Zeitfenstern ausgestattet ist, sodass ein Job nur gültig innerhalb eines seiner Zeitfensters geplant werden kann. Aufgabe ist es eine Untermenge an Jobs zu finden, die gültig verplant werden kann und den Gesamtpreis über alle ausgewählten Jobs maximiert. Das zweite Problem ist das wohlbekannte klassische "longest common subsequence (LCS)" Problem, das aus mehreren Eingabezeichenketten über ein endliches Alphabet besteht. Das Ziel ist die längste Teilsequenz zu finden, die von allen Eingabezeichenketten abgeleitet werden kann. Das LCS Problem hat Anwendungen zum Beispiel in der Bioinformatik, wo Zeichenketten oft RNA oder DNA Segmente repräsentieren. Für beide Probleme sind wir in der Lage, in kürzerer Zeit relaxierte EDs mit der auf A*-basierenden Kompilierungsmethode zu erzeugen, die kleiner sind und zu stärkeren dualen Schranken führen als relaxierte EDs kompiliert mit traditionellen Methoden aus der Literatur.

Für die JSOCMSR Variante erzeugen wir darüber hinaus beschränkte EDs, indem wir strukturelle Information eines zuvor erzeugten relaxierten EDs ausnützen, sodass die Kompilierungszeit von beschränkten EDs beträchtlich beschleunigt wird. Die Idee, strukturelle Information eines relaxierten EDs auszunutzen, wird in dieser Arbeit weiterverfolgt, indem weitere Suchheuristiken beschleunigt werden. Im Speziellen verwenden wir ein zuvor kompiliertes beschränktes ED, um einen hybriden Ansatz aus "limited discrepancy search (LDS)" und BS zu beschleunigen, um damit die Auswahl-Variante des JSOCMSR mit zusätzlichen Präzedenz-Beschränkungen heuristisch zu lösen. Ausgiebige Versuche bestätigen, dass der LDS/BS-Ansatz mit der Ausnützung des relaxierten EDs wesentlich schneller ist als ein eigenständiger Ansatz ohne relaxiertes ED. Dadurch ist es möglich, in der selben Zeit größere Teile des Lösungsraumes zu durchsuchen als mit einer eigenständigen Variante ohne relaxierten ED, um bessere heuristische Lösungen zu finden.

Abschließend, betrachtet diese Arbeit das "repetition-free longest common susequence (RFLCS)" Problem, dass aus zwei Eingabezeichenketten über ein endliches Alphabet besteht. Die Aufgabe ist eine LCS von beiden Eingabezeichenketten zu finden, die wiederholungsfrei ist, d.h. jeder Buchstabe in der Teilsequenz darf nicht öfter als einmal vorkommen. Eine Probleminstanz wird gelöst indem sie zu einer Instanz des "maximum independent set (MIS)" Problems transformiert wird, welche dann über ganzzahligen linearen Programmierung (GLP) gelöst wird. Unser Beitrag besteht aus der Nutzung eines relaxierten ED, um die Größe des Konfliktgraphen des MIS-Problems beträchtlich zu reduzieren. Numerische Experimente bestätigen, dass als Konsequenz der Reduzierung das entsprechende GLP Modell substantial schneller gelöst werden kann.

Contents

CHAPTER 1

Introduction

The first time I had worked on a *discrete optimization problem* was shortly before I started to study computer engineering. Back then I had the opportunity to write together within a small group of talented software developers a program to assist teachers of my old school to create school timetables. The main task was to develop a graphical user interface that supports a client server architecture so that multiple users can work together on the same school timetable. As soon as we finished the task we started to wonder if it is possible to use the computational power of our computers to create timetables automatically. This was a tough problem consisting over 200 teachers, 2000 pupils, many classrooms, heavily shared objects like gyms (partially shared with other nearby schools too) and different kinds of laboratories. Besides different resources and shareable objects, there are many additional restrictions ranging from special availability times of teachers to regulatory requirements (e.g. mandatory lunch breaks) that must be considered in order to get a feasible timetable. At this time I did not have any knowledge about computational complexity or combinatorial optimization at all and at the end we came up with a greedy construction algorithm that, after even hours of computation time, was often not able to provide a feasible timetable. Nevertheless, in retrospect, I learned a lot about the practical field of optimization. Despite the fact that we are interested in a best solution for a problem it may sometimes be even challenging to find any feasible solution. Moreover, it may not be clear how to differentiate between “good” and “bad” solutions. Involved people may have divergent opinions on the quality of a specific solution. For instance, pupils may assess timetables differently as teachers and teachers in turn may have a different opinion of “good” timetables than the principle of the school.

Later, during my studies I learned about the theoretical fundamentals of computer science and discrete optimization. At that time I had also the great pleasure to work at the operations research startup company DESTION that provides consulting and software solutions in the field of industrial optimization. In particular different kinds of large scaled vehicle routing problems (VRPs) were heuristically solved with up to ten thousand of orders, which have to be delivered to costumers. The considered VRPs included a heterogeneous fleet of vehicles, multiple depots, and different kinds of constraints, e.g., time window constraints, (multi-dimensional) capacity constraints, or site dependency constraints. The objective functions consisted of various penalty terms including among others the minimization of the total fuel consumption, waiting times, workload of drivers, or the number of used vehicles. Due to my work at the startup company I became more and more interested in optimization. Consequently, I decided to write my master thesis about site-dependent VRPs under supervision of Günther Raidl and Andreas Chwatal [76]. In the following I accepted a PhD position at the Algorithms and Complexity group, TU Wien, in the field of combinatorial optimization supervised by Günther Raidl as well and funded by the Vienna Graduate School on Computational Optimization (VGSCO).

So what are combinatorial optimization problems (COPs)? Such problems do not just ask for a feasible solution but rather they ask for a *best solution* of all feasible solutions. The term "combinatorial" means that we are dealing with discrete objects (e.g. graphs, permutations, integers, strings, . . .) such that the set of all feasible solutions, sometimes denoted as *solution space*, is countable. To differentiate between "good" and "worse" solutions, a value is usually assigned to each solution such that solutions with smaller values are considered as better/worse than solutions with higher values. Most practical optimization problems are NP-hard and it is therefore unlikely that an efficient algorithm exists to compute the optimal solution, unless P=NP. To find an optimal solution or a near optimal solution for such NP-hard optimization problems we *search* for it in the set of feasible solutions in a *clever* way. Therefore the solution space is also called *search space* and the word clever is here the crucial point since the search space usually grows exponentially with the problem size, an exhaustive enumeration of all feasible solutions will not be practicable even for small instances in most cases.

This search process can be frequently represented, explicitly or implicitly, as a *search graph* (or *search tree*). For instance, consider trajectory based (meta)heuristics, which move during the search from one solution to another (better) solution by considering neighborhood structures. This search process can be represented by a graph such that each solution is mapped to a node and arcs between nodes represent the possible movement from one solution to another solution. Although this search graph is usually not explicitly created during the search, it has its application in landscape analysis of neighborhood structures [146, 163]. Another example, where search graphs appear are exact methods that are based on the divide-and-conquer principle. These methods partition the solution space into disjoint subspaces in a recursive way. This partitioning can be represented as a tree, where the root node represents the whole solution space and child nodes represent subspaces [28]. This tree can be traversed by using one of the standard tree traversal strategies, including uninformed search strategies like depth-first search or breath-first search or informed search strategies like best-first search. A further example is to consider the state space representation of the solution space, which is used e.g. in the field of artificial intelligence [131, 139] but also construction algorithms may be based on such representations, e.g. the earlier mentioned greedy construction algorithm for school timetabling. A state represents (a set of) partial (i.e. incomplete) solutions and a transition from one state to another state represents the feasible extension of those partial solutions. Thereby a state contains all necessary information to perform all feasible transitions to successor states and to check if the state represents a complete solution. The state space can be represented as a *state graph* where each node is associated to a state and an outgoing arc represents the transition to a successor state. Furthermore, each arc is typically labeled with transition costs, caused by taking the corresponding transition. Usually there is a single root state representing an empty solution and one or more goal states representing complete solutions. Solving the optimization problem means to find a sequence of transitions from the root state to a goal state such that the corresponding total transition cost are minimized. Due to the in general exponential size, the state graph is usually only implicitly given by using a transition function that defines how successor states are obtained from a given state. Searches on such state graphs can be again performed by diverse tree traversal techniques.

This thesis focuses mainly on state space representations and we will apply different search techniques to obtain heuristic solutions as well as proven optimal solutions. Most of the approaches presented in this thesis are based on A* search [70], which is a well known path-finding algorithm in artificial intelligence on possible huge graphs. The search belongs to the class of informed search algorithms meaning that it uses a heuristic function to guide the search towards the solution space. Depending on some properties of the heuristic function, A* is able to terminate with a proven optimal solution as soon as the first complete solution is explored. A common problem of A* search is that all explored nodes of the underlying state graph must be kept in memory, leading to the frequent case that the search runs out of memory before a feasible solution could be found. In those cases A* is not able to provide any feasible solution at all. The first part of this thesis focuses therefore on turning A* search into an anytime algorithm by combining best first search with beam search (BS). An anytime algorithm is expected to find a feasible (usually non-optimal) solution shortly after the start and then continuously updates it until a proven optimal solution is finally found during the search. Hence, an anytime algorithm can be interrupted at almost anytime and yields a heuristic or finally optimal solution.

The second part of this thesis is about decision diagrams (DDs) which are well known in computer since for decades in the fields of formal verification and logic circuit design. In the last decade they got popular in the field of combinatorial optimization too, by obtaining new state-of-the-art results on several classical problems [14]. In essence, decision diagrams provide a graphical representation of solutions of COPs. They are (weighted) directed acyclic multigraphs and have a strong relationship to the already discussed state space representation. Each node can be associated to a specific state and paths originating from the root node encode solutions of the considered combinatorial optimization problem. *Exact* DDs represent precisely the set of feasible solutions of the considered problem. For NP-hard problems such exact DDs tend to grow exponentially with the problem size. Heuristic methods try to overcome this combinatorial explosion by, e.g., traversing only promising parts of the state graph. A similar strategy is used for compiling *restricted* DDs which represent only a subset of feasible solutions by ignoring non-promising parts of exact DDs. Thus they provide heuristic solutions. An alternative approach to approximate exact DDs is to compile relaxed DDs, which represent a more compact discrete relaxation of the solution space by superimposing (merging) nodes of exact DDs. Thus, relaxed DDs represent a superset of all feasible solutions and can provide, for instance, dual bounds. In recent years relaxed DDs have been successfully applied to several classical optimization problems, including the maximum independent set (MIS), set covering, and maximum cut problems as well as diverse sequencing and scheduling problems [15, 40, 99]. This thesis presents a novel construction scheme for relaxed DDs which is based on principles of A* search. It uses problem specific upper bounds to guide the construction of relaxed DDs. For the considered prize-collecting scheduling problem, which will be described below, more compact relaxed DDs could be obtained within shorter computation time providing stronger dual bounds than relaxed DDs compiled with standard methods from the literature. In a second step, we use the compiled relaxed DDs to speed up the compilation time of restricted DDs as well as to substantially accelerate standard heuristic search techniques. In particular we will demonstrate how a hybrid approach of limited discrepancy search (LDS) and BS can be accelerated in order to scan larger regions of the solution space than a standalone approach without using relaxed DDs can do in the same time.

The anytime A* algorithm as well as the DD-based approaches are applied specifically on different variants of a job sequencing problem with one common and multiple secondary resources (JSOCMSR). A set of non-preemptive jobs needs to be scheduled, where each job requires two resources: (1) a common resource that is shared by all jobs and (2) a secondary resource, which is shared with only a subset of the other jobs. While the common resource is only required for a part of the job's processing time, the secondary resource is required for the whole duration. Figure 1.1 shows an exemplary solution of an instance with three secondary resources and seven jobs. An interesting property, from a scientific point of view, is that a specific resource (the common resource) is needed by all jobs and is therefore a bottleneck resource. Besides applications in the production of certain goods on a single machine involving mixtures or molds, the problem appears (as subproblem) also in the field of scheduling treatments for cancer patients who are to receive a particle therapy [41, 90, 120]. In this rather novel treatment technique a particle beam consisting of proton or carbon particles is accelerated in a particle accelerator to almost the speed of light, and then directed into one of a few treatment rooms that are differently equipped for specific kinds of radiations. Our considered sequencing problem appears here as a simplified daily subproblem where the treatment rooms corresponds to the secondary resources and the single particle beam, which can only be directed in one of those rooms at a time, corresponds to the common resource.

In the first part of this thesis, our goal will be to minimize the makespan in the JSOCMSR setting by using the anytime A* algorithm. The second part will extend the problem

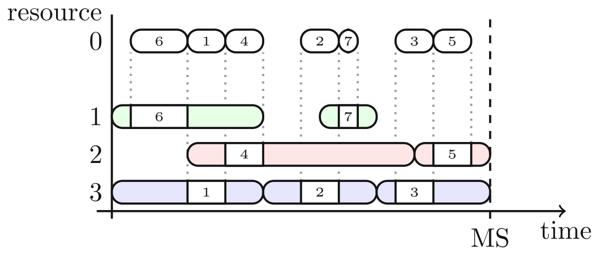


Figure 1.1: Example of a (possible not optimal) solution of a JSOCMSR instance with seven jobs, common resource 0, and three secondary resources 1–3. The makespan MS should be minimized. See Chapter 3 for further details on the JSOCMSR.

formulation by a selection aspect, yielding the so-called prize-collecting job sequencing problem with one common and multiple secondary resources (PCJSOCMSR). Each job is additionally equipped with a prize and a set of time windows where the job is allowed to be scheduled. The task is to select a subset of jobs such that they can be feasibly scheduled and to maximize the total prize over the selected jobs. These extensions are considered mainly to have a more accurate model for the real world patient scheduling problem where not all jobs can be selected for one single day and the start times of the jobs are often limited due to underlying resources. Moreover, the extension makes it also better suited to another application: pre-runtime scheduling of electronic systems within aircraft, called avionic systems, as introduced in [21, 91]. There, a decomposition based approach is used to tackle the complex and large-scaled industrially relevant instances where the PCJSOCMSR appears as an important sub-structure. Simply said, an avionic system consists of a set of nodes and each of these contains a set of modules (processors) with jobs to be scheduled. In each node, there is a single module called the communication module, which corresponds to the common resource. Moreover, each node also has a set of application modules, which correspond to the secondary resources. The task is to create schedules for the nodes by scheduling as many jobs as possible. Large instances of the PCJSOCMSR are solved by compiling relaxed as well as restricted DDs. Furthermore, since in avionic systems jobs often need to be finished before other jobs may start, we extend the PCJSOCMSR with precedence constraints to address this aspect too. This problem variant is solved heuristically by a hybrid LDS/BS approach using relaxed DDs for speed-up.

Finally, we consider the longest common subsequence (LCS) problem consisting of a set of input strings and a finite alphabet. The task is to find the longest subsection that is common to all input strings. A subsequence is a string that can be derived by another string by deleting zero or more characters from it, and it is common to the input strings if the same subsequence can be derived from all input strings. Furthermore, we consider the repetition-free longest common susequence (RFLCS) problem, which is a variant of the LCS problem and usually limited to two input strings. The task is to find the LCS that s repetition free, i.e., no character is allowed to appear more than once in the string. Figure 1.2 shows an example of a LCS/RFLCS instance. The LCS problem has a wide

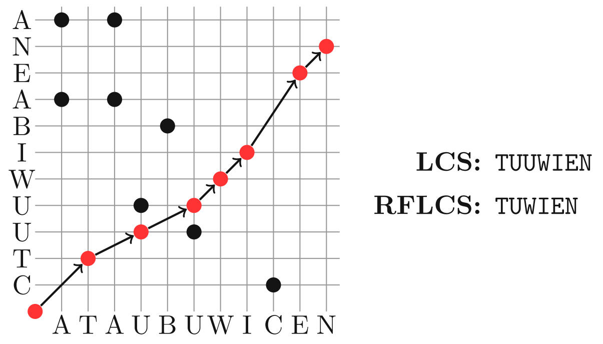


Figure 1.2: Example of an RFLCS instance with two input strings ATAUBUWICEN and CTUUWIBAENA. See Chapter 4 and 6 for further details.

range of applications, for example, in computational biology where strings often represent segments of RNA or DNA [87, 145]. The RFLCS problem arises, for example, in the context of gene duplication in the domain of genome rearrangement. We will improve existing solving techniques by using relaxed multivalued decision diagrams (MDDs).

1.1 Outline of the Thesis

The remainder of this thesis is organized as follows. The next chapter gives an overview of the relevant methodologies used to solve combinatorial optimization problem and starts with a formal definition of combinatorial optimization problems. The following sections will review several standard search algorithms followed by two sections that give an overview on exact as well as heuristic methods with emphasis on approaches that are used within this thesis. The last section discusses DDs in more depth by presenting exact, relaxed, and restricted DDs as well as standard construction methods to compile DDs.

Chapter 3 is dedicated to the novel anytime A* search algorithms to tackle the JSOCMSR. After discussing related work we give a formal problem definition and prove the NP-hardness of the considered problem. Then we present different kinds of lower bounds on the makespan objective and utilize them in a greedy construction algorithm called least lower bound heuristic (LLBH). Afterwards we design an A* search algorithm that uses the mentioned lower bounds as search guidance. Moreover, the A* search is turned into an anytime algorithm by obtaining primal solutions during the search with an advanced diving mechanism. This diving mechanism uses BS, which is based on LLBH, to find primal solutions which are then further improved by a local search (LS) procedure. Furthermore, for comparison purposes we provide mixed integer linear programming (MILP) and constraint programming (CP) formulations. Finally, we present for hard-to-solve instances a general variable neighborhood search (GVNS) that uses an efficient evaluation scheme to scan neighboring solutions of the current incumbent solution. This chapter is based on the following publications

- M. Horn, G. R. Raidl, and C. Blum. Job sequencing with one common and multiple secondary resources: A problem motivated from particle therapy for cancer treatment. In G. Giuffrida, G. Nicosia, P. Pardalos, and R. Umeton, editors, *MOD 2017: Machine Learning, Optimization, and Big Data – Third International Conference*, volume 10710 of *LNCS*, pages 506–518. Springer, 2017

- M. Horn, G. Raidl, and C. Blum. Job sequencing with one common and multiple secondary resources: An A*/Beam Search based anytime algorithm. *Artificial Intelligence*, 277(103173), 2019

- T. Kaufmann, M. Horn, and G. R. Raidl. A variable neighborhood search for the job sequencing with one common and multiple secondary resources problem. In T. Bäck, M. Preuss, A. Deutz, H. Wang, C. Doerr, M. Emmerich, and H. Trautmann, editors, *Proceedings of PPSN XVI: Parallel Problem Solving from Nature*, volume 12270 of *LNCS*, pages 385–398. Springer, 2020.

In Chapter 4 we tackle the prize-collecting variant of the job sequencing problem to obtain new state-of-the-art heuristic solutions as well as the LCS problem to obtain new best dual bounds on the LCS length. For this purpose a novel A*-based construction (A*C) algorithm is presented to compile relaxed DDs. This relaxed DD is then used to further compile a restricted DD to get besides a dual bound also a heuristic solution for the PCJSOCMSR. The first section of the chapter presents the basic concepts of the A*C algorithm. The next sections are dedicated to a detailed problem formulation of the PCJSOCMSR, related work, and different upper bounds on the total prize. The strength of those different upper bounds are evaluated by performing classical A* searches on rather small instance sizes. The best combination of these upper bounds is then used to compile relaxed DDs with the A*C algorithm for larger instances. Moreover, for the experimental evaluation we provide besides the relaxed and restricted DD approaches also an integer linear programming (ILP) formulation and a CP formulation. The last part of Chapter 4 is dedicated to the LCS problem where relaxed DDs are compiled with A*C to obtain dual bounds that in several cases are stronger than the best known dual bounds from the literature. This chapter covers the publications

- M. Horn, G. R. Raidl, and E. Rönnberg. An A* algorithm for solving a prize-collecting sequencing problem with one common and multiple secondary resources and time windows. In E. K. Burke, L. Di Gaspero, B. McCollum, N. Musliu, and E. Özcan, editors, *Proceedings of the 12th International Conference of the Practice and Theory of Automated Timetabling, PATAT 2018*, pages 235–256, Vienna, Austria, 2018

- M. Horn, G. R. Raidl, and E. Rönnberg. A* search for prize-collecting job sequencing with one common and multiple secondary resources. *Annals of Operations Research*, 2020

- M. Horn, J. Maschler, G. R. Raidl, and E. Rönnberg. A*-based construction of decision diagrams for a prize-collecting scheduling problem. *Computers & Operations Research (COR)*, 126:105125, 2021
- M. Horn and G. R. Raidl. A*-based compilation of relaxed decision diagrams for the longest common subsequence problem. volume 12735 of *LNCS*, 2021. To appear. Accepted for the 18th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR 2021.

Chapter 5 reveals how relaxed DDs can be used to accelerate heuristic search techniques. In particular a hybrid approach of LDS and BS is accelerated to solve heuristically large instances of the PCJSOCMSR with precedence constraints. After explaining how relaxed DDs are compiled with A*C, a detailed description of the LDS/BS approach is given. At the end of the chapter experimental results are provided by comparing the performance of our approach with the heuristic performance of MILP and CP solvers. This chapter is based on the publication

- M. Horn and G. R. Raidl. Decision diagram based limited discrepancy search for a job sequencing problem. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *Proceedings of the 17th International Conference of Computer Aided Systems Theory, EUROCAST 2019*, volume 12013 of *LNCS*, pages 344–351. Springer, 2020.

The RFLCS problem is tackled in Chapter 6 by transforming instances of the RFLCS problem to instances of the MIS problem. The MIS instance is than solved by the MILP solver CPLEX. To reduce the size of the MIS instance a relaxed MDD is utilized. The chapter gives first a formal definition of the RFLCS problem followed by a description of the transformation to the MIS problem. Then the compilation of relaxed MDDs is explained in detail as well as the reduction of the size of the corresponding MIS instance. The chapter concludes with an experimental evaluation of the achieved reduction of the MIS instance and the implications of the solving time of the MILP solver. This chapter is based on the publication

- M. Horn, M. Djukanovic, C. Blum, and G. R. Raidl. On the use of decision diagrams for finding repetition-free longest common subsequences. In N. Olenev, Y. Evtushenko, M. Khachay, and V. Malkova, editors, *Proceedings of the XI International Conference Optimization and Applications, OPTIMA 2020*, volume 12422 of *LNCS*, pages 134–149. Springer, 2020.

Chapter 7 concludes this thesis and gives an outlook of further research directions.

CHAPTER 2

Basic Methodologies for Combinatorial Optimization Problems

This chapter presents some of the fundamental solving techniques to tackle combinatorial optimization problems (COPs) which are used and/or extended in the upcoming chapters. However, this chapter is not indented to cover all methodologies in detail since this would be clearly out of the scope of this thesis. There are a lot of excellent books on these topics which provide in-dept coverage, e.g. [28, 127]. Section 2.1 starts with some formal definition of COPs. Most solving techniques perform searches for feasible solutions on search trees or search graphs. Therefore, Section 2.2 presents some of the standard tree traversal strategies including uninformed search strategies like depth-first search or breadth-first search as well as informed search strategies including A* search. Section 2.3 gives an overview of exact methods including mathematical programming approaches, constraint programming approaches as well as dynamic programming. Exact methods are able to find an optimal solution, however, their computation time may increase dramatically with the size of the problem instance. On the contrary, heuristic and metaheuristic approaches, presented in Section 2.4, are expected to find high-quality solutions in reasonable computation time but usually cannot guarantee optimality. The last Section 2.5 introduces decision diagrams (DDs) for combinatorial optimization by explaining fundamental concepts like exact, relaxed and restricted DDs and describes standard construction methods to compile DDs.

2.1 Combinatorial Optimization Problems

In this section we provide some fundamental definitions mainly following Papadimitriou and Steiglitz [127], and Blum and Roli [29]. We start with the definition of general optimization problems.

Definition 2.1.1 (Instance, Papadimitriou and Steiglitz [127])
An instance of an optimization problem is a pair $(\mathscr{S}, f)$ where $\mathscr{S}$ is any set of feasible solutions or points and f is an objective function $f : \mathscr{S} \mapsto \mathbb{R}$ that maps any solution from $\mathscr{S}$ to a real number.

Definition 2.1.2 (Optimization problem, Papadimitriou and Steiglitz [127])
An *optimization problem* is a set of instances.

Depending on the problem, the task is to find a solution from $\mathscr{S}$ that minimizes or maximizes f. Corresponding problems are called minimization or maximization problems. Since each maximization problem can be reformulated to a minimization problem by multiplying f with -1, we consider w.l.o.g. only minimization problems within this chapter.

Definition 2.1.3 (Global optimum, Papadimitriou and Steiglitz [127])
A *global optimum* is any solution $x^* \in \mathscr{S}$ that satisfy

$$f(x^*) \leq f(x) \quad \forall x \in \mathscr{S}. \tag{2.1}$$

Solution x^* is also called *globally optimal solution* or if the context is clear and no confusion can arise just *optimal solution*. It is important to distinguish between optimization problem and an instance of an optimization problem since the latter is a specific *realization* of an optimization problem providing enough input data to obtain a solution. On the contrary, when we talk about the optimization problem then we mean the whole problem including all possible instances. Optimization problems can be divided into continuous optimization problems and into COPs. The former use continuous decision variables and $\mathscr{S}$ is usually a set of real numbers or functions whereas COPs use discrete decision variables and $\mathscr{S}$ is a *countable set* of discrete objects. In this thesis we will in almost all cases deal with COPs, and therefore we assume that $\mathscr{S}$ is a countable set. The only exceptions are linear programming (LP) models in Section 2.3.2, which are based on continuous decision variables. Usually it is not practicable to describe set $\mathscr{S}$ by stating each single solution. Instead, set $\mathscr{S}$ is described implicitly by input parameters and constraints that allows expressing precisely set $\mathscr{S}$ and the corresponding objective function f. For COPs we will therefore frequently use the description of Blum and Roli [29] to express the set $\mathscr{S}$ by decision variables and constraints.

Definition 2.1.4 (Instance of a COP, Blum and Roli [29])
An instance of a combinatorial optimization problem $\mathcal{P}$ consists of a tuple $(\mathbf{x}, \mathcal{C}, \mathcal{D}, f)$ of n decision variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and a (possible empty) constraint set $\mathcal{C} =$

$\{C_1, C_2, \dots\}$. Each variable x_i, $1 \le i \le n$ has a finite domain $\mathcal{D}_i$ and constraints are defined on subsets of $\mathcal{D} = \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n$ such that a constraint is either satisfied or violated by any given $\mathbf{x}$. If $\mathbf{x} \in \mathcal{D}$ satisfies all of the constraints in $\mathcal{C}$ then $\mathbf{x}$ is a feasible solution of $\mathcal{P}$. Objective function $f : \mathcal{D} \to \mathbb{R}$ maps any $\mathbf{x} \in \mathcal{D}$ to a real number $f(\mathbf{x})$, called objective value of $\mathbf{x}$.

The set of feasible solution can be expressed by $\mathscr{S} = \{\mathbf{x} \in \mathcal{D} \mid \mathbf{x} \text{ satisfies } \mathcal{C}\}$. Note that the description of $\mathscr{S}$ is in general not unique and different solving approaches may obtain better/worse results depending on the used formulation of $\mathscr{S}$.

In general, set $\mathscr{S}$ may grow exponentially with the instance size such that checking all solutions in $\mathscr{S}$ to find x^* will be not practicable. Therefore, in practice it is difficult for many relevant COPs to find the optimal solution. One reason for this difficulty, from a theoretical point of view, is that such problems are often NP-hard. Consequently, under the assumption that NP≠P, finding an optimal solution of an NP-hard problem requires in the worst case an exponential number of steps. In general, the literature distinguishes between three kinds of approaches to solve such problems.

Exact Methods: Guarantee that the optimal solution will be found but may require exponential time in the worst case. Section 2.3 provide an overview of the most common exact approaches.

Approximation Algorithms: Guarantee that the algorithm will terminate after a polynomial number of steps and provide a quality guarantee typically of the form that the obtained objective value does not exceed the optimal value times a certain approximation factor. These algorithms are just mentioned here for completeness and are not considered in this thesis.

Heuristics: The goal of heuristic methods is to find high quality solutions in polynomial time but with no optimality guarantee. Some heuristic methods and concepts are described in Sections 2.2 and 2.4.

As we will see in the remaining sections of this chapter, there are plenty of different exact methods as well as heuristic approaches. The reason for this diversity of exact and heuristic solving strategies can be explained by the *no free lunch theorem* by Wolpert and Macready [165]. According to Ho and Pepyne the theorem can be interpreted such that "*a general-purpose universal optimization strategy is theoretically impossible, and the only way one strategy can outperform another is if it is specialized to the specific problem under consideration*" [73]. In other words, there is no "best" algorithm that dominates all other algorithms on each problem for each possible instance. For different NP-hard COPs and different instance classes there will be different approaches superior.

2.2 Combinatorial Search

This section describes search strategies as discussed by Russel and Norvig [139] and Poole and Mackworth [131]. While these authors describe search strategies in the context of artificial intelligence, where so-called agents are searching for the best sequence of actions to reach a certain goal, we will keep the discussion more in the context of combinatorial optimization by searching for the best sequence of actions to construct an optimal solution of a COP. For this purpose we consider the *state space representation*, where a *state* represents a set of partial solutions of the considered COP. The set of all possible states is denoted by state space $\mathcal{S}$, and a *transition* from one state to another state represents the feasible extension of the corresponding partial solutions such that a state contains all necessary information to perform such feasible transitions. Transition function $\tau : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ returns for a given state $s \in \mathcal{S}$ the set of successor states $\tau(s)$ reachable by any single feasible transition from s[1]. A single initial state $s_0 \in \mathcal{S}$ represents an empty solution whereas a state that represents a complete solution is called goal or target state. A state $s \in \mathcal{S}$ contains all required information to check if s is a target state or not. Each transition from state s to another state s' causes some costs $c(s, s')$ such that the task is to find a sequence of transitions $s_0, s_1, \ldots, s_k$ from the initial state to a goal state s_k that minimizes the total cost $f(s_0, s_1, \ldots, s_k) = \sum_{i=1}^{k-1} c(s_i, s_{i+1})$. This sequence can be used to construct the optimal solution of the considered COP. Hence, the *state space* consists of a set of states $\mathcal{S}$, transition function $\tau(\cdot)$, an initial state $s_0 \in \mathcal{S}$ and transition costs $c(\cdot, \cdot)$ for each transition. Note that the state space can also be represented by a *state graph* where each state is associated to a node and arcs between two nodes represent a transition between the corresponding states. Since the number of nodes and/or the number of paths in such a state graph has exponential size, the state graph is usually given implicitly by using transition function $\tau(\cdot)$. Furthermore, we do not consider cases where the state graph has an infinite number of nodes or where the state graph contains cycles.

The idea is now to find (optimal) solutions by searching through the state space and find sequences of states from the initial state to a goal state. This search process can be represented by creating a *search tree* $G = (V, A)$ with vertex set $V(G)$ and arc set $A(G)$ where each vertex $v \in V(G)$ is associated to a state $s \in \mathcal{S}$ such that function $\sigma : V(G) \rightarrow \mathcal{S}$ maps each node $v \in V(G)$ to an associated state $\sigma(s) \in \mathcal{S}$. The root node $\mathbf{r} \in V(G)$ is associated with the initial state s_0, i.e., $\sigma(\mathbf{r}) = s_0$. Consequently, nodes that are associated with a goal or target state are called goal or target nodes. An arc $(u, v) \in A(G)$ between nodes $u \in V(G)$ and $v \in V(G)$ corresponds to a transition $\sigma(v) \in \tau(\sigma(u))$ labeled with transition costs $c(\sigma(u), \sigma(v))$. For convenience, we denote the transition costs of an arc also as the length of the arc. Furthermore, $Z^{\text{sp}}(v)$ indicates for node $v \in V(G)$ the length of the so far shortest known path from $\mathbf{r}$ to node v.Algorithm 2.1

[1]Note that sometimes transition function $\tau(\cdot)$ is slightly differently defined as $\tau : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ by mapping a given state $s \in \mathcal{S}$ to a specific successor state $\tau(s, a)$ when taking action $a \in \mathcal{A}$ from the set of possible *actions* $\mathcal{A}$. We will use this definition in some subsequent chapters and describe it there in more detail. In this section it suffices to use the introduced definition of $\tau(\cdot)$.

demonstrates a general search procedure by maintaining an *open list* of nodes that must be investigated further to find a solution. The search starts by initializing the open list with the root node $\mathbf{r} \in V(G)$ of the search tree associated with s_0. Depth $d(u)$ indicates the number of transitions from s_0 to node u. At each iteration a node u is selected and removed from the open list. The specific selection of a node as well as the implementation of Q depends on the used *search strategy*. Different search strategies will lead to different behaviors of the search algorithm, which will be described in the following subsections. After selecting u we first check if u is associated to a goal state. If this is the case then we return the found solution by considering the shortest path from $\mathbf{r}$ to u. Otherwise, we *expand* node u by considering all successor states $\tau(\sigma(u))$ and insert corresponding newly created nodes into Q. Then we continue by selecting the next node from Q.

Algorithm 2.1: Generic Search Algorithm

```
   Input: initial state $s_0$
1  create root node $\mathbf{r}$ with $\sigma(\mathbf{r}) \leftarrow s_0$ and $Z^{\mathrm{sp}}(\mathbf{r}) \leftarrow 0$;
2  $Q \leftarrow \{\mathbf{r}\}$;
3  for $Q$ not empty do
4      pop $u$ from $Q$;
5      if $\sigma(u)$ is goal state then
6          return solution with minimum cost $Z^{\mathrm{sp}}(u)$;
7      end
8      for $s \in \tau(\sigma(u))$ do                    // expand node $u$
9          create node $v$ with $\sigma(v) \leftarrow s$;
10         $Z^{\mathrm{sp}}(v) \leftarrow Z^{\mathrm{sp}}(u) + c(\sigma(u), \sigma(v))$;
11         $Q \leftarrow Q \cup \{v\}$;
12     end
13 end
14 return no solution found
```

Note that Algorithm 2.1 terminates as soon as a solution is encountered. Depending on the used search strategy this solution may be just a non-optimal solution. Therefore the return statement can be seen as a suggestion and the search may also continue after the first solution was found to obtain a better solution. Furthermore, Algorithm 2.1 generates a search tree, meaning that there may be multiple nodes associated with the same state. Some search algorithms will create a *search graph* where such redundancies are avoided. This is usually done by maintaining a *closed list* in addition to the open list, which contains all states that are already encountered during the search. If a state is re-encountered then an incoming arc is appended to the already existing node that is associated to the re-encountered state, instead of creating a new node for the re-encountered state.

2.2.1 Uninformed Search

Uninformed searches select nodes in a way that does not take any additional problem specific information into account, except $Z^{\mathrm{sp}}(\cdot)$ and $d(\cdot)$. There are different standard search strategies which differ mainly in the order of selecting the next node to expand depending on the implementation of Q from Algorithm (2.1).

Breadth-First Search

In breadth-first search (BFS) the open list is implemented as a first-in, first-out queue. Thus, Algorithm 2.1 always selects the node from Q that was inserted earliest. The algorithm systematically explores all nodes in G with a certain depth d before paths of depth $d+1$ are considered. This strategy, however, in general needs exponential memory size as well as exponential time in $O(b^l)$ where b indicates the largest branching factor over all nodes and l is the length of the first solution found at depth l. To see this consider that the root node on the first level of the search tree generates b nodes for the second level which in turn will generate b^2 nodes for the third level. Thus, if the first solution will be located at level l then at most $\sum_{i=0}^{l} b^i$ nodes will be expanded before finding a solution. However, the search strategy guarantees to find a target state and thus a feasible solution if one exists and this solution will contain the smallest number of transitions. If the transition costs coincide with the number of transitions then this solution is an optimal solution.

Due to the typically high memory requirements, BFS is only applicable for small problem instances where e.g. the state graph is explicitly given or when solutions should be found with the smallest number of arcs. It can rather not be recommended for problems where the state graph has exponential size.

Depth-First Search

The depth-first search (DFS) strategy always selects the last node that was inserted in Q. Thus, Q is implemented as a last-in, first-out stack. This means that Algorithm 2.1 follows always the same path to its completion in G until another alternative path is considered. For DFS the memory requirement is linear, but the time complexity is still exponential. Suppose the search tree has a maximum depth $d_{\max}$ and branching factor b. Then only $d_{\max}$ nodes are expanded and $(b-1)d_{\max}$ un-expanded nodes are in Q until depth $d_{\max}$ is reached the first time. Thus in total $bd_{\max}$ nodes are generated. The time complexity is in the worst case, however, still $O(b^{d_{\max}})$ since the whole search tree may be expanded until a solution is found. If there are many solutions then DFS can be faster than BFS since DFS may expand only a small part of the search tree whereas BFS has to expand all nodes until a certain depth is reached. Note also that Algorithm 2.1 does not specify the order in which successor nodes are added to Q. The search strategy, however, is sensitive to this ordering and informed search strategies will use heuristic information to decide an order, which may decrease the number of expanded nodes until finding a solution.

From a theoretical point of view DFS may never stop if the underlying state graph contains cycles or has infinitely many nodes when using Algorithm 2.1. Note that we do not consider such cases within this thesis, however, if the depth of the search tree is rather large then DFS may not find a solution within reasonable time since it may happen that DFS first follows paths that do not lead to feasible solutions. There are two prominent DFS like search strategies to overcome these disadvantages. *Depth-limited search* imposes a maximum depth limit of the search tree. This guarantees that depth-limited search will finally terminate even if the underlying state graph contains paths with a larger number of transitions. However, the success of depth-limited search strongly depends on the chosen maximum depth limit. If it is chosen too small then depth-limited search will not even find a feasible solution. Therefore, *iterative deepening search* will perform multiple depth-limited searches by increasing systematically the maximum depth limit, starting by one. In this way the algorithm can guarantee that it will eventually find a solution and, similar to BFS, this solution will contain the smallest number of arcs. In particular iterative deepening search is useful if BFS requires a lot amount of memory.

Lowest-Cost-First Search

Methods discussed so far could not guarantee that the first encountered solution by Algorithm 2.1 is an optimal solution with fewest total transition cost. They have not considered any information of transition costs at all. Thus, the lowest-cost-first search will always select the node from Q with smallest Z^{sp}-value; open list Q is therefore implemented as a priority queue. The lowest-cost-first search can guarantee to find an optimal solution if all transition costs are greater than a positive constant, the branching factor is finite and a feasible solution exists. Since all nodes with a smaller Z^{sp}-value as the Z^{sp}-value of the optimal solution will be generated before the optimal solution is found, lowest-cost-first search is exponential in space and time.

2.2.2 Informed Search

In contrast to uninformed search strategies, where besides $Z^{\mathrm{sp}}(\cdot)$ and $d(\cdot)$ no further information is used to guide the search, informed search strategies use some problem specific heuristic information to guide the search. More precisely, they use a *heuristic function* $Z^{\mathrm{h}} : V(G) \to \mathbb{R}_{\geq 0}$ that estimate for each node $v \in V(G)$ the remaining cost-to-go to any target node. Furthermore, we assume that function $Z^{\mathrm{h}}(\cdot)$ maps target nodes to the value zero. Nodes with smaller Z^{h}-values are supposed to be a better choice to expand next than nodes with higher values. We call function $Z^{\mathrm{h}}(\cdot)$ an *admissible heuristic* if it never overestimates the real cost-to-go, i.e., if for each node $v \in V(G)$ the estimated costs $Z^{\mathrm{h}}(v)$ to reach any target nodes are always smaller or equal to the actual costs of a shortest path from v to any goal node. In this case $Z^{\mathrm{h}}(\cdot)$ is a lower bound to the actual costs and we denote $Z^{\mathrm{h}}(\cdot)$ also as $Z^{\mathrm{lb}}(\cdot)$.

Heuristic functions are problem specific such that they need some knowledge of the actual problem to solve in order to compute some estimations of the remaining cost-to-go. This is frequently done by considering a simpler problem that can be solved efficiently and use the optimal cost of the simpler problem as heuristic value. There is usually a tread-off between the amount of time to spend on solving the simpler problem and the quality of the obtained heuristic value.

Search strategies that order the nodes in Q (see Algorithm 2.1) such that always the most promising node is expanded first are called *best-first-search* strategies.

Heuristic Depth-First Search

Heuristic DFS is similar to DFS in the sense that the open list Q from Algorithm 2.1 is implemented as a last-in, first-out stack. However, heuristic DFS uses a heuristic function $Z^{\mathrm{h}}(\cdot)$ to sort successor nodes before they are inserted into Q such that the most promising successor node according to $Z^{\mathrm{h}}(\cdot)$ will be selected next for expansion. In other words, the search strategy always selects locally the best node to select next. However, heuristic DFS has in principle the same disadvantages as DFS since the search may not find a feasible solution in reasonable time if the underlying state graph contains paths with a rather large number of transitions.

Greedy Best-First Search

Greedy best-first search always selects the most promising node of Q according to heuristic function $Z^{\mathrm{h}}(\cdot)$. The open list Q is therefore be implemented as a priority queue in Algorithm 2.1. Again greedy best-first search may suffer if the underlying state graph contains paths with a rather large number of transitions, i.e. a feasible solution may not be found in reasonable time.

A* Search

A* search was introduced by Hart et al. [70] in 1968. The search maintains an open list Q of open nodes that is implemented as priority queue sorted in non-decreasing order according to priority function

$$f_*(u) = Z^{\mathrm{sp}}(u) + Z^{\mathrm{h}}(u) \tag{2.2}$$

where $Z^{\mathrm{sp}}(u)$ is the cost of the so far cheapest path from $\mathbf{r}$ to node u and $Z^{\mathrm{h}}(u)$ is a heuristic function that estimates the remaining cost-to-go from u to any target node. Usually, $Z^{\mathrm{h}}(u)$ is an admissible heuristic, i.e., a lower bound on the remaining cost-to-go. Then priority function $f_*(u)$ can be interpreted as the cost of the cheapest path from $\mathbf{r}$ to any target node through node u.

In principle Algorithm 2.1 can be used to perform A* search with priority function $f_*(\cdot)$. However, this generic search algorithm creates a search tree, meaning that there may by multiple nodes that are associated with the same state, which may cause some redundancies in the search tree since isomorphic sub-trees may emerge multiple times. To save memory, A* is usually implemented as graph search where each encountered state is associated to exactly one node of the generated search graph, as presented by

Algorithm 2.2: A* search

```
   Input: initial state $s_0$, search heuristic $Z^{\mathrm{h}}(\cdot)$
1  create root node $\mathbf{r}$ with $\sigma(\mathbf{r}) \leftarrow s_0$ and $Z^{\mathrm{sp}}(\mathbf{r}) \leftarrow 0$;
2  $Q \leftarrow \{\mathbf{r}\}$;
3  for $Q$ not empty do
4      pop $u$ from $Q$ that minimizes $f_*(u) = Z^{\mathrm{sp}}(u) + Z^{\mathrm{h}}(u)$;
5      if $\sigma(u)$ is goal state then
6          return solution with minimum cost $Z^{\mathrm{sp}}(u)$;
7      end
8      for $s \in \tau(\sigma(u))$ do                                  // expand node $u$
9          if $\nexists v \in V(G) : \sigma(v) = s$ then
10             create node $v \in V(G)$ with $\sigma(v) \leftarrow s$;
11             $Z^{\mathrm{sp}}(v) \leftarrow Z^{\mathrm{sp}}(u) + c(\sigma(u), \sigma(v))$;
12             $Q \leftarrow Q \cup \{v\}$;
13         else if $\exists v \in V(G) : \sigma(v) = s \wedge Z^{\mathrm{sp}}(u) + c(\sigma(u), \sigma(v)) < Z^{\mathrm{sp}}(v)$ then
14             $Z^{\mathrm{sp}}(v) \leftarrow Z^{\mathrm{sp}}(u) + c(\sigma(u), \sigma(v))$;
15             $Q \leftarrow Q \cup \{v\}$;
16         end
17     end
18 end
19 return no solution found
```

Algorithm 2.2. Similar to Algorithm 2.1, A* search selects and removes at each iteration a node u from Q that minimizes $f_*(u)$. If u is not a goal node then u gets expanded by creating all successor states $\tau(\sigma(u))$. This time, however we check for each successor state $s \in \tau(\sigma(u))$ if we have already generated a node $v \in V(G)$ with the same state $\sigma(v) = s$. If not then we proceed as in Algorithm 2.1 be creating a new node that is inserted into Q. Otherwise, if such a node v exists and we found a cheaper path via node u, i.e., $Z^{\mathrm{sp}}(u) + c(\sigma(u), \sigma(v)) < Z^{\mathrm{sp}}(v)$ then we update the Z^{sp}-value of node v accordingly and (re)insert v into Q. Checking if a node with a specific states already exists is usually efficiently implemented as a hash-table that maps states to nodes. Note that Algorithm 2.2 uses the set of already generated nodes $V(G)$ to check if a node with a specific state already exists. Instead of $V(G)$, A* search sometimes maintains in the literature a so-called *closed list* that contains all nodes that got already expanded during the search.

If A* search selects a goal node the first time then the algorithm terminates with a solution that can be obtained from following the predecessor chain back to the root node. Under some conditions, A* search is an admissible search meaning that whenever a feasible solution exists, A* will return the optimal solution. Following theorem gives sufficient conditions to guarantee the admissibility of A* search.

Theorem 2.2.1 (A^* admissibility, Poole and Mackworth [131])
If there is a solution, A^* using heuristic function Z^{h} always returns an optimal solution, if

- the branching factor is finite (each node has a bounded number of neighbors),
- all arc costs are greater than some $\varepsilon > 0$, and
- Z^{h} is an admissible heuristic

Proof. **Part A:** A solution will be found. If the arc costs are all greater than some $\varepsilon > 0$, we say the costs are bounded above zero. If this holds and with a finite branching factor, $Z^{\mathrm{sp}}(u)$ will eventually exceed any finite number for all nodes u in the open list and, thus, will exceed a solution cost if one exists. Because the branching factor is finite, only a finite number of nodes must be expanded before the search could get to this point, but the A^* search would have found a solution by then.

Part B: The shortest path to the first selected goal node corresponds to an optimal solution. Heuristic $Z^{\mathrm{h}}(\cdot)$ is admissible, this implies that the f_*-value of a node on an optimal solution path is less than or equal to the cost of an optimal solution, which, by the definition of optimal, is less than the cost for any non-optimal solution. The f_*-value of a solution is equal to the cost of the solution if the heuristic is admissible. Because an element with minimum f_*-value is chosen at each step, a non-optimal solution can never be chosen while there is a node in Q that belongs to an optimal solution path. So, before it can select a non-optimal solution, A^* will have to pick all of the nodes on an optimal path, including an optimal solution. □

As mentioned above, generates Algorithm 2.2 a search graph where each state is associated to exactly one node. However, it can still happen that at Line 13 a node that was already expanded is reinserted into Q if a cheaper path to that node could be discovered. We call the event, that an already expanded node get selected for expansion again, a *re-expansion*. Such cases force A^* to reevaluate whole parts of the already generated search graph. To avoid such re-expansions we have to consider *consistent heuristics*.

Definition 2.2.1 (Consistent Heuristic)
A heuristic $Z^{\mathrm{h}}(\cdot)$ is consistent if the constraints

$$Z^{\mathrm{h}}(u) \leq \mathrm{cost}(u,v) + Z^{\mathrm{h}}(v) \quad \forall (u,v) \in A(G) \text{ and} \tag{2.3a}$$
$$Z^{\mathrm{h}}(u) = 0 \text{ for each target node } u \tag{2.3b}$$

are satisfied, where $\mathrm{cost}(u,v)$ is the cheapest cost of a path between node u and node v.

Note that a consistent heuristic $Z^{\mathrm{h}}(\cdot)$ satisfies constraint $Z^{\mathrm{h}}(u) \leq \mathrm{cost}(u,v) + Z^{\mathrm{h}}(v)$ for each pair of nodes $u, v \in V(G)$. Furthermore, each consistent heuristic is always an admissible heuristic too.

Theorem 2.2.2
A consistent heuristic $Z^{\mathrm{h}}(\cdot)$ is also admissible.

Proof. By induction. Let $Z^{\mathrm{h}}_*(u)$ denotes the cost of the cheapest path from u to any goal node. To prove admissibility we have to show that $Z^{\mathrm{h}}(u) \leq Z^{\mathrm{h}}_*(u)$ for each node u. Thus let the induction hypotheses be that for an arbitrary node u, $Z^{\mathrm{h}}(u) \leq Z^{\mathrm{h}}_*(u)$ holds.

Base Case: Take a goal node v and consider an arbitrary predecessor node u. Then we have $Z^{\mathrm{h}}(u) \leq \mathrm{cost}(u,v) + Z^{\mathrm{h}}(v) = \mathrm{cost}(u,v) = Z^{\mathrm{h}}_*(u)$ by Definition 2.2.1. Hence, the consistent heuristic $Z^{\mathrm{h}}(u) \leq Z^{\mathrm{h}}_*(u)$ behaves like an admissible heuristic in this case.

Induction Step: Consider an arbitrary node v and an arbitrary predecessor node u of v. Since $Z^{\mathrm{h}}(\cdot)$ is consistent we know that $Z^{\mathrm{h}}(u) \leq \mathrm{cost}(u,v) + Z^{\mathrm{h}}(v)$. By induction step we further know that $Z^{\mathrm{h}}(v) + Z^{\mathrm{h}}_*(v)$ implying that

$$Z^{\mathrm{h}}(u) \leq \mathrm{cost}(u,v) + Z^{\mathrm{h}}(v) \leq \mathrm{cost}(u,v) + Z^{\mathrm{h}}_*(v) = Z^{\mathrm{h}}_*(u) \tag{2.4}$$

where the last equality is true since $\mathrm{cost}(u,v)$ is the cost of the cheapest path from u to v and $Z^{\mathrm{h}}_*(v)$ indicates the cheapest cost to get from v to a goal node. □

Consistent heuristics are sometimes called *monotone heuristics* since the f-values along the paths in G are monotonically non-decreasing. Thus, f-values of nodes that get selected for expansion do not get smaller. If a consistent heuristic is used for Algorithm 2.2 then no re-expansions occur.

Theorem 2.2.3 (Re-expansions)
If Algorithm 2.2 uses priority function $f_*(\cdot) = Z^{\mathrm{sp}}(\cdot) + Z^{\mathrm{h}}(\cdot)$ and heuristic $Z^{\mathrm{h}}(\cdot)$ is a consistent heuristic then no node is re-expanded.

Proof. We proof by contradiction. Suppose that during the expansion of node u we encounter a successor state $s \in \tau(\sigma(u))$ and an already expanded successor node v s.t. $s = \sigma(v)$ and $Z^{\mathrm{sp}}(u) + c(\sigma(u), \sigma(v)) < Z^{\mathrm{sp}}(v)$. Then it must be that $f_*(v) \leq f_*(u)$ since node u was selected before node v and consequently $Z^{\mathrm{sp}}(v) + Z^{\mathrm{h}}(v) \leq Z^{\mathrm{sp}}(u) + Z^{\mathrm{h}}(u)$. From the two inequalities we conclude that

$$\mathrm{cost}(\sigma(u), \sigma(v)) < Z^{\mathrm{sp}}(v) - Z^{\mathrm{sp}}(u) \leq Z^{\mathrm{h}}(u) - Z^{\mathrm{h}}(v) \tag{2.5}$$

which cannot happen if $Z^{\mathrm{h}}(\cdot)$ is consistent, i.e., if $Z^{\mathrm{h}}(u) \leq \mathrm{cost}(u,v) + Z^{\mathrm{h}}(v)$. □

Furthermore, A* search is *optimally efficient* if a consistent heuristic is used. Meaning that up to tie breaking mechanism, there exists no other A* like algorithm using the same consistent heuristic $Z^{\mathrm{h}}(\cdot)$ as A* search that expands a fewer number of nodes. See Dechter and Pearl [47] for a detailed study of optimality criteria of A* search.

Although A* is optimally efficient, the time and space complexity is still exponential in the worst case by $O(b^d)$, where b is the average branching factor and d denotes the solution depth, i.e., the number of transitions to obtain the optimal solution. The practical performance of A*, however, depends on the used heuristic function $Z^{\mathrm{h}}(\cdot)$ since a powerful heuristic can prune away a substantial number of the b^d nodes.

Beam Search

Beam search is a greedy search heuristic that builds a search tree by using a limited best-first search strategy. The first use of beam search (BS) was in a speech recognition system by Lowerre [110] and for image recognition by Rubin [138]. The search tree is created level by level starting with the root node. Thereby a maximum number of β nodes for each level of the search tree is imposed. At each major step, all nodes of the current level are expanded and the newly created nodes are inserted into the next level of the search tree. Then those newly created nodes are sorted according to some heuristic $Z^{\mathrm{h}}(\cdot)$ and the β best nodes are kept. All other nodes are removed from the search tree. Thus, the number of nodes of each level of the search tree is limited by β. According to BS terminology, the set of nodes of the current level is denoted as *beam* B and parameter β is denoted as *beam width*. If β is infinite then BS becomes BFS and if $\beta = 1$ then BS behaves like heuristic depth-first search.

Algorithm 2.3 demonstrates a general BS approach. Note that Algorithm 2.3 terminates as soon as the first goal node is selected for expansion. Another also common termination strategy is to continue the search as long as there is a remaining node to expand, i.e., as long as B is not empty. This has the advantage that a better heuristic solution may be obtained, but requires longer computation time. BS has in general a worst case time and space complexity of $O(\beta d_{\max})$ where $d_{\max}$ is the maximum depth of the search tree. However, in practice the time and space complexity depends strongly on the used heuristic $Z^{\mathrm{h}}(\cdot)$.

Algorithm 2.3: Beam Search

```
   Input: initial state $s_0$, search heuristic $Z^{\mathrm{h}}(\cdot)$, beam width $\beta$
 1 create root node $\mathbf{r}$ with $\sigma(\mathbf{r}) \leftarrow s_0$;
 2 $B \leftarrow \{\mathbf{r}\}$;                                   // current beam
 3 for $B$ not empty do
 4     if $\exists v \in B : \sigma(v)$ is goal state then
 5         return solution with minimum cost $Z^{\mathrm{sp}}(u)$;
 6     end
 7     $B' \leftarrow \emptyset$;
 8     expand all nodes from $B$ and insert newly created nodes in $B'$;
 9     sort nodes in $B'$ according to $Z^{\mathrm{h}}(\cdot)$;
10     $B \leftarrow$ select $\beta$ best nodes of $B'$;
11     remove nodes $B' \setminus B$ from search tree;
12 end
13 return no solution found
```

Limited Discrepancy Search

Limited discrepancy search was first introduced by Harvey and Ginsberg [72] on binary optimization problems. Each state $s \in S$ of such binary problems has only two possible successor states, i.e. $|\tau(s)| \leq 2$. Thus, a search tree generated by a search algorithm will only have a branching factor of two. The idea of limited discrepancy search (LDS) is that a heuristic depth-first search using a greedy heuristic $Z^{\mathrm{h}}(\cdot)$ will frequently lead directly to a feasible heuristic solution. However, this solution is usually not an optimal solution, since $Z^{\mathrm{h}}(\cdot)$ is just a heuristic which may select not always the right successor state to expand further. Depending on the optimization problem, in some cases the heuristic depth-first search will even fail to find any feasible solution. Suppose that there is a given maximum depth of the search tree $d_{\max}$. Then there are only $d_{\max}$ possibilities that $Z^{\mathrm{h}}(\cdot)$ will fail one single time at some point and prefer the wrong successor state, leading to a non-optimal solution or to no feasible solution at all. If $Z^{\mathrm{h}}(\cdot)$ fails two single times then there are only $d_{\max}(d_{\max}-1)/2$ possibilities. Now, LDS allows during the search a small number k of so-called *discrepancies*, where the search algorithm decides against heuristic $Z^{\mathrm{h}}(\cdot)$. Thus, LDS explores in a systematic way the paths on the search tree that differs from the suggested path by $Z^{\mathrm{h}}(\cdot)$ by at most k discrepancies as described by Algorithms 2.4 and 2.5. The latter is called by Algorithm 2.4 iteratively by increasing k each time. Algorithm 2.5 performs recursively a depth-first traversal strategy exploring all paths with maximum k discrepancies. At the beginning Algorithm 2.5 behaves with $k = 0$ like heuristic depth-first search. With larger values for k Algorithm 2.5 explores larger and larger regions of the search space until k reaches finally the maximum depth of the search tree $k_{\max}$ where Algorithm 2.5 carries out an exhaustive search. Thus, Algorithm 2.4 is a complete search algorithm that guarantees to finally find a feasible solution if one exists.

Algorithm 2.4: Limited Discrepancy Search, adapted from [72]

```
   Input: initial state $s_0$, search heuristic $Z^{\mathrm{h}}(\cdot)$
1  for $k \leftarrow 0$ to $d_{\max}$ do
2      create new root node $\mathbf{r}$ with $\sigma(\mathbf{r}) \leftarrow s_0$;
3      $u \leftarrow$ LDS-Probe($\mathbf{r}$, $k$, $Z^{\mathrm{h}}(\cdot)$);
4      if $u \neq \perp$ then return $u$;
5  end
6  return $\perp$
```

For the sake of simplicity Algorithm 2.4 creates a whole new search tree at each major iteration. Of course, this can be avoided by maintaining only one search tree, which is extended accordingly by each call of Algorithm 2.5. Nevertheless, even when only one search tree is maintained, LDS has to revisit large parts of the search tree by every call of Algorithm 2.5. This drawback is avoided by the improved limited discrepancy search (ILDS), introduced by Korf [102]. To this end, the remaining depth of the search tree is

Algorithm 2.5: LDS-Probe, adapted from [72]

```
  Input: node $u$, maximum discrepancies $k$, search heuristic $Z^{\mathrm{h}}(\cdot)$
1 if $\sigma(u)$ is goal state then return $u$ ;
2 else if $\tau(\sigma(u)) = \emptyset$ then return $\perp$ ;
3 expand node $u$ by creating two new nodes $u_1$ and $u_2$ s.t. $Z^{\mathrm{h}}(u_1) \leq Z^{\mathrm{h}}(u_2)$;
4 if $k = 0$ then return LDS-Probe($u_1$, 0, $Z^{\mathrm{h}}(\cdot)$);
5 else
6     $v \leftarrow$ LDS-Probe($u_2$, $k-1$, $Z^{\mathrm{h}}(\cdot)$);
7     if $v \neq \perp$ then return $\perp$;
8     return LDS-Probe($u_1$, $k$, $Z^{\mathrm{h}}(\cdot)$);
9 end
```

additionally forwarded to Algorithm 2.5 allows the search to prune some paths that are already investigated at previous iterations.

Karoui et al. [93] proposed a further improvement of LDS for instances that are unsolvable. Following observation is used to terminate LDS early in such cases. If a call of Algorithm 2.5 with k discrepancies did not find any solution and did not consume the full quota of discrepancies k, i.e. the if-statement at Line 4 is never true during the recursion, then subsequent calls of Algorithm 2.5 with increased values of k will not find a feasible solution too. Therefore, in those cases LDS can be terminated early and the main loop in Algorithm 2.4 do not need to iterate over the whole range from 0 to $d_{\max}$.

Another important detail of LDS-based approaches is the order in which discrepancies are taken. The original work of Harvey and Ginsberg [72] assumes that the used search heuristic will make wrong suggestions rather at the beginning of the search when just a few decisions have been made. Consequently, Algorithm 2.5 investigates paths of the search tree first, where discrepancies are taken near the root node. However, this assumption was inadvertently lost when Korf [102] proposed ILDS by taking discrepancies as late as possible. That motivate Prosser and Unsworth [132] to compare ILDS when taking discrepancies late or early during the search process. They concluded that LDS-based approaches should take discrepancies rather early.

Both, LDS and ILDS can be extended to optimization problems with a lager branching factor then two. However, there is no single unique way how this can be achieved. For instance, Karoui et al. [93] consider problems with larger branching factors by taking the i-th branch, ordered by heuristic $Z^{\mathrm{h}}(\cdot)$, as the i-th discrepancy. Furcy and Koenig [58] follow a different strategy by taking the first branch with the lowest Z^{h}-value as zero discrepancy and each other branch as one discrepancy.

Besides various combination of LDS with other search techniques [9, 160] (e.g. iterative deepening search) in the literature, we will in particular focus on the combination of BS and LDS in Chapter 5 of this thesis. Therefore, let us describe here in more detail a possible combination of both search techniques as suggested by Furcy and Koenig [58].

Their algorithm, called *beam search using limited discrepancy backtracking*, does not only expand one node at each call of Algorithm 2.5, but rather the algorithm expands a set of β nodes similar to BS. The newly created nodes are sorted according to $Z^{\mathrm{h}}(\cdot)$ and partitioned into slices such that the first slice contains the first β best nodes and so on. Taking the i-th discrepancy means in this context that the i-th slice is expanded. The term *backtracking* in the name of their algorithm refers to the ability to undo decision, i.e. to take a discrepancy during the search process.

2.3 Exact Methods

In this section we provide an overview as well as basic concepts of the most prominent exact solution approaches. For optimization problems that are known to be solvable in polynomial time there is usually a specific algorithm available, which constructs a globally optimal solution by exploiting some problem specific properties. This section concentrates more on NP-hard optimization problems which cannot be solved in polynomial time, unless P=NP. The simplest method to obtain a globally optimal solution for such problems is to systematically enumerate the set of all feasible solutions $\mathscr{S}$ of a COP. However, this method is only viable for very small instance sizes due to the *combinatorial explosion* of the number of feasible solutions with respect to the instance size in general. Therefore exact methods try to consider larger regions of the solution space only implicitly by ruling out regions of the solution space where it is guaranteed that no new best solution can be found. Almost all exact methods are based on the *divide-and-conquer* principle where the solution space is partitioned into smaller disjoint subspaces in a recursive way. The partitioning can be represented as a search tree and ruling out regions means to prune the search tree. Therefore, the effectiveness of such approaches depends on the efficiency of the pruning mechanism. Next we will discuss the basic branch and bound (BB) approach which performs pruning based on lower and upper bounds on the objective value.

2.3.1 Branch and Bound

The branch and bound (BB) approach partitions the solution space of a COP into smaller disjoint subspaces in a recursive way. The partitioning, also called *branching*, is done by setting decision variables to a fixed value or by adding additional constraints. A naive approach will recursively partition the subspaces further and further until single solutions are obtained or the subspaces are empty. An effective BB approach aims to stop parts of this recursive partitioning as soon as possible, by proving that the optimal solution cannot be found in the considered subspace of feasible solutions. Consequently, splitting this subspace into further smaller disjoint subspaces will be needless. This partitioning process can be represented as a search tree where the root node corresponds to the whole solution space and child nodes to disjoint subspaces. Search methods from Section 2.2 can be used to traverse the search tree in a systematic way. Thereby, nodes are pruned from the search tree where in the corresponding subspaces no better solution can be found than the current best known solution. The pruning is done by deriving lower and upper

Algorithm 2.6: Generic Branch-and-Bound [28]

```
Input: COP instance $(\mathscr{S}, f)$
Output: globally optimal solution $x^*$
1  $Q \leftarrow \{\mathscr{S}\}$;                                  // list of open subspaces
2  $U \leftarrow \infty$;                                         // global upper bound
3  for $Q$ not empty do
4      pop subspace $P$ from $Q$;
5      $L_P \leftarrow$ derive lower bound from $P$;
6      if $L_P < U$ then                                       // pruning
7          $x \leftarrow$ search for heuristic solution in $P$;
8          if $\exists x \wedge f(x) < U$ then
9              $x^* \leftarrow x$; $U \leftarrow f(x)$;                 // found new best solution
10         end
11     end
12     if $L_P < U$ then
13         partition $P$ into $P_1, \dots, P_k$ disjoint subspaces;      // branching
14         $Q \leftarrow Q \cup \{P_1, \dots, P_k\}$;
15     end
16 end
17 return $x^*$
```

bounds for each subspace of $\mathscr{S}$. For minimization problems an upper bound corresponds to any feasible solution that is contained in the corresponding subspace. Algorithm 2.6 shows the principle of BB by maintaining a list of open subspaces that needs to be considered further, and a currently best known upper bound U. At each iteration a subspace P is taken from Q and a lower bound on the objective value is derived. If the lower bound is smaller than U then there may still be a better solution in P then U. Therefore we search for a heuristic solution in P by using a problem dependent heuristic procedure. If we find a solution x that has a better objective value than U we update x^* and U accordingly. Afterwards we check again if the derived lower bound is smaller than U. If this is the case then there may be still a better solution in P and we start to partition P into disjoint subspaces which are added to Q. Otherwise if the derived lower bound is greater or equal then U we proved that there is no better solution contained in subspace P. Consequently, we do not consider P further and continue with by selecting the next open subspace from Q.

There are major non-trivial design decisions for devising an effective BB algorithm. Two of the most important aspects are the ways to derive lower and upper bounds for subspaces. Strong bounds allow the BB in general to prune nodes that are close to the root node, which has usually a substantial impact on the computation time of the whole algorithm. Other important choices are the selection of the next subspace from Q or the partitioning of the current search space.

2.3.2 Mathematical Programming

In the section we consider only basics of linear and (mixed) integer linear programming, as these are used in the context of this thesis by following the concepts and ideas of Bertsimas and Tsitsiklis [19], Nemhauser and Wolsey [125], and Wolsey [166]. Note, however, that the field of mathematical programming includes many other advanced techniques like quadratic programming, semidefinite programming, or stochastic programming. The idea behind mathematical programming is to formulate an optimization problem in a mathematical way by using decision variables that are restricted by a set of (in our case) linear inequalities. Such formulations can then be solved by powerful general purpose solvers. In Section 2.3.2 we consider linear programming (LP) formulations that consists of continuous decision variables. Such formulations have the advantage that the can be efficiently solved, but they can only represent a small subset of optimization problems. To formulate COPs we have to consider mixed integer linear programming (MILP) formulations, in Section 2.3.2, that use discrete decision variables. However, MILP formulations are in general NP-hard.

Linear Programming

A linear programming (LP) problem can be stated in the form

$$\min \mathbf{c}'\mathbf{x} \tag{2.6a}$$

$$\text{subject to } \mathbf{A}\mathbf{x} \geq \mathbf{b}, \tag{2.6b}$$

$$\mathbf{x} \in \mathbb{R}^n \tag{2.6c}$$

with n continuous decision variables in the form of vector $\mathbf{x} \in \mathbb{R}^n$, m linear constraints expressed by coefficient vector $\mathbf{b} \in \mathbb{R}^m$ and coefficient matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ as well as a linear objective function $f(\mathbf{x}) = \mathbf{c}'\mathbf{x}$ given by cost vector $\mathbf{c} \in \mathbb{R}^n$. The set of feasible solutions can be expressed by $\mathscr{S} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \geq \mathbf{b}\}$. The goal is to find a globally optimal solution $\mathbf{x}^* \in \mathscr{S}$ that minimizes $f(\cdot)$. To solve the LP (2.6a)-(2.6c) we first have to consider some important geometrical interpretations of a LP problem. The set of feasible points $\mathscr{S}$ forms a convex polytope which is described by the linear inequalities of Equation (2.6b).

Figure 2.1a shows an example of a convex polytope, completely described by four linear inequalities. If the polytope is not empty then there exists an optimal solution to the LP being an extreme point of the polytope. Vector $\mathbf{x} \in \mathscr{S}$ is an extreme point of polytope $\mathscr{S}$ if there do not exists two vectors $\mathbf{y}, \mathbf{z} \in \mathscr{S}$, both different from $\mathbf{x}$, and a scalar $\lambda \in [0, 1]$ such that $\mathbf{x} = \lambda\mathbf{y} + (1-\lambda)\mathbf{z}$ [19]. Informally spoken, the optimal solution of a LP problem is located on one of the corner points of the convex hull of the polytope described by the LP's set of linear inequalities (see Figure 1.2a).

Linear programming problems of the form (2.6a)-(2.6c) were first independently considered by Leonid Kantorovich, Frank L. Hitchcock, and George B. Dantzig in the years between 1939 and 1946 [45]. Perhaps the best known solving approach to find optimal

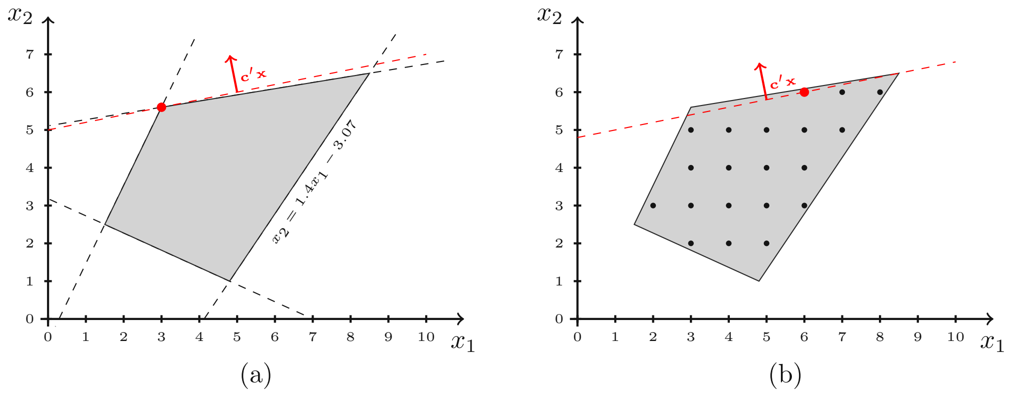


Figure 2.1: (a): A LP problem with two decision variables $\mathbf{x} = (x_1, x_2)$ and four linear inequalities. The gray area depicts the set of feasible solutions, a two dimensional polytope, of the LP problem. The optimal solution, represented by a red dot, is located at one of the corners of the polytope. The red line represents the cost function (or more precisely a level set of the cost function) and the red arrow indicates the direction in which we are optimizing. (b): An integer linear programming (ILP) problem with the same variables and constraints as in (a), but with additional integrality constraints. The set of feasible solutions is depicted by the black dots whereas the gray area represents the set of feasible solutions of the LP relaxation. The optimal solution is not located at one of the polytope's corners anymore.

solutions is the *simplex method* proposed by Dantzig [44]. This approach starts at an extreme point of the polytope and then traverse along the convex hull in a direction which decreases the objective value towards the next extreme point. Since the number of extreme points is finite this method will eventually find the optimal solution. Although modern variants of the simplex method are highly efficient in practice, there are some exceptions where they may need an exponential number of steps until an optimal solution is found [101]. Nevertheless, the simplex method is widely used in leading LP solvers due to an excellent average case performance. Another approach that is also present in many solvers is the interior-point algorithm with a guaranteed polynomial computation time [92]. From a theoretical point of view, the ellipsoid algorithm also is of interest which has a polynomial computation time too [98].

Mixed Integer Linear Programming

Linear programming problems belong to the class of continuous optimization problems and cover a wide range of practical problems. To express (NP-hard) COPs, however, we require mixed integer linear programming (MILP) formulations of the general form

$$\min \mathbf{c}'\mathbf{x} + \mathbf{h}'\mathbf{y} \tag{2.7a}$$
$$\text{subject to } \mathbf{Ax} + \mathbf{Gy} \geq \mathbf{b}, \tag{2.7b}$$
$$\mathbf{x} \in \mathbb{R}^n_{\geq 0}, \tag{2.7c}$$
$$\mathbf{y} \in \mathbb{Z}^p_{\geq 0}, \tag{2.7d}$$

with p additional discrete decision variables represented by vector $\mathbf{y} \in \mathbb{Z}^p_{\geq 0}$, coefficient matrix $G \in \mathbb{R}^{m\times p}$ and cost coefficient vector $\mathbf{h} \in \mathbb{R}^p$. The feasible set of a COP instance $(\mathscr{S}, f)$ is described by

$$\mathscr{S} = \left\{(\mathbf{x},\mathbf{y}) \in \mathbb{R}^n_{\geq 0} \times \mathbb{Z}^p_{\geq 0} \mid \mathbf{Ax} + \mathbf{Gy} \geq \mathbf{b}\right\} \tag{2.8}$$

and the objective function is $f(\mathbf{x},\mathbf{y}) = \mathbf{c}'\mathbf{x}+\mathbf{h}'\mathbf{y}$. Due to the integrality constraints (2.7d), MILP problems are in general NP-hard. If a MILP formulation consists only of integer decision variables then we call this formulation also integer linear programming (ILP) formulation. See Figure 2.1b for an example of a graphical interpretation of an ILP. A common way to solve MILP problems is to utilize BB based approaches as described in Section 2.3.1. There we pointed out that for a BB approach several major design decision have to be made. First we require some mechanism to compute lower bounds in order to prune subproblems. A common way to obtain lower bounds is to consider *relaxations*, where some constraints of a problem are omitted or weakened such that the problem is simpler and can be efficiently solved. Solutions of relaxations are not necessarily a feasible solution to the original problem, but they provide lower bounds. For MILP problems a natural relaxation is to omit the integrality constraints (2.7d) and make the variables continuous. The obtained relaxation is a LP problem denoted as *LP relaxation* which can be efficiently solved. Clearly the set of feasible solutions of a LP relaxation is a superset of the set of feasible solutions of the original MILP problem. Thus, every MILP solution is also a solution of the LP relaxation. Moreover, if a globally optimal solution of the LP relaxation is integral, i.e., all decision variables are integral then this solution is also a feasible optimal solution of the MILP problem.

A second major design decision for BB approaches is the way how problems are partitioned into smaller disjoint subproblems. For MILP problems the branching is typically done by selecting a decision variable with a fractional value in the solution of the LP relaxation y_j^{LP} and creating the first subproblem with the additional constraint $y_j \leq \lfloor y_j^{\mathrm{LP}} \rfloor$ and the second subproblem with constraint $y_j \leq \lceil y_j^{\mathrm{LP}} \rceil$. The LP relaxations of these subproblems can be later again solved by a LP solver possible yielding increased lower bounds.

Algorithm 2.7 depicts the LP-based BB algorithm for MILP problems [42]. The list of open subproblems is initialized with the set of feasible solutions of the LP relaxation of the MILP problem. At each iteration a set of feasible LP solutions p is selected and removed from Q. First, the corresponding optimal solution $(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}})$ is computed. If p does not contain any feasible solution then we prune the current subproblem by infeasibility. Otherwise, if the obtained solution $(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}})$ is worse than the globally best known

Algorithm 2.7: LP-based Branch-and-Bound [42]

```
   Input: a MILP of the form $\min\{\mathbf{c}'\mathbf{x} + \mathbf{h}'\mathbf{y} \mid (\mathbf{x},\mathbf{y}) \in \mathscr{S}\}$
   Output: globally optimal solution $(x^*, y^*)$
 1 $Q \leftarrow$ {linear programming relaxation of $\mathscr{S}$};
 2 $U \leftarrow \infty$;                                      // global upper bound
 3 for $Q$ not empty do
 4     pop $p$ from $Q$;
 5     obtain optimal LP solution $(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}})$ for $p$;
 6     if $p = \emptyset$ then prune $p$ by infeasibility;
 7     else if $f(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}}) > U$ then prune $p$ by bound;
 8     else if $(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}}) \in \mathscr{S}$ then           // $\mathbf{y}^{\mathrm{LP}}$ part is integral
 9         if $f(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}}) < U$ then
10             update upper bound $U \leftarrow f(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}})$;
11             update incumbent $(\mathbf{x}^*, \mathbf{y}^*) \leftarrow (\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}})$;
12         end
13         prune $p$ by optimality;
14     else
15         choose an integer variable $y_j$ that is still fractional;
16         $Q \leftarrow Q \cup \{(\mathbf{x},\mathbf{y}) \in p \mid y_j \leq \lfloor y_j^{\mathrm{LP}} \rfloor\} \cup \{(\mathbf{x},\mathbf{y}) \in p \mid y_j \leq \lceil y_j^{\mathrm{LP}} \rceil\}$;
17     end
18 end
19 return $(\mathbf{x}^*, \mathbf{y}^*)$
```

upper bound U then we prune the subproblem since no better solution can be obtained by considering p further. Next, we check if $\mathbf{y}^{\mathrm{LP}}$ of the obtained solution $(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}})$ is integral. If this is the case than $(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}})$ is a feasible solution w.r.t. the MILP problem and we have found the globally optimal solution of the current subproblem p. Thus, we do not need to consider p further and can prune the subproblem by optimality. If the feasible solution $(\mathbf{x}^{\mathrm{LP}}, \mathbf{y}^{\mathrm{LP}})$ is better than the current incumbent solution we update the incumbent solution and the global best upper bound accordingly. Finally, if the obtained solution contains decision variables $\mathbf{y}^{\mathrm{LP}}$ with fractional values and is better than U then we branch the current subproblem by creating two new problems as described above.

2.3.3 Constraint Programming

Constraint programming (CP) relies on a problem formulation using decision variables and constraints like MILP, but is not restricted to linear constraints and can express substantially more complex constraints. It is primarily targeted towards constraint satisfaction problems (CSPs) which do not have an objective function and where the goal is to obtain any feasible solution that satisfies all constraints. Thus, a CSP has the form of $(\mathbf{x}, \mathcal{C}, \mathcal{D})$ with n decision variables expressed by vector $\mathbf{x} \in \mathcal{D}$ with a corresponding variable domain $\mathcal{D} = \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n$ where domain $\mathcal{D}_i$ corresponds to the i-th variable x_i. Moreover, the variables are constrained by constraint set $\mathcal{C}$. For an overview of CP see the Handbook of Constraint Programming by Rossi et al. [137].

There are two fundamental methods that are combined to solve CSPs: *constraint propagation* and *tree search*. The latter is related to the traversal strategies which we already discussed in Section 2.2. The idea behind constraint propagation is to reduce the search space by removing values from the domains of decision variables that cannot be part of any feasible solution. For instance, most readers will be familiar with this concept when solving Sudokus or other logical puzzles from newspapers. Constraint propagation tries to achieve some level of *local consistency*. We will describe the three most common levels in the following. The simplest form of local consistency is the so-called *node consistency* that is applied to domains of decision variables belonging to unary constraints, i.e., to constraints where only one decision variable is involved. Thus, all values are removed from the variable's domain that are in conflict with the unary constraint. The next level of local consistency is *arc consistency* involving two decision variables and a binary constraint. We say that a variable is arc consistent with another variable w.r.t. a binary constraint if there exists for each value of the domain of the first variable at least one value of the domain of the second variable such that the binary constraint can be fulfilled. Values are removed from the domain of the first variable for which no valid assignment of the second variable exists. This process is iteratively applied to all pairs of decision variables for which binary constraints exist until no domain can be further reduced. *Path consistency* generalizes arc consistency by considering a third decision variable and all corresponding binary constraints. Powerful constraint propagation algorithms are known for achieving different levels of local consistency, e.g. the AC-3 algorithm by Mackworth [111].

Constraint propagation is performed until no domain of a decision variable can be further reduced by considering any of the described local consistency levels. There are three possible outcomes after the constraint propagation: (1) there exists at least one variable with an empty domain; in this case the CSP is not feasible and no solution exists, (2) each domain is reduced to exactly one value in which case a feasible solution is obtained, and (3) no domain is empty and some domains contain more than one value. In this last case we have to continue the search to find a feasible solution or prove infeasibility. This is done by branching, i.e., selecting one of the decision variable and partition the search space into disjoint subspaces which are then considered further. As described in Section 2.2 different traversal strategies can be applied to decide which subproblem to consider next. As soon as a feasible solution is found the search is finished. To prove that no feasible solution exists each branch of the search must be closed.

To solve COPs with objective function $f(\cdot)$ a sequence of CSPs is usually solved by adding a variable x_{n+1} and an additional constraint $x_{n+1} = f(\mathbf{x})$ to the problem which is denoted as *objective constraint*. Hence, variable x_{n+1} expresses the objective value of objective function $f(\cdot)$. The corresponding variable domain $\mathcal{D}_{n+1}$ contains all possible objective values. In the literature there are several proposed ways how to solve such problems in the context of constraint programming. For instance Van Hentenryck [158] proposed a BB scheme based on constraint programming where initially a tree search is used to find a first incumbent solution $\mathbf{x}^*$. Afterwards the COPs is augmented by the constraint $x_{n+1} < f(\mathbf{x}^*)$ to exclude solutions that are not better than $\mathbf{x}^*$. If the augmented COPs has no feasible solution then $\mathbf{x}^*$ is a globally optimal solution of the origin COP. Otherwise, $\mathbf{x}^*$ is updated to the newly found better solution and a corresponding constraint is again added to the COP. This process is repeated until no feasible solution can be found. It is important that constraint propagation techniques are applied to the objective constraint in order to get an effective solving approach for COPs.

2.3.4 Dynamic Programming

The dynamic programming (DP) approach was developed by Bellman [10, 11] and is based on the divide-and-conquer principle. Hence, a problem is recursively divided into multiple simpler subproblems until the subproblems are simple enough such that they can be solved. The solution of the subproblems are then used to solve the whole problem. To efficiently apply the DP approach, subproblems must be *overlapping*. Then solutions of subproblems are stored to reuse them for solving other subproblems. In this way DP avoids solving the same subproblem multiple times. Note that this is contrary to other exact methods that are based on the divide-and-conquer principle, e.g. the BB approach from Section 2.3.1 where subproblems are assumed to be disjoint. Another important condition to efficiently apply DP is if the problem exhibits an *optimal substructure*, i.e. if an optimal solution of the problem can be constructed by optimal solutions of its subproblems. A prominent example of an optimization problem that exhibits an optimal substructure is finding the shortest path on a graph width non-negative edge weights. If the shortest path from node v_1 to node v_4 passes though node v_2 and node v_3 then the shortest path from v_1 to v_3 must pass through v_2 too. For a more detailed introduction we refer to textbooks [18, 48].

Dynamic programming formulations of a COP are usually defined in terms of recursive equations which are based on the state space representation as defined in Section 2.2. Thus, they consist of a state space $\mathcal{S}$ and transition function $\tau(\cdot)$ such that a state $s \in \mathcal{S}$ represents a subproblem and $\tau(s)$ maps to successor states that represent simpler subproblems as the subproblem represented by s. The initial state $s_0 \in \mathcal{S}$ represents the whole problem and cost function $c(s, s')$ maps to the costs that occurs when taking state s' after state s. The total minimum cost that occur when considering a subproblem represented by state s can be expressed by

$$Z^*(s) = \min\{Z^*(s') + c(s, s') \mid s' \in \tau(s)\} \tag{2.9}$$

where $Z^*(s_0)$ denotes the minimum total costs of the whole considered COP. The attraction of DP is that the recursive Equation (2.9) can be efficiently solved due to the overlapping property of subproblems and their optimal substructure by storing into a table already encountered states and their so far best known Z^*-value. Note that formulation (2.9) is based on the notation introduced in Section 2.2 and may not be the most general form of DP formulations. Furthermore, the states of a DP formulation are frequently partitioned into so-called stages such that a transition from one stage to a subsequent stage represents a variable assignment to a specific value.

A well known example where DP can be successfully applied is the discrete 0–1 knapsack problem, which can be solved by a DP-based pseudo-polynomial algorithm [113]. Another prominent algorithm using the ideas of DP is the Floyd-Warshall algorithm to compute all pairs of shortest paths in a graph [55, 162].

2.4 (Meta-)Heuristic Methods

Section 2.3 discusses exact methods that are able to find proven optimal solutions for even NP-hard optimization problems, however, with an exponential worst case complexity. Therefore, such methods are only applicable for small- and middle-sized instances. Industrial relevant problems have usually to deal with larger instance sizes that can frequently not be solved with exact methods in reasonable time. Apart from that, in industry it is often not so important to obtain a globally optimal solution as long as the obtained solution is near enough to an optimal solution. Therefore, heuristics (from Greek *εὑρίσκω* "I find, discover") try to efficiently find solutions near the optimal solution by searching through promising parts of search space $\mathscr{S}$. However, heuristic approaches usually can not provide any guarantee on the quality of the found solution.

Heuristics range from *constructive heuristics* over *improvement heuristics* to *metaheuristics*. The former assembles solutions from scratch within a polynomial number of steps. On the contrary, improvement heuristics try to efficiently improve a given solution by making small changes of the given solution. Finally, metaheuristics provide a powerful problem independent description on a higher level to find near optimal solutions. They describe a wide range of ideas how to combine different constructive and improvement heuristics to efficiently search though the COP's solution space. In general, metaheuristics balance between two mechanism: *intensification* and *diversification*. In intensification phases metaheuristics try to explore similar solutions to a currently considered (incumbent) solution in order to find better ones. This is usually achieved with variants of local search (LS)-based approaches. If the intensification phase does not find further improved solutions then the diversification phase guides the search into a different promising region of the search space. This is usually done in a non-deterministic way.

The field of metaheuristics has been extensively studied over the last 50 years. Consequently, a lot of different metaheuristics exists and there are several ways how to classify them [29, 152]. One meaningful way is to differentiate between *trajectory based methods* and *population based methods*. The former follows a single trajectory through the search space by moving from one solution to another (better) solution. Such metaheuristics have in common that they find some kind of global optima within the intensification phase and then try to escape that local optima within the diversification phase. Examples of such methods are simulating annealing (SA), tabu search (TS), or variable neighborhood search (VNS). One of the first metaheuristics, proposed by Kirkpatrick et al. [100] is SA, which tries to escape local optima by mimic the annealing process of crystalline solid. At each diversification phase a random solution is selected, which is accepted with a certain probability even if the selected solution is worse than the current incumbent solution. The acceptance probability depends on the quality of the selected solution as well as on the elapsed time such that worse solutions are less accepted over time. Another prominent metaheuristic is TS. Note that the term metaheuristic was coined by Glover [62] when introducing TS. This method maintains a tabu list of previously found solutions in order to prevent the search from visiting these solutions again as long as they are on the tabu list. It explores the search space in a LS-based way by accepting also possible worse solutions that are not in the tabu list. In this way, the search prevents to stuck at local optima. The tabu list is usually limited in size such that solutions are removed from it as soon as a new solution should be entered and the tabu list will thereby get too large. The metaheuristic VNS will be discussed in more detail in Section 2.4.4.

On the contrary, population based methods improve multiple solutions simultaneously and are often inspired by nature. For instance, Dorigo et al. [52] proposed the metaheuristic ant colony optimization (ACO), which is inspired by ants that are seeking a source of food. If an ant finds such a source then the ant will lay down a pheromone trail from the source back to their colony. Consequently, other ants will follow the pheromone trail too. The process of finding food as a metaphor for constructing a feasible solution from scratch. Another nature inspired metaheuristic is genetic algorithm (GA) which mimic the process of natural selection by selecting only the best solutions of a set of solution population at each iteration. Then those selected solutions are used to create new solutions by recombination and mutation. A further population based metaheuristic is particle swarm optimization (PSO), proposed by Kennedy and Eberhart [97], which is inspired by the movements of a bird flock or a fish swarm.

Note that Sörensen [151] states that in the last two decades the field of combinatorial optimization was flooded by “novel” metaheuristics that are inspired from biology, physics, chemistry, or in general from nature [54]. However, there is evidence that a lot of these novel metaheuristics do not provide any new solving concepts that are interesting from a scientific point of view. For example, Weyland [164] showed that harmony search [60], which is inspired from musicians playing together and has lead to massive follow-up research, is just a special case of the earlier introduced evolution strategies [20], where all concepts are relabeled accordingly.

We just mentioned a few very prominent metaheuristics. There are much more proposed approaches and discussing all of them will be clearly out of the scope of this thesis. For a comprehensive overview we refer to textbooks [28, 29, 152]. In the remaining of this section we are content to discuss in more detail only the heuristics used within this thesis. We start in Section 2.4.1 with constructive heuristics followed by basic improvement heuristics which are covered by Sections 2.4.2 and 2.4.3. Finally, Section 2.4.4 describes the metaheuristic VNS.

2.4.1 Constructive Heuristics

The simplest heuristic techniques are construction algorithms where solutions are assembled step by step. Constructive heuristics are usually to a high extent problem specific algorithms and the author of this thesis is not aware of any classification of construction algorithms. There is typically a set of solution components that compose a feasible solution. A construction algorithm starts with an empty solution and selects at each step a promising solution component to extend the current partial solution until a feasible solution is obtained. The crucial points are the selection of the next solution component and the way a solution is extended by it. Usually there are multiple ways to extend a partial solution by a given solution component. The algorithm can check at each step each possible solution component and each possible way to extend the current partial solution with it and take then that component that increases the objective function at least. This may, however, too time expensive. Therefore, *greedy construction algorithm* choose the next solution component in a greedy way by ranking the components according to some criterion. Constructive heuristics do in general not lead to an optimal solution. Note that some of the search algorithms from Section 2.2.2 can be seen as typical (advanced) greedy construction algorithms, e.g. the heuristic depth-first search, BS, or LDS. Frequently, constructive heuristics are also randomized by selecting next solution component at random or by breaking ties at random.

2.4.2 Local Search

The basic idea of local search (LS) is to start from an existing solution and move to another better solution by scanning a well-defined set of neighbor solutions. This process is continued until no better solution can be found among the current considered set of neighbor solutions. To define this more formally let us start with the definition of a *neighborhood structure*.

Definition 2.4.1 (Neighborhood Structure, adapted from Blum and Raidl [28])
Let $(\mathscr{S}, f)$ be an instance of a COP. A neighborhood structure is a mapping

$$\mathcal{N} : \mathscr{S} \to 2^{\mathscr{S}} \tag{2.10}$$

that maps each feasible solution $\mathbf{x} \in \mathscr{S}$ to a set of neighbor solutions $\mathcal{N}(\mathbf{x}) \subseteq \mathscr{S}$, also denoted as neighborhood of $\mathbf{x}$.

A neighborhood structure $\mathcal{N}(\cdot)$ is frequently defined implicitly be stating the changes that must be applied to solution $\mathbf{x}$ to get all neighbor solutions $\mathcal{N}(\mathbf{x})$. The definition of neighborhood structures allows us to define locally optimal solutions.

Definition 2.4.2 (Locally Optimal Solution, adapted from Blum and Raidl [28])
Let $(\mathscr{S}, f)$ be an instance of a COP and $\mathcal{N}$ be a neighborhood structure. A locally optimal solution $\hat{\mathbf{x}} \in \mathscr{S}$ with respect to $\mathcal{N}$ satisfies

$$f(\hat{\mathbf{x}}) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{N}(\hat{\mathbf{x}}). \tag{2.11}$$

Algorithm 2.8: Local Search

```
   Input: An initial solution $\mathbf{x}_{\text{init}}$, a neighborhood structure $\mathcal{N}$
   Output: A probably improved solution $\mathbf{x}$ w.r.t $\mathcal{N}$
1  $\mathbf{x} \leftarrow \mathbf{x}_{\text{init}}$;
2  repeat
3      select $\mathbf{x}' \in \mathcal{N}$;
4      if $f(\mathbf{x}') < f(\mathbf{x})$ then
5          $\mathbf{x} \leftarrow \mathbf{x}'$;
6      end
7  until stopping criteria satisfied;
8  return $\mathbf{x}$
```

A locally optimal solution $\hat{\mathbf{x}}$ is called strictly locally optimal if

$$f(\hat{\mathbf{x}}) < f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{N}(\hat{\mathbf{x}}). \tag{2.12}$$

If the context is clear then a locally optimal solution $\hat{\mathbf{x}}$ is sometimes also called just *local solution*. Algorithm 2.8 aims to find such a locally optimal solution $\hat{\mathbf{x}}$ with respect to a neighborhood structure $\mathcal{N}(\cdot)$. The algorithm starts from a given initial solution $\mathbf{x}_{\text{init}}$ and selects at each iteration a solution $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$ from the neighborhood of the current incumbent solution $\mathbf{x}$. Solution $\mathbf{x}'$ becomes the current incumbent solution if $\mathbf{x}'$ is better than $\mathbf{x}$. Usually the LS procedure terminates if no better solution can be obtained from the neighborhood of the current incumbent solution. Note that in this case the returned solution is a locally optimal solution according to Definition 2.4.2. Additionally, the algorithm may terminate if a certain time limit is exceeded or a certain number of iteration is reached. A crucial design option of Algorithm 2.8 is the selection operator at Line 3. There are three possible selection strategies.

Best improvement: The whole neighborhood $\mathcal{N}(\mathbf{x})$ is scanned and the best solution is returned. This option is the most expensive strategy in terms of computation time. However, it does not depend on the order in which $\mathcal{N}(\mathbf{x})$ is search though.

First improvement: A faster option may be to select the first solution that is better than the current incumbent solution $\mathbf{x}$. This selection strategy, however, depends strongly on the order in which the neighborhood of $\mathbf{x}$ is search through.

Random Neighbor: A neighbor solution $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$ is selected at random. This selection strategy is often with an additionally stopping criterion applied such that the procedure terminates if selecting a better solution than the current incumbent solution fails a certain number of times. Note that selecting a neighbor at random can not guarantee that Algorithm 2.8 will terminate with a locally optimal solution with respect to $\mathcal{N}(\cdot)$.

The neighbor selection strategy depends to some extent on the sizes of the considered neighborhoods. For rather small neighborhoods the best improvement selection strategy could be used whereas for larger neighborhoods the first improvement strategy may be a better choice. For extremely large neighborhoods only the random neighbor selection strategy may be practical. Overall, there is no clear rule which selection strategy should be used. Furthermore, the quality of the obtained locally optimal solution depends also on the used neighborhood structure (and their neighborhood sizes) as well as on the used selection strategy. Often a trade-off must be chosen between solution quality and computation time.

2.4.3 Variable Neighborhood Decent

An extension of LS is to consider more than one neighborhood structure. This is motivated by the fact that locally optima depend on the used neighborhood structure. Thus using another neighborhood structure may lead to better locally optimal solutions and we can escape from the local optima that was obtained by using the first neighborhood structure. Therefore, variable neighborhood descent (VND) search through different neighborhood structures by alternating them in a systematic way. Thereby following principles are considered [67, 68]:

- A local optimum with respect to one neighborhood structure is not necessarily one for another.
- A global optimum is a local optimum with respect to all possible neighborhood structures.
- For many problems local optima with respect to one or several neighborhoods are relatively close to each other.

Algorithm 2.9 shows a VND by considering a finite set of $k_{\max}$ neighborhood structures $\mathcal{N}_k(\cdot)$, $1 \leq k \leq k_{\max}$. The algorithm starts with an initial solution $\mathbf{x}_{\text{init}}$. First, the locally optimal solution $\hat{\mathbf{x}}$ with respect to the first neighborhood $\mathcal{N}_1(\cdot)$ is determined. To escape the current local optima the VND switches to the next neighborhood structure $\mathcal{N}_2(\cdot)$ and determines the best neighbor $\mathbf{x}$. If $\mathbf{x}$ is better than $\hat{\mathbf{x}}$, we take $\mathbf{x}$ as new incumbent solution and switch back to the first neighborhood structure. Otherwise, if $\mathbf{x}$ is worse than $\hat{\mathbf{x}}$ then $\hat{\mathbf{x}}$ is locally optimal with respect to $\mathcal{N}_1(\cdot)$ and $\mathcal{N}_2(\cdot)$. Consequently, we consider the next neighborhood structure $\mathcal{N}_3(\cdot)$. This is continued until we reached the last neighborhood structure. As soon as we obtain a best neighbor solution $\mathbf{x}$ that is better than the current incumbent solution $\hat{\mathbf{x}}$ we switch back to the first neighborhood structure and continue with $\mathbf{x}$ as new incumbent solution. The algorithm terminates with a solution that is locally optimal with respect to all considered neighborhood structures. The neighborhood structures are usually ordered according to their increasing sizes, i.e., $|\mathcal{N}_1(\cdot)| \leq |\mathcal{N}_2(\cdot)| \leq \cdots \leq |\mathcal{N}_{k_{\max}}(\cdot)|$ or after their increasing effort to search through them. However, note that it should be avoided to design them such that $\mathcal{N}_1(\cdot) \subseteq \mathcal{N}_2(\cdot) \subseteq \cdots \subseteq \mathcal{N}_{k_{\max}}(\cdot)$ holds.

Algorithm 2.9: Variable Neighborhood Descent

```
Input: An initial solution $\mathbf{x}_{\text{init}}$,
       a sequence of neighborhood structures $\mathcal{N}_k$, $1 \leq k \leq k_{\max}$
Output: A probably improved solution $\hat{\mathbf{x}}$
1  $\hat{\mathbf{x}} \leftarrow \mathbf{x}_{\text{init}}$;
2  $k \leftarrow 1$;
3  while $k \leq k_{\max}$ do
4      $\mathbf{x} \leftarrow \arg\min_{\mathbf{x}' \in \mathcal{N}_k(\hat{\mathbf{x}})} f(\mathbf{x}')$;
5      if $f(\mathbf{x}) < f(\hat{\mathbf{x}})$ then
6          $\hat{\mathbf{x}} \leftarrow \mathbf{x}$;
7          $k \leftarrow 0$;
8      end
9      $k \leftarrow k + 1$;
10 end
11 return $\hat{\mathbf{x}}$
```

2.4.4 General Variable Neighborhood Search

The VNS was first introduced by Mladenović and Hansen [123] and combines deterministic and stochastic changes of neighborhoods. To this end, two set of neighborhood structures are used $\mathcal{N}_k^{\text{vnd}}$, $1 \leq k \leq k_{\max}^{\text{vnd}}$ and $\mathcal{N}_k^{\text{vns}}$, $1 \leq k \leq k_{\max}^{\text{vns}}$. The basic idea behind VNS is to first obtain a locally optimal solution $\hat{\mathbf{x}}$ with respect to neighborhood structures $\mathcal{N}_k^{\text{vns}}$, $1 \leq k \leq k_{\max}^{\text{vns}}$. This phase is also called descent phase or intensification phase and corresponds to the deterministic part of the VNS. In the shaking or diversification phase a solution is randomly selected in a controlled way from the shacking neighborhood structures $\mathcal{N}_k^{\text{vnd}}$, $1 \leq k \leq k_{\max}^{\text{vnd}}$ in order to escape from local optima. If $k_{\max}^{\text{vns}} = 1$, i.e., local search is applied in the descent phase by considering only one neighborhood structure then VNS is also denoted as basic VNS. If more neighborhood structures are applied and a locally optimal solution is obtained by using a VND approach then we speak from a general variable neighborhood search (GVNS) which is depicted at Algorithm 2.10. At each major iteration we randomly select form the first shacking neighborhood $\mathcal{N}_1^{\text{vnd}}(\mathbf{x})$ of the current incumbent solution $\mathbf{x}$ a neighbor solution $\mathbf{x}'$ which is afterwards improved by applying a VND. If the obtained locally optimal solution $\hat{\mathbf{x}}$ is not better than the incumbent solution, we switch to the next shacking neighborhood structure $\mathcal{N}_2^{\text{vnd}}(\cdot)$ and repeat the procedure. Otherwise, we found a new best incumbent solution. In this case we switch back to the first shacking neighborhood structure. The same happens if the last shacking neighborhood structure is reached. There are different standard termination criteria. The most common ones are to terminate the GVNS when a certain threshold of the number of major iterations is succeeded or a certain time limit is reached.

The shaking neighborhood structures are usually ordered according to their sizes, cf. the VND approach in Section 2.4.3. Furthermore, such neighborhood structures are typically larger as solutions are only sampled from them.

Algorithm 2.10: General Variable Neighborhood Search

```
Input: An initial solution $\mathbf{x}_{\text{init}}$,
       a sequence of neighborhood structures $\mathcal{N}_k^{\text{vnd}}$, $1 \leq k \leq k_{\max}^{\text{vnd}}$,
       a sequence of neighborhood structures $\mathcal{N}_k^{\text{vns}}$, $1 \leq k \leq k_{\max}^{\text{vns}}$
Output: A probably improved solution $\mathbf{x}$
1  $\mathbf{x} \leftarrow \mathbf{x}_{\text{init}}$;
2  repeat
3      $k \leftarrow 1$;
4      while $k \leq k_{\max}^{\text{vns}}$ do
5          randomly select $\mathbf{x}' \in \mathcal{N}_k^{\text{vns}}(\mathbf{x})$;             // shaking phase
6          $\hat{\mathbf{x}} \leftarrow$VND($\mathbf{x}'$, $\mathcal{N}_k^{\text{vnd}}$, $1 \leq k \leq k_{\max}^{\text{vns}}$);    // descent phase
7          if $f(\hat{\mathbf{x}}) < f(\mathbf{x})$ then
8              $\mathbf{x} \leftarrow \hat{\mathbf{x}}$;
9              $k \leftarrow 0$;
10         end
11         $k \leftarrow k + 1$;
12     end
13 until stopping criteria satisfied;
14 return $\mathbf{x}$
```

2.5 Decision Diagrams for Optimization

This section describes decision diagrams (DDs) which are a rather new solution approach in the field of combinatorial optimization. A DD is a graphical data structure which originally represents Boolean functions [4, 105] and has been successfully applied in circuit design and formal verification [33, 85]. In the last decade DDs have shown to be a powerful tool in combinatorial optimization [6, 17, 40]. For a variety of problems that classical MILP and CP techniques cannot address effectively (due, e.g., weak dual bounds), new state-of-the-art methodologies could be obtained with DDs at the core. These problems comprise prominent ones such as the maximum independent set, set covering, and maximum cut problems [14, 15] as well as diverse sequencing and scheduling problems [40, 99], including variants of the traveling salesperson problem (TSP). Bergman et al. [14] state that DDs are an interesting alternative to existing methods, since they provide five primary solution strategies of general-purpose methods: relaxation, branching search, constraint propagation, primal heuristics, and modeling to exploit problem structure. This section does not intend to explain in all details these five solution strategies and their applications in context of DDs since that would be clearly out of the scope of this thesis. Instead, we rather provide a relatively short introduction of DDs such that the reader is able to follow upcoming chapters of this thesis. For a comprehensive reading on DDs we refer to the excellent textbook [14] written by Bergman et al.

2.5.1 Exact, Relaxed, and Restricted Decision Diagrams

Before we define DDs in a more formal way let us assume that we consider a COP with instances of the form $(\mathbf{x}, \mathcal{C}, \mathcal{D}, f)$ as defined by Definition 2.1.4 in Section 2.1. Moreover, let $\mathscr{S} = \{\mathbf{x} \in \mathcal{D} \mid \mathbf{x} \text{ satisfies } \mathcal{C}\}$ be the set of feasible solutions. A DD is defined as a rooted acyclic multi-graph $\mathscr{D} = (V, A)$ with node set $V(\mathscr{D})$, arc set $A(\mathscr{D})$, a single root node $\mathbf{r}$, and a single target node $\mathbf{t}$. The node set is typically partitioned into layers $V(\mathscr{D}) = V_1(\mathscr{D}) \cup V_2(\mathscr{D}) \cup \ldots \cup V_{n+1}(\mathscr{D})$, where n is the number of decision variables. The first layer $V_1(\mathscr{D})$ and the last layer $V_{n+1}(\mathscr{D})$ are singletons containing only $\mathbf{r}$ and $\mathbf{t}$, respectively. Each arc $\alpha \in A(\mathscr{D})$ is directed from a node in some layer $V_i(\mathscr{D})$ to a node in a subsequent layer $V_{i+1}(\mathscr{D})$, $1 \leq i \leq n$ and is associated to a label $\mathrm{val}(\alpha) \in \mathcal{D}_i$ that represents the assignment of value $\mathrm{val}(a)$ to variable x_i. There are no two arcs leaving the same node and having the same label. The assignment of x_i to value $\mathrm{val}(\alpha)$ causes some costs which is encoded by the length $z(\alpha) \in \mathbb{R}$ of arc α. Every path $p = (\alpha_1, \ldots, \alpha_n)$ from $\mathbf{r}$ to $\mathbf{t}$ specifies an assignment $\mathbf{x}^p = (\mathrm{val}(\alpha_1), \ldots, \mathrm{val}(\alpha_n))$ of decision variables x_i, $1 \leq i \leq n$ and the corresponding total length is given by $z(p) = \sum_{i=1}^{n} z(\alpha_i)$. The set of $\mathbf{r}$ to $\mathbf{t}$ paths represents the set of assignments $\mathrm{Sol}(\mathscr{D})$.

Exact. We say that $\mathscr{D}$ is an *exact* DD if the $\mathbf{r}$–$\mathbf{t}$ paths encode precisely the set of feasible solutions $\mathscr{S}$ of the considered COP. More precisely, if

$$\mathscr{S} = \mathrm{Sol}(\mathscr{D}), \tag{2.13a}$$

$$f(\mathbf{x}^p) = z(p), \quad \forall \mathbf{r}\text{–}\mathbf{t} \text{ paths } p \text{ in } \mathscr{D} \tag{2.13b}$$

holds. Given an exact DD $\mathscr{D}$, finding the optimal solution is reduced to finding the shortest $\mathbf{r}$–$\mathbf{t}$ path in $\mathscr{D}$. See Figure 2.2a for an example of an exact DD. If all decision variables are binary, i.e. there are at most two outgoing arcs in a corresponding DD, then the DD is called a binary decision diagram (BDD). Otherwise, if there are nodes with more than two outgoing arcs then we speak from a multivalued decision diagram (MDD). Exact DDs tend to grow exponential in size for NP-hard COPs. Thus, in general it will not be possible to create exact DDs within reasonable time or within a reasonable memory consumption. Therefore exact DDs are approximated by restricted or relaxed DDs by usually imposing a limit on the layer size, i.e. the number of nodes each layer is allowed to contain. Depending on the considered DD type, different mechanisms are used to keep a layer below the maximum allowed layer size.

Furthermore, there is a strong relationship between a recursive DP formulation with state space $\mathcal{S}$, transition function $\tau(\cdot,\cdot)$, and transition cost function $c(\cdot,\cdot)$ and an exact DD $\mathscr{D}$ [74]. Each node $u \in V(\mathscr{D})$ is associated to a state $\sigma(u) \in \mathcal{S}$ and an arc $\alpha = (u, v) \in A(\mathscr{D})$ from node $u \in V_i(\mathscr{D})$, $1 \leq i \leq n$ to a target node $v \in V_{i+1}(\mathscr{D})$ represents the transition from state $\sigma(u)$ to state $\sigma(v) = \tau(\sigma(u), \mathrm{val}(\alpha))$ by assigning value $\mathrm{val}(\alpha) \in \mathcal{D}_i$ to decision variable x_i. The length of the arc is thereby equal to the costs caused by the transition from $\sigma(u)$ to $\sigma(v)$, i.e. $z(\alpha) = c(\sigma(u), \sigma(v))$. If the assignment of value $\mathrm{val}(\alpha) \in \mathcal{D}_i$ to variable x_i is not feasible then the transition function maps to the *infeasible state* $\hat{0} \in \mathcal{S}$, i.e. $\tau(\sigma(u), \mathrm{val}(\alpha)) = \hat{0}$. It is possible to provide a generic framework for compiling different types of DDs by considering the underlying state graph of a recursive DP formulation.

Restricted. The **r**–**t** paths of a *restricted* DD encodes a subset of feasible solutions $\mathscr{S}$ such that

$$\mathscr{S} \supseteq \mathrm{Sol}(\mathscr{D}), \tag{2.14a}$$

$$f(\mathbf{x}^p) \geq z(p), \quad \forall \mathbf{r}\text{–}\mathbf{t} \text{ paths } p \text{ in } \mathscr{D} \tag{2.14b}$$

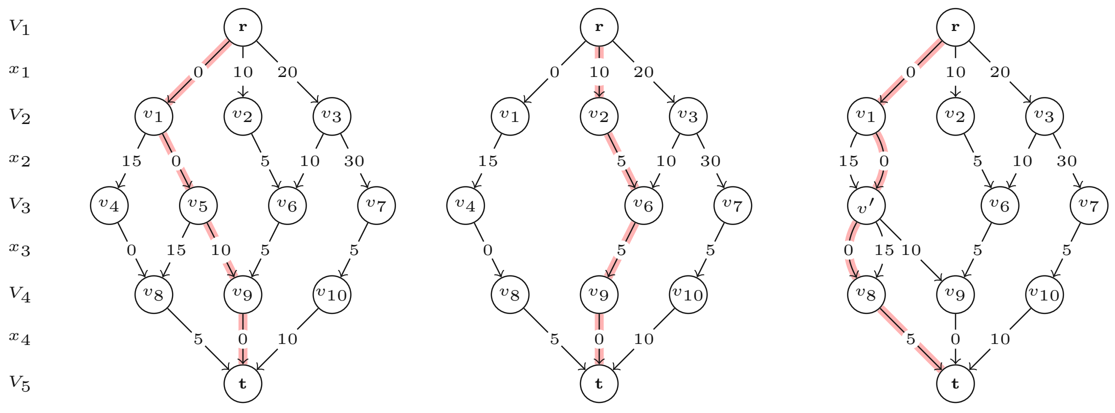
holds. Thus, the shortest path of a restricted DD encodes a heuristic solution of the considered COP. Restricted DD are created by removing or ignoring nodes and incident arcs of a corresponding exact DD as demonstrated in Figure 2.2b. By considering the connection of the state space representation, each search tree or search graph created by one of the search methodologies described in Section 2.2 can be seen as a restricted DD. Consequently, each search approach may be a candidate to compile restricted DDs. In particular a BS-based approach is used in the literature to compile restricted DDs which is called top-down compilation (TDC) in DD-jargon. Hence, a restricted DD $\mathscr{D}$ is compiled layer by layer starting with the root node $\mathbf{r}$ at layer $V_1(\mathscr{D})$. All nodes of the current layer $V_i(\mathscr{D})$, $1 \leq i \leq n$ are expanded using transition function $\tau(\cdot,\cdot)$ and the newly created nodes are inserted into the subsequent layer $V_{i+1}(\mathscr{D})$. If the layer size of $V_{i+1}(\mathscr{D})$ exceeds some threshold β then $|V_{i+1}(\mathscr{D})| - \beta$ nodes are heuristically selected and removed from $\mathscr{D}$. In this way the size of each layer is kept under β.

Relaxed. We say that a DD is *relaxed* if the **r**–**t** paths encode a superset of feasible solutions $\mathscr{S}$. More formally, a DD is relaxed if

$$\mathscr{S} \subseteq \mathrm{Sol}(\mathscr{D}), \tag{2.15a}$$

$$f(\mathbf{x}^p) \leq z(p), \quad \forall \mathbf{r}\text{–}\mathbf{t} \text{ paths } p \text{ in } \mathscr{D} \text{ for which } \mathbf{x}^p \in \mathscr{S} \tag{2.15b}$$

holds. The shortest path of a relaxed DD may not be feasible, however, the length of the shortest path is a feasible lower bound on the optimal objective value. Relaxed DDs are compiled by superimposing nodes of a corresponding exact DD. Thereby a subset of nodes $U \subseteq V_i(\mathscr{D})$, $1 < i \leq n$ from the exact DD is replaced by a new node such that all incoming and outgoing arcs from the selected nodes are redirected to the new node. See Figure 2.2c for an example of a relaxed DD. In this way the size of layer $V_i(\mathscr{D})$ can be limited to a desired number of nodes. Due to this *merging operation* no paths of the exact DD are removed, but new paths may emerge. Those new paths must correspond to non-feasible assignments. To compute the state of the merged node a problem specific *state merger* $\oplus(U)$ operation must be specified. Relaxed DDs represent a *discrete relaxation* of the considered COP which may be a real alternative to e.g. a LP-relaxation (see Section 2.3.2). They are able to provide strong dual bounds for classical problems including maximum independent set problem [15] or variants of scheduling and sequencing problems [40, 99]. Note that sometimes it may be necessary to additionally modify the length of incoming



(a) Exact DD (b) Restricted DD (c) Relaxed DD

Figure 2.2: An example of an exact, restricted, and relaxed DD of a COP with four decision variables $\mathbf{x} = (x_1, x_2, x_3, x_4)$. The nodes of the DDs are partitioned into layers V_i, $1 \leq 5$, with $V_1 = \{\mathbf{r}\}$ and $V_5 = \{\mathbf{t}\}$. Arcs are labeled with the corresponding transition costs and a shortest **r**–**t** path is highlighted in each case. (a) Exact DD with shortest **r**–**t** path of length 10. (b) Restricted DD obtained by removing node v_5 and all incident arcs such that the shortest **r**–**t** path encodes a heuristic solution with length 20. (c) Relaxed DD where nodes v_4 and v_5 are replaced by the new merged node v' leading to a new shortest **r**–**t** path of length 5 which represents a non-feasible variable assignment.

arcs of merged nodes in order to obtain a feasible relaxation. However, in the following we will ignore this detail since this thesis do not consider cases where this is required.

To conclude, DDs are a promising rather new solution approach where COPs can be described in a recursive way by using a DP-based formulation. Furthermore, they provide mechanisms to obtain heuristic solutions (restricted DDs) as well as dual bounds from discrete relaxations (relaxed DDs) of the considered problem. All three concepts are for instance combined in a DD-based BB approach which was able to provide state-of-the art results on classical problems including maximum independent set problem, maximum cut problem, or maximum 2-satisfiability problem [15].

2.5.2 Compilation Methods for Relaxed Decision Diagrams

We describe now a generic framework to compile relaxed DDs which is based on recursive DP formulations. The idea is to extract relaxed DDs from the underlying state graph of a DP formulation by imposing a maximum number of nodes β at each layer. In the literature there are two prominent approaches for relaxed DDs compilation. The TDC is a constructive approach which compiles relaxed DDs layer by layer. Alternatively, the incremental refinement (IR) approach is an iterative procedure that starts with an initial relaxed DD and strengthens the relaxed DD at each iteration by separating a constraint, i.e. removing infeasible assignments from the relaxed DD.

Top-Down Compilation

Algorithm 2.11 shows the TDC in detail. To compile a relaxed DD $\mathscr{D}$ the algorithm checks if the current layer $V_i(\mathscr{D})$, $1 \leq i \leq n$ exceeds the maximum allowed number of nodes β at each iteration. If this is the case then we start to merge nodes by selecting some nodes $U \subseteq V_i(\mathscr{D})$ from layer $V_i(\mathscr{D})$ and replacing them with a new node u' with a corresponding merged state $\sigma(u') = \oplus(U)$. Moreover, all incoming arcs from a node $u \in U$ are redirected to the new node u'. Note that the selection of nodes at Line 4 is problem specific. The most common way is to define some greedy criterion, e.g. based on the length of the shortest path from **r** to nodes of the current layer, and select according to this criterion $|V_i(\mathscr{D})| - \beta + 1$ nodes at once. But also more advance strategies are possible, e.g. selecting only pairs of nodes until $|V_i(\mathscr{D})|$ is reduced to β. After it is guaranteed that the size of $V_i(\mathscr{D})$ is below β, Algorithm 2.11 starts to expand all nodes of layer $V_i(\mathscr{D})$ by considering for each node $u \in V_i(\mathscr{D})$ all possible assignments $d \in \mathcal{D}_i$ that can be feasible assigned to decision variable x_i. Newly created nodes are inserted into the subsequent layer $V_{i+1}(\mathscr{D})$. Finally, all nodes in the last layer $V_{n+1}(\mathscr{D})$ are merged to one target node **t**.

Algorithm 2.11: Top-down compilation, adapted from Bergman et al. [14]

```
Input: Initial state $s_0$, transition function $\tau(\cdot,\cdot)$,
       maximum width $\beta$, merge operator $\oplus$
Output: A relaxed DD $\mathscr{D}$
1  create node $\mathbf{r}$ with $\sigma(\mathbf{r}) = s_0$ and let $V_1(\mathscr{D}) = \{\mathbf{r}\}$;
2  for $i = 1$ to $n$ do
3      while $|V_i(\mathscr{D})| > \beta$ do
4          $U \leftarrow$ select nodes from $V_i(\mathscr{D})$;
5          replace nodes in $U$ with new node $u'$ with $\sigma(u') = \oplus(U)$;
6          forall $u \in V_{i-1}(\mathscr{D})$ and $(u,v) \in A(\mathscr{D})$ with $v \in U$ do
7              remove $(u,v)$ from $A(\mathscr{D})$ and insert arc $(u,u')$;
8          end
9      end
10     $V_{i+1}(\mathscr{D}) \leftarrow \emptyset$;
11     forall $u \in V_i(\mathscr{D})$ and $d \in \mathcal{D}_i$ do
12         if $\tau(u,d) \neq \hat{0}$ then
13             create node $v$ with $\sigma(v) = \tau(u,d)$ and add $v$ to $V_{i+1}(\mathscr{D})$;
14             create arc $\alpha = (u,v)$ with $z(\alpha) = c(\sigma(u),\sigma(v))$;
15         end
16     end
17 end
18 merge nodes in $V_{n+1}(\mathscr{D})$ into target node $\mathbf{t}$;
19 return $\mathscr{D}$
```

Incremental Refinement

An alternative procedure to obtain relaxed DDs is the IR approach that modifies a relaxed DD iteratively until an exact DD is obtained or the size of each layer is equal to the maximum allowed width β. The modification is done by separating constraints in order to remove infeasible variable assignments from the DD. Hence, at each iteration a stronger relaxation of the considered COP is obtained. This can be seen as an analogy to separation procedures in ILP where a continuous relaxation is strengthened by adding

Algorithm 2.12: Incremental refinement, adapted from Bergman et al. [14]

```
Input: Initial relaxed DD $\mathscr{D}$, initial state $s_0$,
       transition functions $\tau_C(\cdot,\cdot)$, $\forall C \in \mathcal{C}$, maximum width $\beta$
Output: A possible strengthened relaxed DD $\mathscr{D}$
while $\exists$ constraint $C \in \mathcal{C}$ violated in $\mathscr{D}'$ do
    $\sigma(u) = \chi$ for all nodes $u \in V(\mathscr{D})$;
    $\sigma(\mathbf{r}) = s_0$;
    for $i = 1$ to $n$ do                                  // filtering step
        for $u \in V_i$ do
            foreach $\alpha = (u, v) \in A(\mathscr{D}')$ do
                if $\tau_C(\sigma(u), \mathrm{val}(\alpha)) \neq \hat{0}$ then
                    remove arc $\alpha$ from $A(\mathscr{D})$;
                end
            end
        end
        for $u \in V_i$ do                                 // refinement step
            foreach $\alpha = (u, v) \in A(\mathscr{D})$ do
                if $\sigma(v) = \chi$ then
                    $\sigma(v) \leftarrow \tau_C(\sigma(u), \mathrm{val}(\alpha))$;
                else if $\sigma(v) \neq \tau_C(\sigma(u), \mathrm{val}(\alpha))$ and $|V_{i+1}(\mathscr{D})| < \beta$ then
                    remove arc $\alpha$ from $A(\mathscr{D})$;
                    create new node $v'$ with $\sigma(v') = \tau_C(\sigma(u), \mathrm{val}(\alpha))$;
                    add arc $(u, v')$;
                    copy outgoing arcs from $v$ as outgoing arcs from $v'$;
                    $V_{i+1}(\mathscr{D}) \leftarrow V_{i+1}(\mathscr{D}) \cup \{v'\}$;
                else
                    update $\sigma(v)$ with $\tau_C(\sigma(u), \mathrm{val}(\alpha))$;
                end
            end
        end
    end
end
return $\mathscr{D}$
```

linear inequalities, so-called cuts, to the ILP model. Instead of linear inequalities we use now the form of DP models to modify the DD. For this purpose let us assume that for each constraint $C \in \mathcal{C}$ an own DP model is defined with its own states and transition function $\tau_C(\cdot,\cdot)$. Algorithm 2.12 depicts the general IR procedure. It starts with an initial relaxed DD $\mathscr{D}$, usually a DD that contains only one node at each layer, and removes at each major iteration all violations of a specific constraint $C \in \mathcal{C}$ in a top-down fashion by splitting corresponding nodes. The algorithm terminates if there is no further violated constraint or each layer has reached the maximum allowed width β. Another additional criterion may be to terminate as soon as the shortest path corresponds to a feasible variable assignment, since in this case optimality could be proven.

At each major iteration a violated constraint $C \in \mathcal{C}$ is selected and all states are reset by a value χ. Afterwards two main steps, *filtering* and *refinement* are applied on each layer starting with the first layer $V_1(\mathscr{D})$.

Filtering step The filtering step consists of removing arcs between layer $V_i(\mathscr{D})$ and $V_{i+1}(\mathscr{D})$, $1 \leq i \leq n$ that belong only to paths that encode an infeasible variable assignment. Algorithm 2.12 identifies such infeasible arcs by checking if the corresponding transition leads to an infeasible state $\hat{0}$. If an upper bound on the objective value is given then the algorithm may also remove arcs that belong only to paths that total lengths exceed the given upper bound. Note that the filtering step is here demonstrated in a rather abstract way. An IR approach for a specific COP will usually apply problem specific filtering rules that sometimes requires also to augment the used states in order to gather enough information for identifying infeasible arcs.

Refinement step The refinement step *split* nodes to resolve violations of constraint C by considering each arc (u, v) between layer $V_i(\mathscr{D})$ and $V_{i+1}(\mathscr{D})$, $1 \leq i \leq n$ separately. If (u, v) is infeasible according to $\tau_C(\cdot,\cdot)$ then it is removed from $\mathscr{D}$. Note that nodes without incoming or outgoing arcs are deleted automatically. If node v is not associated to any state then we set $\sigma(v)$ to the corresponding state represented by arc (u, v). Otherwise, if node v is already associated to another state and the maximum allowed width of layer $V_{i+1}(\mathscr{D})$ does not exceed β we *split* node v by creating a new node v' with the corresponding state and removing arc (u, v). Outgoing arcs from v are copied by v'.

Note that filtering and refinement operations can be modified and applied in any order that is suitable for the problem at hand. Moreover, the filtering operation can still be applied even if the maximum allowed width is already met. Another problem dependent crucial point is the order in which violated constraints are considered. Furthermore, problem specific IR-based approaches may not consider all constraints during the refinement step.

CHAPTER 3

Anytime A* Search

Considering the classical A* search from Section 2.2.2, good solutions are typically only obtained in the very last phase of the search. If A* terminates early due to limited memory or limited computation time then often no feasible solution is available at all. In this chapter we will consider a novel anytime A* search that periodically changes the search strategy from best-first-search to beam search (BS) in order to obtain intermediate heuristic solutions. This anytime A* algorithm is used to tackle a job sequencing problem that has its application as a subproblem in the field of patient scheduling for cancer treatments. The objective is to minimize the makespan.

Preliminary work was presented at the *Third International Conference on Machine Learning, Optimization and Big Data* (MOD 2017) [82]. There we proposed an anytime A* search that uses a greedy construction algorithm as a diving mechanism to obtain intermediate heuristic solutions. Furthermore, we provided an NP-hardness proof for the considered job sequencing problem and we derived a basic lower bound on the makespan objective such that the A* search is able to prove optimality. A substantially extended version of this work was published in the *Artificial Intelligence* (AI) journal [79], where both the exact and the heuristic performance of the anytime A* search got improved. The former was achieved by additionally deriving two strengthened lower bounds for the makespan, which enables A* to prove more instances faster to optimality whereas the latter got boosted by using an advanced diving mechanism based on a combination of BS and local search (LS) to provide high quality intermediate heuristic solutions in regular intervals until a proven optimal solution is found. Finally, we presented a general variable neighborhood search (GVNS) at the *16th International Conference on Parallel Problem Solving from Nature* (PPSN XVI) [95] to obtain heuristic high quality solutions for hard-to-solve instances that could not be solved to proven optimality by the anytime A* search. This work arises from Thomas Kaufmann's master thesis [94] on heuristically solving the patient scheduling problem for cancer treatment.

The remainder of this chapter is structured as follows. First we informally introduce and motivate the job sequencing problem in Section 3.1 followed by a detailed discussion of related work in Section 3.2. A formal definition of the problem as well as a NP-hardness proof is provided in Section 3.3 whereas Section 3.4 reveals three different functions for lower bounds on the makespan. The next Section 3.5 describes a greedy construction algorithm that utilizes the lower bounds as search guidance. Section 3.6 is about the novel anytime A* algorithm that searches on a special state graph structure to efficiently exploit symmetries. Moreover, the greedy construction algorithm is extended to an advanced BS that is in turn used within the A* search as an advanced diving mechanism to obtain regularly intermediate heuristic solutions. Hence, the A* algorithm is able to return a feasible heuristic solution whenever the algorithm is stopped (except for a short time period until the first solution is found). In addition, a LS procedure is applied to any intermediate heuristic solution obtained from BS to further improve these solutions. For comparison purposes a mixed integer linear programming (MILP) formulation and a constraint programming (CP) formulation is provided in Sections 3.7 and 3.8, respectively. A GVNS is proposed in Section 3.9 for hard-to-solve instances. The GVNS is based on move and exchange neighborhood structures and uses an efficient evaluation scheme to scan the neighborhoods of the current incumbent solution. Extensive experiments in Section 3.10 on difficult classes of problem instances demonstrate the excellent performance of the hybrid A* search. Many even large instances with up to 2000 jobs can be solved to proven optimality within seconds, and for the remaining instances solutions with small optimality gaps of usually less than one percent can be obtained. These remaining optimality gaps can be even substantially further reduced by the GVNS.

3.1 Introduction

This chapter considers the following combinatorial optimization problem. A finite set of jobs must be executed without preemption. Each job requires two resources during its execution: (1) a *common resource*, which is required during a certain part of the job's processing period, and (2) a *secondary resource*, which is required during the whole processing period. The common resource is shared by all jobs whereas the secondary resource is shared only by a subset of the other jobs.

Problems with such characteristics arise, for example, in the context of the production of certain products. Imagine a single machine (the common resource) sequentially processing some fixtures or molds (the secondary resource) that contain some raw material. Before the processed fixtures/molds are available for further usage again, some postprocessing (e.g., cooling) might be required. However, our motivation for tackling this problem has a different source: the scheduling of patients in modern particle therapy for cancer treatment [115, 118, 120]. In this rather novel cancer treatment technique, carbon or proton particles are accelerated in a cyclotron or synchrotron (i.e., a specific kind of particle accelerator) to almost the speed of light. The particle beam is then directed into a treatment room where it is used to radiate a patient. Typically, between two and four differently equipped treatment rooms are available but only one particle accelerator, and the single particle beam can only be directed into one of these rooms at a time. The treatment room for each patient is determined in advance and in dependence on the patient's specific needs. Each patient generally requires a specific preparation (related to positioning, fixation, sedation, etc.) in the room before the actual irradiation can start. Moreover, after the treatment some further time is usually needed for medical inspections before the patient can actually leave the room and the treatment of a next patient may start in the same room. Note that the available rooms correspond to the secondary resources mentioned above, while the particle beam is the common resource. We consider the scheduling of a set of patients in a given time period (for example, one day) in such a facility. The optimization goal is to finish the treatment of all patients as early as possible, which is known as *makespan minimization* in the related literature. This problem, whose technical description will be provided in Section 3.3, is henceforth denoted as job sequencing problem with one common and multiple secondary resources (JSOCMSR).

Note that the JSOCMSR is rather easy to solve when (1) the common resource usage is the exclusive bottleneck and enough secondary resources are available or (2) the pre- and postprocessing times during which only the secondary resources are required, are negligible in comparison to the jobs' total processing times. In such cases the jobs can, essentially, be performed in almost an arbitrary ordering and the common resource is exploited without any breaks. The problem, however, becomes challenging when pre- and postprocessing times are substantial and many jobs require the same secondary resources. In this chapter we consider such difficult scenarios.

3.2 Related Work

We split the further treatment of related work into three parts. First we consider the specific application background in particle therapy patient scheduling. The second part addresses further related problems, and the third part deals with related solution techniques.

Concerning particle therapy patient scheduling, the JSOCMSR clearly is a strongly simplified formulation picking out only certain aspects. The complete practical scenario comprises many more aspects, such as, for example, large time horizons of several weeks, sequences of therapies for patients to be treated, additionally needed resources including medical staff and their availability time windows, and a combination of more advanced objectives and diverse soft constraints. A significant amount of previous work covers patient scheduling for radiotherapy on a higher level without planning the detailed timing within each day; see, e.g., [41, 89].

The single particle accelerator, which is only found in modern particle treatment facilities, is an expensive bottleneck resource that needs to be exploited in conjunction with multiple treatment rooms in the best possible way. As this technology is quite recent, there are only a few existing works. Maschler et al. [120] proposed a greedy construction heuristic which is extended towards an iterated greedy (IG) metaheuristic and a greedy randomized adaptive search procedure (GRASP). These approaches treat the whole problem as a bi-level optimization problem in which the upper level is concerned with the assignment of treatments to days, and the lower level corresponds to the detailed scheduling of the treatments assigned at each day. Thus, our JSOCMSR represents the core of these daily sub-problems. In [119], the IG metaheuristic was further refined by including an improved makespan estimation for the daily scheduling problems, which, however, is still a rather crude approximation. Last but not least, this IG metaheuristic was extended in [115] to additionally consider soft constraints for unwanted deviations of starting times of the individual treatments for each therapy. For more details on solving radiotherapy based scheduling problems we refer to [114].

To the best of our knowledge, there are only a few further publications dealing with other scenarios similar to the JSOCMSR. Among those, Veen et al. [157] consider a problem that has the closest relation to the JSOCMSR. The common resource corresponds to a machine on which the jobs are processed and secondary resources needed during pre- and postprocessing are called templates. However, in this problem it can be assumed that the postprocessing times are negligible compared to the total processing times of the jobs. This implies that the starting time of each job only depends on its immediate predecessor. More specifically, a job j requiring a different resource than its predecessor j' can always be started after a setup time only depending on job j, while a job requiring the same resource can always be started after a postprocessing time only depending on job j'. Due to these characteristics, this problem can be interpreted as a traveling salesperson problem (TSP) with a special cost structure. It is shown that this problem can be solved efficiently in time $O(n \log n)$, where n is the number of jobs.

Another related problem is the no-wait flowshop problem. A survey covering this problem in addition to similar problems can be found in [5]. Each job needs to be processed on each of m machines in the same order and the processing of the job on a successive machine always has to take place immediately after its processing has finished on the preceding machine. This problem can be solved in time $O(n \log n)$ for two machines via a transformation to a specially structured TSP [61]. In contrast, for three and more machines the problem is NP-hard, although it can still be transformed into a specially structured TSP. Röck [135] proved that the problem is strongly NP-hard for three machines by a reduction from the 3D-matching problem.

Finally, note that the JSOCMSR can be modeled as a more general resource constrained project scheduling problem (RCPSP) with maximal time lags. For a survey on RCPSPs with various extensions and respective solution methods see Hartmann and Briskorn [71]. Among the state-of-the-art-techniques for solving general RCPSPs are in particular CP techniques utilizing lazy clause generation and satisfiability modulo theory [7, 142]. A corresponding RCPSP instance can, for example, be obtained from a JSOCMSR instance by splitting each job into three activities which are the preprocessing, the main part also requiring the common resource, and the postprocessing. These activities must be performed for each job in this order with a maximal time lag of zero. Moreover, all resource requirements must be respected. Another way to obtain a corresponding RCPSP instance from a JSOCMSR instance is to use for each job two activities that must be scheduled at respectively related times: one activity for the part where the common resource is required and a second activity for the whole processing time where the secondary resource is required. We essentially make use of this point-of-view in our CP approach presented in Section 3.8.

As mentioned already above, one of our core contributions to the algorithmic side is an A* algorithm for solving the JSOCMSR. A* is one of the standard algorithms in the field of artificial intelligence for path planning and, more generally, finding shortest paths in possibly huge graphs [70, 134]. In particular, our approach belongs to the class of Anytime A* algorithms, which are complete but also can be stopped at almost any time and are then likely to yield a reasonable approximate solution. Most of the Anytime A* algorithms from the literature are based on so-called heuristic weighted A* algorithms that were first introduced by Pohl [130]. These algorithms guarantee that the objective value of the obtained solution is not worse than the objective value of the optimal solution times an approximation ratio $\varepsilon \geq 1$. Thus, ε determines a trade-off between solution quality and the time it takes until the first heuristic solution is found. A large value for ε makes the algorithm greedier, meaning that the algorithm will faster find a heuristic solution. Hansen and Zhou [66] provide an anytime weighted A* algorithm which does not stop after the first solution is found but rather continues with the search. This algorithm produces a sequence of improved solutions as well as a sequence of improved dual bounds until the optimal solution is found. A similar approach is suggested by Likhachev, Gordon and Thrun [108], called anytime repairing A* (ARA*), where ε is decreased each time a new solution is found, instead of letting the parameter constant over the whole search. Restarting weighted A* (RWA*) introduced by Richter, Thayer and Ruml [133] is based on ARA* and restarts the search each time ε is decreased. In this way, the algorithm is able to undo bad decisions, which may have been taken at the beginning of the search earlier. Another A* based anytime algorithm which does not require any parameter is anytime nonparametric A* (ANA*) introduced by van den Berg et al. [156]. The same algorithm was also independently introduced under the name anytime potential search (APTS) by Stern et al. [148, 149] via a different derivation [147]. Both algorithms maximize the greediness of the search based on the current incumbent solution. This leads to the fact that APTS/ANA* finds an initial solution faster and needs less time between improving solutions when compared to ARA*.

As we will see in the remainder of this chapter, our new A* variant is able to rapidly find high quality solutions for the JSOCMSR by exploiting strong lower bounds for the makespan as search guidance. In order to obtain solutions of similar quality with one of the above weighted A* algorithms, parameter ε would need to be set very close to one. However, such a parameter choice typically leads in our case to a long-running time until a first solution is found, which stays in contrast to our goal of achieving a good anytime behavior. Using a larger ε to find a first solution faster in general reduces the quality of this first solution substantially. In Section 3.10.8 we experimentally compare ARA* to our anytime A* algorithm and the results verify this behavior. Therefore, using a weighted A* based algorithm is not promising for solving the JSOCMSR.

An anytime A* algorithm that is not based on weighted A* is anytime window A* (AWA*) as introduced by Aine et al. [3]. This algorithm uses a technique based on sliding windows to reduce the set of partial solution extensions to quickly find a heuristic solution. At each iteration the window width is increased such that improved solutions can be found until AWA* can eventually prove optimality.

Anytime pack search (APS), introduced by Vadlamudi et al. [155], is based on BS. The algorithm consecutively performs BS iterations, keeping partial solutions which are pruned during the BS iterations in memory. A BS iteration ends if a heuristic solution is found. Then a new BS iteration starts by selecting the best partial solution from the set of pruned partial solutions. The algorithm terminates if the set of pruned partial solutions is empty. In this case the last found solution must be an optimal solution. The APS approach is similar to our anytime A* approach since we also use BS to find intermediate heuristic solutions. The difference is that we embed BS into an A* algorithm such that BS is performed each time after a specific number of classical A* iterations. Our experimental comparison shows that for the JSOCMSR our anytime A* algorithm performs better than APS, cf. Section 3.10.4.

Finally, note that our suggested anytime A* algorithm was simultaneously applied in a similar way to the longest common palindromic subsequencing problem by Djukanovic [49].

3.3 Problem Definition and Complexity

In formal terms, an instance of the JSOCMSR consists of a set of n jobs $J = \{1, \dots, n\}$ that are to be executed without preemption, the common resource 0, and a set of m secondary resources $R = \{1, \dots, m\}$. The set of all resources is denoted by $R_0 = \{0\} \cup R$. Each job $j \in J$ has a total processing time $p_j > 0$ during which it fully requires a secondary resource $q_j \in R$. Furthermore, each job j requires the common resource 0 for a duration p^0_j with $0 < p^0_j \leq p_j - p^{\text{pre}}_j$, where $p^{\text{pre}}_j \geq 0$ is the *preprocessing time*. In particular, the need for resource 0 starts at time p^{pre}_j, counted from the start of the job's processing. A solution to the problem is described by the starting times $s = [s_j]_{j \in J}$ of all jobs, with $s_j \geq 0$. Such a solution s is feasible if no two jobs require a resource at the same time.

The objective is to find a feasible schedule that minimizes the finishing time of the job that finishes last. This optimization criterion is known as the *makespan*, and it can be calculated for a solution s by

$$\text{MS}(s) = \max_{j \in J} (s_j + p_j). \tag{3.1}$$

As each job requires the common resource 0, and only one job can use this resource at a time, a solution implies with respect to $s_j + p^{\text{pre}}_j$ a total ordering of the jobs. Vice versa, any ordering—i.e., permutation—$\pi = [\pi_k]_{k=1,\dots,n}$, of the jobs in J can be decoded into a feasible solution in the straight-forward greedy way by scheduling each job in the given order at the earliest feasible time. We call a schedule in which, for a certain job permutation π, each job is scheduled at its earliest time, a *normalized schedule*. Obviously, any optimal solution is either a normalized schedule or there exists a corresponding normalized schedule with the same objective value. We therefore also use the notation $\mathrm{MS}(\pi)$ for the makespan of the normalized solution induced by the job permutation π.

For convenience, we further define the duration of the postprocessing time by $p_j^{\mathrm{post}} = p_j - p_j^{\mathrm{pre}} - p_j^0$, $\forall j \in J$ and denote by $J_r = \{j \in J \mid q_r = r\}$ the subset of jobs requiring resource $r \in R$ as secondary resource. Note that $J = \bigcup_{r \in R} J_r$. The minimal makespan over all feasible solutions, i.e., the optimal solution value, is denoted by MS^*.

3.3.1 Computational Complexity

The decision variant of the JSOCMSR answers the question if there exists a feasible solution with a makespan corresponding to a given constant MS^*.

Theorem 3.3.1
The decision variant of the JSOCMSR is NP-complete for $m \geq 2$.

Proof. Our problem is in class NP since a solution can be checked in polynomial time. We show that the decision variant of the JSOCMSR is NP-complete by a polynomial reduction from the well-known NP-complete partition problem (PP) [59], which is stated as follows: Given a finite set of positive integers $A \subset \mathbb{N}$, partition it into two disjoint subsets A_1 and A_2 such that $\sum_{a \in A_1} a = \sum_{a \in A_2} a$.

An instance of the PP is transformed into an instance of the JSOCMSR as follows. Let $m = 2$ and J consist of the following jobs:

- For each $a \in A$ there is a corresponding job $j \in \{1, \dots, |A|\} \subset J$ with processing time $p_j = a$ requiring resource $q_j = 1$ and the common resource 0 the whole time, i.e., $p_j^0 = p_j$ and $p_j^{\mathrm{pre}} = 0$.
- Furthermore, there are two jobs $j \in \{|A|+1, |A|+2\} \subset J$ with processing times $p_j = \frac{1}{2}\sum_{a \in A} a + 1$ requiring resource $q_j = 2$ the whole time but the common resource 0 just at the first time slot, i.e., $p^0 = 1$ and $p_j^{\mathrm{pre}} = 0$.

Let $\mathrm{MS}^* = p_{|A|+1} + p_{|A|+2} = \sum_{a \in A} a + 2$. A feasible solution to the JSOCMSR with makespan MS^* must have the jobs $|A|+1$ and $|A|+2$ scheduled sequentially without any gap and all other jobs in parallel to those two. A corresponding solution to the PP can immediately be derived by considering the integers associated with the jobs scheduled in parallel to job $|A+1|$ as A_1 and those scheduled in parallel to job $|A+2|$ as A_2. The obtained solution to the PP must be feasible since $\sum_{a \in A_1} a = \sum_{a \in A_2} a = \frac{1}{2}\sum_{a \in A} a$ holds as the jobs corresponding to the integers do not overlap and there is exactly $\frac{1}{2}\sum_{a \in A} a$ time left at the common resource 0 when processing job $|A|+1$ and job $|A|+2$, respectively. It also follows that if there is no JSOCMSR solution with makespan MS^*, then there cannot exist a feasible solution to the PP.

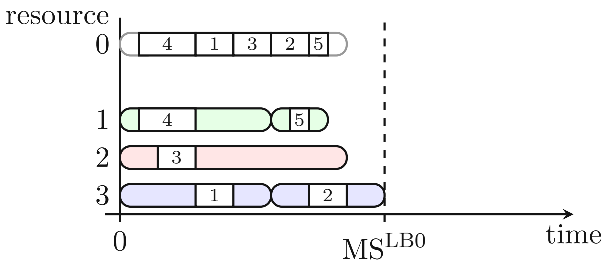


Figure 3.1: Resource-specific individual lower bounds and the overall lower bound $\mathrm{MS}^{\mathrm{LB0}}$ for an example instance with $n = 5$ jobs and $m = 3$ secondary resources.

Clearly, the described transformation of a PP instance into a JSOCMSR instance as well as the derivation of the PP solution from the obtained schedule can both be done in time $O(|A|)$, i.e., in polynomial time.

Consequently, the decision variant of the JSOCMSR is NP-complete. □

Corollary 3.3.1.1
The makespan minimization variant of the JSOCMSR is NP-hard.

3.4 Lower and Upper Bounds

First of all, a trivial upper bound for MS^* is obtained when scheduling all jobs strictly sequentially, yielding $\mathrm{MS}^{\mathrm{UB}} = \sum_{j\in J} p_j$.

A simple lower bound for the makespan can be calculated on the basis of each resource $r \in R$ by taking the total time over all jobs requiring resource r, i.e.,

$$\mathrm{MS}_r^{\mathrm{LB0}} = \sum_{j\in J_r} p_j. \tag{3.2}$$

Similarly, another lower bound can be obtained on the basis of resource 0 by

$$\mathrm{MS}_0^{\mathrm{LB0}} = \min_{j,j'\in J \,|\, j\neq j' \vee |J|=1} (p_j^{\mathrm{pre}} + p_{j'}^{\mathrm{post}}) + \sum_{j\in J} p_j^0. \tag{3.3}$$

Instead of taking only the sum of the time requirements for resource 0, this calculation also considers the minimal times for preprocessing and postprocessing of the first and last scheduled jobs, respectively. Now, we can take the maximum of these $m+1$ resource-specific lower bounds and obtain

$$\mathrm{MS}^{\mathrm{LB0}} = \max_{r\in R_0} \mathrm{MS}_r^{\mathrm{LB0}}. \tag{3.4}$$

Figure 3.1 illustrates these relationships. It follows that taking any normalized solution has an approximation factor of no more than m, since $\mathrm{MS}^{\mathrm{UB}} \leq m \cdot \mathrm{MS}^{\mathrm{LB0}}$.

Theorem 3.4.1
$\mathrm{MS}^{\mathrm{LB0}}$ is a lower bound for the makespan of the JSOCMSR.

Proof. Consider an arbitrary instance of the JSOCMSR and let $\mathrm{MS}^{\mathrm{LB0}}$ be the computed lower bound according to (3.4). Assume that there exists a normalized feasible solution s with $\mathrm{MS}(s) < \mathrm{MS}^{\mathrm{LB0}}$. Let r be a resource that determines the maximum in (3.4), i.e., $\mathrm{MS}_r^{\mathrm{LB0}} = \mathrm{MS}^{\mathrm{LB0}}$. We have to consider two cases:

- $r \neq 0$: The lower bound is determined by the sum of the total processing times of those jobs which require the secondary resource $r \in R$, that is, $\sum_{j \in J_r} p_j$. Since the makespan of solution s is smaller than the sum $\sum_{j \in J_r} p_j$, resource r must be used by more than one job at the same time. Hence, solution s would be infeasible, which contradicts our original assumption.

- $r = 0$: The lower bound is determined by the common resource 0. We have to distinguish between two cases: (1) $\mathrm{MS}(s) < \sum_{j \in J} p_j^0$ and (2) $\sum_{j \in J} p_j^0 \leq \mathrm{MS}(s) < \min_{j,j' \in J \,|\, j \neq j' \vee |J|=1}(p_j^{\mathrm{pre}} + p_{j'}^{\mathrm{post}}) + \sum_{j \in J} p_j^0$. In the first case solution s is not feasible because the common resource 0 must be used by more than one job at the same time to achieve a makespan $\mathrm{MS}(s) < \mathrm{MS}^{\mathrm{LB0}}$. In the second case, the possibility remains that the common resource 0 is not used by more than one job at the same time. But then either there must exist a job $j \in J$ with a starting time $s_j < 0$ or there must exist a job $j' \in J$ such that $s_{j'} + p_{j'} > \mathrm{MS}(s)$. Again, all these cases contradict our original assumption.

We conclude that $\mathrm{MS}^{\mathrm{LB0}}$ is indeed a lower bound for MS^*. □

3.4.1 Strengthened Lower Bounds

The assumption of lower bound $\mathrm{MS}^{\mathrm{LB0}}$ (from Eq. (3.4)) is that regarding a determining resource $r \in R$ all jobs J_r can be scheduled consecutively one after the other without any gaps. $\mathrm{MS}^{\mathrm{LB0}}$ can be further strengthened by checking if the usage of the common resource 0 by all other jobs in $J \setminus J_r$ causes a conflict with this assumed schedule of jobs in J_r. This is done by considering the maximal possible continuous duration in which resource 0 is not used when scheduling the jobs in J_r consecutively in any order. To determine this duration we consider the maximum gap in the usage of resource 0 for any consecutively scheduled pair of jobs from J_r if $|J_r| > 1$, or just the maximum of the preprocessing and postprocessing times in case J_r is singleton, i.e.,

$$p_{\max}^{\mathrm{prepost}}(J_r) = \begin{cases} \max\limits_{j,j' \in J_r \,|\, j \neq j'} (p_j^{\mathrm{pre}} + p_{j'}^{\mathrm{post}}) & \text{if } |J_r| > 1 \\ \max\limits_{j \in J_r}(p_j^{\mathrm{pre}}, p_j^{\mathrm{post}}) & \text{if } |J_r| = 1 \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

Algorithm 3.1: Determine time gaps $[g_i]_{i=1}^k$ for jobs J_r

```
    Input: Jobs $J_r$
    Output: Sequence of time gaps $[g_i]_{i=1}^k$
1   $g_1 \leftarrow p_{\max}^{\text{prepost}}(J_r)$ with corresponding jobs $j_1$ and $j_2$ for $p_{j_1}^{\text{pre}}$ and $p_{j_2}^{\text{post}}$, resp.;
2   $J^{\text{pre}} \leftarrow J^{\text{pre}} \setminus \{j_1\}, \quad J^{\text{post}} \leftarrow J^{\text{post}} \setminus \{j_2\}$;
3   for $k \leftarrow 2$; $J^{\text{pre}} \neq \emptyset \wedge J^{\text{post}} \neq \emptyset$; $k \leftarrow k+1$ do
4       $(j_1, j_2) \leftarrow \arg\max_{(j,j') \in J^{\text{pre}} \times J^{\text{post}}} (p_j^{\text{pre}} + p_{j'}^{\text{post}})$;
5       $g_k \leftarrow p_{j_1}^{\text{pre}} + p_{j_2}^{\text{post}}$;
6       $J^{\text{pre}} \leftarrow J^{\text{pre}} \setminus \{j_1\}, \quad J^{\text{post}} \leftarrow J^{\text{post}} \setminus \{j_2\}$;
7   end
8   return $[g_i]_{i=1}^k$
```

Now, each job $j \in J \setminus J_r$ that requires the common resource 0 longer then $p_{\max}^{\text{prepost}}(J_r)$ causes a delay of at least $p_j^0 - p_{\max}^{\text{prepost}}(J_r)$ in the schedule of the jobs in J_r. Hence, the sum of all minimum delays

$$h_r^1 = \begin{cases} \sum\limits_{j \in J \setminus J_r} \max\left(p_j^0 - p_{\max}^{\text{prepost}}(J_r), 0\right) & \text{if } J_r \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

for each resource $r \in R$ can be added to the corresponding lower bounds MS_r^{LB0}. We obtain $\text{MS}_r^{\text{LB1}} = \text{MS}_r^{\text{LB0}} + h_r^1$ as new bound for each secondary resource $r \in R$ and the overall strengthened lower bound

$$\text{MS}^{\text{LB1}} = \max\left(\text{MS}_0^{\text{LB0}}, \max_{r \in R} \text{MS}_r^{\text{LB1}}\right). \tag{3.7}$$

Lower bound MS^{LB1} can be even further strengthened. Instead of considering always only the largest possible gap $p_{\max}^{\text{prepost}}(J_r)$ in the usage of resource 0 to compute the sum of delays as done in Eq. (3.6), we can more precisely consider a tighter set of maximal time gaps $[g_i]_{i=1}^k$ with $k = |J_r|$ that possibly may occur together in a consecutive scheduling of the jobs in J_r. Algorithm 3.1 shows the calculation of this set, which is obtained as a decreasingly sorted sequence of time gaps $[g_i]_{i=1}^k$, i.e., $g_1 = p_{\max}^{\text{prepost}}(J_r) \geq g_2 \geq \cdots \geq g_k$, for $r \in R$.

Line 1 ensures that the first element of the sequence is always $g_1 = p_{\max}^{\text{prepost}}(J_r)$. Sets J^{pre} and J^{post} contain the jobs whose pre- and postprocessing times are not already consumed for calculating the next time gap, respectively. At each iteration k of the while-loop, jobs $j_1 \in J^{\text{pre}}$ and $j_2 \in J^{\text{post}}$ are determined such that $g_k = p_{j_1}^{\text{pre}} + p_{j_2}^{\text{post}}$ is the next largest time gap. At the end of the loop jobs j_1 and j_2 are removed from sets J^{pre} and J^{post}, respectively.

Algorithm 3.2 shows the calculation of the strengthened sum of minimum delays h_r^2 exploiting the time gaps $[g_i]_{i=1}^k$ for resource $r \in R$. Set J^0 contains all jobs $J \setminus J_r$ which

Algorithm 3.2: Determine the strengthened sum of the minimum delays h_r^2

```
   Input: Resource $r$
   Output: $h_r^2$
1  if $J_r = \emptyset$ then return 0;
2  $h_r^2 \leftarrow 0$;
3  $J^0 \leftarrow J \setminus J_r$;
4  determine time gaps $[g_i]_{i=1}^k$ concerning $J_r$ according to Alg. 3.1;
5  for $i = 1$; $J^0 \neq \emptyset \wedge i \leq k$; $i \leftarrow i+1$ do
6      $j \leftarrow \arg\max_{j \in J^0} p_j^0$;
7      if $p_j^0 < g_i$ then return $h_r^2$;
8      $h_r^2 \leftarrow h_r^2 + p_j^0 - g_i$;
9      $J^0 \leftarrow J^0 \setminus \{j\}$;
10 end
11 return $h_r^2 + \sum_{j \in J^0} p_j^0$
```

are not yet consumed. After determining the time gaps $[g_i]_{i=1}^k$, the for-loop iterates through them starting with the largest time gap g_1. At each iteration, the job j that, among those currently being in J^0, has the largest requirement of resource 0 is determined. The loop terminates if set J^0 becomes empty or p_j^0 is less than the currently largest time gap g_i. If $p_j^0 \geq g_i$ then it is not possible to schedule job j without shifting some of the starting points of the jobs in J_r. Therefore, we add the difference $p_j^0 - g_i$ to h_r^2. Afterwards job j is removed from set J^0. In the last step of Algorithm 3.2 the total processing times from the remaining jobs in J^0 are added to h_r^2.

Once all terms h_r^2 for $r \in R$ are computed by Algorithm 3.2, they are added to the basic individual lower bounds, that is, $\mathrm{MS}_r^{\mathrm{LB2}} = \mathrm{MS}_r^{\mathrm{LB0}} + h_r^2$. Finally, the joined strengthened lower bound is obtained by

$$\mathrm{MS}^{\mathrm{LB2}} = \max\left(\mathrm{MS}_0^{\mathrm{LB0}}, \max_{r \in R} \mathrm{MS}_r^{\mathrm{LB2}}\right). \tag{3.8}$$

Figure 3.2 shows an example where the bound $\mathrm{MS}^{\mathrm{LB2}}$ corresponds to the optimum makespan MS^* and the bounds $\mathrm{MS}^{\mathrm{LB1}}$ and $\mathrm{MS}^{\mathrm{LB0}}$ are weaker. The example illustrates an instance with $n = 6$ jobs and $m = 2$ secondary resources. Jobs $J_1 = \{1, 2, 3, 4\}$ need secondary resource 1 and jobs $J_2 = \{5, 6\}$ resource 2. Note that all jobs in J_1 require the common resource 0 exactly one unit of time whereas all jobs in J_2 need resource 0 during their whole execution. The optimal solution is shown in the upper third of the figure with an optimal makespan of $\mathrm{MS}^* = 14$. However, according to Eq. (3.4) $\mathrm{MS}^{\mathrm{LB0}} = 12$, which is illustrated in the lower part of the figure. The common resource 0 is at most $p_{\max}^{\mathrm{prepost}}(J_1) = 3$ units of time free when considering only the jobs in J_1. Hence, $h_1^1 = 1$ according to (3.6) since job 5 claims resource 0 for more than 3 units of time and job 6 claims resource 0 for exactly 3 units. This gives us a strengthened bound of $\mathrm{MS}^{\mathrm{LB1}} = 13$ according to (3.7), which is illustrated in the lower

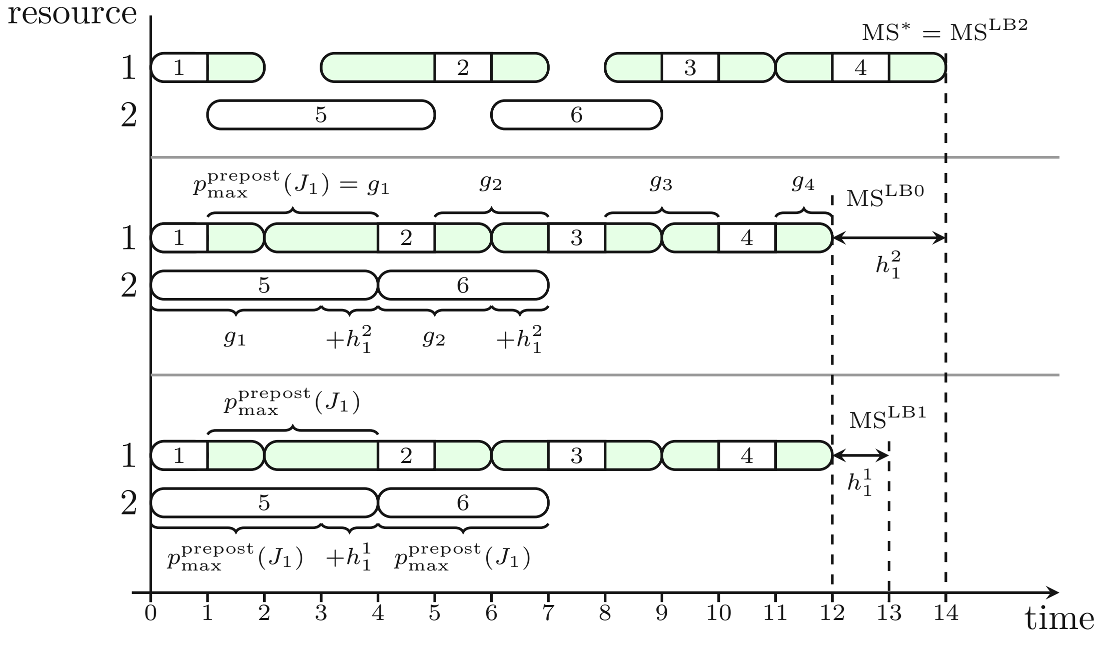


Figure 3.2: Example for the lower bounds MS^{LB0}, MS^{LB1} and MS^{LB2}.

part of Figure 3.2. However, considering the time gaps $[g_i]^4_{i=1} = [3, 2, 2, 1]$ for resource 1 according to Algorithm 3.1, we get $h^2_1 = p^0_5 - g_1 + p^0_6 - g_2 = 4 - 3 + 3 - 2 = 2$. This results in an improved lower bound $\text{MS}^{\text{LB2}} = 14$ as shown in the middle part of Figure 3.2.

Theorem 3.4.2
MS^{LB1} and MS^{LB2} are lower bounds for the makespan and $\text{MS}^{\text{LB0}} \leq \text{MS}^{\text{LB1}} \leq \text{MS}^{\text{LB2}}$ for any instance of the JSOCMSR.

Proof. See the Appendix A. □

3.5 Least Lower Bound Heuristic

We construct a heuristic solution by iteratively selecting a not yet scheduled job, which is executed after all already scheduled jobs at the earliest possible time. The crucial aspect is the greedy selection of the job to be scheduled next, which is based on the lower bound calculation from the previous section. Therefore, we call this heuristic least lower bound heuristic (LLBH).

Let π^{p} be the current partial job permutation representing the current normalized schedule and $J' \subseteq J$ be the set of remaining unscheduled jobs. Initially, π^{p} is empty and $J' = J$. Given π^{p}, the *earliest availability time* for each resource—that is, the earliest time when the resource could be used by a next yet unscheduled job—can be calculated from the respective finishing time of the last job using this resource:

$$t_0 = \begin{cases} \max_{j \in J \setminus J'} s_j + p_j^{\mathrm{pre}} + p_j^0 & \text{for } J' \neq J \\ 0 & \text{else} \end{cases} \tag{3.9}$$

$$t_r = \begin{cases} \max_{j \in J_r \setminus J'} s_j + p_j & \text{for } J_r \setminus J' \neq \emptyset \\ 0 & \text{else} \end{cases} \qquad \forall r \in R \tag{3.10}$$

These times, however, can possibly be further increased (*trimmed*) as the earliest usage time of resource $r \in R$ by a successive job also depends on the remaining unscheduled jobs and the earliest usage time of the common resource 0. We therefore apply the rule

$$t_r \leftarrow \max\left(t_r,\, t_0 - \max_{j \in J_r \cap J'} p_j^{\mathrm{pre}}\right) \qquad \forall r \in R \mid J_r \cap J' \neq \emptyset. \tag{3.11}$$

Moreover, also t_0 might be increased as its earliest usage time also depends on the remaining unscheduled jobs and the earliest usage times of their secondary resources. These relations are considered by applying the rule

$$t_0 \leftarrow \max\left(t_0,\, \min_{j \in J'}(t_{q_j} + p_j^{\mathrm{pre}})\right) = \max\left(t_0,\, \min_{r \in R \mid J_r \cap J' \neq \emptyset}(t_r + \min_{j \in J_r \cap J'} p_j^{\mathrm{pre}})\right). \tag{3.12}$$

Also note that after a successful increase of t_0 by rule (3.12), it might be possible to further increase the t_r-value of some resource $r \in R$ according to the respective rule (3.11). We therefore apply the trimming rules repeatedly until no further increase of any of the earliest availability times is achieved.

Following our general lower bound calculation for the makespan in (3.4), it is possible to derive a more specific lower bound for a given partial permutation π^{p} considering any possible extension to a complete solution on the basis of each resource $r \in R \mid J_r \cap J' \neq \emptyset$ by

$$\mathrm{MS}_r^{\mathrm{LB0}}(\pi^{\mathrm{p}}) = \begin{cases} t_r + \sum_{j \in J_r \cap J'} p_j & \text{for } J_r \cap J' \neq \emptyset \\ 0 & \text{else} \end{cases} \qquad \forall r \in R. \tag{3.13}$$

We define $\mathrm{MS}_r^{\mathrm{LB0}}(\pi^{\mathrm{p}}) = 0$ for any resource r that is not required by any remaining job in J' since these bounds are not relevant for our further considerations.

A lower bound w.r.t. the common resource 0 can be calculated similarly by

$$\mathrm{MS}_0^{\mathrm{LB0}}(\pi^{\mathrm{p}}) = \max\left(t_0 + \min_{j \in J'} p_j^{\mathrm{post}},\, \min_{j,j' \in J' \mid j \neq j' \vee |J'|=1}(t_{q_j} + p_j^{\mathrm{pre}} + p_{j'}^{\mathrm{post}})\right) + \sum_{j \in J'} p_j^0. \tag{3.14}$$

Clearly, an overall lower bound for the partial solution π^{p} is obtained by taking the maximum of the individual bounds

$$\mathrm{MS}_{\max}^{\mathrm{LB0}}(\pi^{\mathrm{p}}) = \max_{r \in R_0} \mathrm{MS}_r^{\mathrm{LB0}}(\pi^{\mathrm{p}}). \tag{3.15}$$

For selecting the next job in LLBH to be appended to π^{p}, we always consider the impact of each job $j \in J'$ on each individual bound $\mathrm{MS}_r^{\mathrm{LB0}}$, $r \in R_0$, as this gives a more fine-grained discrimination than just considering the impact on the overall bound $\mathrm{MS}_{\max}^{\mathrm{LB0}}(\pi^{\mathrm{p}})$, which would often lead to ties.

More specifically, let $\mathbf{f}(\pi^{\mathrm{p}}) = [f_0(\pi^{\mathrm{p}}), \dots, f_m(\pi^{\mathrm{p}})]$ be the vector of the bounds $\mathrm{MS}_r^{\mathrm{LB0}}(\pi^{\mathrm{p}})$ for $r \in R_0$ *sorted in non-increasing value order*, i.e., $f_0(\pi^{\mathrm{p}}) = \mathrm{MS}_{\max}^{\mathrm{LB0}}(\pi^{\mathrm{p}}) \geq f_1(\pi^{\mathrm{p}}) \geq \dots \geq f_m(\pi^{\mathrm{p}})$ holds.

Let $\pi^{\mathrm{p}} \oplus j$ denote the partial solution obtained by appending job $j \in J'$ to π^{p}. We consider $\pi^{\mathrm{p}} \oplus j$ *better than* $\pi^{\mathrm{p}} \oplus j'$ for $j, j' \in J'$ iff there exists an $i \in \{0, \dots, m\}$ such that

$$f_i(\pi^{\mathrm{p}} \oplus j) < f_i(\pi^{\mathrm{p}} \oplus j') \ \wedge \forall i' < i \mid f_{i'}(\pi^{\mathrm{p}} \oplus j) = f_{i'}(\pi^{\mathrm{p}} \oplus j'). \tag{3.16}$$

In other words, the sorted vectors $\mathbf{f}(\pi^{\mathrm{p}} \oplus j)$ and $\mathbf{f}(\pi^{\mathrm{p}} \oplus j')$ are compared in a lexicographic way.

LLBH selects at each iteration a job $j \in J'$ yielding a (locally) best extension. In the case when multiple extensions have equal $\mathbf{f}$-vectors, one of them is chosen at random.

3.5.1 Strengthened Lower Bounds for Partial Solutions

As done in Section 3.4.1, we can also strengthen the lower bound $\mathrm{MS}^{\mathrm{LB0}}(\pi^{\mathrm{p}})$ for a partial solution π^{p} by checking for each secondary resource $r \in R$ if a consecutive schedule of all not yet scheduled jobs $J \cap J_r$ is possible. This is done by adding the strengthening terms $h_r^i(\pi^{\mathrm{p}})$ to $\mathrm{MS}_r^{\mathrm{LB0}}(\pi^{\mathrm{p}})$ for each secondary resource $r \in R$—that is, $\mathrm{MS}_r^{\mathrm{LB}i}(\pi^{\mathrm{p}}) = \mathrm{MS}_r^{\mathrm{LB0}}(\pi^{\mathrm{p}}) + h_r^i(\pi^{\mathrm{p}})$—and taking the maximum of these lower bounds to get the final strengthened lower bound for partial solution π^{p} and strengthening type $i \in \{1, 2\}$, i.e.,

$$\mathrm{MS}^{\mathrm{LB}i}(\pi^{\mathrm{p}}) = \max\left(\mathrm{MS}_0^{\mathrm{LB0}}(\pi^{\mathrm{p}}), \max_{r \in R} \mathrm{MS}_r^{\mathrm{LB}i}(\pi^{\mathrm{p}})\right). \tag{3.17}$$

The terms h_r^1 and h_r^2, for all $r \in R$, can be computed in a way similar to Section 3.4.1, by either considering the maximum time gap $p_{\max}^{\mathrm{prepost}}(J' \cap J_r)$ (see Eq. (3.5)) or by considering a sequence of time gaps $[g_i]_{i=1}^k$ as computed by Algorithm 3.1, respectively. This time, however, we only consider jobs $J' \cap J_r$ that are not already scheduled and require the secondary resource r. Furthermore, we also have to take into account that some resources are already used. If the earliest availability time of resource r is greater than the earliest availability time of resource 0, i.e., $t_r > t_0$ then we also have to consider the additional time lag $t_r - t_0$ for computing h_r^1 or h_r^2. This can be achieved by temporarily adding a *pseudo job* $n+1$ with $q_{n+1} = r$, $p_{n+1}^{\mathrm{pre}} = 0$, $p_{n+1}^0 = t_0$ and $p_{n+1} = t_r$ to set J'. Note that the postprocessing time is $p_{n+1}^{\mathrm{post}} = t_r - t_0$. Hence, for the calculation of the strengthening terms we assume that the already consumed parts of resource r are represented by job $n+1$. Concerning the LLBH, the vector of lower bounds $\mathbf{f}(\pi^{\mathrm{p}})$ can be computed in exactly the same way as described before by using the strengthened lower bounds $\mathrm{MS}_r^{\mathrm{LB}i}(\pi^{\mathrm{p}})$ instead of the basic lower bounds $\mathrm{MS}_r^{\mathrm{LB0}}(\pi^{\mathrm{p}})$ for the secondary resources $r \in R$.

3.5.2 Combined Lower Bounds

Although $\mathrm{MS}^{\mathrm{LB2}}$ is—according to Theorem 3.4.2—the strongest lower bound among the ones described in Section 3.3, in preliminary experiments it turned out that, in practice, the basic lower bound $\mathrm{MS}^{\mathrm{LB0}}$ tends to guide the LLBH better than $\mathrm{MS}^{\mathrm{LB2}}$. This is because, for a given partial solution π^{p}, the evaluation vectors $\mathbf{f}^2(\pi^{\mathrm{p}} \oplus j) = [f^2_0(\pi^{\mathrm{p}} \oplus j), f^2_1(\pi^{\mathrm{p}} \oplus j), \ldots, f^2_m(\pi^{\mathrm{p}} \oplus j)]$ with respect to $\mathrm{MS}^{\mathrm{LB2}}$, for all $j \in J'$, tend to be more similar to each other than the evaluation vectors $\mathbf{f}^0(\pi^{\mathrm{p}} \oplus j) = [f^0_0(\pi^{\mathrm{p}} \oplus j), f^0_1(\pi^{\mathrm{p}} \oplus j), \ldots, f^0_m(\pi^{\mathrm{p}} \oplus j)]$ concerning the basic lower bound $\mathrm{MS}^{\mathrm{LB0}}$. Furthermore, the elements of the vectors $\mathbf{f}^2(\pi^{\mathrm{p}} \oplus j)$ tend to be more similar so that they are not such a good discriminator to select more promising extensions. In order to obtain an evaluation vector with both properties—that is, (1) a good search guidance for the LLBH and (2) a strong lower bound for partial solutions π^{p}—we combine vectors $\mathbf{f}^2(\pi^{\mathrm{p}})$ and $\mathbf{f}^0(\pi^{\mathrm{p}})$ in an interleaved way. More precisely, in order to make use of a combination of lower bounds $\mathrm{MS}^{\mathrm{LB0}}$ and $\mathrm{MS}^{\mathrm{LB2}}$, the vector of lower bounds $\mathbf{f}$ used in LLBH is defined as

$$\mathbf{f}(\pi^{\mathrm{p}}) := \left[f^2_0(\pi^{\mathrm{p}}), f^0_0(\pi^{\mathrm{p}}), \ldots, f^2_m(\pi^{\mathrm{p}}), f^0_m(\pi^{\mathrm{p}})\right]. \tag{3.18}$$

Hereby, the first element is always equal to the stronger lower bound $\mathrm{MS}^{\mathrm{LB2}}$, whereas the second element is always equal to the basic lower bound $\mathrm{MS}^{\mathrm{LB0}}$.

3.6 A* Algorithm and Extensions

The proposed A* algorithm follows more or less the classical principle as described in Section 2.2.2. It performs a minimum cost path search on a weighted directed acyclic state graph from a root node $\mathbf{r}$ to a goal node. The algorithm manages an *open list* Q and the set of all already considered nodes. Each node x can be evaluated by a function $f_*(x) = Z^{\mathrm{sp}}(x) + Z^{\mathrm{h}}(x)$, where the term $Z^{\mathrm{sp}}(x)$ represents the so far smallest known cost from the root node $\mathbf{r}$ to node x, and the heuristic term $Z^{\mathrm{h}}(x)$ represents estimated cost from node x to a goal node. Initially, the root node $\mathbf{r}$ is created and inserted into Q. At each iteration of A* a node x minimizing function $f_*(x)$ is taken from the open list. This node is *expanded* by deriving all possible successor states. For each successor state x' it is checked if a corresponding node already exists. If this is the case, $Z^{\mathrm{sp}}(x')$ is updated if a *dominating* (i.e., shorter) path to x' has been found by reaching x' from x; otherwise, the transition from x to x' can be skipped. If no node exists yet for state x', a new one is created and inserted into the open list Q. The algorithm terminates when A* selects a goal node for expansion.

If function $Z^{\mathrm{h}}(\cdot)$ is *admissible*, meaning that $Z^{\mathrm{h}}(x)$ never overestimates the minimum cost from node x to a goal node for all nodes x and $Z^{\mathrm{h}}(x') = 0$ holds for any goal node x', then A* is known to be complete, i.e., it yields a proven minimum cost path. Under some restrictions and conditions, it can further be shown that—using the same heuristic information of function $Z^{\mathrm{h}}(\cdot)$—there exists no algorithm which expands fewer nodes than A* to find a proven minimum cost path, see Section 2.2.2 for more details.

Our idea is to extend the solution construction principle of LLBH to perform a systematic search for a proven optimal solution. Let us reconsider the evaluation vector $\mathbf{f}(\cdot)$ from the LLBH. Since the first entry of this evaluation vector represents a lower bound for the JSOCMSR according to Theorem 3.4.2, it can be used as admissible evaluation function $f_*(\cdot)$. The vector $\mathbf{t} = [t_r]_{r \in R_0}$ of the trimmed earliest availability times t_r, $r \in R_0$ as defined by Equations (3.9)–(3.12) can be seen as the already occurred costs, and the remaining sums of processing times are the heuristic information used to estimate the makespan of the optimal solution.

It remains to define the nodes in our weighted directed acyclic state graph in more detail. In principle, each node represents any (partial-)schedule that already schedules exactly a specific set of jobs. More precisely, each node contains the following information:

- an unordered set $\hat{J} \subset J$ of already scheduled jobs, implemented by a bit-vector, and
- a set of so-called non-dominated times (NDT) records.

Each NDT record represents a specific partial solution by storing:

- the vector $\mathbf{t}$ of earliest availability times,
- the last scheduled job $j^{\mathrm{last}} \in \hat{J}$ after which $\mathbf{t}$ was obtained, and
- an evaluation vector $\mathbf{f}'$ similar to $\mathbf{f}$ with one modification that will be described below.

Thus, each node aggregates all partial solutions π^{p} having the same jobs $\hat{J}$ scheduled, and each NDT record provides more specific information for each non-dominated partial solution. In our state graph there is exactly on goal node $\mathbf{g}$ which represents any complete schedule and one root node $\mathbf{r}$ which represents the empty schedule. The ordering of the scheduled jobs $\hat{J}$ of a specific partial solution is indirectly given. For a pair of a node and one of its NDT records—henceforth called a (node, NDT record) pair—the corresponding ordering can be derived in a reverse iterative manner by considering the fitting preceding (node, NDT record) pairs, always continuing with a pair where the node is characterized by $\hat{J} \setminus \{j^{\mathrm{last}}\}$ and the NDT record has times t_r allowing to schedule job j^{last} without exceeding the t_r values of the previous (node, NDT record) pair.

Initially a starting (node, NDT record) pair corresponding to the empty schedule is generated with $\hat{J} = \emptyset$, $\mathbf{t} = \mathbf{0}$, $j^{\mathrm{last}} = \text{none}$, and $\mathbf{f}' = (\mathrm{MS}^{\mathrm{LB}i}, \ldots, \mathrm{MS}^{\mathrm{LB}i})$ with a chosen lower bound type $i \in \{0, 1, 2\}$. The goal node is the node with $\hat{J} = J$, corresponding to all complete solutions. See Figure 3.3 for an illustration of a state graph.

The set of all so far considered nodes is implemented by a hash-table with $\hat{J}$ as key in order to efficiently find already existing nodes for reached states. The open list is more specifically realized by a priority queue Q containing references to all *open* (node,

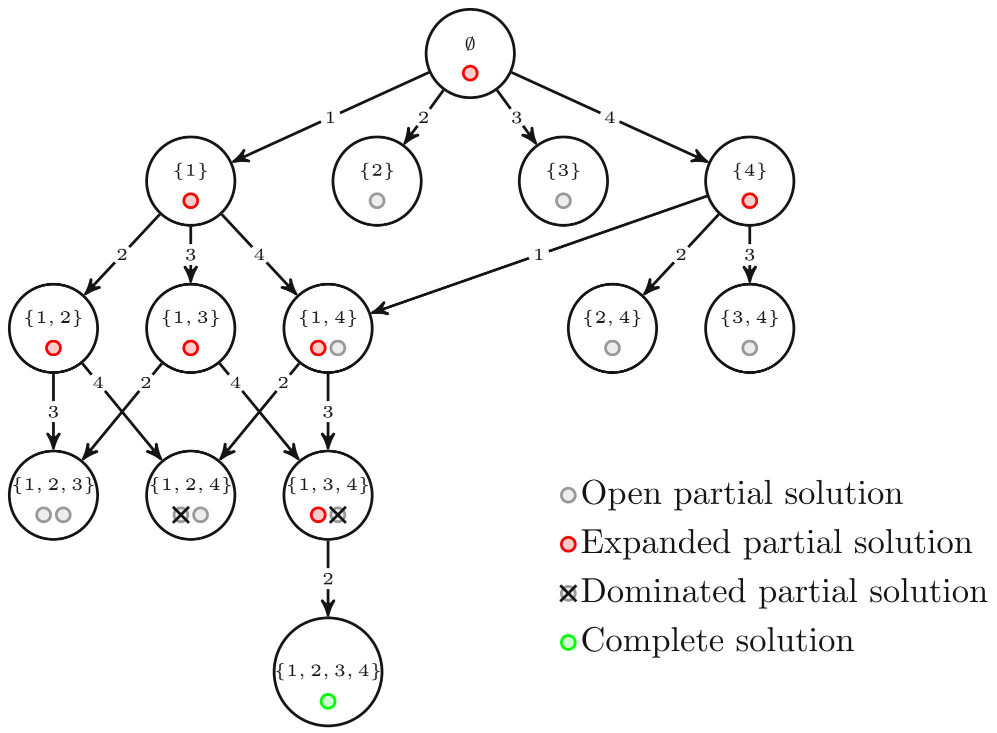


Figure 3.3: Example of a partial state graph created by A* search with 4 jobs $J = \{1, 2, 3, 4\}$. Each node contains an unordered set $\hat{J} \subseteq J$ of already scheduled jobs and a set of NDT records depicted as small circles. Gray NDT records are currently in the open list whereas red NDT records are already expanded during the search. Crossed NDT records got dominated by another NDT record and are therefore removed from the set of NDT records as well as from the open list. Arc labels corresponds to a scheduled job $j \in J \setminus \hat{J}$. Note that this is just an abstract example to sketch a state graph. Therefore, we omit further details including earliest availability times, last scheduled jobs, and evaluation vectors of NDT records.

NDT record) pairs, that is, the non-dominated partial solutions that have not yet been expanded. As order criterion the *is-better* relation from the LLBH (see Eq. 3.16) is used.

Note that the first entry of our evaluation vector $\mathbf{f}$ from LLBH is not necessarily monotonically non-decreasing when considering any path from $\mathbf{r}$ to $\mathbf{g}$. Thus, it can happen that the lower bound $f_0^i(\pi^p \oplus j)$ of an extension of a partial solution π^p with job $j \in J \setminus \hat{J}(\pi^p)$ is less than the lower bound $f_0^i(\pi^p)$. To establish monotonicity at least for the first element in vector $\mathbf{f}$—which corresponds to the lower bound of π^p—we use a modified evaluation vector $\mathbf{f'}$ during the A* search, where the first element $f_0'^i(\pi^p \oplus j)$ is always set to $\max(f_0'^i(\pi^p), f_0'^i(\pi^p \oplus j))$. In this way, the largest lower bound along a path from $\mathbf{r}$ to $\mathbf{g}$ is always preserved.

Algorithm 3.3 provides a pseudo-code of our A* algorithm, already including details about the embedded beam search and local search, which will be described in the following subsections. In each major iteration, a best (node, NDT record) pair is taken from the open list Q in line 12 and expanded in line 19 by considering the addition of each job $j \in J \setminus \hat{J}$ as shown separately in Algorithm 3.4. Hereby, the corresponding node x is looked up—or created, in case it does not yet exist—and a respective NDT record is determined by calculating the earliest usage times $\mathbf{t}$ and the evaluation vector $\mathbf{f}'$. If the lower bound of the first element of vector $\mathbf{f}'$ is larger than the makespan of our current best solution $\mathrm{MS}(\pi^{\mathrm{best}})$, this new NDT record cannot lead to a better solution. Therefore, in this case the new NDT record is disregarded and we move on to the next unscheduled job $j \in J \setminus \hat{J}$. Otherwise, the possibly multiple NDT records in the node are checked for dominance: Only non-identical and non-dominated entries are kept. An NDT record with time vector $\mathbf{t}$ *dominates* (denoted by $\lhd$) another NDT record with time vector $\mathbf{t}'$ iff $\forall r \in R_0\ (t_r \leq t'_r) \wedge \exists r \in R_0\ (t_r < t'_r)$. Algorithm 3.4 returns all newly created (node, NDT record) pairs which are then inserted into the open list. Note that we implicitly assume that (node, NDT record) entries in the open list Q corresponding to dominated and therefore removed NDT records are also deleted or skipped when selected for expansion. The A* algorithm stops when the goal node representing a complete solution is selected for expansion. This complete solution must be optimal since f'_0 is an admissible lower bound for the makespan.

3.6.1 Advanced Diving with Beam Search

The A* algorithm described above aims at finding a proven optimal solution as quickly as possible. Feasible solutions are, however, usually only found very late. To turn A* into an anytime algorithm which is able to find also intermediate complete heuristic solutions significantly earlier than when terminating with a proven optimum we proposed in our preliminary work [82] a LLBH-based diving extension: In regular intervals, the search switches from the A* strategy temporarily to depth-first search node expansion following the successor selection from LLBH until a complete solution is obtained. In this work we extend this simple diving to a more powerful beam search (BS).

Beam search is a heuristic search method to solve combinatorial optimization problems. The search is performed on a graph and can be seen as an extension of depth first search. However, instead of expanding always the most promising node at each level of the generated search tree, BS expands the k_{bw}-most promising nodes, where parameter k_{bw} is also called *beam width*. Note that if $k_{\mathrm{bw}} = 1$ BS behaves exactly like greedy depth first search until a complete solution has been reached. Beam search was first used by Lowerre [110] in a speech recognition system and by Rubin [138] for image recognition. Furthermore, BS was also successfully applied to scheduling problems such as job shop scheduling [56, 140] and single machine scheduling [159].

Our Algorithm 3.3 switches at the very beginning and after each δ regular iterations from its classical best-first strategy temporarily to a BS-based completion strategy in order to find promising complete solutions at regular intervals. After such a switch to diving

Algorithm 3.3: A*+BS+LS Algorithm for the JSOCMSR

```
1  initialize open list $Q$ and $B$ with $(\emptyset, (\mathbf{0}, \text{none}, (\text{MS}^{\text{LB0}}, \ldots, \text{MS}^{\text{LB0}}))$;
2  $iter \leftarrow 0$, diving $\leftarrow$ $true$, $B_{\text{ext}} \leftarrow \emptyset$;
3  repeat
4      if $diving$ then
5          if $B = \emptyset$ then
6              $B \leftarrow$ the best $k_{\text{bw}}$ entries from $B_{\text{ext}}$ according to $\mathbf{f}'$;
7              $B_{\text{ext}} \leftarrow \emptyset$;
8          end
9          $(\hat{J}, (\mathbf{t}, j^{\text{last}}, \mathbf{f}')) \leftarrow$ select randomly one entry from $B$;
10         remove $(\hat{J}, (\mathbf{t}, j^{\text{last}}, \mathbf{f}'))$ from Q and $B$;
11     else
12         $(\hat{J}, (\mathbf{t}, j^{\text{last}}, \mathbf{f}')) \leftarrow$ Q.pop();
13     end
14     if $|\hat{J}| = n$ and not $diving$ then
15         $\pi^{\text{best}} \leftarrow$ derive complete solution from $(\hat{J}, (\mathbf{t}, j^{\text{last}}, \mathbf{f}'))$;
16         return $proven\ optimal\ solution\ \pi^{\text{best}}$
17     end
18     if $iter$ mod $\delta = 0$ then diving $\leftarrow$ $true$;
19     $E \leftarrow$ expand$((\hat{J},(\mathbf{t}, j^{\text{last}}, \mathbf{f}')), \pi^{\text{best}}, \text{diving})$;          // see Alg. 3.4
20     Q.insert$(E)$;
21     apply LS to all complete solutions in $E$;          // see Alg. 3.5
22     if $new\ \pi^{\text{best}}\ obtained\ from\ LS$ then
23         start injection of $\pi^{\text{best}}$;          // see Alg. 3.6
24     end
25     if $diving$ then $B_{\text{ext}} \leftarrow B_{\text{ext}} \cup E$;
26     if $not\ diving$ then $iter \leftarrow iter + 1$;
27     if $B = \emptyset \wedge B_{\text{ext}} = \emptyset$ then diving $\leftarrow$ $false$;
28 until $time$- $or\ memory$-$limit\ reached$;
29 return $heuristic\ solution\ \pi^{\text{best}}\ and\ lower\ bound\ f_0$
```

mode (indicated by setting the parameter diving to *true*), the currently selected node is expanded. From all obtained extensions, only those that are new and non-dominated are kept and added to the open list Q and to the set of current beam extensions B_{ext}. Afterwards only the best k_{bw} extensions from set B_{ext} are filtered out as beam set B. The BS continues by expanding all extensions from set B with Algorithm 3.4. Again, the hereby created extensions are added to open list Q and set B_{ext} if they are new and non-dominated. If an extension represents a complete solution then we also keep it even if the corresponding earliest availability times are dominated. The reason for this exception is that we apply a local search procedure to all complete solutions later in line 21 of Algorithm 3.3. If all extensions from set B are expanded, set B is replaced by

Algorithm 3.4: A* Node Expansion Algorithm

```
Input: (node, NDT record) pair $(\hat{J},(\mathbf{t}, j^{\text{last}}, \mathbf{f}'))$, current best solution $\pi^{\text{best}}$,
       parameter diving indicating if in diving mode or not
Output: set $E$ of extensions, i.e., new non-dominated (node, NDT record) pairs
1  $E \leftarrow \emptyset$;
2  forall $j \in J \setminus \hat{J}$ do
3      find or create node $x$ with $\hat{J}(x) = \hat{J} \cup \{j\}$;
4      calculate new NDT record $(\mathbf{t}_{\text{new}}, j, \mathbf{f}'_{\text{new}})$ from $\mathbf{t}$;
5      if diving or $\text{MS}(\pi^{\text{best}}) \geq f'_{0,\text{new}}$ then
6          if $\nexists(\mathbf{t}_{\text{d}}, j_{\text{d}}^{\text{last}}, \mathbf{f}'_{\text{d}}) \in NDTs(x) \mid \mathbf{t}_{\text{d}} = \mathbf{t}_{new} \vee \mathbf{t}_{\text{d}} \lhd \mathbf{t}_{new}$ then
7              Remove every $(\mathbf{t}_{\text{d}}, j'_{\text{d}}, \mathbf{f}'_{\text{d}}) \in \text{NDTs}(x) \mid \mathbf{t}_{\text{new}} \lhd \mathbf{t}_{\text{d}}$;
8              Add $(\mathbf{t}_{\text{new}}, j, \mathbf{f}'_{\text{new}})$ to $\text{NDTs}(x)$;
9              $E \leftarrow E \cup \{(\hat{J}, (\mathbf{t}_{\text{new}}, j, \mathbf{f}'_{\text{new}}))\}$;
10         else if $|\hat{J}(x)| = n$ then
11             $E \leftarrow E \cup \{(\hat{J}, (\mathbf{t}_{\text{new}}, j, \mathbf{f}'_{\text{new}}))\}$;
12         end
13     end
14 end
```

the best k_{bw} extensions from set B_{ext}. The process continues until set B as well as set B_{ext} become empty, in which case the algorithm switches back to A*'s normal best-first search strategy. Note that for beam width $k_{\text{bw}} = 1$, our A* embedded BS corresponds to the simple diving from [82]. Furthermore, each (node,NDT) pair which is expanded during the BS phase is kept in the A* search tree such that A* will avoid expanding them a second time.

3.6.2 Local Search and Solution Injection

We attempt to further improve any complete intermediate solution by the following local search (LS) procedure based on the *insertion neighborhood*. Given a solution π, the insertion neighborhood contains all solutions that can be obtained by removing a job $j \in J$ from π, which we denote by $\pi \ominus j$, and reinserting j at another position. Following a best-improvement strategy, a best neighbor of the current solution always is selected as incumbent solution for the next step until no further improvement is possible and thus a local optimum has been reached. Algorithm 3.5 sketches this procedure.

Each insertion neighborhood is efficiently searched by only considering job removals that actually may improve the makespan. To this end, let us consider the dependency graph $G^{\text{d}}(\pi) = [V^{\text{d}}(\pi), A^{\text{d}}(\pi)]$ of solution π whose nodes $V^{\text{d}}(\pi)$ correspond to the jobs J and whose arcs $A^{\text{d}}(\pi)$ represent the dependencies in such a way that there is an arc (j, j') between any two jobs $j, j' \in J$, $j \neq j'$ iff job j' starts to use some resource immediately after job j has released this resource. All the paths in this dependency graph from any

Algorithm 3.5: Local Search Procedure

```
   Input: solution $\pi$
1  $\pi^{\text{best}} \leftarrow \pi$;
2  repeat
3      $\pi \leftarrow \pi^{\text{best}}$;
4      determine critical jobs $J^{c}(\pi)$ by breadth-first search of dependency graph of $\pi$;
5      forall $j \in J^{c}(\pi) \mid \mathrm{MS}(\pi \ominus j) < \mathrm{MS}(\pi^{best})$ do
6          $\pi' \leftarrow$ greedily insert $j$ into $\pi \ominus j$ by trying all alternative positions;
7          if $\mathrm{MS}(\pi') < \mathrm{MS}(\pi^{best})$ then
8              $\pi^{\text{best}} \leftarrow \pi'$;
9          end
10     end
11 until $\mathrm{MS}(\pi^{best}) = \mathrm{MS}(\pi)$;
12 return $\pi^{best}$
```

scheduled job starting at time point zero to any job finishing last are so-called *critical paths*, and together, they define the makespan. Now, observe that only the removal of some node (job) lying on *all* these critical paths will yield an immediate reduction of the makespan, and only these jobs are therefore of interest to find a better solution within the insertion neighborhood. We call these jobs *critical jobs* $J^{c}(\pi)$ and determine them by breadth-first search (BFS) of the dependency graph in time $O(nm)$. Figure 3.4 shows an example of such a dependence graph, for more information about critical paths we refer to [129]. Moreover, only those critical jobs whose removal results in a partial solution with a makespan that is lower than the makespan of the current best solution are considered for greedy insertion at a best alternative position in order to determine a best neighbor π^{best}. In preliminary experiments, we also tested an exchange neighborhood in which pairs of jobs are swapped as an alternative or in addition to the insertion neighborhood. However, it turned out that the evaluation of the exchange neighborhood is too expensive regarding running time such that its application usually does not pay off.

If the LS procedure could improve the solution obtained from diving/BS then the solution is injected into the A* search graph to possibly benefit from the respective nodes and NDT records. Furthermore, we thereby want to guide the A* search better into more promising areas of the search space so that A* is able to find faster an optimal solution or to return with a smaller optimality gap in case of a termination due to the time or memory limit, and we want to avoid expanding nodes a second time. Algorithm 3.6 shows the details of the injection operation. Let π^{imp} be the improved solution. The nodes are expanded along the improved solution π^{imp}, starting at the root node with the set of already scheduled jobs $\hat{J} = \emptyset$ and the corresponding NDT record with the vector of earliest availability times $\mathbf{t}_0 = \mathbf{0}$. Note, that we assume that this root node/NDT pair is already created by the A* algorithm before Algorithm 3.6 is executed. If the root node/NDT pair is not yet expanded then we expand it. Starting with $k = 1$, job

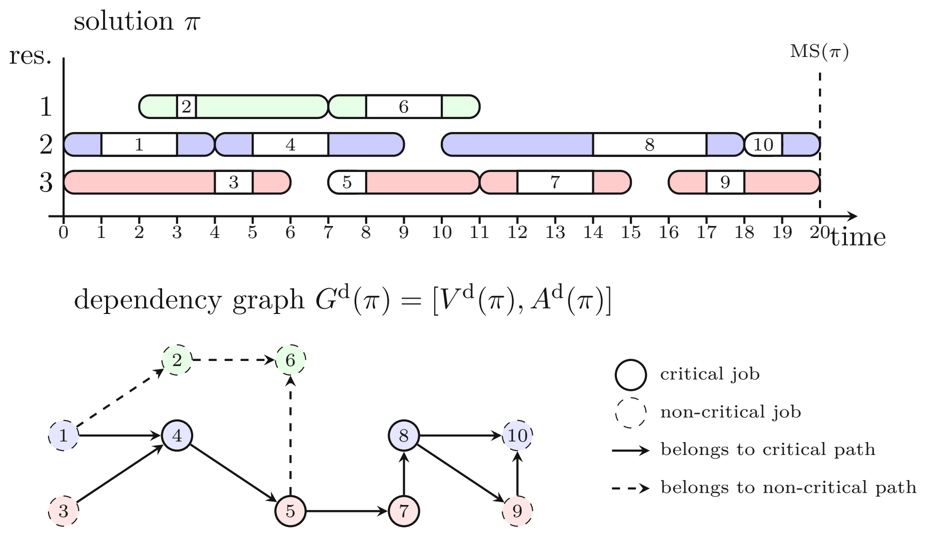


Figure 3.4: Example for dependency graph of solution π with critical jobs $J^{\mathrm{c}} = \{4, 5, 7, 8\}$.

π_k^{imp} is dealt with by setting $\hat{J} \leftarrow \hat{J} \cup \{\pi_k^{\mathrm{imp}}\}$ and computing $\mathbf{t}_k$ from $\mathbf{t}_{k-1}$ and π_k^{imp}. Moreover, node x with $\hat{J}(x) = \hat{J}$ is retrieved and, among the NDT records of x, we search for an NDT record η whose earliest availability times are equal to $\mathbf{t}_k$. If such an NDT record does not exist, there must be at least one dominant NDT record whose earliest availability times dominate $\mathbf{t}_k$. Therefore we select one such dominant NDT record η with earliest availability time $\mathbf{t}$ and set $\mathbf{t}_k \leftarrow \mathbf{t}$. If the (node, NDT record) pair (x, η) is not yet expanded then we expand it by considering all possible extensions and keep only new and non-dominated (node, NDT record) pairs, which are also inserted into the open

Algorithm 3.6: Solution Injection

```
    Input: open list $Q$, improved solution $\pi^{\mathrm{imp}}$
 1  initialize $\hat{J} \leftarrow \emptyset$, $\mathbf{t}_0 \leftarrow \mathbf{0}$;
 2  retrieve starting node $x$ with $\hat{J}(x) = \emptyset$;
 3  get NDT record $\eta = (\mathbf{t}, j^{\mathrm{last}}, \mathbf{f}) \in \mathrm{NDTs}(x)$ s.t. $\mathbf{t} = \mathbf{0}$;
 4  for $k = 1 \ldots n$ do
 5      if $(\hat{J}, \eta)$ is not expanded then
 6          $E \leftarrow$ expand$((\hat{J}, \eta), \pi^{\mathrm{imp}}$, diving=false);        // (see Alg. 3.4)
 7          Q.insert($E$);
 8      end
 9      $\hat{J} \leftarrow \hat{J} \cup \{\pi_k^{\mathrm{imp}}\}$ and calculate $\mathbf{t}_k$ from $\mathbf{t}_{k-1}$ and job $\pi_k^{\mathrm{imp}}$;
10      retrieve node $x$ with $\hat{J}(x) = \hat{J}$;
11      get NDT record $\eta = (\mathbf{t}, j^{\mathrm{last}}, \mathbf{f}') \in \mathrm{NDTs}(x)$ s.t. $\mathbf{t} = \mathbf{t}_k \vee \mathbf{t} \lhd \mathbf{t}_k$;
12  end
```

list Q. Afterwards, k is incremented and the next job in solution π^{imp} is considered. This procedure is repeated until the last job of solution π^{imp}—that is, π_n^{imp}—has been dealt with. Preliminary tests show that injecting improved solutions from LS back into the A* search graph, leads to an overall performance boost of the A* search. In particular for larger instances, A* benefits from the fact that the search is led faster to promising areas and avoids the expansion of nodes a second time so that on average smaller optimality gaps can be obtained faster. In the following, we denote the A* algorithm variant with the embedded BS and LS as A*+BS+LS.

3.7 Mixed Integer Linear Programming Formulation

For comparison purposes, we consider the following position-based MILP formulation from [82], which models solutions to the JSOCMSR in terms of permutations of all jobs. Index $i \in \{1, \dots, n\}$ refers to position i in a permutation. Decision variables $x_{j,i} \in \{0, 1\}$, for all $j \in J$ and $i \in \{1, \dots, n\}$, are set to one iff job j is assigned to position i in the permutation. Variables $s_i \geq 0$ represent the starting time of the jobs scheduled at each position $i = 1, \dots, n$ in the permutation. Finally, $\mathrm{MS} \geq 0$ is the makespan variable to be minimized.

$$\min \mathrm{MS} \tag{3.19}$$

$$\sum_{j \in J} x_{j,i} = 1 \qquad i = 1, \dots, n \tag{3.20}$$

$$\sum_{i=1}^{n} x_{j,i} = 1 \qquad j \in J \tag{3.21}$$

$$s_i + \sum_{j \in J} x_{j,i} \cdot p_j \leq \mathrm{MS} \qquad i = 1, \dots, n \tag{3.22}$$

$$s_1 = 0 \tag{3.23}$$

$$s_i + \sum_{j \in J} x_{j,i} \cdot p_j^{\mathrm{pre}} \geq s_{i-1} + \sum_{j \in J} x_{j,i-1} \cdot (p_j^{\mathrm{pre}} + p_j^0) \qquad i = 2, \dots, n \tag{3.24}$$

$$s_{i'} - s_i + \sum_{j \in J_r} x_{j,i'}(M + p_j) + \sum_{j \in J_r} x_{j,i} M \leq 2M$$
$$i = 2, \dots, n, \; i' = 1, \dots, i-1, \; r \in R \tag{3.25}$$

$$x_{j,i} \in \{0, 1\} \qquad j \in J, \; i = 1, \dots, n \tag{3.26}$$

$$s_i \geq 0 \qquad i = 1, \dots, n \tag{3.27}$$

$$\mathrm{MS} \geq 0 \tag{3.28}$$

Equations (3.20) ensure that exactly one job is assigned to the i-th position of the permutation and (3.21) ensure that each job is assigned to exactly one position. The makespan is determined by inequalities (3.22). Equation (3.23) sets the starting time of the first job in the permutation to zero, and the remaining two sets of inequalities make sure that no resource is used by more than one job at a time. Hereby, inequalities (3.24) take care of the common resource 0, while (3.25) consider the secondary resources. The Big-M constant in these latter inequalities is set to the makespan obtained by LLBH.

Note that we considered three different MILP models for the JSOCMSR in the course of our already mentioned preliminary work on the JSOCMSR [82]: (1) a time-indexed based formulation, (2) a position-based formulation and (3) a disjunctive MILP model. All three variants were clearly outperformed by our anytime A* approach. The best results among the three MILP models were obtained from the position based formulation, and its results were therefore included in [82]. For the sake of completeness we reconsider here also the position-based MILP formulation.

3.8 Constraint Programming Formulation

The CP model proposed in the following for the JSOCMSR is also used for comparison purposes. It makes use of so-called *interval variables* which represent intervals of time and are a specific feature of ILOG CP Optimizer.[1] For each job $j \in J$ we use two such interval variables: (1) x_j, indicating the time interval during which resource q_j is consumed (p_j units of time), and (2) x_j^0, indicating the time interval during which the common resource 0 is consumed (p_j^0 units of time) by job j. One feature of ILOG CP interval variables is that they can be absent or present in the CP model to consider also scenarios where for instance not all jobs have to be scheduled. Since in our case all jobs needs to be scheduled to get a feasible solution, all used interval variables are present in the CP model. The CP model is given by

$$\min \max_{j \in J} \mathrm{end}(x_j) \tag{3.29a}$$

$$\mathrm{startAtStart}(x_j, x_j^0, p_j^{\mathrm{pre}}) \qquad \forall j \in J \tag{3.29b}$$

$$\mathrm{noOverlap}(\{x_j \mid j \in J_r\}) \qquad \forall r \in R \tag{3.29c}$$

$$\mathrm{noOverlap}(\{x_j^0 \mid j \in J\}) \tag{3.29d}$$

$$x_j : \text{interval variable of size } p_j \qquad \forall j \in J \tag{3.29e}$$

$$x_j^0 : \text{interval variable of size } p_j^0 \qquad \forall j \in J \tag{3.29f}$$

where start(x_j) and end(x_j) represents the start time and the end time of interval x_j, respectively. Constraints (3.29b) ensure that each job $j \in J$ consumes the common resources 0 exactly p_j^{pre} units of time after the starting time of j using the ILOG CP constraint startAtStart. Constraints (3.29c) guarantee that each secondary resource $r \in R$ is not used more than once at the same time, whereas Constraint (3.29d) ensures that the common resource 0 is not used more than once at the same time. This is done using the ILOG CP constraint noOverlap which ensures that all intervals in a given set of interval variables are pairwise non-overlapping.

[1]https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-cp-optimizer

3.9 Variable Neighborhood Search

To heuristically solve instances of the JSOCMSR that could not be solved to proven optimality by A*+BS+LS we use a GVNS, where two different sets of neighborhood structures $\mathcal{N}^{\text{vnd}}_{i=1\dots k_{\max}}$ and $\mathcal{N}^{\text{vns}}_{i=1\dots l_{\max}}$ are alternatingly applied in intensification and diversification phases. See Section 2.4.4 for a detailed introduction to GVNS. In the intensification phase, a deterministic variable neighborhood descent (VND) uses a set of $k_{\max} = 4$ intensification neighborhood structures, which are searched, depending on their computational cost, in either a first-improvement or best-improvement manner. In the diversification phase a set of $l_{\max} = 23$ increasingly perturbative shaking neighborhood structures are used to perform random moves in order to reach parts of the search space that are farther away from the incumbent solution. Algorithm 3.7 illustrates this procedure. The initial solution—represented by permutation π—is created uniformly at random. The GVNS terminates if a certain time-limit is exceeded or the incumbent solution's objective value corresponds to the strongest lower bound MS^{LB2} obtained from Section 3.4. In the latter case a proven optimal solution has been found.

Algorithm 3.7: GVNS for JSOCMSR

```
   Input: initial solution $\pi$, $\mathcal{N}^{\text{vnd}}_{i=1,\dots,k_{\max}}$, $\mathcal{N}^{\text{vns}}_{j=1,\dots,l_{\max}}$
 1 $\pi^{\text{best}} \leftarrow \pi$; $l \leftarrow 1$;
 2 repeat
 3     $\pi' \leftarrow \text{Shake}(\mathcal{N}^{\text{vns}}_l, \pi^{\text{best}})$;              // diversification
 4     $\pi'' \leftarrow \text{VND}(\mathcal{N}^{\text{vnd}}, \pi')$;                 // intensification
 5     $l \leftarrow l + 1$;
 6     if $\text{MS}(\pi') < \text{MS}(\pi^{best})$ then         // new incumbent solution found
 7         $\pi^{\text{best}} \leftarrow \pi'$; $l \leftarrow 1$;
 8     else if $l > l_{\max}$ then
 9         $l \leftarrow 1$;
10     end
11 until $\text{MS}^{\text{LB2}} = \text{MS}(\pi^{best}) \vee$ time-limit reached;
12 return $\pi^{best}$
```

3.9.1 Solution Representation and Evaluation

As mentioned in Section 3.3, our GVNS interprets solutions of the JSOCMSR as linear permutations that state the order in which the jobs acquire the common resource 0. To obtain the makespan $\text{MS}(\pi)$ of such a permutation π, the exact starting time s_j for each job $j \in J$ must be determined. This is done by a linear time decoder that greedily schedules each job as soon as its resources become available. As it becomes quite inefficient to naively apply this decoder during neighborhood evaluation, we propose an incremental evaluation scheme in which it is not always necessary to (re-)determine the starting time for each job to obtain its makespan.

However, due to the incremental nature of the decoding mechanism and a solution's consequential characteristic, that even small structural changes—like the removal of a job from its current position–potentially propagate to distant sections in the solution, a strictly constant-time incremental evaluation schema is not possible. Instead, we concentrated on an alternative approach, where a certain subsection of a neighboring solution is evaluated until a point of *synchronization* with respect to the incumbent solution is identified. After this point, no structural differences besides a fixed time offset occur. This point of synchronization in the permutation resides at the end of a so-called *synchronization border*, consisting of a minimal set of jobs on different secondary resources which are aligned w.r.t. their starting times in the incumbent solution and the respective neighboring solution in the same way. In the following we define this formally.

Definition 3.9.1 (Synchronization Border)
Given two solutions π, π' and the respective normalized starting times s and s', where π' is a neighbor of π w.r.t. some neighborhood structure $\mathcal{N}$. Assume further that the underlying permutation of jobs has only changed up to position i, $0 \leq i < n$. The synchronization point is then the smallest position i' with $i < i' \leq n$, where a set of jobs $\mathcal{B} \subseteq \{\pi_k \mid k = i+1, \ldots, i'\}$, denoted as the synchronization border, satisfies the following conditions:

1. The set contains exactly one job for each secondary resource that is still claimed by a job in the permutation at or after the synchronization point i'.
2. The jobs are aligned with respect to their starting times in the same way in s and s', i.e., $\exists c \in \mathbb{Z} \ \ \forall j \in \mathcal{B} : s_j - s'_j = c$.

In order to evaluate the makespan of a neighbor π' of the incumbent solution π, our approach starts at the first position in the permutation subject to the structural change induced by the move in the neighborhood and scans through the permutation to identify the synchronization border. As soon as the synchronization border is established we are able to determine the *alignment offset* c, i.e., the time difference between the solutions concerning the border, and, consequently, can immediately derive the makespan $\mathrm{MS}(\pi')$ of the neighbor solution π'. Figure 3.5 illustrates this approach, where a neighboring solution π' on the bottom is derived from π by removing job 4 from position 9 and reinserting it at its new position 4. In this example, the synchronization border $\mathcal{B} = \{5, 7, 6\}$ can be determined already after three steps, allowing to derive the makespan of π' already at position 8.

As identifying the synchronization border in a naive iterative way requires time $\mathcal{O}(nm)$ in the worst case, we use additional auxiliary data structures for each incumbent solution that frequently allow skipping certain parts of the scan through the permutation. In this way the synchronization border can typically be identified much quicker and as a consequence the exploration of the neighborhoods is more efficient. Besides simple lookup tables to detect, for instance, the last job on a particular resource, most importantly,

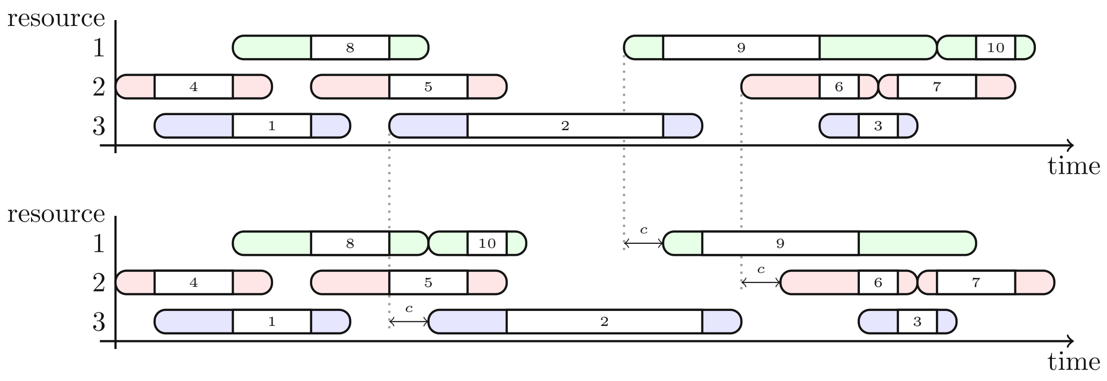


Figure 3.5: Illustration of an incumbent solution (top) and a neighboring solution obtained after moving job 10 (bottom) and their synchronization border $\mathcal{B} = \{2, 6, 9\}$.

our approach relies on a data structure $\alpha(\pi) = (\alpha_{i,r}(\pi))_{i=1,\ldots,n,\, r \in R_0}$ indicating for each position i in permutation π the time from which on each resource r is available for scheduling a job at this position i. Thus, $\alpha_{i,r}(\pi)$ can be used to quickly determine the starting time of a job which should be inserted in π at position i. As all our neighborhood structures are essentially defined by removing and re-inserting jobs in the permutation representation in certain ways, this data structure allows us to immediately determine the starting time of an inserted job at any position, subsequently requiring only the identification of the synchronization border to determine the implied change in the makespan. Although the preparation of these data structures comes with an additional computational cost of $\mathcal{O}(nm)$ per incumbent solution for which the VND is started, our experiments in Section 3.10 indicate that in practice the whole approach requires only constant amortized runtime with respect to the number of jobs for identifying the synchronization border and thus the makespan of a neighboring solution.

3.9.2 Intensification

The VND, which is responsible for intensification within the GVNS, makes use of a set of neighborhood structures for linear permutations, as formally defined by Schiavinotto and Stützle [141].

The *insertion neighborhood* $\mathcal{N}_I(\pi)$ of an incumbent solution π consists of any solution π' obtained by removing any job j from its current position in π and reinserting it at any other position. We efficiently evaluate the whole neighborhood by considering the removal of each job $j \in J$ in an outer loop, yielding a partial solution $\pi \ominus j$ for which the corresponding auxiliary data structure $\alpha(\pi \ominus j)$ is derived and the partial neighborhood $\mathcal{N}'_I(\pi \ominus j, j)$ corresponding to the re-insertion of j at any position except the original one is evaluated in an inner loop. Algorithm 3.8 shows in more detail how the neighbor solution in which job j is re-inserted at a position i in the partial solution $\pi \ominus j$ is evaluated by determining the synchronization border and the respective alignment offset.

Algorithm 3.8: Evaluation of the neighbor: reinsert job j at position i

```
   Input: partial solution $\pi \ominus j$, insertion position $i$, resource avail. times $\alpha(\pi \ominus j)$
1  $t_r \leftarrow \alpha_{i,r}(\pi \ominus j), \forall r \in R_0$;
2  synchronization border $\mathcal{B} = \emptyset$, aligned offset $c \leftarrow 0$;
3  for $k = i, \dots, |\pi \ominus j|$ do                // evaluate $\pi \ominus j$ from $i$ onwards
4      $j' \leftarrow (\pi \ominus j)_k$;
5      $s_{j'} \leftarrow \max\{t_0 - p^{\mathrm{pre}}_{j'}, t_{q_{j'}}\}$;   // evaluate new starting time for $j'$
6      $t_0 \leftarrow s_{j'} + p^{\mathrm{pre}}_{j'} + p^0_{j'}$; $t_{q_{j'}} \leftarrow s_{j'} + p^{\mathrm{pre}}_{j'}$;
7      update $\mathcal{B}$ with job $j'$;
8      if $\mathcal{B}$ satisfies conditions from Definition 3.9.1 then
9          $c \leftarrow$ derive alignment offset from $\mathcal{B}$ and $\pi$;
10         break;
11     end
12 end
13 return $\mathrm{MS}(\pi \ominus j) + c$
```

Based on this evaluation scheme, it turned out to be advantageous in the implementation to further divide the insertion neighborhood $\mathcal{N}_I(\pi)$ into forward and backward insertion neighborhoods such that jobs are only allowed to move forward or backward in the permutation, respectively. This allows reusing some part of the auxiliary data structures for the entire neighborhood evaluation.

The exchange neighborhood $\mathcal{N}_X(\pi)$, contains any solution derived from the incumbent π by exchanging any pair of jobs in the permutation. Again, the neighborhood evaluation is based on determining synchronization borders, but instead of using intermediate partial solutions, a dual synchronization approach has been devised, where the neighborhood operation is essentially reduced to two insertion operations, where both the offset between the respective exchanged jobs and the offset of the latter job to the makespan are obtained with the synchronization technique.

In addition to efficient evaluation schemes for the considered neighborhood structures, we further studied different approaches to reduce neighborhood sizes in order to avoid the evaluation of unpromising neighbors at all. Besides neighborhood reduction based on critical jobs as proposed in Section 3.6.2, we also considered heuristic approaches like avoiding to schedule two jobs of the same secondary resource consecutively or reducing the size of neighborhoods by limiting the maximum distance of move operations. While these pruning techniques bring the danger of quickly approaching local optima of rather poor quality, concentrating on critical jobs is particularly advantageous in the very beginning of the search. Limiting the maximum distance of move operations particularly showed its effectiveness for exchange neighborhoods, where instead of the dual synchronization evaluation scheme, it becomes more the better option to partially evaluate the entire range between the positions of the two exchanged jobs and perform a single synchronization step at the end of this range. Experimentally, we determined a move distance limitation of $k = 50$ to provide a good trade-off between the size of the neighborhood and its evaluation's efficiency in the context of our benchmark instances. Nevertheless, note that these restricted neighborhoods are primarily used in early VND phases, while more comprehensive neighborhoods become important in latter phases to compensate the limitations. More details on the pruning techniques and their impacts can be found in Thomas Kaufmann's master thesis [94]. Here, we will only look more closely on the limitation of move distances.

We used findings of a landscape analysis, where the average quality and depth of local optima were studied to prepare a meaningful parameter tuning configuration, and then applied `irace` [109] to select concrete neighborhood structures and parameters like the step function by which the neighborhoods are searched in the VND. For details regarding the parameter tuning setup we refer to [94]. Finally, we investigated the temporal behavior of our algorithm in a set of experiments to decide the neighborhood change function in the VND [69]. Again, more details on this preliminary investigations can be found in [94].

The finally resulting VND configuration uses four neighborhood structures, subject to a piped neighborhood change function [69]. First, an exchange neighborhood structure with a move distance limitation of 50 is used in conjunction with a first-improvement step function to quickly identify local optima of already relatively high quality. This is followed by the backward insertion neighborhood structure searched in a best improvement manner. Next, the unconstrained exchange neighborhood structure is used and finally the unconstrained insertion neighborhood structure, again searched in first and best improvement manners, respectively.

3.9.3 Diversification

For diversification, the GVNS applies moves from a total of $l_{\max} = 23$ shaking neighborhood structures to the incumbent solution, where each shaking neighborhood $\mathcal{N}_i^{\text{vns}}$ is parameterized by κ_i describing the number of subsequent applications of the underlying neighborhood move. In order to enable our shaking procedure to introduce fine-grained structural changes into the incumbent solution, we use the exponentially growing function

$$\kappa_i = \left\lceil \exp\left(\frac{i \cdot log(n)}{\kappa_{\max} - 1}\right)\right\rceil, \tag{3.30}$$

with a maximum number of applied moves per shaking neighborhood of $\kappa_{\max} = 32$, to generate two sets of 10 insertion and exchange shaking neighborhood structures respectively. Starting with insertions, those sets are then interleaved and at positions four, ten and twenty extended by a subsequence inversion shaking neighborhood applying one, two and four inversions of five jobs respectively.

This configuration was mainly hand-crafted based on characteristics of the ruggedness of the respective neighborhood structures and the general structure of the search space. We used the autocorrelation function on random walks of length 10^6 to estimate the ruggedness of neighborhood structures and analyzed a large set of globally optimal solutions obtained from 3.75×10^6 runs on a diverse set of 300 instances with $n = 30$ jobs, to gain insight on the distribution of globally optimal solutions in the search space. We found that the studied instances contain a relatively high number of distinct globally or at least nearly optimal solutions, being widely distributed in the search space. A primary reason for this is likely the dependency structure inherent to the problem and the induced symmetries, caused by resource imbalance or utilization gaps on secondary resources, frequently allows exchanging of jobs on secondary resources without affecting the makespan. For more details, see [94].

3.10 Experimental Evaluation

The proposed approaches were implemented in C++ using G++ 5.4.1 for compilation. All tests were performed on a cluster of machines with Intel Xeon E5-2640 v4 processors with 2.40 GHz in single-threaded mode and 15GB RAM. The CP model from Section 3.8 was solved with ILOG CP Optimizer 12.7 whereas the MILP model from Section 3.7 was solved with CPLEX 12.7.

This section is organized as follows. First, the benchmark instances that are used are described and the tuning experiments are summarized. Then, Section 3.10.3 presents the impact of different algorithmic components used within our A* framework on the solution quality. The anytime behavior of our A* in conjunction with the different options for the lower bound calculation are analyzed in Section 3.10.5, the improvements of the strengthened lower bound $\mathrm{MS}^{\mathrm{LB2}}$ over $\mathrm{MS}^{\mathrm{LB0}}$ are studied in Section 3.10.6, further, the numbers of considered NDT records in our runs (i.e., essentially the sizes of our state graphs) are discussed in Section 3.10.7. Afterwards, Section 3.10.8 presents detailed main results for pure A*, LLBH, A*+BS+LS, the CP approaches with and without using LLBH as search guidance—henceforth denoted by CP+LLBH/ILOG and CP/ILOG, respectively—, the MILP approach, henceforth denoted by MIP/CPLEX, and the weighted A*-based anytime algorithm ARA*. Whereas, Section 3.10.8 compares mainly the ability to prove instances to optimality or terminate with small remaining optimality gaps, Section 3.10.9 compares the pure heuristic performance of the GVNS with A*+BS+LS and CP/ILOG.

3.10.1 Benchmark Instances

In our preliminary work [82] we created two non-trivial sets of random instances to test the presented algorithms. Basic characteristics of these instances are roughly inspired from the particle therapy patient scheduling scenario. Each of these sets consists of 50 instances for each combination of $n \in \{10, 20, 50, 100, 200, 500, 1000, 2000\}$ jobs and $m \in \{2, 3, 5\}$ secondary resources. As pointed out in Section 3.1 there are typically only two to four treatment rooms available in the particle therapy scenario, and we therefore do not consider here larger numbers of secondary resources than five. The difference between the two benchmark sets lies in the workload concerning the secondary resources.

The first set B is characterized by a *balanced* (B) workload over all resources from R. This was achieved by sampling the secondary resource q_j for each job $j \in J$ uniformly at random from the discrete distribution $\mathcal{U}\{1, m\}$. The second set S has a *skewed* (S) workload. For the S-instances this was achieved by assigning to resource m a probability of 0.5 for resource m and a probability of $1/(2m-2)$ for each of the remaining secondary resources 1 to $m-1$. In this way the expected workload on the common resource roughly corresponds to the expected workload of the dominant resource m. Note that in the preliminary work [82] we considered slightly different S-instances by assigning to resource m a probability twice as high as the probabilities for the remaining secondary resources. This scheme, however, has the disadvantage that for $m > 2$ the common resource tends to become the sole bottleneck, making the instances rather easy to solve. The preprocessing times p_j^{pre} and postprocessing times p_j^{post} were sampled from $\mathcal{U}\{0, 1000\}$ for both instance sets, while times p_j^0 were sampled from $\mathcal{U}\{1, 1000\}$ in case of set B and from $\mathcal{U}\{1, 2500\}$ in case of set S.

Finally, note that we made use of a third set P of instances derived from a real-world particle therapy patient scheduling scenario [120], which consists of 699 instances. However, this instance set consists to a large extent of instances with a similar structure as the balanced instances of type B and of a few instances with a similar structure as the skewed instances of type S. For the sake of completeness we nevertheless show the main results obtained for those set P instances in the main result table in Section 3.10.8. There, we partitioned the whole set of instances into groups with up to 10, 11 to 20, 21 to 50, and 51 to 100 jobs with 51, 39, 207 and 402 instances, respectively. All these instances use $m = 3$ secondary resources. All instance sets are available from `https://www.ac.tuwien.ac.at/research/problem-instances/`.

3.10.2 Parameter Tuning

The parameters of our hybrid A*+BS+LS algorithm were tuned for the goal to obtain as good solutions as possible within a CPU time limit of 900s per run using the automatic parameter configuration tool irace [109] (version 2.1). We distinguish between two different instance sizes: (1) instances with up to $n = 500$ jobs and (2) instances with more jobs. For these we created two independent tuning instance sets, respectively. Both sets contain only instances of type S, because it turned out that these instances are much more difficult to be solved than instances of type B. The first tuning set $T_{\leq 500}$ consists of 10 instances for each combination of $n \in \{10, 20, 50, 100, 200, 500\}$ jobs and $m \in \{2, 3, 5\}$ resources, whereas the second tuning set ($T_{>500}$) consists of 10 instances for each combination of $n \in \{1000, 2000\}$ jobs and $m \in \{2, 3, 5\}$ resources. We tuned four parameters with irace: (1) the lower bound to be used from $\{\text{MS}^{\text{LB0}}, \text{MS}^{\text{LB1}}, \text{MS}^{\text{LB2}}, (\text{MS}^{\text{LB0}}, \text{MS}^{\text{LB2}})\}$ where $(\text{MS}^{\text{LB0}}, \text{MS}^{\text{LB2}})$ denotes the interleaved combination of MS^{LB0} and MS^{LB2} as described in Section 3.5.2, (2) the usage of LS (on or off), (3) the iteration interval for applying BS $\delta \in \{1, 100, 200, 500, 1\text{k}, 2\text{k}, 5\text{k}, 10\text{k}, 20\text{k}, 50\text{k}, 100\text{k}, 200\text{k}\}$ and (4) the beam width $k_{\text{bw}} \in \{1, 2, 5, 10, 20, 50, 100, 200, 500\}$. For the tuning set $T_{>500}$ we use the different domain $\{1, 2, 3, 4, 5, 6, 7, 8\}$ for the beam width k_{bw} since larger values sometimes do not allow a single beam search to be completed within our time limit. Also remember that the case $k_{\text{bw}} = 1$ corresponds to simple diving. The irace tool was applied with a budget of 7000 runs to tuning set $T_{\leq 500}$ and with a budget of 4000 runs to tuning set $T_{>500}$. In this way, the following configurations were obtained by irace: $\mathcal{C}_{\leq 500} = \{(\text{MS}^{\text{LB0}}, \text{MS}^{\text{LB2}}), \text{LS turned on}, \delta = 1000, k_{\text{bw}} = 200\}$ and $\mathcal{C}_{>500} = \{(\text{MS}^{\text{LB0}}, \text{MS}^{\text{LB2}}), \text{LS turned on}, \delta = 100, k_{\text{bw}} = 8\}$, respectively. These results already indicate that the improved lower bound calculations, our advanced diving by BS, as well as the local search procedure are in practice indeed advantageous.

3.10.3 Analysis of Algorithmic Components

In this section, the impact of different algorithmic components of the A* framework—in particular BS and LS—is analyzed. For this purpose we compare first LLBH and a standalone variant of BS, both with LS and without. The variants with LS are denoted in the following by LLBH+LS and BS+LS, respectively. Figure 3.6 shows results obtained for middle-sized instances of types B and S with $n \in \{100, 200, 500\}$ jobs and $m = 3$

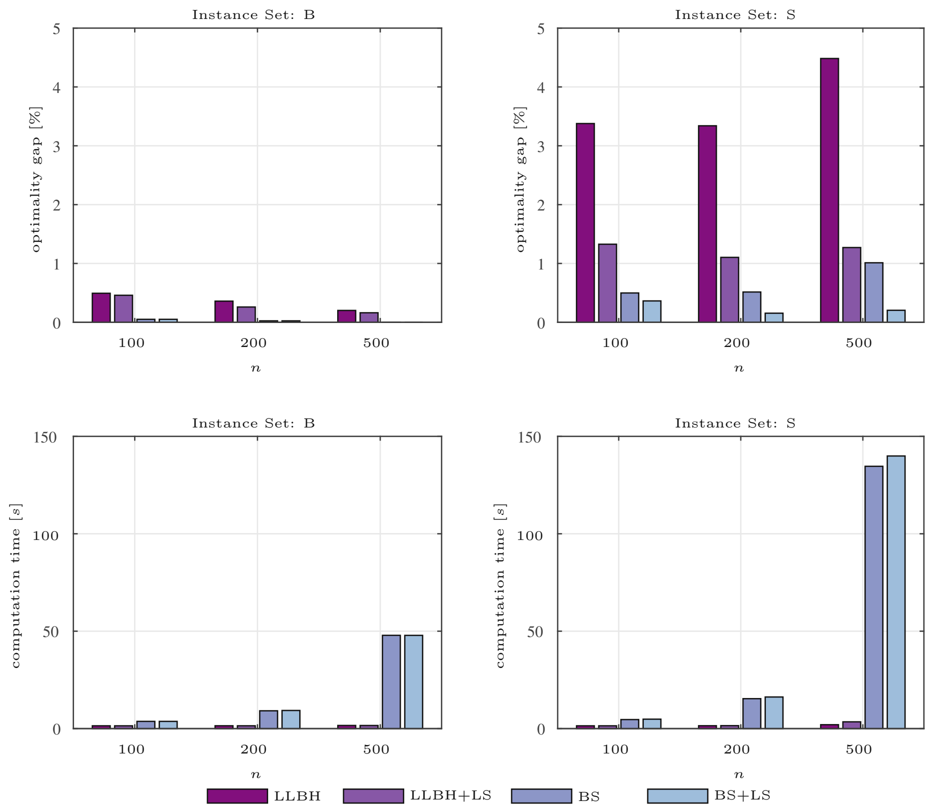


Figure 3.6: Comparison of standalone LLBH and BS with and without LS applied afterwards. Results are visualized as barplots which are grouped according to the instance type and number of considered jobs.

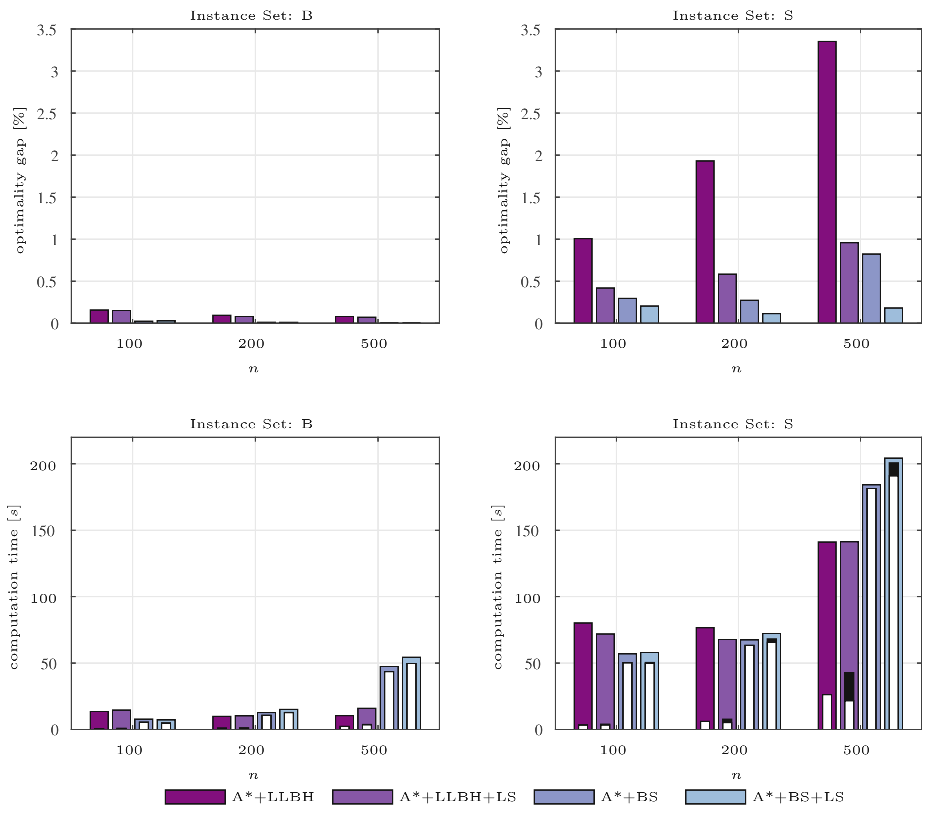


Figure 3.7: Comparison of A*+LLBH, A*+LLBH+LS, A*+BS and A*+BS+LS. Results are visualized as barplots which are grouped according to the instance type and number of considered jobs.

secondary resources. The diagrams on the top present average optimality gaps of obtained solutions π, which are calculated as $100\% \cdot (\mathrm{MS}(\pi) - \mathrm{LB})/\mathrm{LB}$, where LB corresponds to the strengthened lower bound $\mathrm{MS}^{\mathrm{LB2}}$ according to Eq. (3.8). The diagrams on the bottom show corresponding average computation times. Note that LLBH described in Section 3.5 actually is just the special case of BS with beam width $k_{\mathrm{bw}} = 1$. For the standalone BS a beam width of $k_{\mathrm{bw}} = 200$ was used. As expected, LLBH is fastest, however it also yields the highest average optimality gaps in all considered cases compared to the other considered approaches. Conversely, the BS+LS approach leads to the smallest obtained average optimality gaps with the largest obtained average computation times. Remarkable is that BS with or without LS applied always leads to significantly smaller average optimality gaps than LLBH or LLBH+LS. However, these BS variants also require more computation time. Concerning LS, in particular for larger skewed instances of type S, its application has a substantial impact on the obtained average optimality gaps.

Next, we repeat the comparison of the four different heuristics LLBH, LLBH+LS, BS, and BS+LS, however this time embedded in our A* algorithm. Corresponding results are shown in Figure 3.7. In the calculation of the optimality gaps, the lower bounds obtained from the corresponding A* runs are now used as LB. In case of A*+LLBH we set the beam width to $k_{\text{bw}} = 1$ otherwise to $k_{\text{bw}} = 200$. In all cases a BS iteration is initiated after every $\delta = 1000$ classical A* iterations. Concerning the computation times, we split each into three parts: the first white block on the bottom of the bars corresponds to the average accumulated time needed by BS. If LS is applied then the black block in the middle corresponds to the average accumulated time used by LS. Finally, the top-most remainder of the bar represents the average time consumed by classical A* iterations. Consequently, the whole bar corresponds to the average total running time of the whole algorithm. At the first glance the results show similar relationships as the results in Figure 3.6. Again, in all considered cases the best optimality gaps are obtained by A*+BS+LS whereas the worst ones are obtained by A*+LLBH. However, this time A*+LLBH is not always fastest. In particular for skewed instances with $n \in \{100, 200\}$ jobs A*+LLBH requires more time on average compared to all other considered approaches. This has the following reason: Typically the lower bound obtained from the root node of A* is already rather tight, therefore in order to further reduce the optimality gap it is more important to find good feasible solutions instead of further improving the current best known lower bound by performing classical A* iterations. As we have observed in Figure 3.6, the solution quality achieved by LLBH is on average worse than the solution quality achieved by the other considered approaches. Therefore A*+LLBH has to perform more LLBH iterations until a comparably small optimality gap is achieved. Putting more effort in finding the optimal solution by applying LS afterwards or using BS leads to faster convergence since A* needs less LLBH+LS or BS(+LS) iterations even if a single of such iteration takes more time. For larger instances with $n = 500$ jobs this effect does not pay off anymore. Here, A*+BS and A*+BS+LS are able to return smaller average optimality gaps than A*+LLBH or A*+LLBH+LS, however they also need more time.

3.10.4 Comparison to Anytime Pack Search

As described in Section 3.2, anytime pack search (APS) from Vadlamudi et al. [155] is an approach similar to our BS-enhanced A* algorithm and based on consecutive BS iterations. In APS terminology the beam width k_{bw} is called *pack size*. There are two major differences: (1) APS does not perform classical A* iterations between consecutive BS iterations—which corresponds in our case to a setting of the diving/BS interval to $\delta = 1$—and (2) if APS starts a new BS iteration then APS selects the best k_{bw} nodes—called *seeds*—from the open list Q instead of just selecting the best node as our A*+BS+LS algorithm does. For comparison purposes we implemented APS based on our combined lower bound. A comparison of A*+BS+LS and APS is shown in Figure 3.8. APS was applied to all instances of type B and S, with a pack size corresponding to the beam width k_{bw} of the corresponding A*+BS+LS runs. The results are grouped according to the instance type and number of secondary resources and visualized as boxplots.

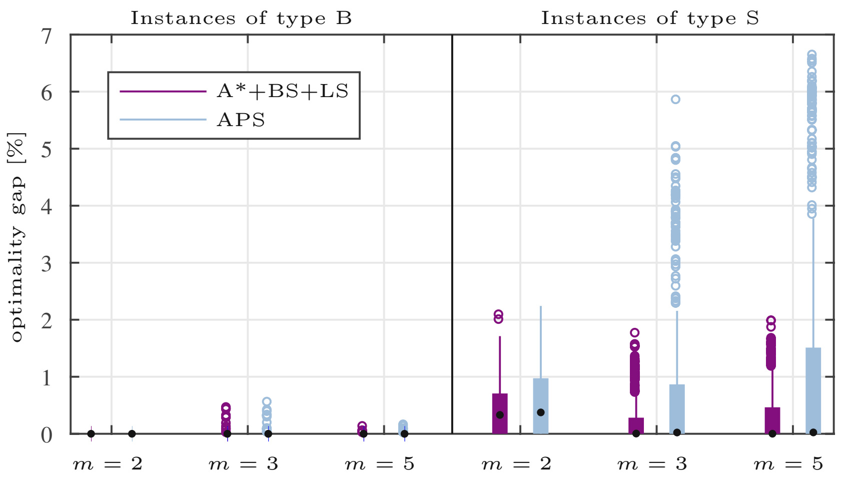


Figure 3.8: Comparison of A*+BS+LS and APS. Results are visualized as boxplots which are grouped according to the instance type and number of secondary resources.

These boxplots indicate the following: The obtained results for balanced instances of type B are comparable. The median optimality gaps concerning the subsets of balanced instances of type B are always zero for both algorithms. In particular, for instances with $m = 2$ secondary resources both algorithms could solve all instances to optimality, and for the instances with $m \in \{3, 5\}$ both algorithms could solve almost all instances to optimality, except a few outliers with optimality gaps $< 0.6\%$. In contrast to the results for the instances of type B, the results for the instances of type S show clear differences between A*+BS+LS and APS. Concerning A*+BS+LS, the medians of the obtained optimality gaps for the subsets of instances are 0.330%, 0.004% and 0.001% for 2, 3 and 5 secondary resources, respectively. In contrast, the corresponding medians obtained from APS are 0.375%, 0.022% and 0.024% for 2, 3 and 5 secondary resources, respectively. The largest optimality gaps obtained from A*+BS+LS are $< 2.1\%$ for all skewed instances of type S, whereas APS yielded a substantial number of times larger gaps of up to 6.7%, in particular for $m = 5$ secondary resources.

3.10.5 Anytime Behavior

In order to study the anytime-behavior of the A* variants, Figure 3.9 and 3.10 show solution quality over time (SQT) plots for the middle-sized instances of type S (skewed) with $n \in \{100, 200, 500\}$ jobs and $m = 3$ secondary resources. For this purpose, the current optimality gap of A*+BS+LS was taken every 0.01 seconds during each run. At each of the resulting $90,000$ time points (remember our time limit of 900s), the average optimality gap for the 50 corresponding problem instances is shown. In addition, in order to visualize the variance, boxplots are shown for time points 5, 50 and 500 seconds. We skip here results showing the anytime behavior of balanced instances if type B, since in most cases the first solution obtained by BS+LS is due to the excellent search guidance of our lower bounds already an optimal solution so that there is no further improvement.

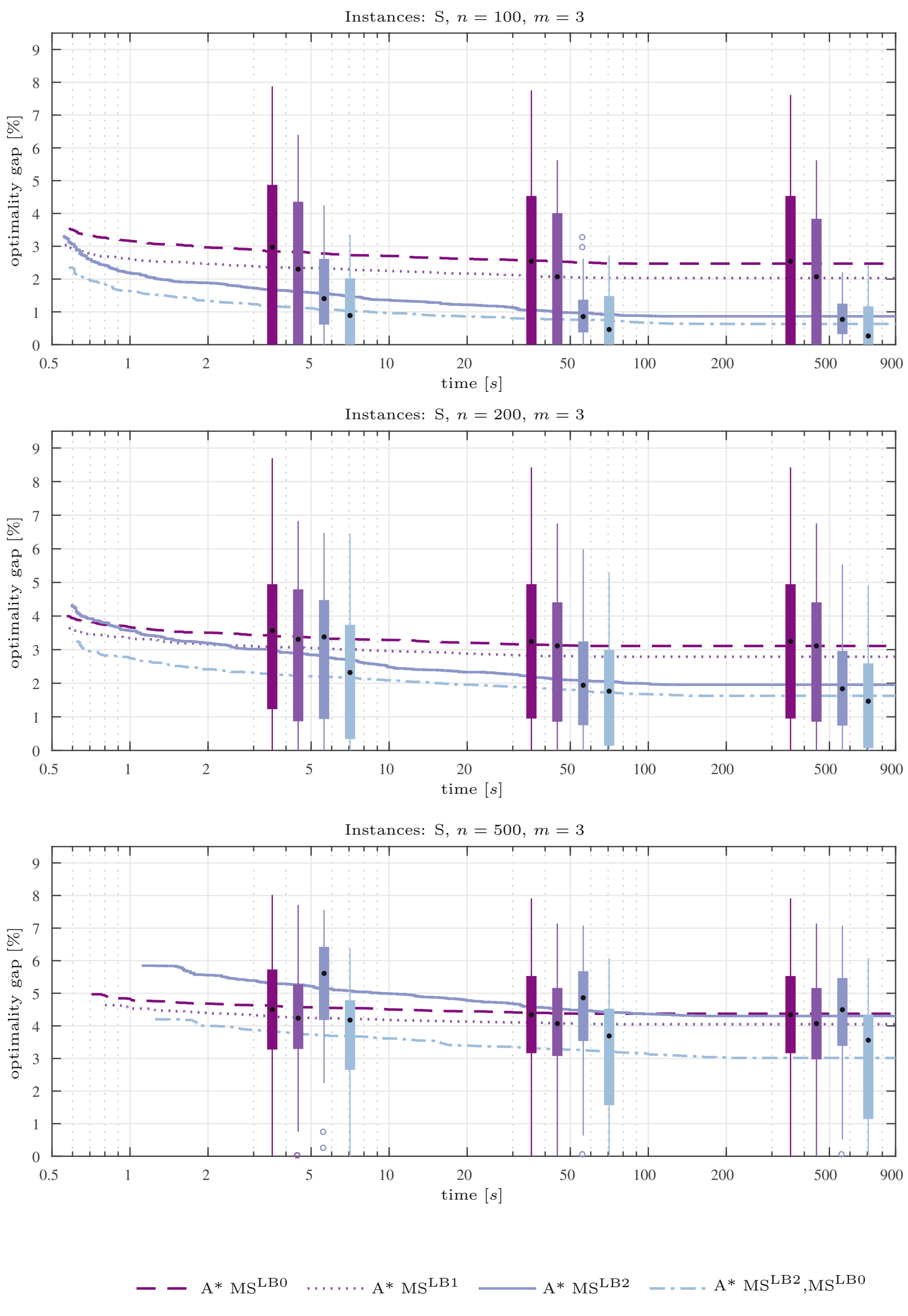


Figure 3.9: SQT plots obtained from A* with simple diving ($k_{\mathrm{bw}} = 1$), $\delta = 1000$ and no LS for different choices of the used lower bounds.

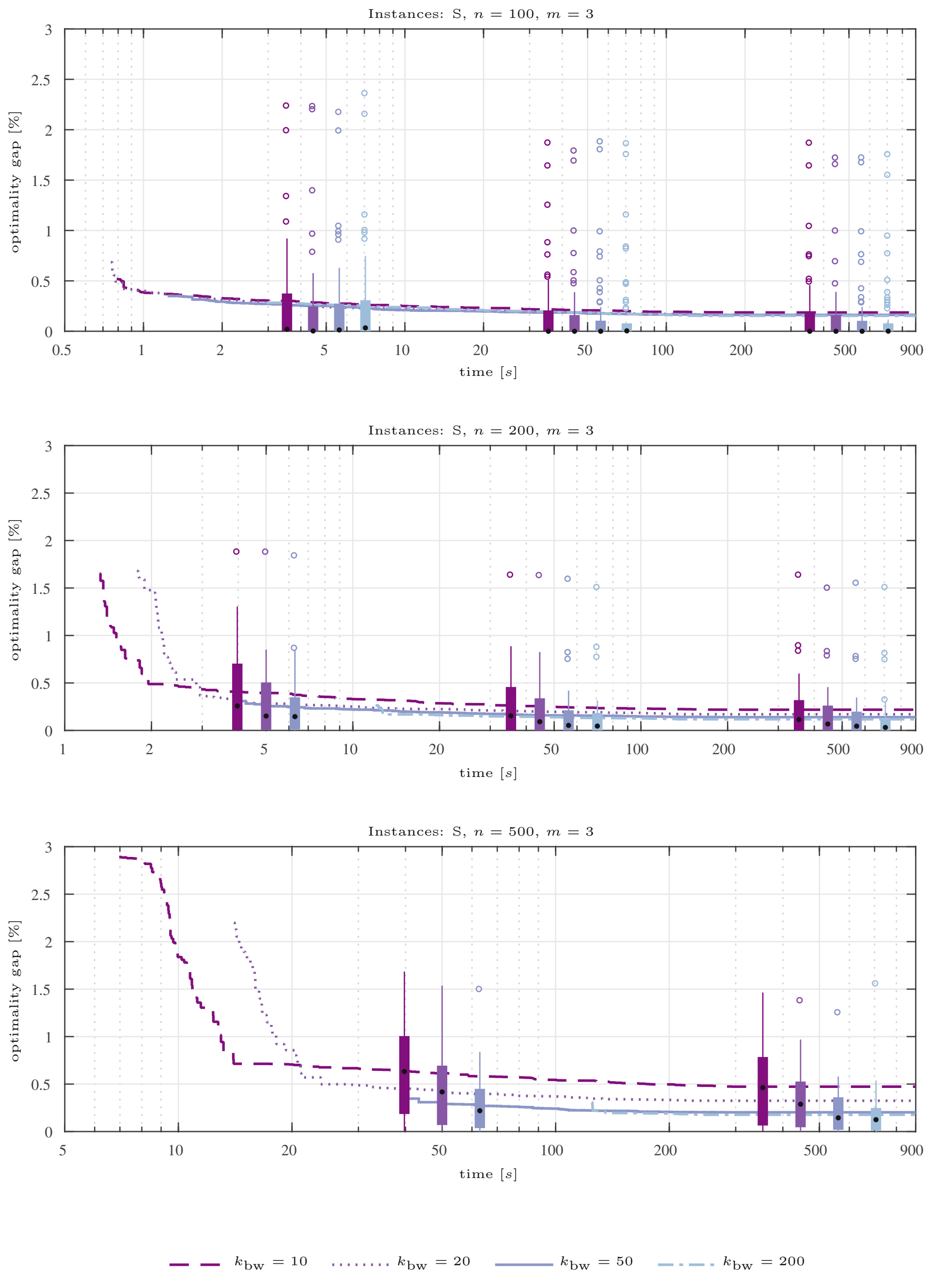


Figure 3.10: SQT plots obtained from A*+BS+LS for different choices of the used beam widths. The algorithm was run by using the combined lower bound of $\mathrm{MS}^{\mathrm{LB0}}$ and $\mathrm{MS}^{\mathrm{LB2}}$ and with $\delta = 1000$ and LS turned on.

In Figure 3.9, the results are shown for four A* variants that differ in the used lower bound: $\mathrm{MS}^{\mathrm{LB0}}$, $\mathrm{MS}^{\mathrm{LB1}}$, $\mathrm{MS}^{\mathrm{LB2}}$ and the interleaved combination of $\mathrm{MS}^{\mathrm{LB0}}$ and $\mathrm{MS}^{\mathrm{LB2}}$. The A* algorithm uses just simple diving ($k_{\mathrm{bw}} = 1$) every $\delta = 1000$ iterations, and heuristic solutions are here *not* further improved by LS. It can be observed that, for all three values of n, the choice of the combined lower bounds $\mathrm{MS}^{\mathrm{LB0}}$ and $\mathrm{MS}^{\mathrm{LB2}}$ almost always provides the smallest average optimality gap at the end of a run.

In Figure 3.10 we show SQT plots of A*+BS+LS runs where the combined lower bounds $\mathrm{MS}^{\mathrm{LB0}}$ and $\mathrm{MS}^{\mathrm{LB2}}$ are used and BS is started every $\delta = 1000$ iterations with the four different beam widths $k_{\mathrm{bw}} \in \{10, 20, 50, 200\}$. Each obtained solution from BS is further improved by LS. The smallest average optimality gaps are obtained from A*+BS+LS when the largest beam width with $k_{\mathrm{bw}} = 200$ is used for all three values of n. Furthermore, it can be observed that, with increasing beam width, we obtain smaller and smaller average optimality gaps at the end of a run. Overall the SQT plots show a continuous decrease of the optimality gaps over time. Finally, note that in case of the high beam width $k_{\mathrm{bw}} = 200$—for instances with $n = 500$ jobs—A*+BS+LS took about 125 seconds until the first solution is obtained. To counteract this initial long waiting time for a first feasible solution, using a smaller beam width for the very first BS application would be an obvious possibility.

3.10.6 Comparison of Lower Bounds

We finally aim to quantitatively evaluate the improvement obtained by lower bound $\mathrm{MS}^{\mathrm{LB2}}$ from Equation (3.8) in comparison to the basic lower bound $\mathrm{MS}^{\mathrm{LB0}}$ from Equation (3.4). To this end, we computed both lower bounds for all instances of our benchmark sets and consider the relative improvement $\Delta = (\mathrm{MS}^{\mathrm{LB2}} - \mathrm{MS}^{\mathrm{LB0}})/\mathrm{MS}^{\mathrm{LB0}}$.

In case of the balanced instances of type B, these differences are always zero, thus, no benefits could be observed. The reason for this is that instances of type B are created

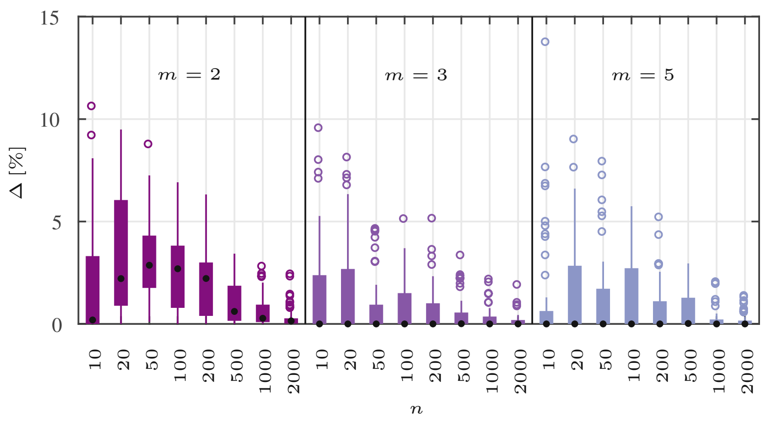


Figure 3.11: Relative improvement of bound $\mathrm{MS}^{\mathrm{LB2}}$ over bound $\mathrm{MS}^{\mathrm{LB0}}$ in the context of the instances of type S (skewed).

in such a way that it is likely that the assumption that all jobs J_r which require a secondary resource $r \in R$ can be scheduled consecutively one after the other without any gaps holds. Hence, in such cases the basic lower bound $\mathrm{MS}^{\mathrm{LB0}}$ will be the same as $\mathrm{MS}^{\mathrm{LB2}}$, see Section 3.4.1 for further details. For the more difficult skewed instances (type S), however, the improvements are sometimes substantial. Figure 3.11 shows them as boxplots grouped according to the number of secondary resources m and the number of jobs n. It can be observed that these relative improvements are strongest for the skewed instances with only two secondary resources and in particular just a moderate number of jobs. There they are, however, also most needed: These are generally the instances that are most difficult to solve for our A* variants in terms of the remaining optimality gap, and we have already seen in the SQT plots of Section 3.10.5 that the usage of the strengthened lower bound boosts the performance significantly.

3.10.7 Number of Considered NDT Records

Each bar in Figure 3.12 shows the average number of NDT records considered by A*+BS+LS, over all 50 instances (per instance type) and using the parameter setting as determined by irace. In essence, these bars show the size of the considered part of

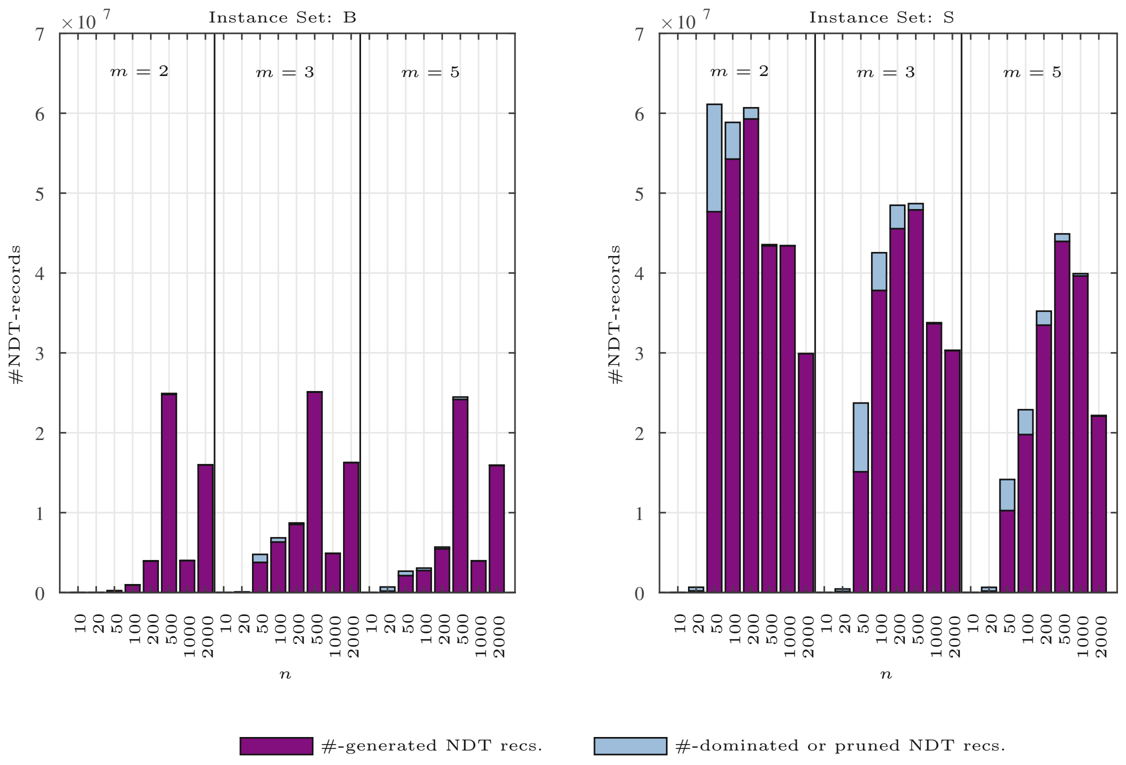


Figure 3.12: Number of considered NDT records during the runs of A*+BS+LS, over all instances of type B and S with the parameter setting as determined by irace.

the state graph. Each bar is split into a lower part and an upper part. The lower part presents the average number of created NDT records that were never dominated by another NDT record. Thus, this average number of created NDT records essentially is also proportional to the memory usage of A*. The upper part shows the average number of NDT records that were either dominated by another NDT record, or pruned since the lower bound was greater than (or equal to) the makespan of the current incumbent solution. Note that these records are either immediately discarded at the time of their creation or removed during the A*+BS+LS run since they got dominated by another NDT record.

First of all, these plots show that the skewed instances of set S are much more demanding concerning the total number of created NDT records, especially those instances with only two external resources. Clearly, this can be explained with the lower bound calculation being not that tight for these instances as for the balanced set. The number of NDT records for the skewed instances has its peak for $n = 50$ jobs and $m = 2$ secondary resources and generally decreases again for instances with more jobs, since fewer node expansions are possible within our CPU time limit of 900s. Overall, there are instances for which A*+BS+LS explores more than $6 \cdot 10^7$ NDT records.

3.10.8 Comparison of LLBH, A*+BS+LS, CP, MIP, and ARA*

Table 3.1 presents the aggregated results for each combination of instance type and the different numbers of jobs and secondary resources for LLBH, A*+BS+LS, the CP approach, the MILP approach, pure A* and ARA*. Concerning CP we tested: (1) the CP ILOG solver with default search heuristic—denoted as CP/ILOG—and (2) CP ILOG solver with a custom search heuristic denoted as CP+LLBH/ILOG, which, similar to LLBH, uses the combined lower bound $\mathrm{MS}^{\mathrm{LB2}}$ and $\mathrm{MS}^{\mathrm{LB0}}$ to decide which job should be selected and appended to a current fixed partial schedule first.

In order to compare our A*+BS+LS algorithm also to another state-of-the-art A*-based anytime algorithm we implemented ARA* [108] (see also Section 3.2, where ARA* was already discussed). ARA* makes use of an inflated evaluation function $f_*(x) = Z^{\mathrm{sp}}(x) + \varepsilon \cdot Z^{\mathrm{h}}(x)$, where ε is the approximation ratio. The algorithm has two parameters: $\varepsilon_{\mathrm{init}} > 1$ and $\varepsilon_{\mathrm{step}} > 0$. Initially, ε is set to $\varepsilon_{\mathrm{init}}$ such that the first heuristic solution that is found is sub-optimal by a factor of at most $\varepsilon_{\mathrm{init}}$. The quality of subsequently encountered heuristic solutions is forced to increase by continuously decreasing ε by step size $\varepsilon_{\mathrm{step}}$. If enough time is given for ε to finally reach value one, ARA* has reached a proven optimal solution. To achieve a reasonable anytime behavior on our benchmark instances we tested different values for $\varepsilon_{\mathrm{init}}$ and $\varepsilon_{\mathrm{step}}$. According to these preliminary experiments we set $\varepsilon_{\mathrm{init}} = 1.5$ and the step size $\varepsilon_{\mathrm{step}} = 0.01$. Moreover, our implementation makes use of the inflated version of evaluation vector $\mathbf{f}$ with the combined lower bound $\mathrm{MS}^{\mathrm{LB2}}$ and $\mathrm{MS}^{\mathrm{LB0}}$.

Columns %-opt of Table 3.1 state the percentage of instances that were solved to proven optimality. Columns %-gap list average optimality gaps of final solutions π, which are

Table 3.1: Average results of LLBH, A*+BS+LS, CP/ILOG, CP+LLBH/ILOG, MIP/CPLEX, pure A*, and ARA* for the instance set B, S, and P.

			LLBH				A*+BS+LS				CP/ILOG				CP+LLBH/ILOG				MIP/CPLEX				A*		ARA*			
type	n	m	%-opt	%-gap	$\sigma_{\%\text{-gap}}$	t[s]	%-opt	%-gap	$\sigma_{\%\text{-gap}}$	t[s]	%-opt	%-gap	$\sigma_{\%\text{-gap}}$	t[s]	%-opt	%-gap	$\sigma_{\%\text{-gap}}$	t[s]	%-opt	%-gap	$\sigma_{\%\text{-gap}}$	t[s]	%-opt	t[s]	%-opt	%-gap	$\sigma_{\%\text{-gap}}$	t[s]
B	10	2	90	0.197	0.87	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	40	0.007	0.01	22.6	**100**	0.8	**100**	**0.000**	0.00	<0.1
B	20	2	96	0.074	0.37	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	-	-	-	900.0	**100**	15.8	**100**	**0.000**	0.00	<0.1
B	50	2	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	1.1	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	-	-	-	900.0	0	106.7	**100**	**0.000**	0.00	<0.1
B	100	2	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	2.0	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	-	-	-	900.0	0	90.1	78	0.163	0.40	1.3
B	200	2	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	5.4	**100**	**0.000**	0.00	0.1	**100**	**0.000**	0.00	0.1	-	-	-	900.0	0	74.2	30	0.639	0.56	255.0
B	500	2	**100**	**0.000**	0.00	0.3	**100**	**0.000**	0.00	35.3	**100**	**0.000**	0.00	1.3	**100**	**0.000**	0.00	0.3	-	-	-	900.0	0	73.3	2	1.381	0.55	323.3
B	1000	2	**100**	**0.000**	0.00	1.2	**100**	**0.000**	0.00	8.9	**100**	**0.000**	0.00	9.2	**100**	**0.000**	0.00	1.3	-	-	-	900.0	0	66.7	2	2.191	0.68	434.0
B	2000	2	**100**	**0.000**	0.00	6.1	**100**	**0.000**	0.00	46.3	98	<0.001	<0.01	63.5	**100**	**0.000**	0.00	5.1	-	-	-	900.0	0	48.6	0	3.117	0.66	900.0
B	10	3	76	0.819	1.92	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	48	0.007	0.01	19.2	**100**	0.8	**100**	**0.000**	0.00	<0.1
B	20	3	76	0.865	1.87	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	2	-	-	900.1	**100**	14.2	**100**	**0.000**	0.00	0.1
B	50	3	74	0.702	1.32	<0.1	**96**	**0.017**	0.08	1.1	92	0.068	0.30	<0.1	90	0.091	0.32	<0.1	-	-	-	900.0	0	119.8	2	1.028	0.37	284.6
B	100	3	68	0.625	1.13	<0.1	**92**	**0.021**	0.09	2.0	78	0.226	0.55	4.2	78	0.204	0.47	<0.1	-	-	-	900.0	0	101.4	0	1.436	0.34	179.0
B	200	3	68	0.439	0.83	<0.1	**92**	**0.016**	0.06	5.9	56	0.556	1.12	319.4	80	0.198	0.51	0.1	-	-	-	900.0	0	82.5	0	1.595	0.41	153.5
B	500	3	56	0.265	0.42	0.3	**98**	**<0.001**	<0.01	35.9	20	2.212	1.83	900.0	66	0.162	0.29	0.3	-	-	-	900.0	0	75.2	0	2.109	0.80	205.3
B	1000	3	68	0.154	0.32	1.3	**98**	**0.001**	<0.01	6.1	2	3.094	1.46	899.9	74	0.083	0.20	1.3	-	-	-	900.0	0	62.8	0	2.737	0.78	358.9
B	2000	3	66	0.087	0.17	6.3	**98**	**0.005**	0.04	23.8	0	4.220	1.20	900.0	70	0.075	0.16	5.1	-	-	-	900.0	0	50.4	0	3.229	0.29	767.1
B	10	5	50	2.062	3.14	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	74	0.004	0.01	2.2	**100**	0.8	**100**	**0.000**	0.00	<0.1
B	20	5	32	1.971	2.15	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	44	-	-	900.1	**100**	18.6	**100**	**0.000**	0.00	0.1
B	50	5	40	0.623	0.83	<0.1	96	**0.000**	0.00	1.2	**100**	**0.000**	0.00	0.7	94	0.002	0.02	<0.1	34	-	-	900.1	0	142.0	68	0.154	0.30	15.6
B	100	5	28	0.305	0.36	<0.1	96	**0.000**	0.00	2.2	**100**	**0.000**	0.00	9.5	94	0.006	0.04	<0.1	-	-	-	900.0	0	105.7	24	0.453	0.34	290.5
B	200	5	38	0.149	0.20	<0.1	96	<0.001	<0.01	6.5	**100**	**0.000**	0.00	91.3	98	0.002	0.02	0.1	-	-	-	900.0	0	95.8	0	0.729	0.26	256.5
B	500	5	38	0.053	0.06	0.3	**100**	**0.000**	0.00	42.3	86	<0.001	<0.01	499.7	90	0.007	0.02	0.3	-	-	-	900.0	0	87.0	0	0.738	0.25	221.8
B	1000	5	38	0.020	0.03	1.5	**100**	**0.000**	0.00	7.9	0	0.359	0.12	900.0	98	0.002	0.01	1.3	-	-	-	900.0	0	64.3	0	0.670	0.25	329.7
B	2000	5	48	0.011	0.02	7.4	**100**	**0.000**	0.00	30.4	0	0.478	0.14	900.0	94	<0.001	<0.01	5.4	-	-	-	900.0	0	52.1	0	0.767	0.42	715.1
B			68	0.436	0.73		**98**	**0.003**	0.01		76	0.467	0.28		93	0.035	0.09		-	-	-		25		38	0.964	0.32	
S	10	2	56	0.837	1.34	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	0.2	**100**	**0.000**	0.00	0.1	26	0.008	0.01	9.0	**100**	0.8	**100**	**0.000**	0.00	<0.1
S	20	2	28	1.410	1.60	<0.1	**100**	**0.000**	0.00	0.9	24	0.005	0.04	899.8	14	0.486	0.79	899.8	-	-	-	900.0	**100**	4.3	**100**	**0.000**	0.00	0.3
S	50	2	0	2.595	1.89	<0.1	**40**	0.268	0.38	11.4	0	**0.210**	0.28	899.9	0	2.147	1.59	899.8	-	-	-	900.0	0	152.3	0	2.532	1.16	247.2
S	100	2	0	2.589	1.50	<0.1	**26**	0.367	0.49	44.8	0	**0.323**	0.47	900.0	0	2.350	1.33	899.8	-	-	-	900.0	0	135.5	0	3.849	1.17	234.2
S	200	2	0	2.912	1.30	<0.1	**2**	**0.440**	0.33	65.2	0	0.642	0.51	900.0	0	2.865	1.31	899.9	-	-	-	900.0	0	148.0	0	4.606	1.04	244.5
S	500	2	**0**	3.691	1.05	0.5	**0**	**0.532**	0.18	88.7	**0**	2.736	0.51	900.0	**0**	3.679	1.02	899.9	-	-	-	900.0	**0**	222.5	**0**	4.940	0.85	222.7
S	1000	2	**0**	4.381	0.87	3.2	**0**	**0.725**	0.20	176.8	**0**	4.636	0.43	900.0	**0**	4.365	0.87	900.9	-	-	-	900.0	**0**	306.6	**0**	5.026	0.48	276.8
S	2000	2	**0**	4.976	0.86	22.7	**0**	**0.786**	0.18	252.7	**0**	4.784	0.39	900.0	**0**	4.991	0.84	900.0	-	-	-	900.0	**0**	353.6	**0**	5.294	0.40	574.7
S	10	3	30	1.675	1.91	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	0.1	**100**	**0.000**	0.00	<0.1	56	0.005	0.01	2.2	**100**	0.8	**100**	**0.000**	0.00	<0.1
S	20	3	14	1.790	1.42	<0.1	**100**	**0.000**	0.00	0.8	54	**0.000**	0.00	255.0	42	0.446	0.79	899.3	40	-	-	899.3	**100**	6.0	**100**	**0.000**	0.00	<0.1
S	50	3	8	1.937	1.63	<0.1	**80**	0.053	0.21	1.3	54	**0.035**	0.15	27.7	30	1.580	1.63	899.7	10	-	-	900.0	0	194.3	26	0.936	0.97	339.0
S	100	3	12	2.785	1.83	<0.1	**50**	0.153	0.37	16.5	44	**0.060**	0.15	899.7	20	2.720	1.85	899.9	-	-	-	900.0	0	190.0	8	2.483	1.79	268.8
S	200	3	4	3.221	1.89	0.1	34	**0.117**	0.26	26.4	**36**	0.135	0.21	899.8	10	3.202	1.91	899.8	-	-	-	900.0	0	194.5	4	2.939	1.57	212.1
S	500	3	0	4.127	1.70	0.7	**14**	**0.177**	0.24	121.6	4	1.360	0.76	900.0	4	4.124	1.70	899.9	-	-	-	900.0	0	265.9	0	3.781	0.93	158.9
S	1000	3	0	4.123	1.81	4.2	**2**	**0.621**	0.47	248.0	0	2.872	0.93	900.0	0	4.122	1.81	899.9	-	-	-	900.0	0	337.9	0	3.747	0.47	211.9
S	2000	3	**0**	4.142	1.97	31.3	**0**	**0.701**	0.41	480.2	**0**	4.296	0.97	900.0	**0**	4.141	1.97	900.0	-	-	-	900.0	**0**	435.7	**0**	3.794	0.37	735.9
S	10	5	32	1.695	2.20	<0.1	**100**	**0.000**	0.00	0.8	**100**	**0.000**	0.00	0.0	**100**	**0.000**	0.00	<0.1	68	0.004	0.01	0.9	**100**	0.8	**100**	**0.000**	0.00	<0.1
S	20	5	16	1.832	1.97	<0.1	**100**	**0.000**	0.00	0.8	60	**0.000**	0.00	0.6	46	0.708	1.43	899.2	50	7.886	12.25	581.1	**100**	11.0	**100**	**0.000**	0.00	<0.1
S	50	5	16	2.258	1.94	<0.1	**80**	0.077	0.19	1.4	54	**0.045**	0.14	15.0	24	2.023	1.93	899.8	22	-	-	899.7	0	196.9	38	1.096	1.27	362.3
S	100	5	10	2.578	2.06	<0.1	**64**	0.064	0.18	6.1	48	**0.019**	0.04	899.6	18	2.534	2.06	899.8	-	-	-	900.0	0	219.4	14	2.291	1.78	454.2
S	200	5	2	3.472	1.74	0.1	**34**	0.281	0.49	38.8	28	**0.161**	0.25	900.0	6	3.449	1.76	899.8	-	-	-	900.0	0	243.4	8	3.247	1.57	438.5
S	500	5	2	3.653	1.85	0.9	**16**	**0.347**	0.34	188.6	8	1.229	0.95	899.9	4	3.645	1.86	899.9	-	-	-	900.0	0	384.0	4	3.346	1.24	319.7
S	1000	5	**0**	4.371	1.95	5.9	**0**	**0.702**	0.50	387.3	**0**	2.478	1.11	900.0	**0**	4.371	1.95	900.0	-	-	-	900.0	**0**	423.0	**0**	3.613	0.73	396.4
S	2000	5	**0**	4.460	2.07	52.0	**0**	**0.915**	0.54	789.3	**0**	4.229	1.22	900.0	**0**	4.461	2.07	900.0	-	-	-	900.0	**0**	558.4	**0**	3.631	0.50	900.0
S			9	3.040	1.76		**43**	**0.305**	0.25		40	1.272	0.43		22	2.600	1.35		-	-	-		25		29	2.548	0.76	
P	≤10	3	71	0.464	0.93	<0.1	**100**	**0.000**	0.00	0.9	**100**	**0.000**	0.00	<0.1	**100**	**0.000**	0.00	<0.1	63	**0.000**	0.00	<0.1	**100**	1.0	**100**	**0.000**	0.00	<0.1
P	≤20	3	54	0.567	0.86	<0.1	**100**	**0.000**	0.00	0.9	**100**	**0.000**	0.00	<0.1	97	0.011	0.07	<0.1	59	**0.000**	0.00	63.5	**100**	1.7	**100**	**0.000**	0.00	<0.1
P	≤50	3	42	0.634	0.96	<0.1	**87**	**0.110**	0.35	1.0	86	0.120	0.36	0.3	70	0.374	0.78	<0.1	27	-	-	900.0	10	179.3	52	0.580	0.89	257.8
P	≤100	3	48	0.494	0.85	<0.1	83	0.086	0.27	1.2	**85**	**0.070**	0.23	1.6	67	0.396	0.81	<0.1	4	-	-	900.0	0	153.5	21	1.049	1.06	311.1
P			54	0.540	0.90		**93**	0.049	0.16		**93**	**0.048**	0.15		84	0.195	0.42		28	-	-		53		68	0.407	0.49	

calculated by $100 \cdot (\mathrm{MS}(\pi) - \mathrm{LB})/\mathrm{LB}$, where LB is the largest lower bound returned from A*+BS+LS, pure A*, CPLEX or from ILOG CP Optimizer. Columns $\sigma_{\%\text{-gap}}$ provide the corresponding standard deviations. Columns t[s] show the median computation times in seconds. The last row of each instance type shows the average percentage of instances that were solved to proven optimality, the average optimality gap as well as the corresponding average standard deviations over all instances. Remember that for instances with $n \leq 500$ jobs, A*+BS+LS was run with configuration $\mathcal{C}_{\leq 500}$, and otherwise with configuration $\mathcal{C}_{>500}$. Moreover, LLBH and CP+LLBH/ILOG was applied using the combined lower bounds of $\mathrm{MS}^{\mathrm{LB2}}$ and $\mathrm{MS}^{\mathrm{LB0}}$. Again, the CPU time limit for each run was 900s. Since the pure A* algorithm does not provide any solution in case of early termination due to the time limit or memory limits, Table 3.1 just states the percentages of instances that were solved to optimality and the total computation times.

The following observations can be made with respect to the balanced instances of set B. First, A*+BS+LS was able to solve all instances with $m = 2$ secondary resources and more than 96% of the instances with $m > 2$ to proven optimality. In contrast, while CP/ILOG also solves nearly all instances with $m = 2$ secondary resources to optimality, its performance degrades significantly with growing n and m. In particular, instances with $m = 3$ secondary resources are hard to solve for CP/ILOG, since those instances are created in a way such that the expected workload of the common resource as well as the expected workloads of the secondary resource are equal, which makes instances hard to solve, whereas for instances with $m \in \{2, 5\}$ secondary resources the common resource becomes the sole bottleneck. Moreover, the average optimality gaps obtained by A*+BS+LS are always smaller than (or equal to) the ones of CP/ILOG. The largest average optimality gap of A*+BS+LS is only 0.021% while that of CP/ILOG is 4.220%. Clearly, the most important ingredient for this success of A*+BS+LS is the excellent guidance of the search by our lower bound calculation so that even for instances with $m = 3$ secondary resources excellent results are obtained. This is documented by the fact that even LLBH yields comparably good results with average optimality gaps of no more than 2.062%. Due to this excellent search guidance it can further be observed that CP+LLBH/ILOG can solve significantly more instances to optimality as CP/ILOG. Over all balanced instances of set B, CP+LLBH/ILOG is able to solve on average 93% of the instances to proven optimality whereas CP/ILOG can only solve on average 76% to proven optimality. However, A*+BS+LS can solve on average 98% of the instances to proven optimality. Moreover, CP+LLBH/ILOG is able to provide better average optimality gaps as CP/ILOG for larger instances. Nevertheless, CP+LLBH/ILOG cannot provide better average optimality gaps than A*+BS+LS, since the combination of BS and LS embedded in A* seems superior to the CP approach in combination with LLBH as search heuristic.

The observations concerning the benchmark set S are as follows. First of all, these instances are clearly more difficult to be solved than the instances from set B. In particular, A*+BS+LS was only able to solve instances up to size $n = 20$ consistently to optimality. However, it still holds that the optimality gaps obtained by A*+BS+LS are, in most cases, smaller than (or equal to) the ones of CP/ILOG. In those cases in which CP/ILOG returns smaller gaps, A*+BS+LS is usually able to solve more instances to optimality. The average optimality gaps obtained by A*+BS+LS never exceed 0.915%, whereas the ones of CP/ILOG range up to 4.784%. Finally, LLBH again provides reasonable results with optimality gaps of no more than 4.976% in the shortest computation times, although as a pure heuristic, LLBH itself does not accompany its results with any lower bound. For CP+LLBH/ILOG, the picture looks different than it does concerning the balanced instances of type B. For instances in data set S, the CP+LLBH/ILOG approach is not able to solve on average more instances to optimality or to provide better optimality gaps than A*+BS+LS or CP/ILOG. Since, CP+LLBH/ILOG uses LLBH as search guidance it is not surprising that in particular for larger instances the average results are similar to those results obtained from LLBH. However, for small and middle-sized instances CP+LLBH/ILOG is able to provide smaller average optimality gaps than LLBH.

Regarding the observations of the benchmark set P: the A*+BS+LS approach, CP/ILOG as well as the pure A* approach could solve all instances with up to 20 jobs to optimality. For instances with 21 to 50 jobs A*+BS+LS is able to return the smallest average optimality gaps and can solve most instances to proven optimality. For instances with more than 51 jobs the results of A*+BS+LS and CP/ILOG dominate, and they are essentially on par concerning optimality gaps and instances solved to proven optimality.

The MIP approach is not even able to solve instances with up to $n = 10$ jobs to optimality and in most cases not even able to derive primal solutions for instances with more than 10 jobs. In general, MIP/CPLEX is not able to yield solutions for instances with more than 200 jobs. A reason for this weak performance are clearly the Big-M constraints and the resulting weak linear programming relaxation of the model.

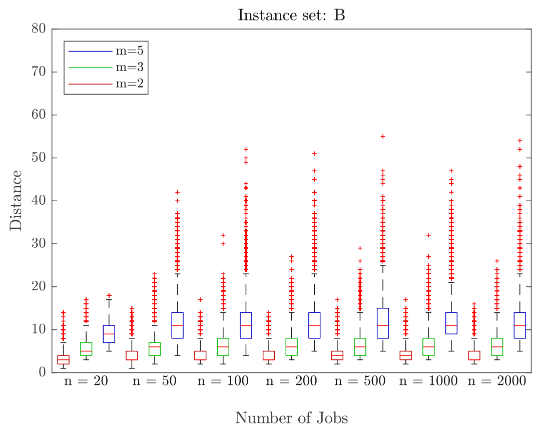
Regarding the pure A* approach, only instances with up to 20 jobs can be solved to proven optimality. For instances with more than 20 jobs A* runs out of memory for each single instance before the proven optimal solution could be found. In particular for balanced instances of type B where A*+BS+LS is able to solve 98% of the instances to proven optimality, the poor performance of the pure A** algorithm seems surprising at the first glance, since in those cases A* without BS and LS might be expected to require fewer node expansions due to A*'s optimality condition. However, as already mention in Section 3.6 this optimality condition holds only under certain conditions. Most importantly, the way how ties are handled can substantially impact the number of expanded nodes. In our case the interleaved lower bounds $\mathrm{MS}^{\mathrm{LB2}}$ and $\mathrm{MS}^{\mathrm{LB0}}$ are rather tight and ties occur in general relatively frequently. At least for the smaller instances, it is therefore in many cases sufficient to find an optimal solution quickly in order to also prove optimality. The A*+BS+LS approach frequently achieves this with its strong BS heuristic using $\mathrm{MS}^{\mathrm{LB2}}$ and $\mathrm{MS}^{\mathrm{LB0}}$ as search guidance, whereas the pure A* algorithm tends in case of the ties towards a breadth-first-search behavior, such that it runs rather soon out of memory. This behavior of the pure A* algorithm can be fixed by adding a tie breaking criterion which prefers partial schedules where more jobs are already scheduled over partial schedules with fewer jobs scheduled as we studied it already in [82]. In conjunction with A*+BS+LS, however, this kind of tie breaking would be counter-productive as it is in fact beneficial to initiate the embedded BS from a more diversified set of initial partial solutions, i.e., starting nodes.

The ARA* approach is only competitive to our A*+BS+LS approach in the context of small instances with up to 20 jobs. Concerning larger problem instances A*+BS+LS is generally able to solve more instances to proven optimality and to provide smaller optimality gaps in those cases in which optimality could not be proven. In accordance with observations in the literature [108], with an initial setting of $\varepsilon = 1.5$ ARA* behaves rather greedily and quickly finds a heuristic solution. However, the quality of this solution is generally worse than the initial solution obtained by A*+BS+LS. This seems to be the case because using BS with the non-inflated evaluation vector $\mathbf{f}$ as search guidance is more powerful. To find better solutions ARA* decreases ε during the search. In particular, the closer ε is to value one, the more ARA* behaves like the pure A* search. However, as already mentioned above, the behavior of pure A* tends to be similar to breadth-first search. Therefore, in many cases, ARA* runs either out of memory or out of time before solutions of similar quality as those obtained by A*+BS+LS are found.

In summary, we have shown that embedding BS and LS in our A* framework has a clearly positive impact on the quality of the obtained solutions and can sometimes even reduce the computation time in contrast to using A*+LLBH. Furthermore, we analyzed the anytime-behavior of A* by using different lower bounds and different beam widths for the BS. Here it turned out that the combined lower bound $\mathrm{MS}^{\mathrm{LB0}}$ and $\mathrm{MS}^{\mathrm{LB2}}$ almost always leads to the smallest average optimality gaps at the end of a run. Regarding different beam widths we could observe that selecting a higher beam width will usually provide solutions with smaller average optimality gaps. This, however, comes at the cost of a higher waiting time until a first complete solution is obtained by the search.

3.10.9 GVNS Evaluation

In this section we present results of our experimental evaluation of the GVNS for hard to solve instances. In order to study the practical efficiency of our incremental evaluation approach, an experiment was conducted where 10^4 randomly selected neighborhood moves in exchange and insertion neighborhoods where applied and the distance from the structural change to the last job in the synchronization border—that is the number of steps until the synchronization border could be determined—was traced. Figure 3.13 shows the synchronization distance of balanced and skewed instances of different sizes. For the considered instances, it can be observed that our approach exhibits an average amortized computation time behavior that is constant in the number of jobs, but increases with the number of secondary resources due to the nature of the synchronization border. Moreover, Figure 3.13 illustrates the sensitivity of the approach to significant resource imbalance, indicated by a higher number of outliers observed in the skewed instance set, likely due to large sections in the schedules where secondary resources are not utilized.



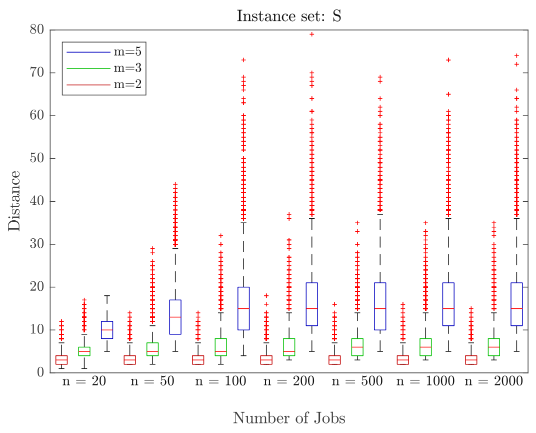


Figure 3.13: Synchronization distance: Number of steps required to identify the synchronization border in balanced and skewed instances, starting from the position of structural change due to a neighborhood move.

Finally, Table 3.2 compares average results of our GVNS on different instance classes to the baselines A*+BS+LS and CP/ILOG. Columns %-gap state the final optimality gaps in percent, which is calculated by $100\% \cdot (\mathrm{MS}(\pi) - \mathrm{LB})/\mathrm{LB}$, where LB refers to the strongest lower bound obtained from either A*+BS+LS or CP/ILOG, whereas columns %-opt lists the percentage of proven optimal solutions. Both columns use the best lower bound obtained from experiments in Section 3.10.8. Columns $\sigma_{\text{\%-gap}}$ show the standard deviations of the corresponding average optimality gaps. Column t provides the median time the GVNS required to obtain its best solution in a run. To obtain statistically more stable results, we executed the GVNS ten times for each of the 50 instances per instance class. For the anytime A* algorithm and for the CP solver, column t shows the median time when the algorithms terminated either because the optimal solution has been found or the time- or memory limit was exceeded.

Generally, Table 3.2 shows that the GVNS manages to obtain heuristic solutions comparable to those of the A* search, while both approaches show their specific advantages on particular subsets of instances. For balanced instances, on the one hand, A* search already showed its effectiveness, where even large instances up to 2000 jobs could be solved to proven optimality. The GVNS obtains similar results for instances with $m = 2$ and $m = 5$, with respect to solution quality, although the temporal performance decreases with increasing instance size in comparison. For instances with $m = 3$ the GVNS's solutions are clearly worse than those of the A* search, although the average optimality gap of $\leq 0.288\%$ is still small and much better than the one of the CP approach. In Kaufmann [94] we show that providing an initial solution obtained LLBH can further improve the solution quality for this particular instance set, however, A* is still superior both with respect to quality as well as temporal behavior.

For the harder skewed instances, on the other hand, our GVNS shows a significant improvement compared to both baseline methods with an average optimality gap below

Table 3.2: Average results of GVNS, A*+BS+LS, and the CP approach.

			GVNS				A*+BS+LS				CP/ILOG			
type	n	m	%-gap	$\sigma_{\%\text{-gap}}$	%-opt	t[s]	%-gap	$\sigma_{\%\text{-gap}}$	%-opt	t[s]	%-gap	$\sigma_{\%\text{-gap}}$	%-opt	t[s]
B	50	2	**0.000**	0.00	100.0	<0.1	**0.000**	0.00	100.0	1.1	**0.000**	0.00	100.0	<0.1
B	100	2	**0.000**	0.00	100.0	<0.1	**0.000**	0.00	100.0	2.0	**0.000**	0.00	100.0	<0.1
B	200	2	**0.000**	0.00	100.0	0.2	**0.000**	0.00	100.0	5.4	**0.000**	0.00	100.0	<0.1
B	500	2	**0.000**	0.00	100.0	2.4	**0.000**	0.00	100.0	35.3	**0.000**	0.00	100.0	1.3
B	1000	2	**0.000**	0.00	100.0	13.0	**0.000**	0.00	100.0	8.9	**0.000**	0.00	100.0	9.2
B	2000	2	**0.000**	0.00	100.0	83.5	**0.000**	0.00	100.0	46.3	<0.001	0.01	98.0	63.5
B	50	3	0.050	0.22	91.2	<0.1	**0.017**	0.08	96.0	1.1	0.068	0.30	92.0	<0.1
B	100	3	0.112	0.29	79.6	0.1	**0.021**	0.09	92.0	2.0	0.226	0.55	78.0	4.2
B	200	3	0.176	0.45	74.0	2.1	**0.016**	0.06	92.0	5.9	0.556	1.12	56.0	319.4
B	500	3	0.260	0.42	47.0	422.3	**<0.001**	<0.01	98.0	35.9	2.212	1.83	20.0	900.0
B	1000	3	0.216	0.33	31.0	385.0	**0.001**	<0.01	98.0	6.1	3.094	1.46	2.0	899.9
B	2000	3	0.288	0.34	15.0	843.2	**0.005**	0.04	98.0	23.8	4.220	1.20	0.0	900.0
B	50	5	<0.001	<0.01	99.4	0.1	**0.000**	0.00	100.0	1.2	**0.000**	0.00	100.0	0.7
B	100	5	**0.000**	0.00	100.0	0.4	**0.000**	0.00	100.0	2.2	**0.000**	0.00	100.0	9.5
B	200	5	**0.000**	0.00	100.0	2.3	<0.001	0.00	98.0	6.5	**0.000**	0.00	100.0	91.3
B	500	5	**0.000**	0.00	100.0	14.3	**0.000**	0.00	100.0	42.3	<0.001	<0.01	86.0	499.7
B	1000	5	<0.001	<0.01	96.0	49.2	**0.000**	0.00	100.0	7.9	0.359	0.12	0.0	900.0
B	2000	5	<0.001	<0.01	86.6	128.8	**0.000**	0.00	100.0	30.4	0.478	0.14	0.0	900.0
S	50	2	**0.163**	0.23	42.0	4.8	0.268	0.38	40.0	11.4	0.210	0.28	42.0	899.9
S	100	2	**0.172**	0.32	33.8	115.5	0.367	0.49	26.0	44.8	0.323	0.47	12.0	900.0
S	200	2	**0.111**	0.18	14.8	606.0	0.440	0.33	2.0	65.2	0.642	0.51	0.0	900.0
S	500	2	**0.095**	0.08	0.0	831.7	0.532	0.18	0.0	88.7	2.736	0.51	0.0	900.0
S	1000	2	**0.105**	0.06	0.0	813.3	0.725	0.20	0.0	176.8	4.636	0.43	0.0	900.0
S	2000	2	**0.214**	0.11	0.0	892.6	0.786	0.18	0.0	252.7	4.784	0.39	0.0	900.0
S	50	3	**0.035**	0.15	82.0	0.2	0.053	0.21	82.0	1.3	**0.035**	0.15	80.0	27.7
S	100	3	**0.030**	0.10	82.8	3.5	0.153	0.37	50.0	16.5	0.060	0.15	52.0	899.7
S	200	3	**0.025**	0.11	78.8	21.5	0.117	0.26	34.0	26.4	0.135	0.21	36.0	899.8
S	500	3	**0.006**	0.02	42.4	370.5	0.177	0.24	14.0	121.6	1.360	0.76	4.0	900.0
S	1000	3	**0.009**	0.02	19.2	584.5	0.621	0.47	2.0	48.0	2.872	0.93	0.0	900.0
S	2000	3	**0.041**	0.05	5.8	863.3	0.701	0.41	0.0	80.2	4.296	0.98	0.0	900.0
S	50	5	0.046	0.14	83.7	<0.1	0.077	0.19	80.0	1.4	**0.045**	0.14	84.0	15.0
S	100	5	**0.006**	0.02	88.4	1.2	0.064	0.18	66.0	6.1	0.019	0.04	70.0	899.6
S	200	5	**0.034**	0.14	77.2	18.3	0.281	0.49	34.0	38.8	0.161	0.25	28.0	900.0
S	500	5	**0.009**	0.02	46.8	351.7	0.347	0.34	16.0	188.6	1.229	0.95	8.0	899.9
S	1000	5	**0.012**	0.02	22.2	625.3	0.702	0.50	0.0	387.3	2.478	1.11	0.0	900.0
S	2000	5	**0.105**	0.10	2.0	893.6	0.915	0.54	0.0	789.3	4.229	1.22	0.0	900.0

0.214%. Instances with two secondary resources tend to be among the more difficult ones, where even for small instances with 50 jobs, optimality could be proven with the lower bound only in 42% of the runs. This, however, could as well be an indicator for the lower bounds being off the optimum. Interestingly, the GVNS still shows an improvement with respect to the number of obtained proven optimal solutions, where particularly for small to moderately large instances up to 88% could be solved to proven optimality, despite the inherent incompleteness of the GVNS.

3.11 Conclusion

In this chapter we considered the problem of scheduling a set of jobs where each job requires two resources: a common resource shared by all jobs for part of their processing, and a secondary resource for the whole processing time. The objective is to minimize the makespan. We showed that this problem is NP-hard and that we could derive tight lower bounds for the makespan. These lower bounds are exploited in the fast construction heuristic LLBH and a complete A* search. The A* algorithm features in particular a special state graph structure with nodes holding sets of non-dominated time-records in order to exploit symmetries and to keep the memory consumption reasonable. Since the basic A* algorithm may find a complete solution for large, difficult instances only after significant search, we enhanced the search by switching the search strategy in regular intervals to a beam search. Nodes where the lower bound is greater than or equal to the makespan of the so far best solution are pruned. Furthermore, complete solutions undergo a LS based on the insertion neighborhood. In this way excellent heuristic solutions are obtained early and the method exhibits a good anytime behavior while remaining complete.

Our experimental evaluation shows that this A*+BS+LS can solve very large non-trivial instances with up to 2000 jobs either to proven optimality or with typically very small remaining gaps of less than one percent. We observed, however, that the difficulty of problem instances is not so much related to the number of jobs, but primarily determined by the distribution of the workload on the resources. More specifically, if the instances exhibit a skewed workload and the workload of the common resource is similar to the workload of the dominant resource among the secondary resources, then these instances are typically hard to solve. In our experiments we therefore considered such difficult skewed instances as well as easier balanced instances and studied the impact of diverse parameters like the different lower bounds, the beam width, and the usage of the LS. Furthermore, we compared our approach to the anytime A* variants APS from [155] and ARA* from [108], the pure A* algorithm and a basic CP model solved by the ILOG CP solver. In particular for large instances A*+BS+LS significantly outperforms the ILOG CP approach and in almost all cases where the ILOG CP solver provides smaller average optimality gaps our approach is able to solve more instances to optimality. In addition, we compared our anytime A* algorithm to a position-based mixed integer programming (MIP) model solved by CPLEX, which is, however, not competitive at all. Only for small instances was CPLEX able to obtain heuristic solutions.

In addition, we devised a GVNS in order to obtain even better solutions from hard-to-solve instances. We efficiently evaluate solutions in the course of a neighborhood search in incremental ways and applied it to variants of insertion and exchange neighborhood structures. Insertion and exchange moves were utilized in the intensification phase, a piped VND, as well as in the diversification phase as for randomized shaking.

The experimental analysis of the GVNS first dealt with the practical efficiency of the incremental evaluation scheme, which still has a linear computation time in the number of jobs in the worst-case but exhibits a essentially a constant average computation time on all our benchmark instances. When comparing the GVNS to A*+BS+LS and the CP model, we observed the GVNS's ability to obtain high-quality solutions for a diverse set of large instances with an average optimality-gap of $\leq 0.288\%$. Although for balanced instances, the A*+BS+LS algorithm was out of reach, for particularly hard instances, In particular for hard instances with skewed workloads the GVNS showed its effectiveness, where significantly better solutions could be obtained than with A*+BS+LS.

Further research may consider a closer investigation of different starting strategies for the embedded BS as well as an adaptive tuning of the beam width. In fact, the proposed general A*+BS+LS framework is rather problem-independent, and its applications in other problem domains appears promising. Concerning the GVNS it would be interesting to investigate the computation time of the incremental evaluation scheme also from a theoretical point-of-view, in the hope that the constant amortized time observed here in practice can even be proven for a larger class of instances.

CHAPTER 4

A*-based Construction of Multivalued Decision Diagrams

This chapter presents a novel construction technique for multivalued decision diagrams (MDDs) that relies on the principles of A* search and uses problem specific upper bound functions to guide the compilation process. Another specialty of this novel A*-based construction (A*C) mechanism is that the created MDDs are not necessarily based on a layered structure. This has the advantage that multiple nodes representing the same state can be avoided and consequently also many redundant isomorphic substructures. In particular, A*C can be advantageous over traditional compilation methods from the literature if the considered problem exhibits selection and sequencing aspects.

Sections 4.1 and 4.2 introduce the basic ideas of the A*-based compilation process and of non-layered MDDs. Section 4.3 provides in particular for relaxed MDDs the problem independent details of A*C. The A*C is then applied on two NP-hard optimization problems to demonstrate the benefits over traditional compilation methods. The first problem, considered in Section 4.4, is an extension of the job sequencing problem with one common and multiple secondary resources (JSOCMSR) from the previous Chapter 3 by considering in addition time window constraints as well as a prize for each job. Instead of scheduling all jobs and minimizing the makespan, a subset of jobs must be selected that can be feasibly scheduled and the total prize of scheduled jobs is maximized. Preliminary work on this prize-collecting job sequencing problem with one common and multiple secondary resources (PCJSOCMSR) was first presented at *the 12th International Conference of the Practice and Theory of Automated Timetabling* (PATAT 2018) [83], where a classical A* search was used to solve small and middle-sized instances. An extended version of this work was published in the special issue *The Practice and Theory of Automated Timetabling* of the journal *Annuals of Operations Research* [84], where besides the application of treatment scheduling for cancer patients another application is additionally considered: pre-runtime scheduling of electronic systems within aircraft. Works [83] and [84] are captured in the first part of Section 4.4. Moreover, this whole chapter is mainly based on work [78], published in the *Computers & Operations Research* journal. This work presents the layer-free compilation of relaxed MDDs with A*C to solve the PCJSOCMSR. In order to obtain not only an upper bound from the compiled relaxed MDDs, work [78] proposes in addition a method to compile restricted MDDs by using a previously compiled relaxed MDD to obtain heuristic solutions. In this way the compilation time of the restricted MDD can be substantially decreased. Note that the development of this restricted MDD compilation method was mainly conducted by my coauthors and is here presented for the sake of completeness only. Section 4.5 reveals how to compile relaxed MDDs with A*C for the longest common subsequence (LCS) problem. A corresponding article of Section 4.5 is submitted to the *18th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (CPAIOR 2021) [81]. Finally, Section 4.6 concludes this chapter.

4.1 Introduction

Decision diagrams, as introduced in Section 2.5, are in essence data structures that provide graphical representations of the solution space of a combinatorial optimization problem (COP). More specifically, a relaxed decision diagram (DD) represents a superset of all feasible solutions in a compact way and can therefore be seen as a discrete relaxation of the COP. A relaxed DD can be used to obtain a dual bound, but it also provides a fruitful basis for alternative branching schemes [14] and constraint propagation [6], for example.

For the kind of problems where a subset of elements is to be selected, so-called binary decision diagrams (BDDs) are typically used. Here, solutions are usually represented by binary vectors and each layer of the BDD is associated with a Boolean decision variable indicating whether an element is selected or not. In contrast, for problems in which an optimal sequence of elements shall be found, it is more natural to apply so-called MDDs. Then, a solution is represented by a permutation of the given elements, and consequently a layer in the MDD is associated with the decision which element appears at the respective position in the permutation.

In this chapter we consider DDs for COPs that combine the selection aspect with the sequencing aspect, i.e., problems in which a subset of initially unknown size needs to be selected from some ground set of elements and the selected elements need to be ordered to form a complete solution. Problem-specific constraints restrict the solution space so that not all subsets have a feasible order. More specifically, in this chapter we consider the PCJSOCMSR and the LCS problem. The former problem, consists of a set of jobs such that each job is associated with a prize and the objective is to select a subset of jobs and find a feasible schedule such that the total prize is maximized. The well known classical standard LCS problem consists of a set of input strings defined over a finite alphabet and the task is to find a longest subsequence that is common to all input strings. A subsequence can be derived from an input string by removing zero or more characters from the input string. For the LCS problem the finite alphabet can be seen as the ground set of elements such that characters must be selected in a specific order to obtain a common subsequence. In contrast to PCJSOCMSR, where jobs can be selected only one time, the LCS problem allows selecting characters multiple times, i.e. a common subsequence can contain the same character more than once. Note that besides the PCJSOCMSR and the LCS problem, the type of problems that combines the element selection aspect with the sequencing aspect is not uncommon. For example, the prominent class of orienteering problems [64], also called selective traveling salesperson problem (TSP), falls into this category as well as order acceptance and single machine scheduling [126, 144], prize-collecting single machine scheduling [43], and other scheduling problems in which the number of tardy jobs shall be minimized (which corresponds to selecting and scheduling a subset of the jobs), see, e.g., [106, 124]. More generally, similar problems also appear as pricing problems in column generation approaches, for example for vehicle routing and parallel machine scheduling problems. Last but not least, assortment problems also exhibit the selection aspect—although sometimes with decisions beyond binary ones—and occasionally the sequencing aspect, for example when optimizing over time [46].

For the type of problem with both selection and sequencing decisions we consider, it is natural to build upon MDDs similar to those from Cire and van Hoeve [40], as solutions can be represented by permutations of the chosen elements. In contrast, it does not seem possible to effectively cover the sequencing aspect in some BDD variant. A particularity of our case is that feasible solutions may have arbitrary size in terms of the number of selected elements. This leads us to a novel technique for constructing relaxed MDDs. The method is inspired by A* search, a commonly used algorithm in path planning and problem solving [70]; see also Section 2.2.2 for more details. A priority queue is maintained for open nodes that still need further processing. A fast-to-calculate exogenous dual bound function is used as heuristic function to iteratively select the next node to be processed. To keep the constructed relaxed MDD compact, nodes are merged in a carefully selected way when the open list reaches a certain size. We show for the PCJSOCMSR as well as for the LCS problem that the relaxed MDDs obtained by the A*-based method yield substantially stronger bounds than relaxed MDDs of comparable size constructed by traditional compilation methods. The main reasons for this advantage are (a) the guidance by the dual bound heuristic, (b) that our construction is able to effectively avoid multiple nodes for identical states at different layers of the MDD, and (c) that similar nodes can also be merged across different layers. Substantial redundancies that cannot be avoided in the standard construction techniques are therefore less problematic in our approach.

In order to not just obtain dual bounds, we further describe the construction of a restricted DD that yields promising heuristic solutions for the PCJSOCMSR. Restricted DDs in general represent subsets of all feasible solutions. Hereby we contribute with a novel way of utilizing a previously constructed relaxed DD in order to substantially speed up the construction of a restricted DD. For the LCS problem we are content with obtaining stronger dual bounds from relaxed DDs compiled with A*C than dual bounds obtained from existing dual bound procedures from the literature.

Furthermore, the proposed methods to compile restricted and relaxed DDs to obtain primal and dual bounds can be straightforwardly combined in an exact branch and bound (BB) approach similar as shown by Bergman at el. [15]. The BB algorithm branches over a subset of nodes in the relaxed MDD, which is compiled by the proposed A* based construction scheme. The same relaxed MDD can be used to obtain a dual bound as well as to speed up the construction of a restricted MDD in order to obtain a primal bound. Such an exact BB method, however, is beyond the scope of this dissertation and therefore left for future work.

4.2 Decision Diagrams for Combinatorial Optimization

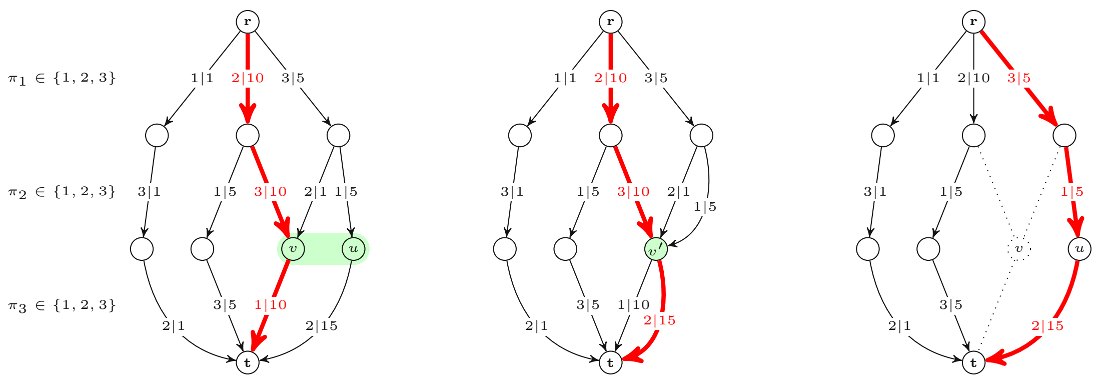
For the sake of completeness and clarity we briefly repeat here the definition of different kinds of DDs as already done in Section 2.5 of Chapter 2. However, we will do this more in the context of sequencing problems that exhibit also a certain selection aspect.

In this chapter a DD is defined as a directed weighted acyclic multi-graph $\mathscr{D} = (V, A)$ with node set $V(\mathscr{D})$ and arc set $A(\mathscr{D})$. In the literature, the node set $V(\mathscr{D})$ is usually partitioned into layers $V(\mathscr{D}) = V_1(\mathscr{D}) \cup \ldots \cup V_{n+1}(\mathscr{D})$, where n corresponds to the number of decision variables representing a solution. The first and the last layer are singletons and contain the root node $\mathbf{r} \in V(\mathscr{D})$ and the destination node $\mathbf{t} \in V(\mathscr{D})$, respectively. Each arc $\alpha = (u, v) \in A(\mathscr{D})$ is associated with a value $\mathrm{val}(\alpha)$ and directed from a source node u in some layer $V_i(\mathscr{D})$ to a destination node v in the subsequent layer $V_{i+1}(\mathscr{D})$, $i \in \{1, \ldots, n\}$. Such an arc refers to the assignment of value $\mathrm{val}(\alpha)$ to the i-th decision variable. While the domain of values in BDDs is restricted to $\{0, 1\}$, MDDs have arbitrary finite domains corresponding to those of the respective decision variables.

Each path from the root node $\mathbf{r}$ to the target node $\mathbf{t}$ corresponds to a solution encoded in the DD. An *exact DD* has a one-to-one correspondence between feasible solutions and the existing $\mathbf{r}$-$\mathbf{t}$ paths. Let us consider a sequencing problem where a subset of the permutations $\pi = (\pi_1, \ldots, \pi_n)$ of the ground set $\{1, \ldots, n\}$ forms the set of feasible solutions. Figure 4.1a shows an example for an exact MDD with $n = 3$ encoding the permutations (1,3,2), (2,1,3), (2,3,1), (3,2,1), and (3,1,2).

Each arc $\alpha \in A(\mathscr{D})$ has a length $z(\alpha)$ (or prize, cost, etc.) which gives the corresponding variable assignment's contribution to the objective value if it is chosen. The total length of an $\mathbf{r}$-$\mathbf{t}$ path thus corresponds to the solution's objective value. We assume throughout this chapter that the considered optimization problem is a maximization problem. Consequently, we are looking for a longest $\mathbf{r}$-$\mathbf{t}$ path.

As long as the DD is not too large, a longest path can be found efficiently as the DD is acyclic. Unfortunately, exact DDs for NP-hard optimization problems will in general have exponential size. This is where relaxed DDs come into play: They are more compact,



(a) Exact MDD (b) Relaxed MDD (c) Restricted MDD

Figure 4.1: Examples of an exact, a relaxed, and a restricted MDD for a sequencing problem with ground set $\{1,2,3\}$. Each arc label shows both the element to be assigned to the corresponding variable and the arc length as $\mathrm{val}(\alpha)|z(\alpha)$. The longest path is highlighted. For the exact MDD, the longest path encodes the optimal permutation $\pi^* = (2,3,1)$ with a total prize of 30. The relaxed MDD approximates the exact MDD by merging nodes u and v into node v'. The corresponding longest path encodes the infeasible solution $\pi = (2,3,2)$ and yields the upper bound 35. In the restricted MDD, node v and its incident arcs are removed and therefore only a subset of all feasible solutions is encoded. Its longest path is the permutation $\pi = (3,1,2)$ of length 25, which provides a lower bound.

and they approximate an exact DD by encoding a superset of all feasible solutions. The longest path of a relaxed DD therefore provides an upper bound to the original problem's optimum solution value. A restricted DD, in contrast, encodes only a subset of the feasible solutions, and its longest path therefore yields a lower bound and a possibly promising heuristic solution. Figure 4.1b and 4.1c show a relaxed and a restricted DD for the exact DD in Figure 4.1a.

Each node $u \in V(\mathscr{D})$ carries problem-specific information through its *state* $\sigma(u)$ that is reached by all the partial solutions corresponding to the paths from the root node to node u. In our case, when the MDDs represent subsets of permutations, each state includes the subset of elements by which the partial solutions may still be extended, thereby defining which outgoing arcs exist; we denote this set as $P(u)$.

Decision diagrams are usually derived from a dynamic programming (DP) formulation of the considered problem, and therefore a strong relationship exists between the DP's state transition graph and the nodes of the a DD [74]. We will see this relationship in more detail when considering the PCJSOCMSR or the LCS specifically in Section 4.4.3 and 4.5.3, respectively.

Section 2.5 in Chapter 2 states that there are two fundamental methods for compiling relaxed DDs of limited size which will be briefly repeated here. These are, to the best of our knowledge, used in almost all so far published works where relaxed DDs are applied to address combinatorial optimization problems. The top-down compilation (TDC) starts with just the root node and iteratively creates the DD layer by layer, essentially performing a breadth-first search. The size of the DD is controlled by imposing an upper bound β, called *width*, on the number of nodes at each layer. If the size of a current layer exceeds β, then some nodes of this layer are selected and *merged* so that the layer's size is reduced to at most β. This merging is done in such a way that no paths corresponding to feasible solutions are lost; new paths corresponding to infeasible solutions may emerge, however.

The second frequently applied approach for constructing relaxed DDs is incremental refinement (IR). It starts with a trivial relaxed DD, e.g., a DD of width one, which has just one node in each layer. Then two major steps are repeatedly applied until some termination condition is fulfilled, e.g., a maximum number of nodes is reached. In the *filtering* step, the relaxation represented by the DD is strengthened by removing arcs that cannot be part of any path corresponding to a feasible solution. In the *refinement* step, nodes are split into pairs of new replacement nodes in order to remove some of the paths that correspond to infeasible solutions.

Besides TDC and IR, Bergman and Cire [12] proposed to consider the compilation of a relaxed DD as an optimization problem and investigated a mixed integer programming (MIP) formulation. While this approach is useful for benchmarking different compilation methods on small problem instances, it is computationally too expensive for any practical application. In another work, Römer et al. [136] suggested a local search framework that serves as a more general scheme to obtain relaxed DDs. It is based on a set of local operations for manipulating and iteratively improving a DD, including the node splitting and merging from IR and TDC, respectively, and arc redirection as a new operator. Again, both approaches are strongly layer-oriented.

Especially in the context of binary DDs, a commonly used extension that frequently yields more compact DDs are so-called *long arcs* [33, 121]. They skip one or more layers and represent multiple variable assignments with one arc. In *zero-suppressed DDs*, variables corresponding to skipped layers take the value zero, while in *one-suppressed* DDs they get value one. Alternatively, a long arc may indicate that the skipped variables can take either value. For example, Bergman et al. [17] suggested to use zero-suppressed DDs for the independent set problem, while Kowalczyk and Leus [103] applied them to solve the pricing problem in a branch-and-price algorithm for parallel machine scheduling.

In conjunction with scheduling and sequencing problems, MDDs were already successfully applied e.g. to single machine scheduling problems [40], the time-dependent traveling salesman problem with and without time windows, the time-dependent sequential ordering problem [99], and job sequencing with time windows and state-dependent processing times [75].

All these approaches utilize MDDs for permutations similar to our example in Figure 4.1. An alternative way of representing sets of permutations as DDs has been described by

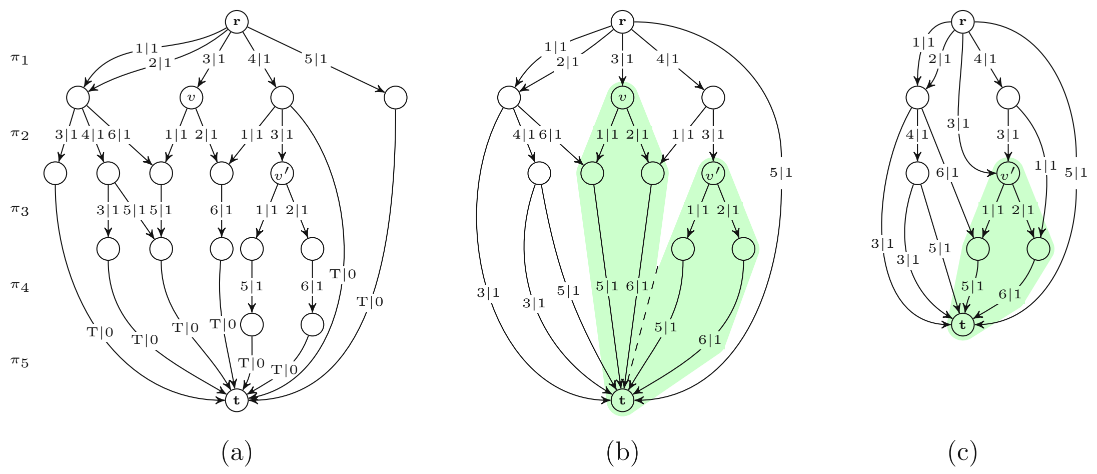


Figure 4.2: MDD variants for a problem with both selection and sequencing decisions encoding the same set of solutions: (a) using artificial termination arcs with val(α) = T; (b) redirecting all arcs leading to non-extendable states directly to the target node (if appending an element may never yield a worse feasible solution); (c) additionally avoiding multiple instances of isomorphic substructures (shaded parts).

Minato [122]. It builds upon zero-suppressed decision diagrams and encodes permutations by binary decision variables that indicate the transposition of pairs of elements. While this approach offers interesting advantages concerning certain algebraic operations, it appears nontrivial to efficiently express typical objective functions from routing and scheduling in terms of arc lengths on such DDs.

4.2.1 MDDs for Problems with Both Selection and Sequencing Decisions

To address problems like the PCJSOCMSR/LCS, the above described MDDs for permutations can be extended in natural ways.

A commonly used approach for modeling problems with multiple different goal states is to use a single target node **t** and connect each other node that corresponds to a feasible end state to this target node with a special *termination arc* of length zero. Such a termination arc α has a special value val(α) = T and does not correspond to any classical variable assignment. See Figure 4.2a for an example of such an approach in our case.

A simpler method can be used for optimization problems where appending an element to a solution, if feasible, always leads to a solution that is not worse. This is, in particular, the case when all arc lengths are non-negative. Here, we can avoid additional artificial arcs and simply redirect all arcs that lead to a non-extendable state directly to the target node **t**; see Figure 4.2b. These redirected arcs may now skip layers. In contrast to the previously discussed long arcs, however, our arcs here still represent single variable assignments. In the remainder of this work, we consider just this simpler redirection approach without explicit termination arcs. However, the algorithmic concepts we present can also be adapted in a straightforward way to the more general DD structure with termination arcs.

A further advantage of our MDDs for problems with both selection and sequencing decisions is illustrated in Figure 4.2b–c. In Figure 4.2b, consider the substructures rooted at nodes v and v' and note thereby that due to the variable solution length, isomorphic substructures appear. In Figure 4.2c, the MDD is condensed by storing this substructure just once, with all arcs leading to the two substructures in Figure 4.2b redirected to the single substructure. In this way, many redundancies might be avoided and substantially more compact MDDs representing the same set of solutions may be obtained. Note, however, that classical DD construction techniques such as TDC and IR are not able to create such an MDD as they rely on the notion that an arc originating at layer i, $i = 1, \dots, n$, (or a long arc passing a layer i) assigns a value to the i-th decision variable. The A*-based construction method we will propose in Section 4.3 does not rely on the layer-to-variable relationship but more generally just assumes that on any **r**-**t** path, the i-th arc represents an assignment to the i-th variable.

4.3 Basic Concepts of A*-based Construction

We propose to construct relaxed MDDs for the PCJSOCMSR/LCS problem and possibly other problems with both selection and sequencing decisions in a novel way that essentially adapts the classical TDC towards the spirit of A* search, i.e., instead of following a breadth-first search we turn towards a best-first search. Since A* search is a purely state based approach, our compilation method does not require to define an ordering of decision variables and to assign them to a fixed number of layers in advance. Hence, layers do not play a role anymore. To compare this with traditional compilation methods in a more systematic way, let us assume that the A* state graph is aligned to the underlying decision variables. This means that in case of sequencing problems the i-th arc of an **r**-**t** path corresponds to the i-th decision variable, whereas for DDs, compiled with traditional layer-based methods, the i-th decision variable corresponds to the i-th layer (cf. Section 4.2.1, Figure 4.2c and 4.2b, respectively). The resulting key characteristics of this scheme are:

1. It naturally avoids multiple nodes for identical states at different layers and consequently multiple copies of isomorphic substructures (cf. Section 4.2.1 and Figure 4.2).
2. Node expansions and the selection of nodes to be merged are guided by an auxiliary upper bound function. This upper bound function may be stronger than longest paths derived from DDs.
3. Partner nodes for merging are selected by considering state similarity and a more flexible merging of nodes across different layers is enabled in a natural way.
4. Solutions of different lengths are represented naturally.

These features may allow obtaining more compact relaxed MDDs that provide tighter upper bounds than the so far used classical construction techniques. Note that, if the A* search is appropriately defined, there may also be connections to non-serial DP, where the choice of the next decision variable may be state dependent. See Hooker [74] for more details about non-serial DP.

The A* based approach may not be applicable or advantageous if it is not possible to prevent the emergence of cycles in the underlying state graph when merging nodes. Traditional methods naturally avoid such cycles by merging nodes only of the same layer. We will discuss this topic and a possible cycle-avoidance mechanism in more detail in Section 4.3.4. Another disadvantage of A* based compilation may be that the time complexity of A* search is in general exponential in the worst case (see Section 2.2.2). This implies that the A* based compilation of relaxed DDs has an exponential worst case time complexity too. The practical compilation time, however, is competitive with the traditional TDC method, which exhibits a polynomial time complexity in the worst case, as our experimental results in Sections 4.4.10 and 4.5.6 will show.

4.3.1 A* Search

A* search [70] is a commonly applied technique in path planning and problem solving. It is well known for its ability to efficiently find best paths in very large (state) graphs. The following brief overview on A* search builds upon the notation already introduced in our DD setting. For more details on A* search, we refer to Section 2.2.2 in Chapter 2. A* search follows a best-first-search strategy and uses as key ingredient a heuristic function $Z^{\mathrm{ub}}(u)$ that estimates, for each node u reached, the cost to get to the target node $\mathbf{t}$ in a best way, the so-called *cost-to-go*[1]. All not yet expanded nodes, called *open nodes*, are maintained in a priority queue, the *open list* Q. This list is partially sorted according to a priority function

$$f_*(u) = Z^{\mathrm{lp}}(u) + Z^{\mathrm{ub}}(u) \tag{4.1}$$

where $Z^{\mathrm{lp}}(u)$ denotes the length of the so far best path from the root node $\mathbf{r}$ to node u. Initially, Q contains just the root node. The A* search then always takes a node with the highest priority from Q and *expands* it by considering all outgoing arcs. Destination nodes that are reached in better ways via the expanded node are updated and newly reached nodes are added to Q. Considering maximization, a heuristic function Z^{ub} that never underestimates the real cost-to-go (i.e., is an upper bound function) is called *admissible*. A* search terminates when $\mathbf{t}$ is selected from Q for expansion. If an admissible heuristic is used, then $Z^{\mathrm{lp}}(\mathbf{t})$ is optimal. From now on let us assume that Z^{ub} is indeed admissible. The efficiency of A* search mostly relies on how well the heuristic function estimates the real cost-to-go.

[1]Note that the term cost-to-go is more fitting in the context of minimization, as introduced in Section 2.2.2. We aim at maximizing the total length, benefit, or prize, and one might therefore consider "length-to-go" more suitable. Nevertheless, we stay here with the term cost-to-go as it is commonly used in the literature.

4.3.2 Constructing Exact MDDs by A* Search

When performing the A* search, all encountered nodes and arcs that correspond to feasible transitions are stored. If the construction process is carried out until the open list becomes empty and thereby all nodes have been expanded, then a complete MDD is obtained. Alternatively, the A* search's criterion can be applied, and then the search is terminated already when the target node is selected for expansion. In this case, typically substantially fewer nodes will have been expanded, and the obtained MDD is in general incomplete. Nevertheless, we know due to the optimality condition of A* search that at least one optimal path is contained and thus an optimal solution is indeed represented.

4.3.3 Constructing Relaxed MDDs

To obtain a compact relaxed MDD we now extend the above A*-based construction (A*C) by *limiting the open list*. This is achieved by merging similar and less promising nodes when the open list exceeds a certain size ϕ. Details on how we choose the nodes to be merged will be presented in Section 4.3.4. Selected nodes are merged in the same problem-specific ways as in traditional relaxed DD construction techniques. In particular, it has to be guaranteed that no paths corresponding to feasible solutions get lost. Sections 4.4.8 and 4.5.5 will show how this is done for the PCJSOCMSR and LCS problem, respectively.

When performing this MDD construction until the open list becomes empty, we now obtain a complete relaxed MDD that indeed represents a superset of all feasible solutions and yields an upper bound on the optimal solution value.

Alternatively, we may also here already terminate early once the target node is selected for expansion. Due to the merging and the optimality condition of A* search, we have then obtained a path whose length is a valid upper bound to the optimal solution value, and this bound cannot be further improved by continuing the MDD construction. Only longer paths corresponding to weaker bounds may later arise due to further node merges. Let us denote this best obtained bound by $Z^{\mathrm{ub}}_{\mathrm{min}}$.

Thus, the termination criterion to be used depends on the goal for which the MDD is constructed. If we are only interested in the upper bound or, for example, a DD-based BB [14] shall be performed, the early termination may be very meaningful and can save much time. However, should we indeed need a representation of a complete superset of all feasible solutions, the construction has to be continued.

Algorithm 4.1 shows the proposed MDD construction technique in pseudo-code. After the initialization phase, the main loop is entered. At each major iteration, a node u with maximum priority $f_*(u)$ is taken from the open list Q. As long as the target node $\mathbf{t}$ was not chosen for expansion, the node's f_*-value provides a valid upper bound and $Z^{\mathrm{ub}}_{\mathrm{min}}$ is updated accordingly in Line 6. If $\mathbf{t}$ was chosen, the optional early termination takes places.

Algorithm 4.1: A*-based construction of a relaxed MDD

```
   Input: open list size limit $\phi$
   Output: relaxed MDD $\mathscr{D} = (V, A)$ and upper bound to optimal solution value
 1 create root node $\mathbf{r}$ corresponding to initial state;
 2 open list $Q \leftarrow \{(\mathbf{r}, f_*(\mathbf{r}) = Z^{\mathrm{ub}}(\mathbf{r}))\}$;
 3 $Z^{\mathrm{ub}}_{\mathrm{min}} \leftarrow Z^{\mathrm{ub}}(\mathbf{r})$; $\mathbf{t}$-expanded $\leftarrow$ *false*;
 4 while $Q \neq \emptyset$ do
 5     $u \leftarrow$ pop node with largest $f_*(u)$ from $Q$;
 6     if *not* $\mathbf{t}$-*expanded* then  $Z^{\mathrm{ub}}_{\mathrm{min}} \leftarrow \min(Z^{\mathrm{ub}}_{\mathrm{min}}, f_*(u))$ ;
 7     if $u = \mathbf{t}$ then
 8         $\mathbf{t}$-expanded $\leftarrow$ *true*;
 9         // optional, if just the upper bound is of interest:
10         return incomplete MDD $\mathscr{D}$ and upper bound $Z^{\mathrm{ub}}_{\mathrm{min}}$;
11     end
12     if $u$ *not yet expanded* then
13         foreach *feasible successor state* $\Sigma$ *of* $\sigma(u)$ do        // expand node $u$
14             if $\nexists v \in V(\mathscr{D}) \mid \sigma(v) = \Sigma$ then  add new node $v$ to $V(\mathscr{D})$ with
                 $\sigma(v) = \Sigma$, $Z^{\mathrm{lp}}(v) = 0$ ;
15             add new arc $\alpha = (u, v)$ to $A(\mathscr{D})$;
16             if $Z^{\mathrm{lp}}(u) + \mathrm{z}(\alpha) > Z^{\mathrm{lp}}(v)$ then
17                 $Z^{\mathrm{lp}}(v) \leftarrow Z^{\mathrm{lp}}(u) + \mathrm{z}(\alpha)$;
18                 $Q \leftarrow Q \cup \{(v, f_*(v) = Z^{\mathrm{lp}}(v) + Z^{\mathrm{ub}}(v))\}$;
19             end
20         end
21     else
22         foreach *arc* $\alpha = (u, v) \in A(\mathscr{D})$ do                 // re-expand node $u$
23             if $Z^{\mathrm{lp}}(u) + \mathrm{z}(\alpha) > Z^{\mathrm{lp}}(v)$ then
24                 $Z^{\mathrm{lp}}(v) \leftarrow Z^{\mathrm{lp}}(u) + \mathrm{z}(\alpha)$;
25                 $Q \leftarrow Q \cup \{(v, f_*(v) = Z^{\mathrm{lp}}(v) + Z^{\mathrm{ub}}(v))\}$;
26             end
27         end
28     end
29     if $|Q| > \phi$ then                                              // reduce size of $Q$
30         try to merge nodes in $Q$ until $|Q| \leq \phi$ according to Alg. 4.2;
31     end
32 end
33 return *relaxed MDD M and upper bound* $Z^{\mathrm{ub}}_{\mathrm{min}}$
```

Algorithm 4.2: Reduce Q by merging nodes

```
   Input: open list $Q$, global set of collector nodes $V^c$ (initially empty)
   Output: possibly reduced open list $Q$
 1 for $u \in Q$ in increasing order of values $Z^{\mathrm{lp}}(\cdot)$ do
 2     if $|Q| \leq \phi$ then break ;
 3     while $u$ not expanded $\wedge\ \exists v \in V^c \mid L(v) = L(u) \wedge u \neq v \wedge\ v$ not expanded do
 4         create new node $v'$ with merged state $\sigma(v') = \sigma(u) \oplus \sigma(v)$;
 5         remove $u$ from $Q$ and $v$ from $Q$ and $V^c$;
 6         if $\exists v'' \in V(\mathscr{D}) \mid \sigma(v'') = \sigma(v')$ then
 7             $f''_{\mathrm{old}} \leftarrow f_*(v'')$;
 8             redirect all incoming arcs from $v'$ to $v''$;
 9             if $f_*(v'') > f''_{\mathrm{old}}$ then $Q \leftarrow Q \cup \{(v'', f_*(v'') = Z^{\mathrm{lp}}(v'') + Z^{\mathrm{ub}}(v''))\}$ ;
10             $v' \leftarrow v''$;
11         else
12             add node $v'$ to $V(\mathscr{D})$;
13             $Q \leftarrow Q \cup \{(v', f_*(v') = Z^{\mathrm{lp}}(v') + Z^{\mathrm{ub}}(v'))\}$;
14         end
15         $u \leftarrow v'$;
16     end
17     if $u$ not expanded then insert $u$ into $V^c$;
18 end
19 return $Q$
```

Next, the case when node u has not yet been expanded is handled by considering all feasible transitions from state $\sigma(u)$ and creating new nodes and arcs accordingly. If thereby a new path to a node v increases $Z^{\mathrm{lp}}(v)$, then node v is (re-)inserted into Q. If node u was already expanded, a re-expansion has to take place because a longer path to u, yielding a larger $Z^{\mathrm{lp}}(u)$, has been found in an iteration after the node's original expansion. This is done by propagating the updated $Z^{\mathrm{lp}}(u)$ to all its successor nodes and evaluating if they also need re-expansions. Note that, in general, we cannot avoid such re-expansions even when the upper bound function is consistent since node merges may lead to new longer paths.

After each node expansion, the algorithm checks if the size of the open list $|Q|$ exceeds the limit ϕ. If this is the case, then the algorithm tries to reduce Q by merging nodes as explained in the next section. Algorithm 4.1 terminates regularly when the open list becomes empty and returns the relaxed MDD together with the best obtained upper bound $Z^{\mathrm{ub}}_{\min}$.

4.3.4 Reducing the Open List by Merging

Merging different nodes usually introduces new paths corresponding to infeasible solutions, and this typically weakens the upper bound obtained. Therefore, we aim at quickly identifying nodes for merging that (a) are less likely to be part of some final longest path; (b) are associated with similar states, since this should imply that the merged state still is a strong representative for both; (c) do not introduce cycles in the MDD as they would lead to infinite solutions; and (d) ensure that the open list gets empty after a finite number of expansions. The last two aspects are crucial conditions to ensure a proper termination of the approach, and they are not trivially fulfilled due to the possibility to merge across different layers.

Aspect (a) is considered by iterating over the nodes in the open list in an increasing Z^{lp}-order and trying to merge each node with a suitably selected *partner node* in a pairwise fashion until the size of the open list does not exceed ϕ anymore. The motivation for the increasing Z^{lp}-order is that A* search has so far postponed the expansion of these nodes while other nodes with comparable Z^{lp} values have already been expanded. Therefore, the nodes with small Z^{lp} values can be argued to be less likely to appear in a longest path.[2]

The selection of the partner node for merging is done considering aspects (b) to (d) by utilizing a global set of so-called *collector nodes* V^{c}. To this end, we define a problem-specific labeling function $L(u)$ that maps the data associated with a node u—in particular its state $\sigma(u)$—to a simpler label of a restricted finite domain $\mathcal{D}_L$, thereby partitioning the nodes into subsets of similar nodes. Our labeling function, for example, may drop, aggregate, or relax parts of the states considered less important and condense the information in this way. Similar principles as in state-space relaxation [39] can be applied. The labeling function, however, may additionally also consider the upper bound $Z^{\mathrm{ub}}(u)$ as criterion for similarity; experimental results in Section 4.4.10 will show the particular usefulness of this. The global set of collector nodes V^{c} is initially empty and realized as a dictionary (e.g., hash table) indexed by the labels so that for each label in $\mathcal{D}_L$, there is at most one collector node in V^{c}, and thus $|V^{\mathrm{c}}| \leq |\mathcal{D}_L|$. In this way, we can efficiently determine for any node u if a related collector node with the same label $L(u)$ already exists and, in this case, directly access it.

Algorithm 4.2 shows the whole procedure to reduce the open list. As long as the open list is too large, nodes are selected in increasing Z^{lp}-order. For a chosen node u, it is checked if it is not yet expanded and if a corresponding collector node v, that is also not yet expanded, exists (Line 3). In this case, u and v are merged, yielding the new node v' with state $\sigma(v') = \sigma(u) \oplus \sigma(v)$, where $\oplus$ denotes the problem-specific state merging operation. All incoming arcs from u and v will be redirected to the new node v'. Consequently, u is removed from Q and v from Q as well as V^{c}. Next, we have to integrate the new node v' into the node set V by avoiding multiple nodes in the set V associated with the same state (Line 6). Furthermore, v' becomes a collector node in V^{c}, essentially replacing the former collector node v. Node v' may, however, have a different label than the former v,

[2]We remark that we considered in preliminary experiments also an increasing f_* order, thus processing the priority queue essentially in reverse order. While we obtained mostly MDDs of roughly comparable quality, they were sometimes significantly larger and more computation time was needed.

and some other collector node with the same label as v' may already exist in V^c. In this case, we iterate the merging with these nodes by continuing the while-loop in Line 3.

Note that Algorithm 4.2 shows the main idea pointing out the important steps. In a concrete implementation, a few additional corner cases need to be considered, in particular when collector nodes get changed (e.g., expanded) between two calls of Algorithm 4.2. Furthermore, our implementation marks merged nodes such that non-merged nodes may be preferred for expansion in case of ties (e.g. see the tie breaking criterion in Section 4.4.8). Tagging merged nodes may also be useful for a later implementation of an exact DD-based BB framework.

4.4 Prize-Collecting Job Sequencing with One Common and Multiple Secondary Resources Problem

In this section we will create relaxed MDDs with the A*C method for the PCJSOCMSR. The goals are (1) to show that for PCJSOCMSR stronger relaxed MDDs can be compiled with A*C than with other standard compilation methods from the literature and (2) to heuristically solve large instances of the PCJSOCMSR. Since the PCJSOCMSR is a newly introduced problem we also consider a classical A* search approach compared with a MIP formulation and a constraint programming (CP) formulation to solve small instances to proven optimality. In this way we get an impression of the instance size from which it makes sense to solve instances heuristically. To obtain heuristic solutions we create restricted MDDs with the TDC by using some structural information of a previously compiled relaxed MDD with A*C. In this way we are able to speed up the compilation process of the restricted MDD.

Chapter 3 introduced the JSOCMSR, which considers the scenario of scheduling a set of jobs where each job, for part of its execution, requires a common resource and, for the whole processing time, requires a secondary resource that is only shared with a subset of the other jobs. The goal of the JSOCMSR is to find a feasible schedule minimizing the makespan.

Besides an application of this problem in manufacturing, a more specific application can be found in the daily scheduling of cancer patients that are to receive particle therapy [120]. There, the common resource corresponds to a sophisticated particle accelerator, which accelerates proton or carbon particles to almost the speed of light. This particle beam is directed into one of a small set of treatment rooms where one patient can be radiated at a time. The treatment rooms are here the secondary resources. During the setup time, a patient is positioned and possibly sedated and after the actual radiation with the particle beam, typically some medical examinations are to be done before the patient can leave the room and it becomes available for a next patient. In such particle therapy treatment centers, there is usually only a single particle accelerator because of its huge cost and about two to three treatment rooms. Since these treatment rooms are typically individually equipped for handling different kinds of treatments, the assignment of patients to rooms is pre-specified. Ideally, patients are scheduled in such a way that the particle beam is directly switched from one room to another so that patients are radiated without any significant breaks in between.

However, the JSOCMSR as presented in Chapter 3 is in most cases only a strongly simplified model of real-world scenarios like the above patient scheduling. Most notably, the jobs start times are in practice frequently constrained to certain time windows arising from the availability of the underlying resources. Furthermore, in practice, it happens frequently that not all jobs can be scheduled due to these time windows and instead, a best subset of jobs that can be feasibly scheduled must be selected.

To also include such aspects is the focus of the current section: We extend the JSOCMSR by considering job-specific time windows, and instead of minimizing the makespan we aim at finding a feasible solution that maximizes the total prize of all scheduled jobs. To this end, each job has an assigned prize, which can simply take the value one if we want to maximize the number of scheduled jobs or it can take a value representing the priority of the job. Another possibility is that these prizes are correlated to the processing times of the jobs to avoid scheduling primarily short jobs.

These new aspects have a substantial impact on the structure of the problem and consequently on the algorithmic side, and existing methods for the JSOCMSR cannot be extended in efficient ways. Most importantly, the rather effective lower bound calculation for the makespan in the JSOCMSR from Section 3.4 is useless for the new problem variant due to the entirely different objective function and new decision variables. Therefore, we propose in Section 4.4.5 a new A* search to solve this prize-collecting variant of the JSOCMSR to proven optimality. In particular, we investigate four different upper bound calculations for partial solutions which are used as heuristic functions to estimate the costs-to-go within the A* search. A further aspect of our A* search is that there is no specific target state known in advance.

There is also another application of the PCJSOCMSR: pre-runtime scheduling of electronics within an aircraft, called avionics. The industrially relevant instances considered in [21, 91] are too complex and large-scale to be addressed directly and instead they need to be solved by some decomposition. The PCJSOCMSR appears as an important sub-structure both in the exact decomposition approach in Blikstad et al. [21] and the matheuristic approach in Karlsson et al. [91].

The considered system consists of a set of nodes and each of these contains a set of modules (processors) with jobs to be scheduled. Each node consists of one communication module, corresponding to the common resource, and a set of application modules, corresponding to the set of secondary resources. There are three types of jobs: partition jobs, communication jobs and regular jobs. Partition jobs, which are executed on the application modules run the system's software applications. Each of these jobs has to communicate with the communication module and we consider only the case where a partition job has to communicate either at the beginning or at the end of its execution. Typically, the processing time of partition jobs is long compared to other jobs and its usage of the common resource is short compared to the total processing time of the job.

The communication is handled by communication jobs and regular jobs, both executed on the communication module. These have short processing times and in order to model that they use the common resource only, an additional artificial secondary resource is included for these jobs. The communication and regular jobs use both the common resource and the artificial secondary resource for their whole processing time. Communication jobs and regular jobs together handle different kinds of communication (system external, inter- and intranode) and the communication jobs have the specific purpose of representing jobs used for sending communication messages [21, 91]. For this reason, the communication jobs can only be scheduled at specific time slots where communication messages can be sent and the length of a communication job's time window is equal to the job's total processing time. This distinguishes them from the regular jobs which have time windows with similar characteristics as those of the partition jobs.

To mimic the situation of creating a partial schedule for a node in the system, only a part of the total length of a schedule is considered and the tasks available exceed what is possible to include in the partial schedule. The prize of a job reflects the individual importance of including this job in the partial schedule. Note that compared to [21, 91], some simplifications are made with respect to the types of jobs included and by omitting precedence relations between jobs. Furthermore, we do not explicitly and fully consider the scheduling of the communication network used for communication between the nodes.

After a more formal problem definition in the next section and a survey of related work in Section 4.4.2, we describe the A* search in Section 4.4.5. This method relies on a specific state strengthening procedure and the use of a good heuristic guidance function corresponding to an upper bound calculation for the total prize that may still be an achieved from a partial solution onward. Concerning the latter, we investigate different possibilities based on linear programming (LP) and Lagrangian relaxations of a multidimensional knapsack problem formulation. For comparison purposes, we further consider, in Section 4.4.6, an order-based MIP model solved by Gurobi Optimizer Version 7.5.1 and, in Section 4.4.7, a constraint programming model solved by MiniZinc. To create relaxed MDDs, Section 4.4.8 describes the problem specific parts of the A*C method in order to apply it on PCJSOCMSR instances and Section 4.4.9 reveals how to compile restricted MDDs with the TDC method by using an earlier compiled relaxed MDD for large instance sizes that could not be solve to optimality by the considered exact approaches. Section 4.4.10 presents and compares computational results for all these approaches on instances of the avionics scenario as well as balanced instances of the particle therapy scenario. The results show that the proposed A* search can solve substantially larger instances with up to 80 jobs to optimality in shorter times than the other considered exact methods. Furthermore, on the heuristic side, experiments including comparisons with a variable neighborhood search heuristic on large benchmark instances with up to 500 jobs show the advantages of the proposed relaxed and restricted MDD construction techniques, respectively.

4.4.1 Problem Formulation

The prize-collecting job sequencing problem with one common and multiple secondary resources (PCJSOCMSR) considers the sequencing of a set of jobs where each job needs to respect resource constraints and time windows. The resource constraints refer both to a common resource that is used by all jobs and a set of secondary resources of which each job uses exactly one. It is assumed that it is usually not possible to find a feasible schedule that includes all jobs; instead each job is associated with a prize and the objective is to choose a subset of the jobs such that the sum of prizes of the sequenced jobs is maximized.

Let the set of all jobs be denoted by J, with $|J| = n$, and let the prize of job j be $z_j > 0$, $j \in J$. The problem is to find a subset of jobs $S \subseteq J$ that can be feasibly scheduled so that the total prize of these jobs is maximized:

$$Z^* = \max_{S \subseteq J} Z(S) = \max_{S \subseteq J} \sum_{j \in S} z_j. \tag{4.2}$$

The set of (renewable) resources is denoted by $R_0 = \{0\} \cup R$, with $R = \{1, \ldots, m\}$. During its execution, job j uses resource 0, referred to as the *common resource*, and one of the secondary resources $q_j \in R$, $j \in J$. Let $p_j > 0$ be the processing time of job j, during which it fully requires the secondary resource q_j, $j \in J$. Further, let $J_r = \{j \in J \mid q_j = r\}$ be the set of jobs that require resource r, $r \in R$. For job j, $j \in J$, the use of the common resource begins $p_j^{\text{pre}} \geq 0$ time units after the start of the job, has a duration of p_j^0, and ends $p_j^{\text{post}} = p_j - p_j^{\text{pre}} - p_j^0 \geq 0$ time units before the end of the job.

If a job j is scheduled, it must be performed without preemption and within one of its ω_j disjunctive time windows $W_j = \{W_{jw} \mid w = 0, \ldots, \omega_j\}$ with $W_{jw} = [w_{jw}^{\text{start}}, w_{jw}^{\text{end}}]$, where $w_{jw}^{\text{end}} - w_{j,w}^{\text{start}} \geq p_j$, $j \in J$. We assume that each job has at least one time window. For job j, let the release time be $T_j^{\text{rel}} = \min_{w=0,\ldots,\omega_j} w_{jw}^{\text{start}}$ and the deadline be $T_j^{\text{dead}} = \max_{w=0,\ldots,\omega_j} w_{jw}^{\text{end}}$. The overall time interval to consider is then $[T^{\min}, T^{\max}]$ with $T^{\min} = \min_{j \in J} T_j^{\text{rel}}$ and $T^{\max} = \max_{j \in J} T_j^{\text{dead}}$. Note that the existence of unavailability periods of resources is also covered by the above formulation since these can be translated into time windows of the jobs.

Since each job requires resource 0 and only one job can use this resource at a time, a solution to PCJSOCMSR implies a total ordering of the scheduled jobs S. Vice versa, a permutation $\pi = (\pi_i)_{i=1,\ldots,|S|}$ of a subset of jobs $S \subseteq J$ that can be feasibly scheduled can be decoded into a feasible schedule in a straight-forward greedy way by, in the order given by π, placing each job from S at its earliest feasible time with respect to when the resources are available after being used by all its preceding jobs. A schedule derived from a job permutation π in this way is referred to as a *normalized schedule*. Note that if this greedy approach is applied to a permutation of jobs and some job cannot be feasibly scheduled in this way, this permutation does not correspond to a feasible solution. Also, an optimal solution is either a normalized schedule or the order of the jobs in this optimal solution can be used to derive a normalized schedule with the same objective value. For

Instance:

j	p_j	p_j^{pre}	p_j^0	q_j	z_j	W_j
1	4	1	2	1	2	$\{[0,8]\}$
2	4	1	2	1	4	$\{[0,8]\}$
3	4	0	3	2	3	$\{[3,8]\}$
4	5	1	3	2	2	$\{[8,14]\}$

Optimal Solution π: $Z(\pi) = 9$

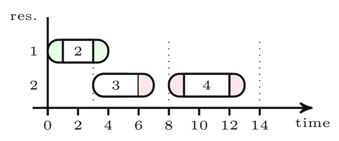


Figure 4.3: A PCJSOCMSR instance with $n = 4$ jobs and $m = 2$ secondary resources. Each job has exactly one time window in W_j. The optimal solution is given by the job sequence $\pi = (2, 3, 4)$ and its normalized schedule visualized on the right side. The white region in each job denotes the part where, in addition to the job's specific secondary resource, also the common resource is needed. Job 1 cannot be additionally scheduled due to the time windows and resource requirements. The solution's total prize is $Z(\pi) = 9$.

this reason the notation $Z(\pi)$ is used for the total prize of the normalized solution given by the job permutation π. Figure 4.3 shows (1) an example of an instance with four jobs and two secondary resources where each job has exactly one time window and (2) a corresponding optimal solution.

It is not difficult to see that the PCJSOCMSR is NP-hard: The decision variant of the JSOCMSR, which looks for a feasible schedule with a makespan not exceeding a given M, has already been shown to be NP-hard in Section 3.3. One can reduce this decision problem to the PCJSOCMSR in polynomial time by setting all time windows to $W_j = \{[0, M]\}$ and all prizes to $z_j = 1$. A solution to the JSOCMSR decision problem exists if and only if a solution to the PCJSOCMSR can be found that has all jobs scheduled.

4.4.2 Related Work

Since the PCJSOCMSR is an extension of the JSOCMSR, most of the related work of JSOCMSR form Section 3.2 is also related to the PCJSOCMSR. Consequently, we do not repeat these works here again.

Concerning the PCJSOCMSR, Maschler and Raidl [116] investigated TDC and IR approaches to construct relaxed MDDs and a TDC for restricted MDDs. In addition, an independent general variable neighborhood search (GVNS) metaheuristic was considered. An extended version [117] was published by the same authors, where in particular a smaller relaxed MDDs can be obtained during the IR approach by using techniques to detect and remove redundancies. Instances with up to 300 jobs were studied, which are so far clearly out of reach to be solved to proven optimality, as our experiments in Section 4.4.10 will also confirm. It turned out that IR frequently yields relaxed MDDs of roughly comparable size with a larger number of jobs. Incremental refinement's running times are, however, in general higher than those of TDC for constructing MDDs of comparable size. The heuristic solutions obtained from the restricted MDDs were usually better than or on par with the solutions obtained from the GVNS, except for the largest skewed instances, where the GVNS performed better. In our experimental investigations in Section 4.4.10 we will also compare to these approaches.

Moreover, the PCJSOCMSR is to some extent related to the well studied orienteering problem (OP), which essentially also combines the tasks of selecting a subset yielding a maximum prize with finding a sequence of the selected elements that make the solution feasible. Different variants of the OP have been studied, including OPs with time windows; for a survey see Gunawan et al. [64]. In the OP, each node is associated with a prize and travel times are known between all pairs of nodes. The task is to find a path from a given start node to an end node within a given time budget such that the total prize of the visited nodes is maximized. Due to the time budget not all nodes can be visited. In such an OP, the arrival time at a visited node only depends on the immediate predecessor, possible time windows, and the (constant) travel time between these two nodes. For PCJSOCMSR, the situation is more complicated due to the secondary resources: A much earlier scheduled job requiring the same secondary resource may impact the earliest starting time of the job to be scheduled next. The PCJSOCMSR, however, also does not generalize the OP with time windows as pairwise travel times are not covered by the PCJSOCMSR.

4.4.3 State Graph

In order to solve the PCJSOCMSR exactly by a classical A* search, presented in upcoming Section 4.4.5 and to compile relaxed MDDs with A*C we have to define a state graph. The mentioned algorithms will work on this state graph.

A state in PCJSOCMSR must describe all relevant aspects in order to determine the earliest starting time of any successive job that can be scheduled. For a node u in this state graph this is the tuple $\sigma(u) = (P(u), t(u))$ consisting of

- the set $P(u) \subseteq J$ of jobs that can still be feasibly scheduled, and
- the vector $t(u) = (t_r(u))_{r \in R_0}$ of the earliest times from which each resource r is available for performing a next job.

To simplify the consideration of the time windows, we introduce the function

$$\mathrm{eft}(j,t) = \min(\{t' \geq t \mid \exists k : [t', t' + p_j] \subseteq W_{jk}\} \cup \{T^{\max}\}), \tag{4.3}$$

that yields the *earliest feasible time*, not smaller than the provided time $t \leq T^{\max}$, at which job j can be scheduled according to the time windows W_j of the given job $j \in J$. The value $\mathrm{eft}(j,t) = T^{\max}$ indicates that job j cannot be feasibly included in the schedule at time t or later.

The state of the root node is $\sigma(\mathbf{r}) = (P(\mathbf{r}), t(\mathbf{r})) = (J, (T^{\min}, \ldots, T^{\min}))$ and represents the original instance of the problem, with no jobs scheduled or excluded yet, and the

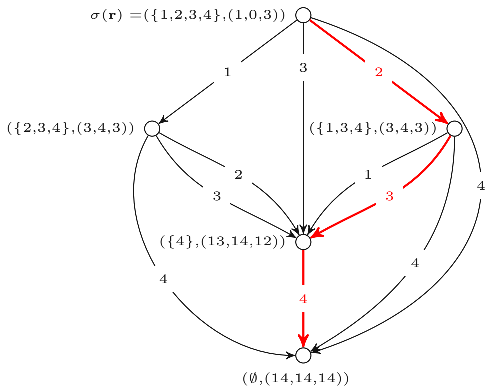


Figure 4.4: State graph for the problem instance from Figure 4.3 with $n = 4$ jobs and $m = 2$ secondary resources. Node labels denote the corresponding states $(P(\cdot), t(\cdot))$, arc labels the scheduled jobs j. The path that represents the optimal solution is highlighted. Note that for the shown state graph the strengthening of states as described in Section 4.4.3 has been applied.

target node's state is $\sigma(\mathbf{t}) = (P(\mathbf{t}), t(\mathbf{t})) = (\emptyset, (T^{\max}, \dots, T^{\max}))$. An arc $\alpha = (u, v)$ represents the transition from state $(P(u), t(u))$ to state $(P(v), t(v))$ that is achieved by scheduling job $j = \mathrm{val}(\alpha)$, $j \in P(u)$, at its earliest possible time w.r.t. vector $t(u)$. When performing this transition, the *start time* of job j w.r.t. state $(P(u), t(u))$ is

$$\mathrm{s}\left((P(u), t(u)), j\right) = \mathrm{eft}\left(j, \max(t_0 - p_j^{\mathrm{pre}}, t_{q_j})\right). \tag{4.4}$$

The transition function to obtain the successor state $(P(v), t(v))$ of state $(P(u), t(u))$ when scheduling job $j \in P(u)$ is

$$\tau\left((P(u), t(u)), j\right) = \begin{cases} (P(u) \setminus \{j\}, t(v)), & \text{if } \mathrm{s}((P(u), t(u)), j) < T^{\max}, \\ \hat{0}, & \text{else,} \end{cases} \tag{4.5}$$

with

$$t_0(v) = \mathrm{s}((P(u), t(u)), j) + p_j^{\mathrm{pre}} + p_j^0, \tag{4.6}$$

$$t_r(v) = \mathrm{s}((P(u), t(u)), j) + p_j, \qquad \text{for } r = q_j, \tag{4.7}$$

$$t_r(v) = t_r(u), \qquad \text{for } r \in R \setminus \{q_j\}, \tag{4.8}$$

and where $\hat{0}$ represents the infeasible state. If a transition results in the infeasible state, the corresponding arc and node are omitted in the MDD.

The prize associated with a state transition is the prize z_j of the scheduled job j. Thus, each path in this state graph originating in the initial state of the root node $\sigma(\mathbf{r})$ and ending in some other state than $\hat{0}$ corresponds to a feasible solution in which the jobs associated with the arcs are greedily scheduled in the order in which they appear in the path. Note that a feasible state $\sigma(u) = (P(u), t(u))$ may, in general, be reached via multiple different paths, i.e., by including different sets of jobs and/or by different orderings of these jobs. Therefore, a feasible state does, in general, not represent a unique solution. Figure 4.4 shows as example the state graph for the problem instance in Figure 4.3.

Using these definitions of states and transitions, we can express the optimal solution value of the PCJSOCMSR subproblem represented by a node u by the recursive DP formulation

$$Z^*(u) = \max\left(\{Z^*(\tau(\sigma(u), j)) + z_j \mid j \in P(u), \tau(\sigma(u), j) \neq \hat{0}\} \cup \{0\}\right) \tag{4.9}$$

and the overall PCJSOCMSR solution value is $Z^*(\mathbf{r})$.

Strengthening of States

A state can be replaced by a dominating state if it is ensured that the latter still allows for the same feasible solutions. This dominance relation is defined as follows. A state $(P'(u), t'(u))$ *dominates* a state $(P(u), t(u))$, denoted by $(P'(u), t'(u)) \rhd (P(u), t(u))$, when $P'(u) \subseteq P(u)$, $t'_r(u) \geq t_r(u)$ for all $r \in R_0$, and $(P'(u), t'(u)) \neq (P(u), t(u))$. The feasible extensions from $(P'(u), t'(u))$ towards complete solutions can then only be a subset of those from $(P(u), t(u))$.

To possibly strengthen a state $(P(u), t(u))$, let $P'(u) = \{j \in P(u) \mid \mathrm{s}((P(u), t(u)), j) \neq T^{\max}\}$ include only the jobs from $P(u)$ that can actually be scheduled. Then, set the times

$$t'_0(u) = \min_{j \in P'(u)} \left(\mathrm{s}((P(u), t(u)), j) + p^{\mathrm{pre}}_j\right), \tag{4.10}$$

$$t'_r(u) = \begin{cases} \min_{j \in P'(u) | q_j = r} \ \mathrm{s}((P(u), t(u)), j), & \text{if } \{j \in P'(u) \mid q_j = r\} \neq \emptyset, \\ T^{\max}, & \text{else,} \end{cases} \qquad r \in R, \tag{4.11}$$

such that they correspond to the earliest possible time when the corresponding resource can actually be used considering the jobs in $P'(u)$. Here, $t'_r(u)$ is set to $T^{\max}$ if no job that requires resource r remains in $P'(u)$. This strengthening also ensures that any state for which no feasible extension exists anymore is mapped to the unique target state $\mathbf{t} = (\emptyset, (T^{\max}, \ldots, T^{\max}))$.

4.4.4 Upper Bounds for the Total Prize of Remaining Jobs

As described in Section 2.2.2, A*-based algorithms use an upper bound function for a given state $(P(u), t(u))$ of node u to guide the search process. An upper bound for the cost-to-go, i.e. for the still achievable total prize for the remaining jobs in $P(u)$ can be calculated by solving the following LP relaxation of a multi-constrained 0–1 knapsack problem

$$
\begin{aligned}
Z^{\text{ub}}_{\text{MKP-LP}}(u) = \max \quad & \sum_{j \in P(u)} z_j x_j && && (4.12)\\
\text{s.t} \quad & \sum_{j \in P(u)} p^0_j x_j \le W_0(P(u), t(u)), && && (4.13)\\
& \sum_{j \in P(u) \cap J_r} p_j x_j \le W_r(P(u), t(u)), && r \in R, && (4.14)\\
& 0 \le x_j \le 1, && j \in P(u), && (4.15)
\end{aligned}
$$

where variables x_j is a continuous relaxation of a binary variable that indicates if job j is scheduled (=1) or not (=0), $j \in P(u)$. The right-hand-sides of the knapsack constraints are

$$
W_0(P,t) = \left| \bigcup_{\substack{j \in P,\\ k=1,\dots,\omega_j \mid \\ w^{\text{end}}_{jk} - p^{\text{post}}_j \ge t_0 + p^0_j}} \left[\max\left(t_0, w^{\text{start}}_{jk} + p^{\text{pre}}_j\right), w^{\text{end}}_{jk} - p^{\text{post}}_j \right] \right| \tag{4.16}
$$

and

$$
W_r(P,t) = \left| \bigcup_{\substack{j \in P \cap J_r,\\ k=1,\dots,\omega_j \mid \\ w^{\text{end}}_{jk} \ge t_r + p_j}} \left[\max\left(t_r, w^{\text{start}}_{jk}\right), w^{\text{end}}_{jk} \right] \right|, \tag{4.17}
$$

where the union of intervals is defined as $\bigcup_{i=1,\dots,k}[\alpha_i, \beta_i] = \{\gamma \in \mathbb{R} \mid \exists i : \gamma \in [\alpha_i, \beta_i]\}$, and function $|\cdot|$ denotes the sum of the lengths of the resulting disjoint continuous intervals of this union. Thus, $W_0(P,t)$ and $W_r(P,t)$ represent the total amount of still available time for resource 0 and resource r, $r \in R$, respectively, considering the current state and the time windows.

To solve this upper bound calculation problem for each state with an LP solver is computationally rather expensive, as our experiments in the next section will document. Instead, simpler upper bounds are determined by solving two types of further relaxations. The first one is obtained by relaxing inequalities (4.14).

$$
\begin{aligned}
Z^{\text{ub}}_0(u) = \max \quad & \sum_{j \in P(u)} z_j x_j && && (4.18)\\
\text{s.t} \quad & \sum_{j \in P(u)} p^0_j x_j \le W_0(P(u), t(u)) && && (4.19)\\
& 0 \le x_j \le 1 && j \in P(u) && (4.20)
\end{aligned}
$$

The second relaxation is obtained by performing a Lagrangian relaxation of inequality (4.13), where $\lambda \geq 0$ is the Lagrangian dual multiplier associated with this inequality.

$$
\begin{aligned}
h^{\mathrm{ub}}(u,\lambda) = \max \quad & \sum_{j \in P(u)} z_j x_j + \lambda \left(W_0(P(u), t(u)) - \sum_{j \in P} p_j^0 x_j \right) && && (4.21)\\
\text{s.t} \quad & \sum_{j \in P(u) \cap J_r} p_j x_j \leq W_r(P(u), t(u)), && r \in R && (4.22)\\
& 0 \leq x_j \leq 1 && j \in P(u) && (4.23)
\end{aligned}
$$

Both, $Z_0^{\mathrm{ub}}(u)$ and $h^{\mathrm{ub}}(u,\lambda)$, are computed by solving LP relaxations of simple knapsack problems. In the latter case, this is possible since the problem separates over the resources and for each resource, the resulting problem is an LP relaxation of a knapsack problem. An LP relaxation of a knapsack problem can be efficiently solved by a greedy algorithm that packs items in decreasing prize/time-ratio order; the first item that does not completely fit is packed partially so that the capacity is exploited as far as possible, see Kellerer et al. [96].

It follows from weak duality (see, e.g., [125], Prop. 6.1) that $h^{\mathrm{ub}}(u,\lambda)$ yields an upper bound on $Z_{\text{MKP-LP}}^{\mathrm{ub}}(u)$ for all $\lambda \geq 0$, but the quality of this upper bound depends on the choice of λ. We have chosen to consider $h^{\mathrm{ub}}(u,\lambda)$ for the values $\lambda = 0$ and $\lambda = z_{\bar{j}}/p_{\bar{j}}^0$, where $\bar{j}$ is the last, and typically partially, packed item in an optimal solution to the problem solved to obtain $Z_0^{\mathrm{ub}}(u)$. The value $\lambda = z_{\bar{j}}/p_{\bar{j}}^0$ is chosen since it is an optimal LP dual solution associated with inequality (4.19) and therefore has a chance to be a good estimate of a value for λ that gives a strong upper bound.

By solving the relaxations introduced above, the strongest bound on $Z_{\text{MKP-LP}}^{\mathrm{ub}}(u)$ we can obtain, and the one that we use in our A*-based construction of a relaxed MDD, is

$$
Z^{\mathrm{ub}}(u) = \min \left\{ Z_0^{\mathrm{ub}}(u), h^{\mathrm{ub}}(u,0), h^{\mathrm{ub}}(u, z_{\bar{j}}/p_{\bar{j}}^0) \right\}. \tag{4.24}
$$

In our experimental comparisons in Section 4.4.10, we will illustrate the practical strengths of the bounds

$$
\begin{aligned}
& Z_{\text{MKP-LP}}^{\mathrm{ub}}(u), && (4.25)\\
& Z_0^{\mathrm{ub}}(u), && (4.26)\\
& Z_{00}^{\mathrm{ub}}(u) = \min \left(Z_0^{\mathrm{ub}}(u), h^{\mathrm{ub}}(u,0) \right), \text{ and} && (4.27)\\
& Z_{0\bar{j}}^{\mathrm{ub}}(u) = \min \left(Z_0^{\mathrm{ub}}(u), h^{\mathrm{ub}}(u, z_{\bar{j}}/p_{\bar{j}}^0) \right), && (4.28)
\end{aligned}
$$

and study which compromise of strength and computational effort pays off the most in the context of our A* search.

4.4.5 Classical A* Search

In this section we describe a classical A* search approach for the PCJSOCMSR. Remember that a feasible state $\sigma(u)$ of node u does not represent a unique solution since this node may be reached via multiple different paths. As we want to find a solution with maximum total prize, we are primarily interested in a path from $\mathbf{r}$ to u with a maximum total prize. Let $Z^{\mathrm{lp}}(u)$ be this maximum total prize to reach a feasible state $\sigma(u)$. In order to solve the PCJSOCMSR we have to find a node u with a feasible state $\sigma(u)$ and maximum $Z^{\mathrm{lp}}(u)$. Such a state cannot have any feasible successor state, hence, either $P(u) = \emptyset$ or $\tau((P(u), t(u)), j) = \hat{0}$, $j \in P(u)$, and otherwise $Z^{\mathrm{lp}}(u)$ is not the maximum achievable prize.

A* search belongs to the class of *informed* search strategies that make use of a heuristic estimate for guidance in order to return a proven optimal solution possibly faster than a more naive uninformed search like breadth- or depth-first-search. See Chapter 2/Section 2.2.2 for more details about A* search. Within our A* search, each encountered feasible state $(P(u), t(u))$ of the state graph is evaluated by a priority function $f_*(u) = Z^{\mathrm{lp}}(u) + Z^{\mathrm{ub}}(u)$ in which $Z^{\mathrm{lp}}(u)$ corresponds to the prize of the so far best (i.e., longest) known path from $\mathbf{r}$ to node u and $Z^{\mathrm{ub}}(u)$ is the A* search's heuristic function estimating the "costs-to-go". The latter is in our case an upper bound on the still achievable prize by extending the path from state $(P(u), t(u))$ onward as described in Section 4.4.4. The value of the priority function $f_*(u)$ is thus an upper bound on the total prize that a solution may achieve by considering node u.

The proposed A* search is shown in pseudo-code in Algorithm 4.3. It maintains the set W of all so far encountered states, implemented by a hash table; for each contained state $(P(u), t(u))$, this data structure also stores the values $Z^{\mathrm{lp}}(u)$ and $Z^{\mathrm{ub}}(u)$ as well as a reference $\mathrm{pred}(u)$ to the predecessor node of a currently longest path from $\mathbf{r}$ to u, and the last scheduled job $\mathrm{j}(u)$. Furthermore, the algorithm maintains the *open list* Q, which contains all nodes queued for further expansion. It is realized by a priority queue data structure that considers the states' priority values $f_*(u)$. Last but not least, our A* search maintains a reference x_{maxlp} to the encountered node u with the so far largest $Z^{\mathrm{lp}}(u)$ value. Both W and Q, as well as x_{maxlp}, are initialized with the root node $\mathbf{r}$ at Lines 2–4 of Algorithm 4.3.

At each major iteration, a node u with maximum priority value $f_*(u)$ is taken from Q at Line 9. This node u is then *expanded*, which means that each job in $P(u)$ is considered as next job to be scheduled by calculating the respective successor state obtained by the transition $x = \tau((P(u), t(u)), j)$. If a job yields the infeasible state $\hat{0}$, it is skipped. Similarly, if the obtained state has been encountered before and the new path via node u is not longer than the previously identified path to the already existing node with state x, we skip job j. Otherwise, if a new feasible state is reached, it is added to set W and a new correspond node v is created. Since a new longest path to node v via u has been found, line 24 sets $Z^{\mathrm{lp}}(v) = Z^{\mathrm{lp}}(u) + z_j$ and stores u as predecessor of v and job j as the last scheduled job. The upper bound $Z^{\mathrm{ub}}(v)$ is then also calculated, and if there is

Algorithm 4.3: A* Algorithm for PCJSOCMSR

```
Input: Root node $\mathbf{r}$ with initial state $\sigma(\mathbf{r})$;
set of encountered states $W \leftarrow \{\mathbf{r}\},\ Z^{\mathrm{lp}}(\mathbf{r}) \leftarrow 0$;
open list $Q \leftarrow \{(\mathbf{r}, f_*(\mathbf{r}) = Z^{\mathrm{ub}}(\mathbf{r}))\}$;
node with maximum $Z^{\mathrm{lp}}$ so far $x_{\mathrm{maxlp}} \leftarrow \mathbf{r}$;
do
    if $Q = \emptyset$ then
        return opt. solution given by $x_{\mathrm{maxlp}}$ and its predecessor chain
    end
    $u \leftarrow$ pop node with maximum $f_*(u)$ from $Q$;
    if $f(u) \leq Z^{\mathrm{lp}}(x_{\mathrm{maxlp}})$ then
        return opt. solution given by $x_{\mathrm{maxlp}}$ and its predecessor chain
    end
    foreach $j \in P(u)$ do                                // expand node $u$
        state $x \leftarrow \tau((P(u), t(u)), j)$; strengthen state $x$;
        if $x = \hat{0} \vee (\exists \sigma(v) \in W : \sigma(v) = x \wedge Z^{\mathrm{lp}}(P,t) + z_j \leq Z^{\mathrm{lp}}(v))$ then
            // infeasible or existing state reached in no better way, skip
            continue
        end
        if $x \in W$ then let $v$ be the node with $\sigma(v) = x$ ;
        else                                               // new state reached
            create new node $v$ with $\sigma(v) \leftarrow x$;
            $W \leftarrow W \cup \{x\}$;
        end
        $Z^{\mathrm{lp}}(v) \leftarrow Z^{\mathrm{lp}}(u) + z_j,\ \mathrm{pred}(v) \leftarrow u,\ \mathrm{j}(v) = j$ ;
        if $Z^{\mathrm{ub}}(v) \neq 0$ then $Q \leftarrow Q \cup (v, f_*(v) = Z^{\mathrm{lp}}(v) + Z^{\mathrm{ub}}(v))$;
        if $Z^{\mathrm{lp}}(x_{\mathrm{maxlp}}) < Z^{\mathrm{lp}}(v)$ then $x_{\mathrm{maxlp}} \leftarrow v$ ;
    end
while time or memory limit not reached;
// terminate early:
$u \leftarrow$ node with maximum $f_*(u)$ from $Q$;
derive solution $\pi$ from $x_{\mathrm{maxlp}}$ following predecessors;
$\pi \leftarrow$ greedily augment $\pi$ with jobs from $P(u)$;
return heuristic solution $\pi$ and upper bound $f_*(u)$;
```

potential for further improvement, i.e., $Z^{\mathrm{ub}}(v) > 0$, node v is added to the open list Q for a possible future expansion. Finally, reference x_{maxlp} is updated if a new overall longest path is obtained.

A special aspect of our A* search therefore is that we do not have a specific target state that is known in advance, and we only add states/nodes that may yield further successor states to the open list. Lines 6 to 11 makes sure that we nevertheless recognize when a proven optimal solution has been reached: This is the case when either the open list Q becomes empty or the priority value $f_*(u)$ of Q's top element u (i.e., the maximum priority value) is not larger than the length $Z^{\mathrm{lp}}(x_{\mathrm{maxlp}})$ of the so far longest identified path. Note that the priority value of Q's top element always is a valid overall upper bound for the total achievable prize. This follows from the fact that $Z^{\mathrm{ub}}(u)$ is an *admissible heuristic* (see Section 2.2.2), i.e., it never underestimates the real prize that can still be achieved from $(P(u), t(u))$ onward. Further, an optimal solution is derived from state x_{maxlp} by following its chain of predecessor states and respectively scheduled jobs, and the corresponding solution is returned.

A particular feature of our A* search is that it can also be terminated early by providing a time or memory limit for the execution and it still yields a heuristic solution together with an upper bound on the optimal solution value in this case. This heuristic solution is derived from the so far best state x_{maxlp} by following its chain of predecessors and additionally considering all remaining jobs in $P(u)$ in their natural order (i.e., as given in the instance specification) for further addition in a greedy way. The returned upper bound is the priority value of Q's top element.

4.4.6 A Mixed Integer Programming Model

For comparison purpose we propose the following mixed integer programming (MIP) model to solve instances of the PCJSOCMSR. We use the binary variable t_j to indicate if job j, $j \in J$, is included in the schedule (=1) or not (=0) and the binary variable t_{jw} to indicate if job j is assigned to time window w (=1) or not (=0), $w = 1, \ldots, \omega_j$, $j \in J$. Let the continuous variable s_j be the start time of job j. Binary variable $y_{jj'}$ is further used to indicate if job j is scheduled before j' w.r.t. the common resource (=1) or not (=0), if both jobs are scheduled, $j, j' \in J,\ j \neq j'$.

Let

$$\delta_{jj'} = \begin{cases} p_j, & \text{if } q_j = q_{j'}, \\ p_j^{\mathrm{pre}} + p_j^0 - p_{j'}^{\mathrm{pre}}, & \text{if } q_j \neq q_{j'}, \end{cases} \tag{4.29}$$

be the minimum time between the start of job j and the start of job j' if job j is scheduled before job j', which depends on whether both jobs use the same resource or not.

A solution to PCJSOCMSR can be obtained by solving the MIP model

$$
\begin{aligned}
\max \quad & \sum_{j \in J} z_j t_j && && \text{(4.30a)} \\
\text{s.t} \quad & t_j = \sum_{w=1,\dots,\omega_j} t_{jw}, && j \in J, && \text{(4.30b)} \\
& y_{jj'} + y_{j'j} \geq t_j + t_{j'} - 1, && j, j' \in J,\ j \neq j', && \text{(4.30c)} \\
& s_{j'} \geq s_j + \delta_{jj'} - (T^{\text{dead}}_j - p_j - T^{\text{rel}}_{j'} + \delta_{jj'})(1 - y_{jj'}), && && \\
& && j, j' \in J,\ j \neq j' && \text{(4.30d)} \\
& s_j \geq T^{\text{rel}}_j + \sum_{w=1,\dots,\omega_j} \left(w^{\text{start}}_{jw} - T^{\text{rel}}_j \right) t_{jw}, && j \in J, && \text{(4.30e)} \\
& s_j \leq T^{\text{dead}}_j - p_j + \sum_{w=1,\dots,\omega_j} (w^{\text{end}}_{jw} - T^{\text{dead}}_j) t_{jw}, && j \in J, && \text{(4.30f)} \\
& t_j \in \{0, 1\}, && j \in J, && \text{(4.30g)} \\
& t_{jw} \in \{0, 1\}, && w = 1, \dots, \omega_j,\ j \in J, && \text{(4.30h)} \\
& s_j \in [T^{\text{rel}}_j, T^{\text{dead}}_j - p_j], && j \in J, && \text{(4.30i)} \\
& y_{jj'} \in \{0, 1\}, && j, j' \in J,\ j \neq j'. && \text{(4.30j)}
\end{aligned}
$$

Equations (4.30b) state that each scheduled job must be assigned to a time window and inequalities (4.30c) ensure that if two jobs j and j' are scheduled, either $y_{jj'}$ or $y_{j'j}$ must be set to one, i.e., one of them needs to precede the other. If a job is to precede another one, inequalities (4.30d) ensure this w.r.t. the jobs' start times. If a job is assigned to a time window, inequalities (4.30e) and (4.30f) make its start time comply with this time window, and otherwise the job only complies with its release time and deadline.

In Chapter 3, a MIP model with position based variables was introduced for JSOCMSR since this model showed better computational performance than a MIP model with order based variables. Such position based model does, however, not extend well to the current setting with multiple time windows since the time windows require explicit knowledge of the start time of each job, and the position based model only has explicit times for the start time of a certain position.

4.4.7 A Constraint Programming Model

We further propose the following constraint programming (CP) model for the PCJSOCMSR, which we implemented in the constraint modeling language MiniZinc[3]. The model makes use of so-called *option type* variables. Such a variable may either have a value of a certain domain assigned or set to the special value $\top$ that indicates the absence of a value. For job $j \in J$ we use the option type variable s_j for the job's start time. An absent start time, i.e., $s_j = \top$, indicates that the job is not scheduled. The CP model is

[3]https://www.minizinc.org

given by

$$\max \sum_{j\in J|\mathrm{occurs}(s_j)} z_j \tag{4.31a}$$

$$\mathrm{disjunctive\_strict}(\{(s_j + p_j^{\mathrm{pre}}, p_j^0) \mid j \in J\}), \tag{4.31b}$$

$$\mathrm{disjunctive\_strict}(\{(s_j, p_j) \mid j \in J \land q_j = r\}), \qquad r \in R, \tag{4.31c}$$

$$\mathrm{occurs}(s_j) \rightarrow \min_{\omega=1,\ldots,\omega_j} w_{j\omega}^{\mathrm{start}} \le s_j \le \max_{\omega=1,\ldots,\omega_j} (w_{j\omega}^{\mathrm{end}} - p_j), \qquad j \in J, \tag{4.31d}$$

$$s_j \in [T_j^{\mathrm{rel}}, \ldots, T_j^{\mathrm{dead}} - p_j] \cup \{\top\}, \qquad j \in J, \tag{4.31e}$$

where for job $j \in J$ the predicate occurs(s_j) yields true if the option type variable s_j is not absent, i.e., job j is scheduled. Objective function (4.31a) considers the total prize over all scheduled jobs. The strict disjunctive constraints (4.31b) and (4.31c) ensure that all scheduled jobs do not overlap w.r.t. their usage of the common resource and the secondary resource $r \in R$, respectively. Absent jobs are hereby ignored. Constraints (4.31d) state that if job $j \in J$ is scheduled, it must be performed within one of the job's time windows. The domains of variables s_j are specified in (4.31e).

4.4.8 A*-based Construction of Relaxed MDDs

In order to compile relaxed MDDs for the PCJSOCMSR with the A*C Algorithm 4.1 from Section 4.3, we have to define some problem specific parts. First the A*C algorithm uses the state graph from Section 4.4.3 and we have to define an appropriate merge operator that merges two states in such a way that no feasible solutions are removed from the relaxed MDD. Second, we need to define a labeling function that selects pairs of nodes from the open list Q for merging such that no cycles emerge and the merged node is still a strong representative for both selected nodes. Furthermore the labeling function has to ensure that the open list will get empty after a finite number of expansions. And finally we reveal some further technical details about dominant node merging and tie breaking criteria.

Merging of States

In order to compile a relaxed MDD $\mathscr{D}$ we further have to define the merging operation. Here, when two nodes $u, v \in V(\mathscr{D})$ are merged into a new single node, the merged state is

$$(P(u), t(u)) \oplus (P(v), t(v)) = \Big(P(u) \cup P(v), (\min(t_r(u), t_r(v)))_{r\in R_0}\Big) . \tag{4.32}$$

By this construction, the merged state allows all feasible extensions that both original states did. Additional extensions and originally infeasible solutions may, however, become feasible due to the merge, as is usually the case in relaxed DDs. If possible, the obtained state is further strengthened as described above.

We proof now the validity of the merge operator $\oplus$. In line with the common definition of a relaxation, Bergman at el. [14] defines a relaxed DD as follows.

Definition 4.4.1
A weighted DD is relaxed for an optimization problem $\mathcal{P}$ if

(i) the DD represents a superset of the feasible solutions to $\mathcal{P}$ and

(ii) each path that represents a feasible solution to $\mathcal{P}$ has a length that is an upper bound on the objective value of this solution.

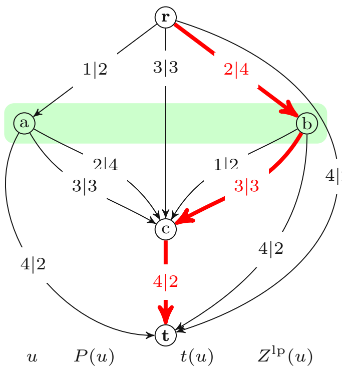
Given an exact DD formulation and a merge operator, this operator is considered valid if (repeatedly) applying it to the DD will result in a DD that is relaxed with respect to the original problem. In order to show this, it is sufficient to show that if the merge operator is applied to a DD that complies with (i) and (ii), so will the resulting DD, and the result will follow by induction. (For the initial step of the induction, we assume the operator is applied to an exact DD, which trivially complies with (i) and (ii)).

Proposition 4.4.0.1
Given a relaxed MDD constructed for the PCJSOCMSR according to Section 4.4.3 that complies with (i) and (ii) in Definition 4.4.1. When the merge operator defined in Equation (4.32) is applied to this MDD, then the resulting MDD will also comply with (i) and (ii).

Proof. A state $\sigma(u) = (P(u), t(u))$ carries the following information. The set $P(u) \subseteq J$ of jobs that can still be feasibly scheduled, and the vector $t(u) = (t_r(u))_{r \in R_0}$ of the earliest times from which each resource r is available for performing a next job. When the merge operator $\sigma(u) \oplus \sigma(v)$ is applied to the two states $\sigma(u) = (P(u), t(u))$ and $\sigma(v) = (P(v), t(v))$, the resulting state is $\left(P(u) \cup P(v), (\min(t_r(u), t_r(v)))_{r \in R_0}\right)$. For the merged state, the set of jobs that can be feasibly scheduled is a superset of both the original sets of jobs, and no feasible solutions are omitted due to the merge. As for the earliest times, since the merged state gets the component-wise earliest time from each of the original states, no feasible solution is lost. Because of this, (i) holds after the merge operation is applied. Note that after the merge operation, paths from the merged state that were not feasible with respect to neither $\sigma(u)$ nor $\sigma(v)$ might become feasible for $\sigma(u) \oplus \sigma(v)$. Condition (ii) follows because the longest path from $\sigma(u) \oplus \sigma(v)$ is selected from a superset of the paths that existed before the merge and that the cost of the arcs are the same. Moreover, note that the DD stays acyclic and therefore feasible thanks to the node selection mechanism. □

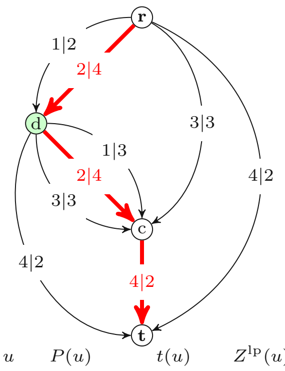
Figure 4.5 shows an example of an exact MDD and a corresponding relaxed MDD for the small PCJSOCMSR instance from Figure 4.3. The states associated with the nodes are detailed in the tables below each MDD. Arc labels indicate the scheduled job and its prize. In the exact MDD, the longest path is highlighted and it has a total length of nine. The corresponding optimal solution is given by the sequence $\pi^* = (2, 3, 4)$ and the respective schedule is depicted on the right side of the figure. A relaxed MDD is shown

Exact MDD



u	$P(u)$	$t(u)$	$Z^{\text{lp}}(u)$
r	$\{1,2,3,4\}$	$(1,0,3)$	0
a	$\{2,3,4\}$	$(3,4,3)$	2
b	$\{1,3,4\}$	$(3,4,3)$	4
c	$\{4\}$	$(9,14,8)$	7
t	$\{\}$	$(14,14,14)$	9

Relaxed MDD



u	$P(u)$	$t(u)$	$Z^{\text{lp}}(u)$
r	$\{1,2,3,4\}$	$(1,0,3)$	0
d	$\{1,2,3,4\}$	$(3,4,3)$	4
c	$\{4\}$	$(9,14,8)$	7
t	$\{\}$	$(14,14,14)$	10

Instance:

j	p_j	p_j^{pre}	p_j^0	q_j	z_j	W_j
1	4	1	2	1	2	$\{[0,8]\}$
2	4	1	2	1	4	$\{[0,8]\}$
3	4	0	3	2	3	$\{[3,8]\}$
4	5	1	3	2	2	$\{[8,14]\}$

Opt. solution π^*: $Z(\pi^*) = 9$

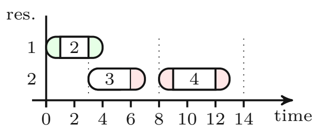


Figure 4.5: Examples of an exact MDD and a relaxed MDD for a PCJSOCMSR instance with $n = 4$ jobs and $m = 2$ secondary resources. In the relaxed MDD, the original nodes a and b have been merged.

in the middle; it has been obtained by merging nodes a and b from the exact MDD, yielding node d. The longest path of this relaxed MDD has length ten and it represents the sequence $\pi^* = (2, 2, 4)$, where job 2 is scheduled twice. It can here be easily verified that all **r**–**t** paths in the exact MDD, which correspond to all feasible solutions of this PCJSOCMSR instance, have corresponding paths in the relaxed MDD, but there exist additional paths representing infeasible solutions such as $(2, 2, 4)$.

Labeling Function for Collector Nodes

As a final major component, closely related to merging, we have to define the labeling function $L(u)$ used for indexing the collector nodes V^{c}. Remember that this function should partition the set of nodes into subsets such that nodes within a subset are similar enough to be promising to merge; thus, similar nodes should tend to get the same label.

In case of PCJSOCMSR, we use for a node u the triple $L(u) = (t_0(u), r(u), Z^{\text{ub}}(u))$ as label, where $t_0(u)$ is again the time from which on the common resource is available, $r(u)$ refers to the secondary resource of the job scheduled last in the so far longest path to node u (ties are resolved by using the resource identified first), and $Z^{\text{ub}}(u)$ is the upper bound for the cost-to-go.

Note that by this definition, we do not explicitly consider $P(u)$, the set of jobs that might still be scheduled, nor t_r, $r \in R$, the individual availability times of the secondary resources. Instead of the latter, $r(u)$ is used as a rough substitute. The upper bound $Z^{\text{ub}}(u)$ is an important additional indicator that can be seen to somehow summarize important information about the state of node u. In summary, two nodes are only merged in our A*-based construction if (a) the common resource 0 is used to the same extent, (b) the last used secondary resource is the same, and (c) the values of the problem-specific upper bounds coincide.

Note that a merged node will have the same t_0 value as the original nodes according to Equation (4.32). Since each job requires the common resource 0 for a positive time, each transition from a node to a successor node increases the corresponding t_0 value. From this follows the important property that the t_0 values strictly increase along any path in our MDD. Consequently, it holds that cycles cannot occur and that the open list gets empty in a finite number of iterations (since when the t_0 values strictly increase along any path, the set of jobs that might be scheduled will decrease due to the deadline of the jobs). Hence, the increase of t_0 in each state transition helps to guarantee that the algorithm terminates with a complete relaxed MDD.

In Section 4.4.10 we will experimentally investigate also the following simpler labeling functions: $L^1(u) = t_0(u)$, $L^2(u) = (t_0(u), r(u))$, and $L^3(u) = (t_0(u), Z^{\mathrm{ub}}(u))$.

Besides the argued theoretical convergence, it might be the case that the practical running time of the algorithm is still too large due to the not strongly limited domain size of the labels: Values $t_0(u)$ as well as $Z^{\mathrm{ub}}(u)$ may be continuous and in the worst case, exponentially many different values may emerge in the course of our algorithm, leading to a potentially exponential number of collector nodes. In our experiments in Section 4.4.10, this situation did not occur. In case that it does, discretizing these values in the labeling function by appropriate rounding can be a solution. This technique will be applied for the LCS problem in Section 4.5.

Dominated Merging

Algorithm 4.2 does not merge already expanded nodes since, in general, the operations of re-evaluating and updating the expanded sub-graphs would be too expensive. However, sometimes it is possible to merge nodes with already expanded collector nodes without further evaluations and updates. Let $v \in Q$ be a not yet expanded node and $u \in V$ be an already expanded node. If $\sigma(v) \oplus \sigma(u) = \sigma(u)$, $Z^{\mathrm{lp}}(v) \leq Z^{\mathrm{lp}}(u)$, and $t_0(v) = t_0(u)$ holds, then it is possible to merge v into u without changing the state of u and without increasing the length of the currently longest path to it. The last two conditions are important to (a) safely omit the re-expansion of node u and (b) to efficiently identify such possible merges by additionally indexing all so far encountered nodes $u \in V$ by their $t_0(u)$ values.

After each node expansion, each new or changed node in Q is considered for this type of merge by checking the condition in conjunction with all other nodes in V that have the same t_0 value. If a pair of nodes u and v that fulfills this condition is found, we remove v from the open list and merge v into u by redirecting all incoming arcs from v to u. Since this kind of merge does not introduce any relaxation loss, we perform this procedure after every node expansion even if $|Q| \leq \phi$.

Tie Breaking in the Priority Function

The nodes in the open list Q are sorted according to the value of the priority function f_*, given in Equation (4.1). It is not uncommon that different nodes have the same f_*-value, and we therefore use the following two-stage tie breaking in order to further guide the algorithm in a promising way. First, if two nodes have the same f_*-value, we always prefer *exact* nodes over *non-exact* nodes. We call a node exact when it has a longest path from the root node that does not contain any merged node where the merging induced a relaxation loss. In other words, an exact node is guaranteed to have a feasible solution that corresponds to this longest path. Such nodes are considered more promising to expand than non-exact nodes with the same f_*-value. In case of a remaining tie, we prefer nodes where the corresponding state has fewer jobs that may still be scheduled, i.e., we prefer nodes u with smaller $|P(u)|$.

4.4.9 Construction of Restricted MDDs based on Relaxed MDDs

A restricted MDD represents only a subset of all feasible solutions. It is primarily used to obtain feasible solutions and corresponding lower bounds. The construction usually follows a layer-by-layer top-down approach [15, 16], see also Chapter 2.1/Section 2.5.2. As for relaxed MDDs, the size of a restricted MDD is typically limited by imposing a maximum width β for each layer. Whenever the allowed width is exceeded, nodes are selected from the current layer according to a greedy criterion and removed together with their incoming arcs.

So far, most previous approaches in the literature construct restricted DDs independently of relaxed DDs. However, an earlier construction of a relaxed DD will, in general, have already collected substantial information. We propose to exploit this information in a successive construction of a restricted DD. The goal is to speed up the construction of the restricted DD and/or to obtain a stronger restricted DD representing better solutions. Note that some recent works derive primal solutions directly from relaxed DDs without compiling a successive restricted DD, e.g., by using exact states of the relaxed DDs as done by Tjandraatmadja and van Hoeve [153] or by applying heuristic search methods on the relaxed DD as done in Chapter 6.

Throughout this section, we denote all elements of restricted MDDs with primed symbols, while corresponding symbols of relaxed MDDs are not primed. Our approach applies the common top-down compilation principle. Each node $u' \in V(\mathscr{D}')$ in the restricted MDD $\mathscr{D}'$ always has a *corresponding node* $u \in V(\mathscr{D})$ in the relaxed MDD $\mathscr{D}$ in the sense that a path from $\mathbf{r}'$ to u' represents a feasible partial solution that is also represented in $\mathscr{D}$ by a path from $\mathbf{r}$ to node u. In other words, the node $u' \in V(\mathscr{D}')$ that corresponds to a node $u \in V(\mathscr{D})$ is the node that can be reached by the same sequence of scheduled jobs. For each newly created node in the restricted MDD, we keep track of its corresponding node in the relaxed MDD.

When expanding node u', this corresponding node u will allow us to skip certain transitions in the restricted MDD without evaluating them, i.e., we avoid introducing the

Algorithm 4.4: Construction of a restricted MDD based on a relaxed MDD

```
Input: relaxed MDD $\mathscr{D} = (V, A)$, maximum width $\beta$
Output: restricted MDD $\mathscr{D}' = (V', A')$
1  $V_1(\mathscr{D}') \leftarrow \{\mathbf{r}'\}$; $A(\mathscr{D}') \leftarrow \emptyset$;
2  $l \leftarrow 1$;
3  while $V_l(\mathscr{D}') \neq \emptyset$ do
4      $V_{l+1}(\mathscr{D}') \leftarrow \{\}$;
5      foreach node $u' \in V_l(\mathscr{D}')$ do
6          let $u \in V(\mathscr{D})$ be the node corresponding to $u'$ w.r.t. the path from $\mathbf{r}$;
7          foreach outgoing arc $\alpha = (u, v)$ of node $u$ do
8              if $\tau(\sigma(u'), \mathrm{val}(\alpha)) = \hat{0}$ then
9                  continue with next arc;
10             end
11             if $|V_{l+1}(\mathscr{D}')| = \beta \wedge$ node $v$ would be removed from $V_{l+1}(\mathscr{D}') \cup \{v\}$
                 then
12                 continue with next arc;
13             end
14             $\Sigma \leftarrow \tau(\sigma(u'), \mathrm{val}(a))$; strengthen $\Sigma$;
15             if $\nexists v' \in V_{l+1}(\mathscr{D}') \mid \sigma(v') = \Sigma$ then
16                 add new node $v'$ to $V_{l+1}(\mathscr{D}')$ and set $\sigma(v') = \Sigma$;
17             end
18             add new arc $\alpha' = (u', v')$ to $A(\mathscr{D}')$;
19             if $|V_{l+1}(\mathscr{D}')| > \beta$ then
20                 select and remove a node from $V_{l+1}(\mathscr{D}')$ with its incoming arcs
                     according to a greedy criterion;
21             end
22         end
23     end
24     $l \leftarrow l + 1$;
25 end
26 return $\mathscr{D}' = (V', A')$ with $V(\mathscr{D}') = V_1(\mathscr{D}') \cup \ldots \cup V_{l-1}(\mathscr{D}')$;
```

corresponding arcs and successor nodes. In this way, a vast amount of arcs and nodes for states that cannot lead to an optimal solution may be omitted.

Algorithm 4.4 shows this compilation of a restricted MDD $\mathscr{D}'$ that utilizes the relaxed MDD $\mathscr{D}$. We start with the first layer that consists of the root node $\mathbf{r}'$. Then, each successive layer $V_{l+1}(\mathscr{D}')$ is built from the preceding layer $V_l(\mathscr{D}')$ by creating nodes and arcs for feasible transitions from the states associated with the nodes in $V_l(\mathscr{D}')$.

Here comes the first novel aspect: For each node u' in layer $V_l(\mathscr{D}')$ we consider only state transitions corresponding to outgoing arcs of the respective node u in the relaxed MDD. Other potentially feasible state transitions do not need to be considered since we know from the relaxed MDD that they cannot lead to an optimal feasible solution. Note, however, that the relaxed node u might have outgoing arcs representing transitions that are actually infeasible for node u' in the restricted MDD. This may happen since the states of u' and u do not need to be the same but u' may dominate u due to merged nodes on the path from $\mathbf{r}$ to u in the relaxed MDD. In Line 8, our algorithm therefore checks the feasibility of the respective transition (remember that $\hat{0}$ represents the infeasible state) and skips infeasible ones. For PCJSOCMSR, this feasibility check simply corresponds to testing if $\mathrm{val}(a) \in P(u)$.

When we have reached the maximum allowed width at the current layer, we can make an efficient pre-check if the node v' that would be created next would be removed later when the set $V_{l+1}(\mathscr{D}')$ is greedily reduced to β nodes. To this end, we evaluate the criterion that is used to decide which nodes are removed from the current layer for the corresponding node v in the relaxed MDD in conjunction with the so far obtained set $V_{l+1}(\mathscr{D}')$. If this criterion is chosen in a sensible way, the evaluation for v will never indicate a removal of node v when v' would not be removed, since either the associated states are identical or the state of v' dominates the state of v. In our algorithm, Line 11 realizes this pre-check and correspondingly skips the respective transitions.

For the remaining transitions, Line 14 calculates the obtained new state Σ and creates the corresponding node v' if no node in $V(\mathscr{D}')$ exists yet for Σ. Then, a new arc (u', v') representing the transition in the restricted MDD is added to $A(\mathscr{D}')$. Finally, if $V_{l+1}(\mathscr{D}')$ has grown to more than β nodes, a node is removed according to the used greedy criterion.

A typical way to select the nodes for removal at each layer is to take the nodes with the smallest lengths of their longest paths from the root node $\mathbf{r}'$, i.e., the nodes with the smallest $Z^{\mathrm{lp}}(v')$, $v' \in V_{l+1}(\mathscr{D}')$ [16, 15]. As already observed by Maschler and Raidl [116, 117] this strategy is not beneficial for PCJSOCMSR since it disregards the advances in the time line. Instead, we remove nodes with the smallest $Z^{\mathrm{lp}}(v')/t_0(v')$ ratios in our implementation for the PCJSOCMSR. When applying this removal criterion to the corresponding node v of the relaxed MDD in Line 11, it holds that $Z^{\mathrm{lp}}(v)/t_0(v) \geq Z^{\mathrm{lp}}(v')/t_0(v')$ as $Z^{\mathrm{lp}}(v) \geq Z^{\mathrm{lp}}(v')$ and $t_0(v) \leq t_0(v')$ since state $\sigma(v')$ is equal to or dominates state $\sigma(v)$. We can even sharpen this estimation by using $(Z^{\mathrm{lp}}(u')+z(\alpha))/t_0(v)$ and thus take advantage of our knowledge of $Z^{\mathrm{lp}}(u')$ and the respective transition cost $z(\alpha)$ to reach node v.

The benefits of exploiting the relaxed MDD in the compilation of the restricted MDD depends on how closely the exact states in the restricted MDD are approximated by the corresponding states in the relaxed MDD as well as the size of the solution space encoded in the relaxed MDD. Various filtering techniques, as for example described by Cire and van Hoeve [40] for sequencing problems, can substantially reduce relaxed MDDs, and consequently, their application to the relaxed MDD before its exploitation in the construction of the restricted MDD may be advantageous.

4.4.10 Experimental Results

The classical A* search, the A*C for compiling a relaxed MDD, as well as the approach from the last section to further derive a restricted MDD were implemented in C++ using GNU g++ 5.4.1. All tests were performed on a cluster of machines with Intel Xeon E5-2640 v4 processors with 2.40 GHz in single-threaded mode with a memory limit of 16 GB per run.

We created two non-trivial sets of benchmark instances with up to 500 jobs for the experimental evaluation. The first set is based on characteristics from the particle therapy application scenario and denoted here as set P, whereas the second instance set is based on the avionic system scheduling scenario and called set A. All instances are available at `https://www.ac.tuwien.ac.at/research/problem-instances` and are described in the upcoming subsection.

This section is structured such that first, we will describe our used benchmark instances in more detail. Then we will study the impact of different problem specific upper bound functions from Section 4.4.4 on the classical A* search from Section 4.4.5. The next experiments are dedicated to exact methods such that we compare the pure A* algorithm, the MIP model from Section 4.4.6, and the CP model from Section 4.4.7 for small and middle-sized instances.

The next part of our experimental evaluation is dedicated to solve large instances of the PCJSOCMSR heuristically by compiling first a relaxed MDD with A*C and then use the structural information of it to compile a restricted MDD with TDC as explained in Section 4.4.9. For this purpose we first present results from studying the impact of different values for the open list size limit ϕ and of different choices for the labeling function $L(u)$ in the compilation of relaxed MDDs with A*C. Thereafter, we compare the quality of upper bounds obtained from relaxed MDDs compiled with A*C to those from other approaches. The last experiments finally compares primal bounds obtained from the derived restricted MDDs to those from other heuristic and exact approaches for PCJSOCMSR.

Benchmark Instances

We created two non-trivial benchmark instance sets in order to test our solution approaches. The first instance set, called P, is inspired from the particle therapy patient scheduling application, while the second set, called A, exhibits characteristics from the avionic system application. Each set consists of 30 instances for each combination of $n \in \{10, 20, 30, 40, 50, 60, 70, 80, 90\}$ jobs for small and middle-sized instances and $n \in \{50, 100, 150, 200, 250, 300, 350, 400, 450, 500\}$ jobs for large instances as well as $m \in \{2, 3\}$ secondary resources for the particle therapy based scenario and $m \in \{3, 4\}$ secondary resources for the avionic system based scenario.

Particle therapy instances (P) For these instances, the following values are sampled for each job $j \in J$: (a) the secondary resource q_j from the discrete uniform distribution

$\mathcal{U}\{1, m\}$, (b) the pre-processing times p_j^{pre} and the post-processing times p_j^{post} from $\mathcal{U}\{0, 8\}$, (c) the times p_j^0 from $\mathcal{U}\{1, 8\}$, and (d) the prize z_j from $\mathcal{U}\{p_j^0, 2p_j^0\}$ (such that this prize correlates to the usage of the common resource of job j). Time windows are chosen such that, on average, roughly 30 % of the jobs can be scheduled. For this purpose let the time horizon be $T = \lfloor 0.3\, n\, \mathrm{E}(\mathrm{p}^0) \rfloor$, where $\mathrm{E}(\mathrm{p}^0)$ is the expected value of the distribution for p_j^0. In the first step, the number of time windows ω_j of job j is sampled from $\mathcal{U}\{1, 3\}$, i.e., a job can have up to three time windows. Second, for time window k, $k = 1, \ldots, \omega_j$, its start time w_{jk}^{start} is sampled from $\mathcal{U}\{0, T - p_j\}$ and its end time from w_{jk}^{end} from $w_{jk}^{\text{start}} + \max\{p_j, \mathcal{U}\{\lfloor 0.1\, T/\omega_j \rfloor, \lfloor 0.4\, T/\omega_j \rfloor\}\}$. Overlapping time windows are merged.

Note that we create here similar to Chapter 3 “balanced particle therapy instances”. In [84] we also considered a third set of “skewed particle therapy instances”, in which the usage of the secondary resources is not uniform. However, especially for large instances the differences between the balanced and skewed instances turned out to be less interesting, and we therefore do not consider the skewed instances here.

Avionic instances (A) For the avionic instance set A, a fixed time horizon of $T = 1000$ is considered, and there are 20% communication jobs, 40% partition jobs, and 40% regular jobs. The time p_j^0 is for partition jobs and regular jobs sampled from the discrete uniform distribution $\mathcal{U}\{36, 44\}$ and for communication jobs $p_j^0 = 40$. Each partition job is assigned a secondary resource and each secondary resource has the same probability to be selected. For partition jobs, the total processing time p_j is sampled from $\mathcal{U}\{5p_j^0, 8p_j^0\}$ and then, with equal probability, p_j^{pre} or p_j^{post} is set to 0 and the respective other value is set to $p_j - p_j^0$. Since the communication jobs and regular jobs do not use a secondary resource in the real scenario, an artificial secondary resource is introduced and assigned to all of these jobs, and $p_j = p_j^0$. The prize z_j is for five of the partition jobs and ten of the communication jobs set to the high value 70 to give these jobs a higher priority, while for the remaining partition jobs and communication jobs the prize is sampled from $\mathcal{U}\{10, 50\}$. For all regular jobs, the prize is sampled from $\mathcal{U}\{10, 25\}$. For partition jobs and regular jobs, the number of time windows and the length of the time windows are computed as in the particle therapy case, but for the communication jobs the structure is different. The communication jobs can only be scheduled at certain points in time when the communication can be performed; these time points are $0, 80, 160, \ldots, 880$. Each time window of a communication job corresponds to one such time point and a job's total set of time windows corresponds to a number of consecutive such time points. The number of time windows for a communication job is obtained by sampling a value from the uniform distribution $\mathcal{U}\{1, 3\}$ and multiplying it by three.

Comparison of Problem Specific Upper Bound Functions

We start by experimentally evaluating the impact of the individual components of our combined upper bound function $Z_*^{\text{ub}}(u) = \min(Z_0^{\text{ub}}(u), h^{\text{ub}}(u, 0), h^{\text{ub}}(u, z_{\bar{j}}/p_{\bar{j}}^0))$ from Section 4.4.4. To this end, we performed the A* search on small- and middle-sized

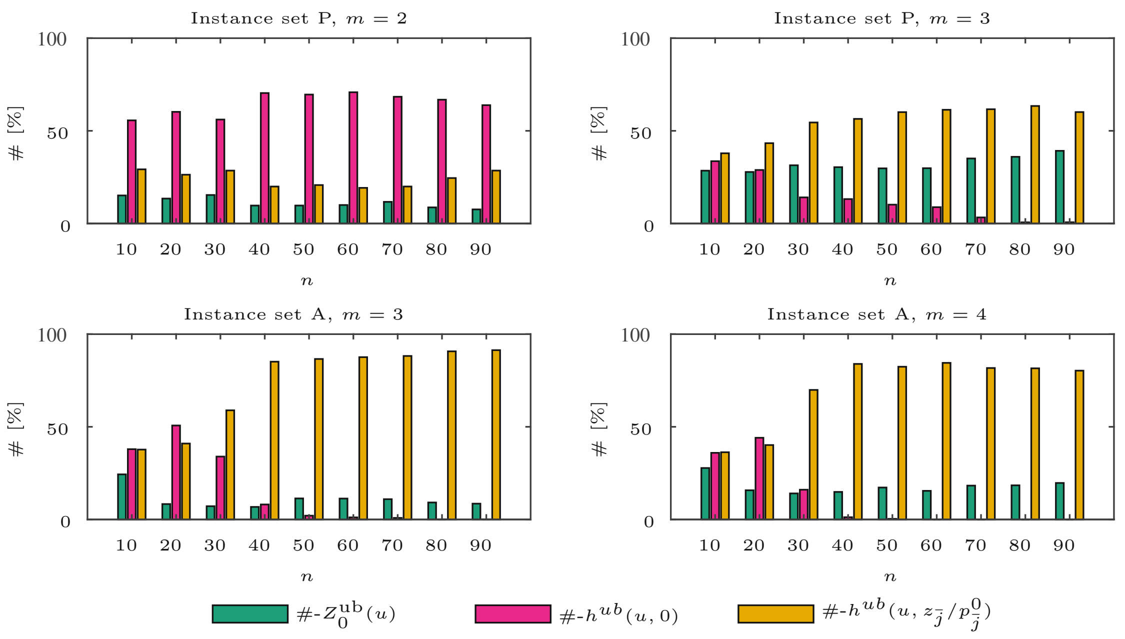


Figure 4.6: Success rates of upper bound subfunctions $Z_0^{\text{ub}}(u)$, $h^{\text{ub}}(u,0)$ and $h^{\text{ub}}(u,z_{\bar{j}}/p_{\bar{j}}^0)$ to yield the smallest value, i.e., to determine $Z_*^{\text{ub}}(u)$.

benchmark instances with $n \in \{10,20,30,40,50,60,70,80,90\}$ jobs, using $Z_*^{\text{ub}}(u)$ to evaluate all states and count for the sub-functions $Z_0^{\text{ub}}(u)$, $h^{\text{ub}}(u,0)$, and $h^{\text{ub}}(u,z_{\bar{j}}/p_{\bar{j}}^0)$ how often each one of them yields the minimum, i.e., determines $Z_*^{\text{ub}}(u)$. Figure 4.6 shows the obtained average success rates grouped according to the instance type, the number of jobs n, and the number of secondary resources m for all three upper bounds.

Most importantly we can see that in most cases not a single sub-function is dominating, i.e., it makes sense to calculate all three functions and to combine their results by taking the minimum in order to get a generally tighter bound. More specifically, the success of each sub-function obviously also depends on the specific characteristics of the problem instances. For instances of type P with two secondary resources, $h^{\text{ub}}(u,0)$ is for each instance class on average more than 50% of the times the strongest upper bound. In all other cases $h^{\text{ub}}(u,z_{\bar{j}}/p_{\bar{j}}^0)$ is on average most successful.

The strongest upper bound function, however, does not necessarily yield the best performing A* search, since the time for calculating the bound also plays a major role. As already stated in Section 4.4.4, we consider the upper bound functions $Z_{\text{MKP-LP}}^{\text{ub}}(u)$, $Z_0^{\text{ub}}(u)$, $Z_{00}^{\text{ub}}(u)$, $Z_{0\bar{j}}^{\text{ub}}(u)$, and $Z_*^{\text{ub}}(u)$. The former is solved by the CPLEX 12.7 LP solver in single threaded mode whereas the other upper bound functions make use of the sub-functions $h^{\text{ub}}(u,\lambda)$, $\lambda \geq 0$ and $Z_0^{\text{ub}}(u)$ in different ways as stated in Equations (4.24)–(4.28).

Table 4.1 presents the aggregated results for each combination of instance type, numbers of jobs, and secondary resources for our A* search using these different upper bound calculations. Columns %-opt show the percentage of instances which could be solved to proven optimality. Columns $\overline{Z^{\mathrm{ub}}}$ state the average final upper bounds and columns $\overline{\%\text{-gap}}$ list the average optimality gaps which are calculated by $100\% \cdot (Z^{\mathrm{ub}} - Z(\pi))/Z^{\mathrm{ub}}$, where π is the final solution and Z^{ub} the final upper bound. Columns t[s] list the median computation times in seconds, whereas columns $\overline{|W|}$ state the average number of encountered states during the A* search. The best values are printed bold.

In almost all cases, A* search with the combined bound $Z^{\mathrm{ub}}_{*}(u)$ provides the tightest final bounds. There are only two exceptions where $Z^{\mathrm{ub}}_{00}(u)$ or $Z^{\mathrm{ub}}_{0\bar{j}}(u)$ yield tighter bounds on average, but $Z^{\mathrm{ub}}_{*}(u)$ is not far behind. Using the original LP relaxation $Z^{\mathrm{ub}}_{\mathrm{MKP-LP}}(u)$ yields in almost all cases where not all instances could be solved to optimality worse final upper bounds than using $Z^{\mathrm{ub}}_{0}(u)$, $Z^{\mathrm{ub}}_{00}(u)$, $Z^{\mathrm{ub}}_{0\bar{j}}(u)$ or $Z^{\mathrm{ub}}_{*}(u)$. The reason for this is that, although the full LP relaxation $Z^{\mathrm{ub}}_{\mathrm{MKP-LP}}(u)$ may provide the tightest upper bound for a single state, substantially fewer nodes could be processed due to the higher computational effort to solve each LP, cf. columns $|W|$.

When considering instance set P, in cases where not all instances could be solved to optimality, the A* search with $Z^{\mathrm{ub}}_{0}(u)$ was able to provide the smallest average optimality gaps in most cases. For instances of set A the smallest average optimality gaps could be obtained from the A* search with $Z^{\mathrm{ub}}_{0\bar{j}}(u)$ or $Z^{\mathrm{ub}}_{*}(u)$. This observation is in accordance with our previous observation concerning Fig. 4.6, where $Z^{\mathrm{ub}}_{0\bar{j}}(u)$ provides frequently more often the strongest upper bounds for instances of type A. We conclude that $Z^{\mathrm{ub}}_{0}(u)$ might be a slightly better guidance for instances in set P for our simple greedy heuristic used to find solutions when terminating early.

Considering only instance classes where all instances could be solved to optimality, the A* algorithm with upper bound function $Z^{\mathrm{ub}}_{\mathrm{MKP-LP}}(u)$ encounters fewer states than A* with $Z^{\mathrm{ub}}_{*}(u)$ which in turn encounters fewer states than A* with one of the other functions $Z^{\mathrm{ub}}_{0}(u)$, $Z^{\mathrm{ub}}_{00}(u)$, or $Z^{\mathrm{ub}}_{0\bar{j}}(u)$, respectively. This is not surprising since function $Z^{\mathrm{ub}}_{\mathrm{MKP-LP}}(u)$ solves the full LP relaxation which provides frequently the strongest upper bounds and $Z^{\mathrm{ub}}_{*}(u)$ dominates the other functions $Z^{\mathrm{ub}}_{0}(u)$, $Z^{\mathrm{ub}}_{00}(u)$, and $Z^{\mathrm{ub}}_{0\bar{j}}(u)$. However, again we see that providing the strongest upper bounds cannot outweigh the disadvantage of the longer computation times such that A* with $Z^{\mathrm{ub}}_{*}(u)$ terminates in almost all cases substantially earlier than A* with $Z^{\mathrm{ub}}_{\mathrm{MKP-LP}}(u)$.

Last but not least, we point out that the memory limit of 16GB was the termination reason in several runs for the largest instances. Thus, memory consumption plays a significant role in our A* algorithm. One way to save memory would be to adopt the technique applied in Chapter 3 where states with the same $P(u)$ are stored in an aggregated fashion. This can be done by storing $P(u)$ only once and include the individual vectors $t(u)$ and further information in an attached list of so-called non-dominated time records.

Table 4.1: Average results of A* search for different upper bound functions.

			$Z^{\text{ub}}_{\text{MKP-LP}}(u)$					$Z^{\text{ub}}_{0}(u)$					$Z^{\text{ub}}_{00}(u)$					$Z^{\text{ub}}_{0\bar{j}}(u)$					$Z^{\text{ub}}_{*}(u)$				
type	n	m	%-opt	$\overline{Z^{\text{ub}}}$	$\overline{\%\text{-gap}}$	t[s]	$\overline{\lvert W\rvert}$	%-opt	$\overline{Z^{\text{ub}}}$	$\overline{\%\text{-gap}}$	t[s]	$\overline{\lvert W\rvert}$	%-opt	$\overline{Z^{\text{ub}}}$	$\overline{\%\text{-gap}}$	t[s]	$\overline{\lvert W\rvert}$	%-opt	$\overline{Z^{\text{ub}}}$	$\overline{\%\text{-gap}}$	t[s]	$\overline{\lvert W\rvert}$	%-opt	$\overline{Z^{\text{ub}}}$	$\overline{\%\text{-gap}}$	t[s]	$\overline{\lvert W\rvert}$
P	10	2	**100**	**30.93**	**0.00**	<0.1	$2.18\cdot10^{1}$	**100**	**30.93**	**0.00**	<0.1	$2.52\cdot10^{1}$	**100**	**30.93**	**0.00**	<0.1	$2.19\cdot10^{1}$	**100**	**30.93**	**0.00**	<0.1	$2.36\cdot10^{1}$	**100**	**30.93**	**0.00**	<0.1	$2.19\cdot10^{1}$
P	20	2	**100**	**50.37**	**0.00**	0.1	$2.06\cdot10^{2}$	**100**	**50.37**	**0.00**	<0.1	$2.66\cdot10^{2}$	**100**	**50.37**	**0.00**	<0.1	$2.07\cdot10^{2}$	**100**	**50.37**	**0.00**	<0.1	$2.55\cdot10^{2}$	**100**	**50.37**	**0.00**	<0.1	$2.06\cdot10^{2}$
P	30	2	**100**	**75.33**	**0.00**	0.6	$1.33\cdot10^{3}$	**100**	**75.33**	**0.00**	<0.1	$1.65\cdot10^{3}$	**100**	**75.33**	**0.00**	<0.1	$1.36\cdot10^{3}$	**100**	**75.33**	**0.00**	<0.1	$1.61\cdot10^{3}$	**100**	**75.33**	**0.00**	<0.1	$1.36\cdot10^{3}$
P	40	2	**100**	**98.93**	**0.00**	5.4	$1.17\cdot10^{4}$	**100**	**98.93**	**0.00**	0.3	$1.69\cdot10^{4}$	**100**	**98.93**	**0.00**	0.2	$1.22\cdot10^{4}$	**100**	**98.93**	**0.00**	0.3	$1.62\cdot10^{4}$	**100**	**98.93**	**0.00**	0.2	$1.20\cdot10^{4}$
P	50	2	**100**	**123.27**	**0.00**	55.2	$1.61\cdot10^{5}$	**100**	**123.27**	**0.00**	5.3	$2.21\cdot10^{5}$	**100**	**123.27**	**0.00**	3.1	$1.69\cdot10^{5}$	**100**	**123.27**	**0.00**	5.2	$2.13\cdot10^{5}$	**100**	**123.27**	**0.00**	3.9	$1.67\cdot10^{5}$
P	60	2	60	149.57	4.51	638.5	$8.69\cdot10^{5}$	97	146.97	0.11	89.5	$2.69\cdot10^{6}$	**100**	**146.80**	**0.00**	51.4	$1.76\cdot10^{6}$	97	146.93	0.22	84.5	$2.58\cdot10^{6}$	**100**	**146.80**	**0.00**	53.7	$1.75\cdot10^{6}$
P	70	2	3	184.90	14.40	900.0	$1.43\cdot10^{6}$	40	182.10	8.39	900.0	$1.18\cdot10^{7}$	**60**	**175.77**	**4.19**	623.1	$9.69\cdot10^{6}$	37	182.10	9.16	900.0	$1.11\cdot10^{7}$	56	175.93	4.41	712.0	$9.57\cdot10^{6}$
P	80	2	0	218.40	22.92	900.0	$1.21\cdot10^{6}$	0	219.00	16.07	900.0	$1.53\cdot10^{7}$	**3**	211.43	12.81	900.0	$1.40\cdot10^{7}$	0	218.10	16.02	900.0	$1.42\cdot10^{7}$	**3**	**211.23**	**12.71**	900.0	$1.38\cdot10^{7}$
P	90	2	**0**	256.07	29.48	900.0	$1.15\cdot10^{6}$	**0**	259.73	22.81	900.0	$1.43\cdot10^{7}$	**0**	251.63	**20.61**	900.0	$1.39\cdot10^{7}$	**0**	258.53	23.38	900.0	$1.36\cdot10^{7}$	**0**	**251.47**	20.69	900.0	$1.33\cdot10^{7}$
P	10	3	**100**	**36.17**	**0.00**	<0.1	$2.94\cdot10^{1}$	**100**	**36.17**	**0.00**	<0.1	$3.11\cdot10^{1}$	**100**	**36.17**	**0.00**	<0.1	$2.94\cdot10^{1}$	**100**	**36.17**	**0.00**	<0.1	$3.10\cdot10^{1}$	**100**	**36.17**	**0.00**	<0.1	$2.94\cdot10^{1}$
P	20	3	**100**	**59.27**	**0.00**	0.1	$2.76\cdot10^{2}$	**100**	**59.27**	**0.00**	<0.1	$3.03\cdot10^{2}$	**100**	**59.27**	**0.00**	<0.1	$2.86\cdot10^{2}$	**100**	**59.27**	**0.00**	<0.1	$2.92\cdot10^{2}$	**100**	**59.27**	**0.00**	<0.1	$2.80\cdot10^{2}$
P	30	3	**100**	**86.30**	**0.00**	1.2	$3.10\cdot10^{3}$	**100**	**86.30**	**0.00**	<0.1	$3.65\cdot10^{3}$	**100**	**86.30**	**0.00**	<0.1	$3.43\cdot10^{3}$	**100**	**86.30**	**0.00**	<0.1	$3.36\cdot10^{3}$	**100**	**86.30**	**0.00**	<0.1	$3.24\cdot10^{3}$
P	40	3	**100**	**112.00**	**0.00**	9.0	$3.37\cdot10^{4}$	**100**	**112.00**	**0.00**	0.4	$3.76\cdot10^{4}$	**100**	**112.00**	**0.00**	0.4	$3.61\cdot10^{4}$	**100**	**112.00**	**0.00**	0.4	$3.62\cdot10^{4}$	**100**	**112.00**	**0.00**	0.5	$3.50\cdot10^{4}$
P	50	3	93	140.90	0.45	143.7	$3.56\cdot10^{5}$	**100**	**140.33**	**0.00**	10.3	$5.09\cdot10^{5}$	**100**	**140.33**	**0.00**	11.6	$5.06\cdot10^{5}$	**100**	**140.33**	**0.00**	9.4	$4.96\cdot10^{5}$	**100**	**140.33**	**0.00**	10.0	$4.85\cdot10^{5}$
P	60	3	40	170.80	6.84	900.0	$1.13\cdot10^{6}$	**90**	166.07	**0.68**	132.3	$4.30\cdot10^{6}$	**90**	166.07	1.06	114.1	$4.19\cdot10^{6}$	**90**	166.03	1.01	126.7	$3.98\cdot10^{6}$	87	**166.00**	1.18	138.4	$3.95\cdot10^{6}$
P	70	3	7	208.73	15.67	900.0	$1.30\cdot10^{6}$	**40**	203.20	**7.10**	900.0	$1.24\cdot10^{7}$	37	202.90	8.84	900.0	$1.24\cdot10^{7}$	37	**202.60**	7.64	900.0	$1.25\cdot10^{7}$	**40**	**202.60**	8.41	871.8	$1.22\cdot10^{7}$
P	80	3	0	252.43	25.45	900.0	$1.22\cdot10^{6}$	**10**	247.87	**14.78**	900.0	$1.57\cdot10^{7}$	**10**	247.87	16.42	900.0	$1.54\cdot10^{7}$	**10**	247.43	16.19	900.0	$1.49\cdot10^{7}$	**10**	**247.40**	16.67	900.0	$1.44\cdot10^{7}$
P	90	3	**0**	285.43	28.03	900.0	$1.36\cdot10^{6}$	**0**	282.80	**19.00**	900.0	$1.65\cdot10^{7}$	**0**	282.73	19.60	869.4	$1.68\cdot10^{7}$	**0**	**282.50**	19.31	900.0	$1.62\cdot10^{7}$	**0**	282.60	20.12	900.0	$1.56\cdot10^{7}$
A	10	3	**100**	**422.13**	**0.00**	0.1	$2.08\cdot10^{2}$	**100**	**422.13**	**0.00**	<0.1	$2.42\cdot10^{2}$	**100**	**422.13**	**0.00**	<0.1	$2.08\cdot10^{2}$	**100**	**422.13**	**0.00**	<0.1	$2.08\cdot10^{2}$	**100**	**422.13**	**0.00**	<0.1	$2.08\cdot10^{2}$
A	20	3	**100**	**707.27**	**0.00**	5.5	$1.19\cdot10^{4}$	**100**	**707.27**	**0.00**	0.1	$1.96\cdot10^{4}$	**100**	**707.27**	**0.00**	0.1	$1.25\cdot10^{4}$	**100**	**707.27**	**0.00**	0.1	$1.28\cdot10^{4}$	**100**	**707.27**	**0.00**	0.1	$1.20\cdot10^{4}$
A	30	3	97	904.27	0.03	82.6	$3.05\cdot10^{5}$	**100**	**903.97**	**0.00**	4.1	$6.87\cdot10^{5}$	**100**	**903.97**	**0.00**	2.3	$4.40\cdot10^{5}$	**100**	**903.97**	**0.00**	2.7	$3.82\cdot10^{5}$	**100**	**903.97**	**0.00**	2.0	$3.53\cdot10^{5}$
A	40	3	43	1051.43	7.18	900.0	$1.10\cdot10^{6}$	83	1033.93	1.55	71.9	$8.10\cdot10^{6}$	90	1031.47	1.10	55.5	$7.49\cdot10^{6}$	**97**	**1030.00**	**0.36**	39.1	$5.48\cdot10^{6}$	**97**	**1030.00**	**0.36**	40.0	$5.42\cdot10^{6}$
A	50	3	0	1216.90	20.69	900.0	$1.45\cdot10^{6}$	27	1197.27	10.92	366.5	$2.04\cdot10^{7}$	27	1192.77	10.64	382.8	$2.03\cdot10^{7}$	**33**	1177.87	**9.89**	377.2	$1.92\cdot10^{7}$	**33**	**1177.67**	9.96	377.2	$1.92\cdot10^{7}$
A	60	3	0	1255.03	24.28	900.0	$1.44\cdot10^{6}$	**3**	1240.90	18.49	368.1	$1.96\cdot10^{7}$	**3**	1239.60	18.41	385.4	$1.96\cdot10^{7}$	**3**	1225.67	17.34	400.4	$1.96\cdot10^{7}$	**3**	**1225.57**	**17.28**	416.6	$1.96\cdot10^{7}$
A	70	3	**0**	1290.57	27.91	900.0	$1.33\cdot10^{6}$	**0**	1286.33	21.74	391.2	$1.89\cdot10^{7}$	**0**	1285.23	21.70	422.2	$1.89\cdot10^{7}$	**0**	1270.63	**20.41**	448.3	$1.89\cdot10^{7}$	**0**	**1270.53**	20.44	442.8	$1.89\cdot10^{7}$
A	80	3	**0**	1319.33	31.93	900.0	$1.31\cdot10^{6}$	**0**	1329.10	26.30	364.7	$1.80\cdot10^{7}$	**0**	1328.67	25.94	377.4	$1.80\cdot10^{7}$	**0**	**1311.53**	**24.78**	407.3	$1.80\cdot10^{7}$	**0**	**1311.53**	**24.78**	412.5	$1.80\cdot10^{7}$
A	90	3	**0**	1346.00	33.26	900.0	$1.13\cdot10^{6}$	**0**	1337.13	26.56	409.8	$1.71\cdot10^{7}$	**0**	1337.03	26.54	413.0	$1.71\cdot10^{7}$	**0**	**1323.67**	**26.13**	461.7	$1.71\cdot10^{7}$	**0**	**1323.67**	**26.13**	445.5	$1.71\cdot10^{7}$
A	10	4	**100**	**451.07**	**0.00**	0.1	$3.11\cdot10^{2}$	**100**	**451.07**	**0.00**	<0.1	$3.29\cdot10^{2}$	**100**	**451.07**	**0.00**	<0.1	$3.11\cdot10^{2}$	**100**	**451.07**	**0.00**	<0.1	$3.12\cdot10^{2}$	**100**	**451.07**	**0.00**	<0.1	$3.12\cdot10^{2}$
A	20	4	**100**	**751.73**	**0.00**	8.3	$1.66\cdot10^{4}$	**100**	**751.73**	**0.00**	0.2	$2.32\cdot10^{4}$	**100**	**751.73**	**0.00**	0.1	$1.71\cdot10^{4}$	**100**	**751.73**	**0.00**	0.1	$1.71\cdot10^{4}$	**100**	**751.73**	**0.00**	0.1	$1.67\cdot10^{4}$
A	30	4	93	952.90	0.36	113.6	$4.20\cdot10^{5}$	**100**	**951.97**	**0.00**	4.0	$7.45\cdot10^{5}$	**100**	**951.97**	**0.00**	3.6	$6.26\cdot10^{5}$	**100**	**951.97**	**0.00**	3.0	$5.00\cdot10^{5}$	**100**	**951.97**	**0.00**	2.9	$4.95\cdot10^{5}$
A	40	4	23	1094.60	10.00	900.0	$1.30\cdot10^{6}$	**93**	1062.80	0.50	81.3	$7.94\cdot10^{6}$	**93**	1062.80	0.50	75.8	$7.78\cdot10^{6}$	**93**	**1062.07**	**0.36**	65.7	$6.08\cdot10^{6}$	**93**	**1062.07**	**0.36**	68.1	$6.08\cdot10^{6}$
A	50	4	0	1240.90	19.66	900.0	$1.47\cdot10^{6}$	17	1208.53	11.40	374.6	$1.95\cdot10^{7}$	17	1208.03	11.37	372.5	$1.95\cdot10^{7}$	**20**	**1196.73**	**9.30**	374.0	$1.86\cdot10^{7}$	**20**	**1196.73**	**9.30**	397.3	$1.86\cdot10^{7}$
A	60	4	0	1294.73	24.38	900.0	$1.26\cdot10^{6}$	3	1259.47	15.72	350.6	$1.99\cdot10^{7}$	3	1259.43	15.75	378.2	$1.99\cdot10^{7}$	**10**	**1250.40**	**14.60**	393.1	$1.96\cdot10^{7}$	**10**	**1250.40**	**14.60**	402.6	$1.96\cdot10^{7}$
A	70	4	0	1328.40	29.09	900.0	$1.23\cdot10^{6}$	**3**	1300.93	19.70	354.1	$1.86\cdot10^{7}$	**3**	1300.93	19.64	362.0	$1.86\cdot10^{7}$	**3**	**1294.20**	**19.28**	393.5	$1.85\cdot10^{7}$	**3**	**1294.20**	**19.28**	393.5	$1.85\cdot10^{7}$
A	80	4	**0**	1356.93	33.87	900.0	$1.33\cdot10^{6}$	**0**	1338.07	26.25	363.2	$1.79\cdot10^{7}$	**0**	1338.07	26.25	388.1	$1.79\cdot10^{7}$	**0**	**1328.47**	**25.43**	388.6	$1.79\cdot10^{7}$	**0**	**1328.47**	**25.43**	405.8	$1.79\cdot10^{7}$
A	90	4	**0**	1374.47	31.83	900.0	$1.25\cdot10^{6}$	**0**	1356.17	24.69	361.8	$1.73\cdot10^{7}$	**0**	1356.17	24.69	379.0	$1.73\cdot10^{7}$	**0**	**1347.33**	**24.32**	415.1	$1.73\cdot10^{7}$	**0**	**1347.33**	**24.32**	425.0	$1.73\cdot10^{7}$

Comparison of A* Search, MIP, and CP

We finally compare our A* search using the generally dominating upper bound function $Z^{\text{ub}}_{*}(u)$ to solve the MIP model from Section 4.4.6 using Gurobi Optimizer 8.1 in single-threaded mode and the CP model from Section 4.4.7 using MiniZinc 2.1.7 with the backend solver Chuffed. Note that we considered besides Chuffed also the backend solvers Gecode and G12 LazyFD, but Chuffed clearly dominated these alternatives concerning the number of instances solved to proven optimality, as it is documented in more detail in [83]. Table 4.2 shows the aggregated results. Regarding the number of instances that could be solved to proven optimality, the A* search consistently outperforms Gurobi and Chuffed. For particle therapy based instances of type P, A* search could solve all instances with up to 50 jobs to proven optimality. The avionic system based instances of type A are harder to solve. Here, A* search was only able to solve all instances with up to 30 jobs to proven optimality.

The largest instance which A* could solve to proven optimality consists of 80 jobs, whereas the largest instances that Gurobi and Chuffed could solve to proven optimality have 50 and 60 jobs, respectively. Gurobi could solve all avionic based instances with up to 20 jobs, all particle therapy based instances with up to 40 jobs to proven optimality. Computation times for those are, however, significantly larger than for A*. In particular for small instances with up to $n = 30$ jobs, A* only required median computation times of no more than 0.1 seconds for instances of type P. The CP solver Chuffed could solve all instances of type P up to $n = 40$ jobs to optimality and all instances of type A up to $n = 30$ jobs to optimality. The A* algorithm was able to provide equally good or better final solutions than Gurobi and Chuffed in almost all cases for instances of type P. Exceptions occurred only for some of the largest instances with 90 jobs, where Gurobi's heuristic performance proved to be superior.

For instances of type A, Gurobi's heuristic performance is also superior for instances of smaller sizes. Note, however, that the derivation of just heuristic solutions is not in the foreground in this section here. Concerning obtained upper bounds, the A* search again clearly outperforms the MIP approach by a large margin, especially on the largest instances with $n = 90$ jobs. Chuffed is not able to return any upper bounds. We conclude that is seems reasonable to heuristically solve instances with more than 90 jobs.

4.4.11 Impact of Open List Size Limit ϕ and Different Labeling Functions

In this and upcoming sections we focus on heuristically solving large instances of the PCJSOCMSR with $n \in \{50, 100, 150, 200, 250, 300, 350, 400, 450, 500\}$ jobs. We start with an experimental evaluation on the impact of parameters used for A*C to compile relaxed MDDs. We tested A*C with different open list size limits ϕ and four different variants of the labeling function $L(u)$ used for mapping nodes to collector nodes. The considered labeling function variants are $L^1(u) = t_0(u)$, $L^2(u) = (t_0(u), r(u))$, $L^3(u) = (t_0(u), Z^{\text{ub}}(u))$ and $L^4(u) = L(u) = (t_0(u), r(u), Z^{\text{ub}}(u))$, as proposed in Section 4.4.8. From now on

Table 4.2: Average results of A*, MIP, and CP.

			A*, $Z^{\mathrm{ub}}_*(u)$					MIP					CP, Chuffed		
type	n	m	%-opt	$\overline{obj}$	$\overline{Z^{\mathrm{ub}}}$	$\overline{\text{\%-gap}}$	t[s]	%-opt	$\overline{obj}$	$\overline{Z^{\mathrm{ub}}}$	$\overline{\text{\%-gap}}$	t[s]	%-opt	$\overline{obj}$	t[s]
P	10	2	**100**	**30.93**	**30.93**	**0.00**	<0.1	**100**	**30.93**	**30.93**	**0.00**	<0.1	**100**	**30.93**	0.8
P	20	2	**100**	**50.37**	**50.37**	**0.00**	<0.1	**100**	**50.37**	**50.37**	**0.00**	0.1	**100**	**50.37**	0.4
P	30	2	**100**	**75.33**	**75.33**	**0.00**	<0.1	**100**	**75.33**	**75.33**	**0.00**	4.4	**100**	**75.33**	0.6
P	40	2	**100**	**98.93**	**98.93**	**0.00**	0.2	**100**	**98.93**	**98.93**	**0.00**	69.0	**100**	**98.93**	3.0
P	50	2	**100**	**123.27**	**123.27**	**0.00**	3.9	30	122.73	144.60	14.13	900.3	**100**	**123.27**	56.3
P	60	2	**100**	**146.80**	**146.80**	**0.00**	53.7	0	143.73	218.37	33.91	900.2	13	142.23	900.0
P	70	2	**56**	**168.00**	**175.93**	**4.41**	712.0	0	165.33	308.00	46.12	900.1	0	154.93	900.0
P	80	2	**3**	184.13	**211.23**	**12.71**	900.0	0	**189.27**	370.97	48.95	900.4	0	171.70	900.0
P	90	2	**0**	198.87	**251.47**	**20.69**	900.0	**0**	**215.27**	457.10	52.80	900.2	**0**	188.53	900.0
P	10	3	**100**	**36.17**	**36.17**	**0.00**	<0.1	**100**	**36.17**	**36.17**	**0.00**	<0.1	**100**	**36.17**	0.6
P	20	3	**100**	**59.27**	**59.27**	**0.00**	<0.1	**100**	**59.27**	**59.27**	**0.00**	0.1	**100**	**59.27**	0.5
P	30	3	**100**	**86.30**	**86.30**	**0.00**	<0.1	**100**	**86.30**	**86.30**	**0.00**	4.6	**100**	**86.30**	0.5
P	40	3	**100**	**112.00**	**112.00**	**0.00**	0.5	**100**	**112.00**	**112.00**	**0.00**	92.2	**100**	**112.00**	4.0
P	50	3	**100**	**140.33**	**140.33**	**0.00**	10.0	10	138.70	175.73	20.02	900.5	93	140.20	116.9
P	60	3	**87**	**163.97**	**166.00**	**1.18**	138.4	0	161.97	235.63	30.87	900.6	13	160.43	900.0
P	70	3	**40**	184.57	**202.60**	**8.41**	871.8	0	**187.20**	319.23	41.22	900.1	0	179.60	900.0
P	80	3	**10**	205.40	**247.40**	**16.67**	900.0	0	**215.97**	388.50	44.21	900.0	0	202.17	900.0
P	90	3	**0**	224.90	**282.60**	**20.12**	900.0	**0**	**240.67**	469.97	48.70	900.0	**0**	220.80	900.0
A	10	3	**100**	**422.13**	**422.13**	**0.00**	<0.1	**100**	**422.13**	**422.13**	**0.00**	14.2	**100**	**422.13**	1.0
A	20	3	**100**	**707.27**	**707.27**	**0.00**	0.1	**100**	**707.27**	**707.27**	**0.00**	15.1	**100**	**707.27**	1.1
A	30	3	**100**	**903.97**	**903.97**	**0.00**	2.0	83	**903.97**	908.40	0.47	42.6	**100**	**903.97**	3.3
A	40	3	**97**	1026.10	**1030.00**	**0.36**	40.0	0	**1027.90**	1126.87	8.76	900.7	37	979.53	900.0
A	50	3	**33**	1056.67	**1177.67**	**9.96**	377.2	0	**1114.87**	1360.57	18.03	900.2	0	887.03	900.0
A	60	3	**3**	1011.97	**1225.57**	**17.28**	416.6	0	**1105.33**	1506.27	26.56	900.1	0	807.13	900.0
A	70	3	**0**	1009.63	**1270.53**	**20.44**	442.8	**0**	**1116.90**	1696.97	34.11	900.1	**0**	803.57	900.0
A	80	3	**0**	985.60	**1311.53**	**24.78**	412.5	**0**	**1106.03**	1876.03	40.98	900.0	**0**	746.60	900.0
A	90	3	**0**	977.37	**1323.67**	**26.13**	445.5	**0**	**1095.27**	2055.27	46.65	906.0	**0**	726.13	900.0
A	10	4	**100**	**451.07**	**451.07**	**0.00**	<0.1	**100**	**451.07**	**451.07**	**0.00**	<0.1	**100**	**451.07**	0.7
A	20	4	**100**	**751.73**	**751.73**	**0.00**	0.1	**100**	**751.73**	**751.73**	**0.00**	2.6	**100**	**751.73**	1.2
A	30	4	**100**	**951.97**	**951.97**	**0.00**	2.9	83	**951.97**	959.73	0.77	208.3	**100**	**951.97**	13.9
A	40	4	**93**	1058.10	**1062.07**	**0.36**	68.1	0	**1058.80**	1191.53	11.11	900.2	17	970.57	900.0
A	50	4	**20**	1082.63	**1196.73**	**9.30**	397.3	0	**1127.73**	1443.30	21.79	900.1	0	874.40	900.0
A	60	4	**10**	1065.37	**1250.40**	**14.60**	402.6	0	**1143.90**	1592.60	28.10	900.1	0	821.60	900.0
A	70	4	**3**	1043.47	**1294.20**	**19.28**	393.5	0	**1145.40**	1769.70	35.25	900.0	0	809.90	900.0
A	80	4	**0**	990.23	**1328.47**	**25.43**	405.8	**0**	**1130.87**	1936.70	41.57	900.0	**0**	764.00	900.0
A	90	4	**0**	1019.07	**1347.33**	**24.32**	425.0	**0**	**1128.27**	2090.50	45.98	900.0	**0**	734.37	900.0

we always use A*C with upper bound function $Z^{\mathrm{ub}}_*(\cdot)$ to guide the compilation process, i.e., $Z^{\mathrm{ub}}(\cdot) = Z^{\mathrm{ub}}_*(\cdot)$. Figure 4.7 illustrates the impact of the different choices for ϕ and the labeling function on instances with 250 jobs of set P with $m = 2$ and of set A with $m = 3$, respectively.

For each combination of value for ϕ and labeling function variant, there is a box plot drawn that summarizes the obtained results over all 30 instances of the corresponding category. The diagrams at the top show the lengths $Z^{\mathrm{lp}}(\mathbf{t})$ of the longest paths from the obtained relaxed MDDs, whereas the diagrams in the middle show the corresponding CPU-times for compiling the MDDs. Moreover, the diagrams at the bottom state the size of the relaxed MDDs in terms of the number of nodes. The diagrams to the left in Figure 4.7 show the results for instance set P using the values $\phi \in \{1000, 2000, 3000, 5000\}$. As one could expect, we see that with increasing ϕ, the lengths of the longest paths of the obtained relaxed MDDs in general get smaller, i.e., the obtained upper bounds

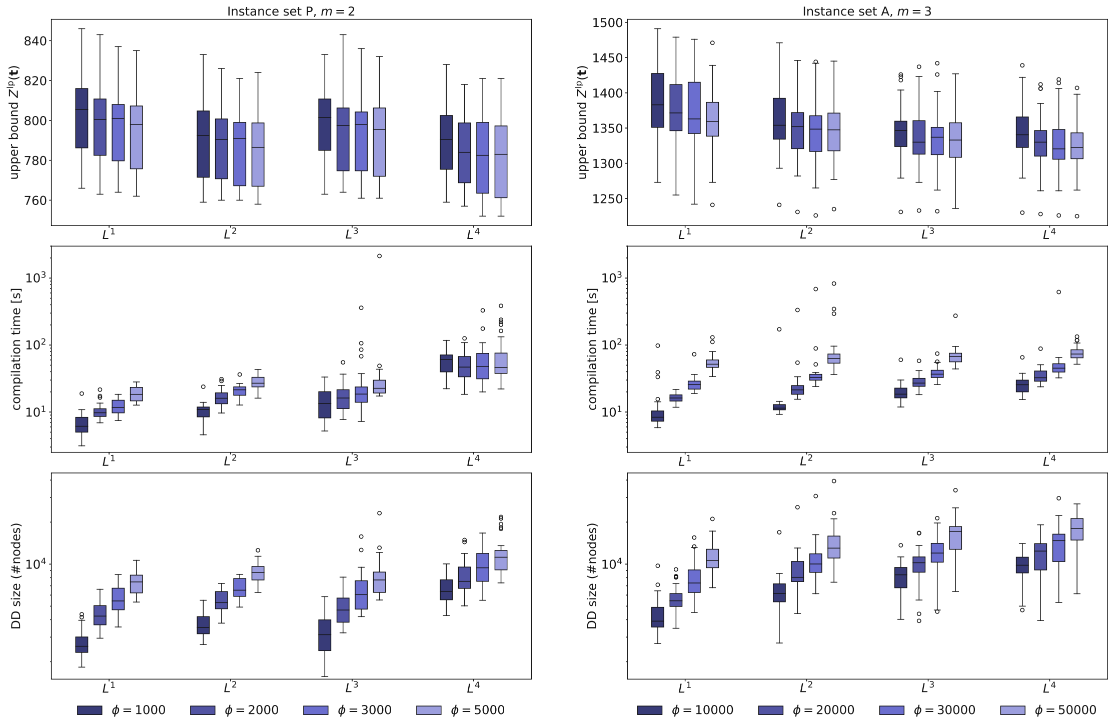


Figure 4.7: Comparison of open list size limits ϕ and labeling functions L^i, $i = 1, \dots, 4$, for instances of sets P and A with 250 jobs and $m = 2$ and $m = 3$ secondary resources, respectively.

become stronger, while the MDD sizes and computation times naturally increase. Thus, parameter ϕ indeed allows to control the MDD's size, although not in such a direct linearly related fashion as the width-limit in a classical top-down construction. This effect can be observed for all labeling functions. Concerning the different labeling functions, $L^1(u) = t_0(u)$ yields relaxed MDDs with in general the smallest sizes, but also the weakest bounds. This is, however, also achieved in the shortest times. The reason for this is that labeling function L^1 does only consider the time from which on the common resource is available and has therefore the smallest domain among the four considered labeling functions. Hence, when using L^1, there are in general far more node merges than with one of the other more complex labeling functions which provide larger domains and therefore a finer differentiation of nodes. It can clearly be seen that the additional consideration of $r(u)$ or $Z^{\mathrm{ub}}(u)$ in the labeling function in general significantly improves the obtained bounds $Z^{\mathrm{lp}}(\mathbf{t})$, and the combination of all these aspects in L^4 provides the best results. This, however, at the cost of larger MDDs and higher running times. The smallest median longest path length of 783 for instances with two secondary resources were obtained when limiting the size of the open list to $\phi = 5000$ nodes and using L^4. In more detail, note that parameter ϕ has more impact when using labeling function L^1 and less when using labeling function L^4. This can again be explained by the domain sizes of the labeling functions, but also the fact that the bounds obtained with L^4 are in general already closer to the optimal solution values and it becomes more and more difficult to find better bounds. When comparing the results of L^2 and L^3, we can see that L^2 yields mostly slightly better results, but this again at the cost of longer running times.

The diagrams to the right in Figure 4.7 shows the results for instance set A using the values $\phi \in \{10000, 20000, 30000, 50000\}$. Note that, since the time horizon in this case never exceeds $T = 1000$, larger values of ϕ were considered than in the experiments for instance set P. This implies that also the MDDs' heights are restricted correspondingly, and larger values for ϕ can be used to utilize roughly comparable computation times. Again, we can see that parameter ϕ allows to control the quality of the obtained relaxed MDDs. Hence, with increasing ϕ, the lengths of the longest paths of the obtained relaxed MDDs in general get smaller, while the computation times and MDD sizes increase.

Structurally similar results are obtained for instances of set P with three secondary resources as well as for instances of set A with four secondary resources, cf. Appendix B.

For all further experiments, we went for a compromise between expected quality of the relaxed MDD and compilation time and fixed the following settings. Instance set P: labeling function $L^3(u)$ and $\phi = 1000$; instance set A: labeling function $L^4(u)$ and $\phi = 20000$.

4.4.12 Upper Bound Comparison from relaxed MDDs

The five types of upper bounds to be compared are the following. The first two are from A*C, namely $Z^{\mathrm{ub}}_{\min}$, which is obtained when the target node is chosen for expansion, and $Z^{\mathrm{lp}}(\mathbf{t})$, which is the longest path length of the completely constructed relaxed MDD. A third one is obtained by solving a MIP model with a commercial solver, while the last two come from MDDs built with traditional TDC and IR algorithms. Remember that $Z^{\mathrm{lp}}(\mathbf{t})$ may be larger than $Z^{\mathrm{ub}}_{\min}$ due to additionally performed merging operations after having found $Z^{\mathrm{ub}}_{\min}$.

The MIP model from Section 4.4.6 is solved again with Gurobi Optimizer 8.1 in single-threaded mode with a CPU time limit of 900 seconds. The TDC and IR methods are those from Maschler and Raidl [116, 117]. The latter is performed with a CPU time limit of 900 seconds and TDC is executed with two different width limits β which were chosen in a way so that the average running times are roughly in the same order of magnitude and usually not smaller than those of A*C: $\beta \in \{300, 500\}$ for set P and $\beta \in \{3000, 5000\}$ for set A.

Figure 4.8 documents the results of this comparative study for instance sets P and A. The diagrams at the top show the obtained upper bounds, the middle diagrams the computation times, and the diagrams at the bottom the sizes of obtained relaxed MDDs in terms of the number of nodes. Each group of bars on the horizontal axes corresponds to a specific instance class with the stated number of jobs, and each bar indicates the average value over all 30 instances of the corresponding instance class and the respective approach.

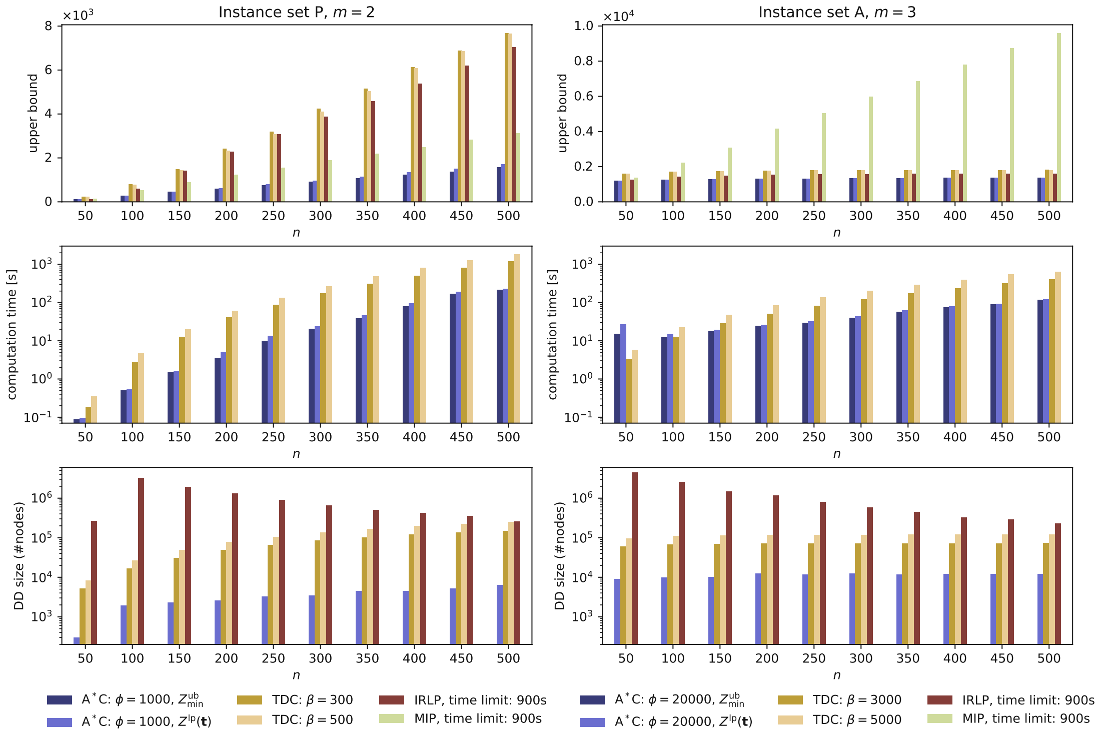


Figure 4.8: Instance sets P and A with two and three secondary resources, respectively, average values of: upper bounds obtained from A*C, the classical TDC, the IR, and the order-based MIP approach; respective computation times; and the sizes of the obtained relaxed MDDs.

Concerning the depicted computation times, each first bar shows A*C's average time to obtain $Z^{\text{ub}}_{\text{min}}$, i.e., when the construction would stop according the classical A* termination criterion, while the second bar shows the average time required for the construction of the complete relaxed MDD. Since the MIP approach as well as IR exhausted the time limit of 900 seconds in almost all runs, we omit corresponding bars. More specifically, the MIP solver could only solve the smallest instances to proven optimality. The percentages of the instances with $n = 50$ jobs are 23.3% and 10% of set P for $m = 2$ and $m = 3$, respectively. The IR approach was not able to solve any instance to proven optimality.

The results for instance set P, shown in Figure 4.8 on the left side, give a rather clear picture. The average upper bounds $Z^{\text{ub}}_{\text{min}}$ obtained by the A*C algorithm are always the strongest. They are in particular substantially better than those obtained from the TDC variants and the IR approach. The difference is more than a factor of four for the largest instances. Even more dramatic are the differences in the sizes of the respectively obtained MDDs. A*C's MDDs are usually more than an order of magnitude smaller than those compiled with TDC and IR. The A*C algorithm clearly can take advantage from avoiding multiple nodes for the same state at different layers, and the merging strategy we proposed appears to be effective. The bounds obtained from the MIP approach are clearly better than those of TDC and IR, but also significantly worse than those of A*C. Differences between $Z^{\mathrm{ub}}_{\min}$ and $Z^{\mathrm{lp}}(\mathbf{t})$ are in comparison to the bounds from the other approaches not that large, but still significant.

For instance set A, Figure 4.8 shows remarkable differences. The upper bounds obtained from the MIP approach are far worse than those obtained from A*C as well as TDC and IR. Differences between A*C, TDC, and IR are not that large anymore, but nevertheless, in each case the strongest upper bounds could be obtained by A*C. The better relative performance of the classical approaches TDC and IR on these instances in comparison to set P can be explained by the constant time horizon and the special prize structure, due to which the height of the MDDs is limited in a stronger way. Concerning the size of the obtained MDDs, A*C exhibits again substantial advantages over TDC: A*C's MDDs only have about half the size of TDC's MDDs, and those of IR are even more than three times larger than those of A*C for the smaller instances.

Similar results are obtained for instances of instance set P with three secondary resources as well as for instances of instance set A with four secondary resources, cf. B.

4.4.13 Lower Bound Comparison to Other Approaches

Finally, we consider the A*C approach to construct a relaxed MDD followed by the construction of a restricted MDD and compare to other heuristic methods to approach larger PCJSOCMSR instances. Now, our focus is primarily on the quality of obtained heuristic solutions, i.e., lower bounds, but since our approach also yields upper bounds from the relaxed MDD, we will also study resulting gaps. We compare to a conventional TDC of a restricted MDD, a general variable neighborhood search (GVNS) metaheuristic, the MIP approach, and a basic CP formulation.

After a relaxed MDD has been constructed by A*C, it is post-processed by filtering in order to reduce its size and strengthen it before deriving the restricted MDD. This is done as follows.

1. A first lower bound (and heuristic solution) is determined in a quick way by compiling a small restricted MDD in an independent way (maximum width $\beta = 100$ for type P instances and $\beta = 15000$ for type A instances).

2. Using the obtained lower bound, cost-based filtering (see, e.g., Cire and van Hoeve [40]) is applied in order to get rid of many arcs and nodes that cannot be part of a path representing a better solution.

3. For each node u in the relaxed MDD, we have the upper bound for the cost-to-go obtained from the auxiliary upper bound function $Z^{\mathrm{ub}}(u)$, but also the length of the longest u-$\mathbf{t}$ path provides an upper bound, which we denote by $Z^{\mathrm{lp}\uparrow}(u)$. We keep the better of these bounds and check if further arcs and nodes may be removed

due to it by cost-based filtering. Note that $Z^{\mathrm{lp}\uparrow}(u)$ can be determined for all $u \in V$ efficiently by a single bottom-up traversal of the MDD.

4. When removing some ingoing arcs of a node, we always re-determine the state of the node, and if the state changes, the auxiliary upper bound $Z^{\mathrm{ub}}(u)$. Changes are always propagated to successor nodes as far as they are affected.

After the relaxed MDD has been filtered, it is used to compile the main restricted MDD. Experiments showed that on average 51.57% and 88.90% of all arcs can be removed from the relaxed MDD over all instance sizes for type P and type A instances, respectively. This substantial reduction leads, in particular for type A instances, to shorter computation times when compiling the main restricted MDD.

The conventional TDC of a restricted MDD uses the same greedy criterion from Section 4.4.9 to select nodes for removal as our compilation method based on the relaxed MDD. Figure 4.9 shows for different maximum widths β a comparison between the conventional TDC and the TDC when utilizing a previously compiled and filtered relaxed MDD by A*C. The relaxed MDDs were compiled with different values of ϕ and used labeling function $L^3(u)$ and $L^4(u)$ for instance set P and A, respectively. Although the choice of ϕ has an impact on the quality of the obtained relaxed MDD, as shown in previous sections, the plots in Figure 4.9 indicate that ϕ does not significant influence the finally obtained objective values from a subsequently applied TDC for the considered instance classes. Regarding computation times, we can see that larger values of ϕ will result in larger computation times. While the obtained objective values are not substantially different compared to those from the conventional TDC, the diagrams at the bottom row indicate substantial time savings when a relaxed MDD is used to compile a restricted MDD. For example for instances with 250 jobs and two secondary resources of instance set P the conventional TDC needs for $\beta = 20000$ 1407 seconds to terminate whereas the A*C+filtering+TDC approach only needs 170 seconds. This time saving of frequently almost an order of magnitude is the benefit of using the structural information of a previously compiled and filtered relaxed MDD.

Moreover, we also tried to create restricted MDDs by a variant of A*C in which nodes are removed from the open list instead of merging them in Line 30 of Algorithm 4.1. This, however, only yielded restricted MDDs with substantially worse objective values than the conventional TDC. Increasing parameter ϕ to allow a larger open list size also did not help much in this case but just led to larger computation times.

Note that the computational experiments reported in Figure 4.9 were done on a somewhat different cluster environment at a later time than all other experiments of this chapter. After a thorough analysis we concluded that computation times of our algorithms differed by a factor of 2.2 between the different environments, and we have scaled the reported times in Figure 4.9 accordingly to make them directly comparable.

For the main results in Table 4.3 we compile restricted MDDs with $\beta = 2000$ and $\beta = 12000$ for benchmark sets P and A, respectively. Moreover, the A*C+filtering+TDC

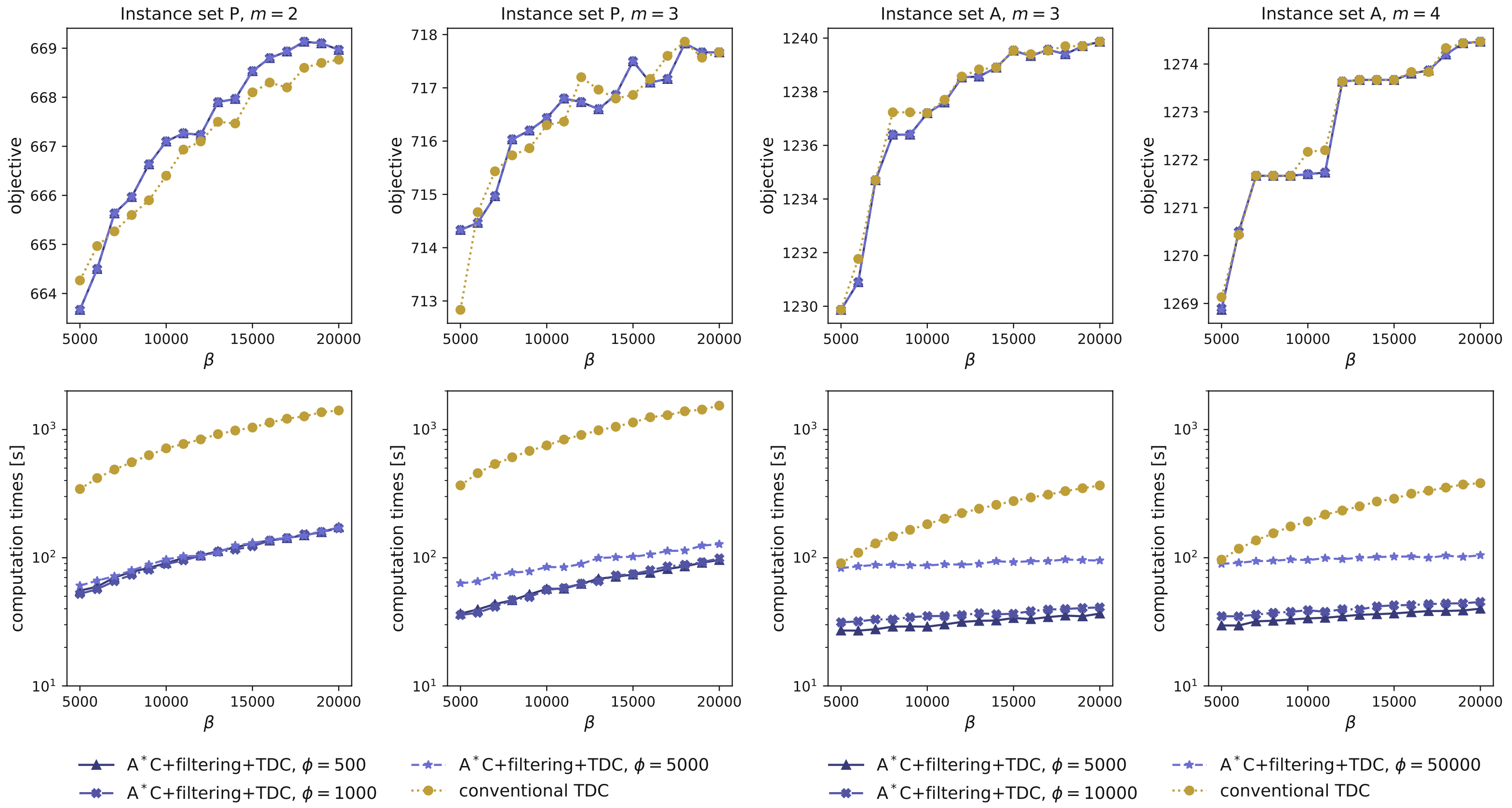


Figure 4.9: Comparison of A*C+filtering+TDC and conventional TDC for instances of sets P and A with 250 jobs: average objective values and median computation times in dependence of β.

approach compiles restricted MDDs with $\beta = 12000$ and $\beta = 45000$ for benchmark sets P and A, respectively. These values have been selected so that the TDC terminates for the largest instances in about 900 CPU-seconds.

The GVNS is the one from Maschler and Raidl [116, 117]. It applies a job permutation encoding, starts with a random initial solution, and combines a classic exchange and insertion neighborhood search for intensification. For diversification (shaking), up to four random insertion moves are performed. The GVNS terminates when reaching a time limit of 900 seconds.

Moreover, we compare to the objective values of the best feasible solutions provided by the order-based MIP formulation from Section 4.4.6, solved again by Gurobi using a single thread with a CPU time limit of 900 seconds.

Last but not least, we also consider the CP model from Section 4.4.7. The model was implemented with MiniZinc 2.1.7 and we apply the backbone solver Chuffed with a time limit of 900 seconds. Results with Chuffed consistently dominated those obtained with the alternative backbone solvers Gecode and G12 LazyFD. Note that we further performed tests with the newer MiniZinc version 2.3.2, but obtained results were inconsistent and mostly worse than those from version 2.1.7.

The results of all approaches are presented in Table 4.3. Each row shows the aggregated results over the 30 benchmark instances with the characteristics given in the first three columns. For all approaches columns $\overline{obj}$ and $\sigma(obj)$ state the mean objective values of obtained heuristic solutions and corresponding standard deviations. For the MDD-based approaches these values correspond to the lengths of the longest paths in the restricted MDDs. Moreover, we list for the MDD-based approaches median total computation times in seconds in the $t[s]$ columns, and for the A*C based approach more specifically in column $t^{\mathrm{f}}[s]$ median times just for filtering the relaxed MDDs including the times for determining the required lower bound and in column $t^{\mathrm{c}}[s]$ median times for compiling the final restricted MDDs. For GVNS, MIP, and CP timing information is omitted as they were always terminated with the time limit of 900 seconds. The only exceptions are MIP and CP runs for the smallest instances with 50 jobs, which finished in some cases earlier with proven optimality. In addition, we list for the A*C based approach average optimality gaps, where %-gap $= 100\% \cdot (Z^{\mathrm{ub}}_{\mathrm{min}} - obj)/Z^{\mathrm{ub}}_{\mathrm{min}}$.

If we disregard the results from the benchmark instances of type P with three secondary resources for now, Table 4.3 gives a clear picture. A*C+filtering+TDC provides in general the best solutions, followed by the GVNS and the TDC. While the TDC performs, on the P instances with two secondary resources, in most cases better than the GVNS, the GVNS is superior to the TDC on the other instances. The weakest solutions have on average been obtained by the MIP and CP approaches, which are only competitive for type P instances with 50 jobs. Especially, for the medium to large instances, the A*C based method typically requires less time than TDC. A*C+filtering+TDC is superior to the conventional TDC in two ways. Not only are we able to construct much larger restricted MDDs, usually yielding better solutions in less time, but since we first also

			A*C+filtering+TDC						TDC			GVNS		MIP		CP	
set	m	n	$\overline{obj}$	$\sigma(obj)$	$\overline{\%\text{-gap}}$	$t^{\mathrm{f}}[s]$	$t^{\mathrm{c}}[s]$	$t[s]$	$\overline{obj}$	$\sigma(obj)$	$t[s]$	$\overline{obj}$	$\sigma(obj)$	$\overline{obj}$	$\sigma(obj)$	$\overline{obj}$	$\sigma(obj)$
P	2	50	**123.3**	10.3	2.2	<1	<1	<1	**123.3**	10.3	<1	123.2	10.4	122.9	10.7	**123.3**	10.3
P	2	100	**259.9**	11.7	9.4	<1	5	6	259.2	11.7	5	259.0	11.9	238.5	13.3	200.7	20.0
P	2	150	**401.2**	18.9	11.5	<1	17	19	398.8	19.3	20	396.6	17.7	328.9	22.3	261.0	23.5
P	2	200	**530.0**	18.7	12.3	<1	43	50	526.1	19.3	54	527.1	19.0	383.4	29.5	273.6	61.2
P	2	250	**667.2**	21.1	12.6	1	84	100	661.7	20.0	108	660.8	22.4	475.6	29.7	281.0	52.9
P	2	300	**797.4**	16.8	13.1	2	145	170	792.0	16.8	193	790.0	17.5	570.8	32.1	308.1	78.5
P	2	350	**931.3**	25.6	13.7	3	231	282	923.9	25.2	313	923.0	26.7	626.5	53.8	326.7	122.2
P	2	400	**1061.6**	21.3	13.9	5	338	442	1054.4	21.4	481	1055.1	19.4	661.1	55.0	329.6	110.9
P	2	450	**1197.0**	28.7	13.5	7	485	679	1187.5	28.6	704	1180.9	27.4	704.7	48.5	348.6	152.4
P	2	500	**1339.1**	25.4	14.6	8	613	845	1330.3	25.0	949	1324.3	28.0	711.6	199.3	403.4	351.5
P	3	50	**140.3**	10.4	1.3	<1	<1	<1	**140.3**	10.4	<1	**140.3**	10.4	139.3	10.2	140.2	10.5
P	3	100	**289.9**	14.5	5.1	<1	5	6	288.4	14.6	7	288.5	14.3	268.7	16.0	240.0	16.4
P	3	150	**437.4**	15.1	7.1	<1	13	15	433.5	14.8	24	437.0	17.1	362.0	21.3	331.3	19.3
P	3	200	581.8	18.3	8.4	<1	27	33	576.9	18.2	59	**582.9**	16.3	460.8	26.7	367.2	44.4
P	3	250	716.7	13.6	9.5	1	44	63	712.0	14.4	117	**721.9**	16.5	573.4	22.2	380.6	89.3
P	3	300	850.3	16.0	11.2	2	71	107	846.0	15.3	210	**864.0**	19.8	675.1	27.6	419.5	109.8
P	3	350	988.0	26.8	11.7	3	107	171	983.3	27.7	341	**1008.2**	29.3	716.4	61.0	405.7	188.8
P	3	400	1124.6	24.5	12.3	5	152	296	1119.1	23.6	530	**1142.5**	24.1	757.8	61.9	527.5	161.1
P	3	450	1266.3	19.7	11.9	7	198	433	1257.7	20.9	751	**1283.5**	26.1	846.6	59.8	524.5	207.3
P	3	500	1397.8	25.3	12.3	10	269	672	1392.1	23.7	1082	**1418.0**	27.1	900.2	53.6	589.3	238.6
A	3	50	**1130.3**	39.8	5.0	7	20	57	1127.8	38.4	6	**1130.3**	39.8	1114.5	41.9	892.0	45.8
A	3	100	1201.1	36.5	5.6	9	81	110	1196.6	36.4	23	**1201.4**	37.0	1108.2	52.0	712.7	44.1
A	3	150	**1215.5**	26.3	6.0	11	141	169	1208.9	28.7	58	**1215.5**	27.0	936.7	58.0	643.4	41.9
A	3	200	1228.9	21.8	6.7	14	219	261	1220.4	25.6	109	**1229.4**	21.4	842.6	132.5	544.5	149.4
A	3	250	1244.7	28.6	6.6	18	279	331	1238.5	30.8	180	**1245.4**	28.7	703.2	79.3	575.3	48.0
A	3	300	**1243.9**	23.4	7.5	22	379	448	1234.4	22.9	265	1243.7	23.5	675.2	80.8	553.8	41.3
A	3	350	**1256.2**	22.6	7.4	30	440	542	1245.6	24.8	370	1255.5	23.6	683.8	84.4	536.5	50.3
A	3	400	**1269.7**	19.1	8.0	34	529	646	1262.5	19.1	493	1267.2	19.6	714.1	57.8	525.2	42.8
A	3	450	**1268.4**	19.2	8.3	41	609	748	1257.5	24.5	647	1268.2	18.3	730.1	79.1	516.6	48.6
A	3	500	**1271.7**	19.0	8.2	46	676	869	1260.4	22.7	799	1271.2	17.9	680.7	63.8	527.5	24.0
A	4	50	1141.8	35.5	3.4	7	2	45	1138.7	35.0	6	**1142.4**	36.2	1127.6	40.0	882.5	42.0
A	4	100	1218.8	40.6	4.0	8	57	82	1215.7	41.5	25	**1218.9**	40.6	1137.3	62.5	708.9	119.8
A	4	150	**1253.8**	30.6	4.4	10	118	146	1248.5	30.4	60	1253.5	30.9	963.7	77.7	655.6	48.5
A	4	200	1259.5	31.3	4.9	15	180	228	1253.6	32.0	118	**1259.6**	31.7	881.0	145.7	576.5	129.5
A	4	250	**1280.4**	27.0	5.9	20	277	339	1273.6	25.7	191	**1280.4**	28.2	702.8	60.0	541.4	134.8
A	4	300	**1293.8**	25.1	5.9	27	346	441	1282.6	28.0	287	1292.3	24.9	691.6	64.0	565.0	46.0
A	4	350	**1298.9**	23.9	5.8	32	394	504	1289.9	24.5	396	1297.2	22.9	684.6	75.4	548.8	38.1
A	4	400	**1304.4**	42.0	6.0	37	477	617	1298.1	41.2	533	1299.6	42.9	700.5	69.8	541.4	27.9
A	4	450	**1308.4**	25.2	6.9	45	551	752	1298.9	29.2	672	1304.7	26.7	696.4	65.2	546.5	56.6
A	4	500	**1315.6**	28.0	6.9	48	644	842	1304.0	25.8	858	1312.5	29.1	715.4	73.1	526.8	33.1

Table 4.3: Comparison of the subsequent application of A*C, filtering, and the construction of restricted MDDs to the conventional top-down construction of restricted MDDs, the GVNS, MIP, and CP approaches.

determine the relaxed MDDs, our approach has the additional bonus of providing upper bounds. Average gaps never exceed 15% and are in particular for instance set A usually not larger than 8%.

On benchmark instances of set P with three secondary resources, GVNS typically provides the best solutions when more than 150 jobs are considered. The relative differences between the obtained objective values from GVNS and our A*C-based approach are typically about one to two percent. We believe that in these cases, the GVNS's local search is particularly effective. Clearly, an option would be to finally "polish" the solutions of the MDD-based methods by applying a local search. Another particularity of the results for set P with $m = 3$ are the required times t^{c} for constructing the restricted MDDs.

Although the same maximum width is used as for the instances with two secondary resources, these median times are considerably shorter for the case with three secondary resources than for two. This indicates an even better exploitation of the relaxed MDD and underlines the consistent performance improvements of A*C+filtering+TDC over the classical TDC of restricted MDDs.

The optimality gaps increase with the problem size on all instance sets, as one might expect for a compilation of relaxed and restricted MDDs with fixed parameter values. In comparison to instance set A, we obtain smaller optimality gaps on type P instances with few jobs but get larger optimality gaps for the instances with many jobs. This can be explained by the problem size independent time horizon of set A instances, which implies a certain maximal number of jobs that can be scheduled independently of the number of available jobs.

Last but not least, for some instances the optimality gap has been closed, i.e., they could be solved to proven optimality. This was the case for nine of the type P instances with two secondary resources and 50 jobs. For type P instances with three secondary resources we could optimally solve ten instances that consider 50 jobs. Furthermore, for a single benchmark instance with 50 jobs and four secondary resources of type A, the lower and upper bound coincided.

4.5 Longest Common Subsequence Problem

In this section we will compile relaxed MDDs with A*C for the longest common subsequence (LCS) problem in order to obtain tight upper bounds on the solution length. Another goal is to investigate the applicability of A*C on the prominent LCS problem in order to see if this construction method has the potential to lead to superior results also on this different kind of problem and not only for the PCJSOCMSR in the previous Section 4.4. Indeed, an extensive experimental evaluation on several standard LCS benchmark instance sets shows that our novel construction algorithm clearly outperforms also for the LCS problem a traditional top-down compilation (TDC) for MDDs. We are able to obtain stronger and at the same time more compact relaxed MDDs than TDC and this in a shorter time. Furthermore, we will compare the A*C approach for the LCS problem not only to a top-down MDD construction but also to several upper bounding procedures for the LCS obtained from the literature. For several groups of benchmark instances new best known upper bounds are obtained. In comparison to existing simple upper bound procedures, the obtained bounds are on average 14.8% better.

The next section introduces the LCS problem in a formal way and Section 4.5.2 discusses related work of the LCS problem. States and transitions for the LCS are defined in Section 4.5.3 and Section 4.5.4 reviews two known procedures to obtain upper bounds for the length of an LCS and presents a new one that extends one of those. Section 4.5.5 explains how relaxed MDDs are compiled for the LCS problem with A*C. Results of computational experiments are discussed in Section 4.5.6.

4.5.1 Problem Formulation

The goal of the LCS problem [112] is to find the longest string which is a common subsequence of a set of m input strings $S = \{s_1, s_2, \ldots, s_m\}$ over a finite alphabet Σ. We denote the length of a string s by $|s|$, and let n be the maximum length of the input strings, i.e., $n = \max_{i=1,\ldots,m} |s_i|$. A subsequence is a string that can be derived from another string by deleting zero or more characters. A common subsequence can be derived from all input strings. For instance, for the input strings `ABCDBA` and `ACBDBA`, an LCS is `ABDBA`. Determining the length of an LCS is a way to measure the similarity of strings and has a wide range of applications, for example in computational biology where strings often represent segments of RNA or DNA [87, 145]. Other applications can be found in text editing, file comparison, data compression, and the production of circuits in field programmable gate arrays, to just name a few [104, 8, 32]. If m is fixed then the LCS problem can be solved by dynamic programming (DP) based algorithms in polynomial time $O(n^m)$ [65]. For an arbitrary number of input strings, however, the problem is known to be NP-hard [112].

Before we proceed, let us define further notation. We denote the character at position j in a string s by $s[j]$, and $s[j, j']$, $j \leq j'$, refers to the continuous subsequence of s starting at position j and ending at position j'. For $j > j'$, substring $s[j, j']$ is the empty string denoted by ε. Last but not least, let $|s|_a$ be the number of occurrences of character $a \in \Sigma$ in string s.

4.5.2 Related Work

In the literature plenty of exact approaches have been proposed for solving the LCS problem. Besides the already mentioned DP based approaches, Blum and Festa [27] investigated a MIP model, which is however not competitive and cannot be practically applied to any of the commonly used benchmark sets in the literature due to its excessive size. Further exact methods are for instance based on dominant point approaches and/or parallelization [36, 107, 128, 161] or on a transformation to the max clique problem [26], but they are still not applicable to practical instances with a large number of long input strings.

Solving LCS instances of practical relevance to proven optimality is still a challenging task in terms of computation time and memory consumption. Therefore, heuristic approaches are used for larger m and n. Fast construction heuristics are, e.g., the expansion algorithm [30] or the best next heuristic [57, 86]. Among the more advanced search strategies, in particular beam search (BS) based approaches have been frequently proposed differing in various details such as the heuristic guidance and filtering. This culminated in a general BS-based framework by Djukanovic at el. [50] which can express essentially all heuristic state-of-the-art approaches from the literature by respective configuration settings. They authors proposed also a novel heuristic guidance function, which approximates the expected length of a LCS for random strings. The BS framework in combination with this novel guidance dominates the other existing approaches on most of the available benchmark instances. The same authors further described novel A* based anytime algorithms by interleaving A* search with BS or anytime column search, respectively [51]. Thereby the novel search guidance from before plays again a crucial role.

4.5.3 State Graph

Since the A*C Algorithm 4.1 from Section 4.3 operates on states and state transitions, we have to define a state graph in order to compile relaxed MDDs for the LCS problem with A*C. Each node u in this state graph is associated with a state which is a *position vector* $\mathbf{p}(u)$, with $p_i(u) \in \{1, \dots, |s_i|\}$, $i = 1, \dots, m$. On the basis of this position vector it is possible to define a subproblem $S[\mathbf{p}(u)]$ of S by considering the substrings $s_i[p_i(u), |s_i|]$, $i, \dots, m$. Thus, $S[\mathbf{p}(u)]$ consists of the right part of each string from S starting from the position indicated in position vector $\mathbf{p}(u)$. The root state represents the original problem S, indicated by $S[\mathbf{p}(\mathbf{r}) = (1, \dots, 1)]$. An arc $\alpha = (u, v)$ represents the transition from state $\mathbf{p}(u)$ to state $\mathbf{p}(v)$ by appending character $c = \mathrm{val}(\alpha)$, $c \in \Sigma$ to the sequences of characters encoded by the paths from $\mathbf{r}$ to u. The prize associated with a state transition is always one, i.e., $z(\alpha) = 1$. The transition function to obtain successor state $\mathbf{p}(v)$ by considering character c is defined as

$$\tau(\mathbf{p}(u), c) = \begin{cases} (p_{1,c}(u) + 1, \dots, p_{m,c}(u) + 1) & \text{if } c \in \Sigma^{\mathrm{nd}}(u) \\ (n + 1, \dots, n + 1) & \text{if } \Sigma^{\mathrm{nd}} = \emptyset, \end{cases} \tag{4.33}$$

where $p_{i,a}(u)$, $i = 1, \dots, m$ denotes for each character $a \in \Sigma$ the position of the first occurrence of a in $s_i[p_i(u), |s_i|]$ and set $\Sigma^{\mathrm{nd}}(u) \subseteq \Sigma$ contains all letters that can be feasibly appended at state $\mathbf{p}(u)$, thus letters that occur at least once in each string in $S[\mathbf{p}(u)]$, and are non-dominated. A character $a \in \Sigma$ dominates character $b \in \Sigma$ iff $p_{i,a}(u) \leq p_{i,b}(u)$ for all $i = 1, \dots, m$, and therefore it never can be better to append a dominated letter next. States that have no further feasible transition, i.e., where $\Sigma^{\mathrm{nd}} = \emptyset$, are mapped to state $(n + 1, \dots, n + 1)$ of target node $\mathbf{t}$.

4.5.4 Independent Upper Bounds

To compile MDDs based on A* search we need a fast-to-calculate independent upper bound on the solution length of LCS subproblems to guide the construction mechanism. We use two well known upper bounds from the literature as well as a third bound which is an adaption of one of the former. The first upper bound from Fraser [57] was tightened by Blum et al. [25] and is based on the number of occurrences of each character. Given a node u and the associated position vector $\mathbf{p}(u)$, this bound calculates the sum of the minimal number of occurrences of each character over all the strings of the corresponding subproblem $S[\mathbf{p}(u)]$, i.e.,

$$\mathrm{UB}_1(u) = \mathrm{UB}_1(\mathbf{p}(u)) = \sum_{a \in \Sigma} \min_{i=1,\dots,m} |s_i\,[p_i(u), |s_i|]|_a\,. \tag{4.34}$$

By using a suitable data structure prepared in pre-processing, UB_1 can be efficiently computed in $O(m\,|\Sigma|)$ time.

The second upper bound is based on DP and was introduced by Wang et al. [161]. Since the LCS for two input strings can be efficiently computed, for each pair $\{s_i, s_{i+1}\} \subseteq S$, $i = 1, \dots, m-1$ a so-called scoring matrix $\mathbf{M}_i^2$ is computed, where an entry $M_i^2[p, q]$ with $p = 1, \dots, |s_i|$ and $q = 1, \dots, |s_{i+1}|$, stores the length of the LCS of strings $s_i[p, |s_i|]$ and $s_{i+1}[q, |s_{i+1}|]$. The scoring matrices are determined in a pre-processing step. Then

$$\mathrm{UB}_2(u) = \mathrm{UB}_2(\mathbf{p}(u)) = \min_{i=1,\dots,m-1} \mathbf{M}_i^2[p_i(u), p_{i+1}(u)] \tag{4.35}$$

is an upper bound for the subproblem $S[\mathbf{p}(u)]$ of a given node u and the associated position vector $\mathbf{p}(u)$.

The third upper bound we consider adapts the above one as follows. For UB_2, $m-1$ scoring matrices are computed, one for each pair of input strings $\{s_i, s_{i+1}\}$, $i = 1, \dots, m-1$. However, the pairs of input strings are just chosen according to their natural order given by the instance specification. We are aiming now to choose pairs of input strings in a more controlled and more promising way by utilizing as guidance the version of the first upper bound function for two strings, i.e., $\mathrm{UB}_1(s_i, s_{i'}) = \Sigma_{a\in\Sigma} \min(|s_i|_a, |s_{i'}|_a)$, $s_i, s_{i'} \in S$, $s_i \neq s_{i'}$. Pairs of strings for which this value is small can be expected to typically also have shorter LCSs, possibly leading to an overall tighter bound. The subset of pairs of input strings for which we will compute corresponding scoring matrices, denoted by P, is determined as follows. We iterate over all pairs of input strings $\{(s_i, s_{i'}) \in S \times S \mid i < i'\}$ sorted according to $\mathrm{UB}_1(\cdot,\cdot)$ in non-decreasing order and add each string pair for which not both strings already appear in some string pair earlier added to P. In this way it is ensured that each input string is used at least once and $|P| = O(m)$. The upper bound of a given node u is then

$$\mathrm{UB}_3(u) = \mathrm{UB}_3(\mathbf{p}(u)) = \min_{(s_i, s_{i'})\in P} \mathbf{M}^3_{s_i, s_{i'}}[p_i(u), p_{i'}(u)], \tag{4.36}$$

where $\mathbf{M}^3_{i,i'}$ is the scoring matrix for string pair $(s_i, s_{i'}) \in P$.

Finally, let

$$\mathrm{UB}(u) = \min\{\mathrm{UB}_1(u),\ \mathrm{UB}_2(u),\ \mathrm{UB}_3(u)\} \tag{4.37}$$

be the strongest upper bound we can obtain.

4.5.5 A*-based Construction of Relaxed MDDs

To construct a relaxed MDD $\mathscr{D} = (V, A)$ for the LCS problem we essentially apply the A*C Algorithm 4.1 from Section 4.3 by using the state graph from Section 4.5.3. In this section we will define further LCS-specific parts of Algorithm 4.1. First, we define an appropriate node merger that merges a set of states in such a way that no feasible common subsequences are removed form the relaxed MDD. Furthermore, we will define labeling functions that selects partner nodes for merging from the open list Q and discuss some further technical details about the adopted A*C method for the LCS problem.

Merging of States

To create relaxed MDDs we have to define a state merger which computes the state of merged nodes. Let U be a set of nodes that should be merged. An appropriate state merger is

$$\oplus(U) = \left(\min_{u \in U} p_i(u) \right)_{i=1,\ldots,m}. \tag{4.38}$$

Since we take always the minimum of each position, each feasible solution of any subproblem $S[\mathbf{p}(u)]$, $u \in U$, will also be a feasible solution of the subproblem $S[\mathbf{p}(\oplus(U))]$. Hence, no feasible solution will be lost in the relaxed MDD, but new paths corresponding to infeasible solutions may emerge. We proof now the validity of the merge operator $\oplus$.

Proposition 4.5.0.1
Given a relaxed MDD constructed for the LCS according to Section 4.5.3 that complies with (i) and (ii) in Definition 4.4.1 from Section 4.4.8. When the merge operator defined in Equation (4.38) is applied to this MDD, then the resulting MDD will also comply with (i) and (ii).

Proof. A state for the LCS is a position vector $\mathbf{p}(u)$ that define a subproblem $S[\mathbf{p}(u)]$ of S by considering the substrings $s_i[p_i(u), |s_i|]$, $i, \ldots, m$. When the merge operator $\oplus(U)$ is applied to a set of nodes U, the resulting state is $(\min_{u \in U} p_i(u))_{i=1,\ldots,m}$. This merged state represents the substrings $s_i[\min_{u \in U} p_i(u), |s_i|]$, $i, \ldots, m$. Moreover, each substring $s_i[p_i(u), |s_i|]$, $u \in U$, $i = 1, \ldots, m$ is contained in substring $s_i[\min_{u \in U} p_i(u), |s_i|]$ since always the smallest position over all position vectors from U is component-wise taken. Thus, no feasible solution is lost and consequently (i) holds after the merge operation is applied. Condition (ii) follows because the longest path from $\oplus(U)$ is selected from a superset of the paths that existed before the merge and that the cost of each single arc is always one. □

Labeling Function for Collector Nodes

If $|Q|$ exceeds ϕ then nodes are selected in a pairwise fashion for merging. As already explained in Section 4.3, this must be done carefully since we have to ensure that no cycles emerge and that the open list gets empty after a finite number of expansions. Furthermore, we do not merge nodes which are already expanded since this would require to update all successor states from the expanded node onward.

To do the selection for pairing, we label each node $u \in V(\mathscr{D})$ of the MDD $\mathscr{D}$ by a labeling function $L(u)$ that maps the state $\mathbf{p}(u)$ to a simpler label of a restricted finite domain $\mathcal{D}_L$. The idea is that nodes with the same label are considered similar such that the merged state is still a reasonable representative for both nodes. Hence, we only merge nodes with the same label. Moreover, labels are chosen in such way that no cycles will emerge through merging and the open list gets empty in a finite number of steps.

To efficiently select partner nodes for merging we use a global set of so-called collector nodes V^{c}. As long as Q is too large, nodes that are not yet expanded are selected from it in increasing Z^{lp}-order. If for a selected node already a collector node V^{c} with the same label exists then the two nodes get merged s.t. all incoming arcs from the two nodes will be redirected to the new merged node. The two original nodes are removed from Q and V^{c} and the new merged node is integrated into $V(\mathscr{D})$ and becomes a new collector node in V^{c}. For more details we refer to Algorithm 4.2 in Section 4.3.

Static Labeling Function. For the LCS problem we label all nodes $u \in Q$ by

$$L^1(u) = (p_{i_{\mathrm{lcs1}}}(u), p_{i_{\mathrm{lcs2}}}(u)) \tag{4.39}$$

where $\{s_{i_{\mathrm{lcs1}}}, s_{i_{\mathrm{lcs2}}}\} \in S$ is a pair of input strings with smallest $\mathbf{M}^3_{s_i,s_{i'}}[0,0]$ over all $(s_i, s_{i'}) \in P$. Hence, we merge only nodes whose states have the same positions in strings $s_{i_{\mathrm{lcs1}}}$ and $s_{i_{\mathrm{lcs2}}}$ and thus partially represent the same subproblems. Consequently, the longest path in the relaxed MDD will never be worse than the upper bound obtained from the corresponding scoring matrix and each path originating from $\mathbf{r}$ will be a feasible common subsequence w.r.t. input strings $\{s_{i_{\mathrm{lcs1}}}, s_{i_{\mathrm{lcs2}}}\} \subseteq S$. Since any merged node will have the same values for $p_{i_{\mathrm{lcs1}}}$ and $p_{i_{\mathrm{lcs2}}}$ as the original nodes, and each transition from a state to a corresponding successor state increases the values from $p_{i_{\mathrm{lcs1}}}$ and $p_{i_{\mathrm{lcs2}}}$, the values $p_{i_{\mathrm{lcs1}}}$ and $p_{i_{\mathrm{lcs2}}}$ strictly increase along each path in the relaxed MDD. Consequently, no cycles can occur and the open list gets empty within a finite number of iterations.

Dynamic Labeling Function. To derive stronger relaxed MDDs we investigate further the static labeling function

$$L^2(u) = (p_{i_{\mathrm{lcs1}}}(u), p_{i_{\mathrm{lcs2}}}(u), p_{i_{\mathrm{lcs3}}}(u), p_{i_{\mathrm{lcs4}}}(u)) \tag{4.40}$$

where $\{s_{i_{\mathrm{lcs3}}}, s_{i_{\mathrm{lcs4}}}\} \in S$ is the additional pair of input strings with smallest $\mathbf{M}^3_{s_i,s_{i'}}[0,0]$ over all $(s_i, s_{i'}) \in P \setminus \{(i_{\mathrm{lcs1}}, i_{\mathrm{lcs2}})\}$. Note that the convergence speed of A*C depends on the size of the domain $|\mathcal{D}_L|$ of the used labeling function L. If the domain size is large then nodes can be grouped into many subgroups and it may be harder to keep the open list size under the desired threshold value ϕ since there are fewer possibilities to merge nodes. If the domain size is small then nodes are merged more aggressively, which makes it easier to keep the open list size under ϕ. However, the finally compiled relaxed MDD will in general be weaker than a relaxed MDD compiled with a labeling function of a larger domain size. For L^1 the domain size is $|\mathcal{D}_{L^1}| = \sum_{a\in\Sigma} |s_{i_{\mathrm{lcs1}}}|_a \, |s_{i_{\mathrm{lcs2}}}|_a$. Preliminary results showed that the domain size of L^2 is already too large to let A*C finish in reasonable time on our benchmark instances, but the obtained relaxed MDDs have the potential to be stronger than relaxed MDDs compiled with L^1. Therefore we follow a different strategy: Instead of using a labeling function that is static over the whole compilation process we use a function that adapts its domain size depending on the current situation. We propose the labeling function

$$L^2_\Delta(u) = (p_{i_{\mathrm{lcs1}}}(u),\, p_{i_{\mathrm{lcs2}}}(u),\, \lfloor p_{i_{\mathrm{lcs3}}}(u)/\Delta \rfloor,\, \lfloor p_{i_{\mathrm{lcs4}}}(u)/\Delta \rfloor) \tag{4.41}$$

which discretizes the values for $p_{i_{\mathrm{lcs3}}}$ and $p_{i_{\mathrm{lcs4}}}$ by discretization factor Δ. A*C starts with $\Delta = 1$ and doubles this parameter after every k consecutive failures of reducing the open list size below ϕ. If the open list size could be reduced to size ϕ then Δ is reset to one. Each time Δ is adapted, the set of collector nodes V^c is cleared.

Further Details

Similar for relaxed MDDs for the PCJSOCMSR in Section 4.4.8, we merge an already expanded node $u \in V(\mathscr{D})$ and a not yet expanded node $v \in Q$ if $\mathbf{p}(v) \oplus \mathbf{p}(u) = \mathbf{p}(u)$, $Z^{\mathrm{lp}}(v) \leq Z^{\mathrm{lp}}(u)$, and $L^1(u) = L^1(v)$ holds since we do not need to update the state of node u. This is efficiently done by indexing each expanded node by labeling function L^1 and checking the condition after each node expansion for each newly created node

4.5.6 Results

To test our approaches we use six benchmark sets from the literature.

BL instance set from Blum and Festa [27]: Consists of 450 instances grouped by different values for m, n, and $|\Sigma|$. For each combination there are ten uniform random instances.

Rat, Virus, and Random instance set from Shyu and Tsai [143]: Three benchmark sets consisting of 20 instances each. The Rat and Virus benchmark sets have a biological background whereas instances of the Random benchmark sets are randomly generated.

ES instance set from Easton and Singireddy [53]: Consists of 600 instances which are grouped by different values for m, n and $|\Sigma|$, where each group includes 50 instances.

BB instance set from Blum and Blesa [22]: Consists of 800 instances that were artificially generated in a way such that input strings have a large similarity to each other. There are ten instances for each combination of m and $|\Sigma|$.

We used all of these instances for the experimental evaluation but report here only some due to the lack of space. In particular, the main results table to come in the following section omits data for sets Virus and Random since they are similar to those obtained for Rat, and from set BL only instances with $n = 100$ are considered as also done in [51]. However, all instances from all mentioned benchmark sets are considered in all the boxplots to come. Complete results over all benchmark instances are available from `https://www.ac.tuwien.ac.at/files/resources/results/LCS/cpaior21_mdds.zip`. The algorithms were implemented using GNU C++ 7.5.0, and all experiments were performed on a single core of an Intel Xeon E5649 with 2.53 GHz and 32 GB RAM.

To evaluate the A*C algorithm we use the standard top-down compilation (TDC) as baseline, which compiles a relaxed MDD $\mathscr{D}$ layer by layer starting with the root node $\mathbf{r}$.

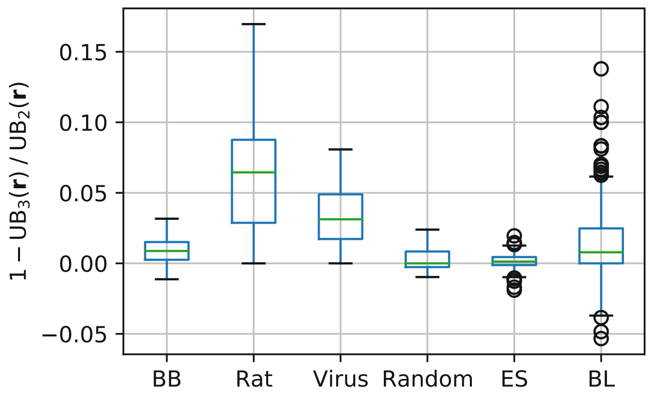


Figure 4.10: Relative differences of upper bounds $\mathrm{UB}_2(\mathbf{r})$ and $\mathrm{UB}_3(\mathbf{r})$.

All nodes of the current layer $V_i(\mathscr{D})$, $i = 1, \ldots, n$, are expanded and the newly created nodes are inserted into layer $V_{i+1}(\mathscr{D})$. The layer size is limited by parameter β. If the size of $V_{i+1}(\mathscr{D})$ exceeds β then $V_{i+1}(\mathscr{D})$ is reduced, after all nodes of $V_i(\mathscr{D})$ are expanded, by sorting the nodes according to priority function f_* in non-increasing order and replacing the last nodes with smallest f_*-values from position β onward into a single merged node. Note that TDC in general yields a MDD with be multiple target nodes at different layers. In this case the notation $Z^{\mathrm{lp}}(\mathbf{t})$ refers to the length of the longest path from $\mathbf{r}$ to any target node.

Comparison of Independent Upper Bounds

We start with a comparison of the upper bounds $\mathrm{UB}_2(\mathbf{r})$ and $\mathrm{UB}_3(\mathbf{r})$ from Section 4.5.4. Figure 4.10 shows boxplots for the relative differences $1 - \mathrm{UB}_3(\mathbf{r})/\mathrm{UB}_2(\mathbf{r})$ over the different benchmark sets. Over all instances, tighter upper bounds can be obtained from $\mathrm{UB}_3(\mathbf{r})$ than from $\mathrm{UB}_2(\mathbf{r})$ in 62.2% of the cases, and in these the relative difference is on average 1.6%. Both upper bounds are equal in 17.6% of all instances. Overall, upper bound $\mathrm{UB}_3(\mathbf{r})$ has on average a relative difference to $\mathrm{UB}_2(\mathbf{r})$ of 0.9%. However, differences vary significantly with the type of benchmarks as the figure shows. The largest relative differences could be observed on benchmark sets Rat and Virus. For randomly generated instances, the relative differences seems to be smaller in general. Overall, we conclude that UB_3 provides in general slightly tighter upper bounds than UB_2 but does not dominate it. As both bounding procedures are relatively fast, we conclude that their joint application makes sense.

4.5.7 Impact of Parameters ϕ and β

Next we investigate the impact of parameter ϕ as well as the choice of the labeling function on the quality of the obtained relaxed MDDs. For this purpose we compile MDDs for middle size instances from benchmark set BB with $m = 100$, $n = 1000$, and

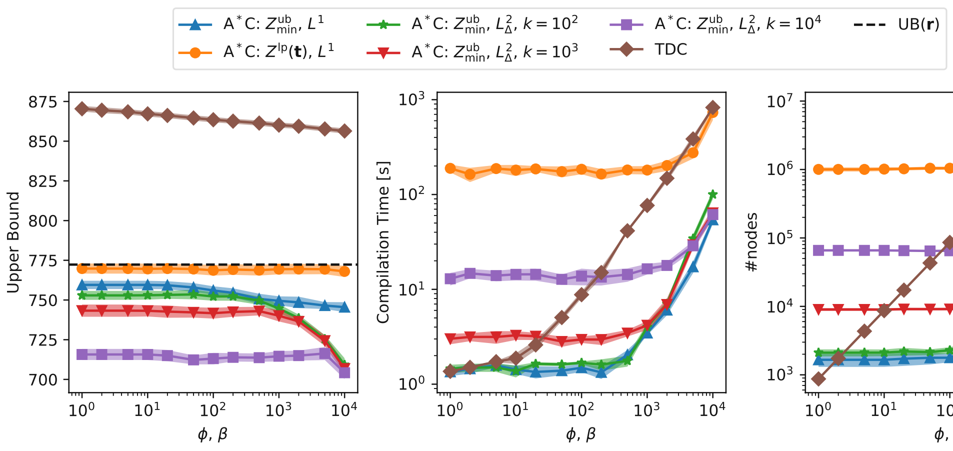


Figure 4.11: Relaxed MDDs obtained by A*C and TDC for different settings of ϕ and β for benchmark set BB, $n = 1000$, $m = 100$, $|\Sigma| = 8$.

$|\Sigma| = 8$. Figure 4.11 depicts aggregated characteristics of the relaxed MDDs created by A*C and TDC, respectively. The diagram to the left shows obtained upper bounds, i.e., average lengths of longest **r**-**t** paths, for different values of ϕ and β in the range of 1 to 10^4. The different solid lines represent different choices of labeling functions for A*C as well as results obtained from TDC. The small tubes around the lines indicate corresponding standard deviations. For A*C we generally report the upper bound values Z^{ub}_{min} obtained when **t** was selected the first time for expansion, and in case of labeling function L^1 additionally the longest path lengths in the complete relaxed MDDs. The dashed line indicates the combined bound UB(**r**) from Section 4.5.4. The diagrams in the middle and to the right report the corresponding average computation times in seconds and average numbers of nodes of the relaxed MDDs, respectively.

In general, we can observe that tighter upper bounds can be obtained when choosing larger values for ϕ or β. Naturally, this comes at the cost of larger compilation times and lager relaxed MDDs. In comparison to TDC, A*C provides consistently much better results in terms of tightness of obtained upper bounds and for larger values of ϕ and β also in terms of compilation time and compactness of obtained relaxed MDDs. A*C with the dynamic labeling function L^2_Δ yields stronger bounds than with L^1, requires, however, more time than L^1. This is not surprising since domain $\mathcal{D}_{L^2_\Delta}$ is larger than $\mathcal{D}_{L^1}$ and thus leads less frequently to merges. The tightest upper bounds can be obtained with function L^2_Δ where the discretization factor Δ is doubled after every $k = 10^4$ consecutive failures of reducing Q below ϕ. Again this can be explained due to less merges than with other parameter settings. For the same reason these settings need in general more computation time and produce larger relaxed MDDs. Note also that, even for small values of ϕ, upper bounds obtained from A*C are substantially smaller than UB(**r**).

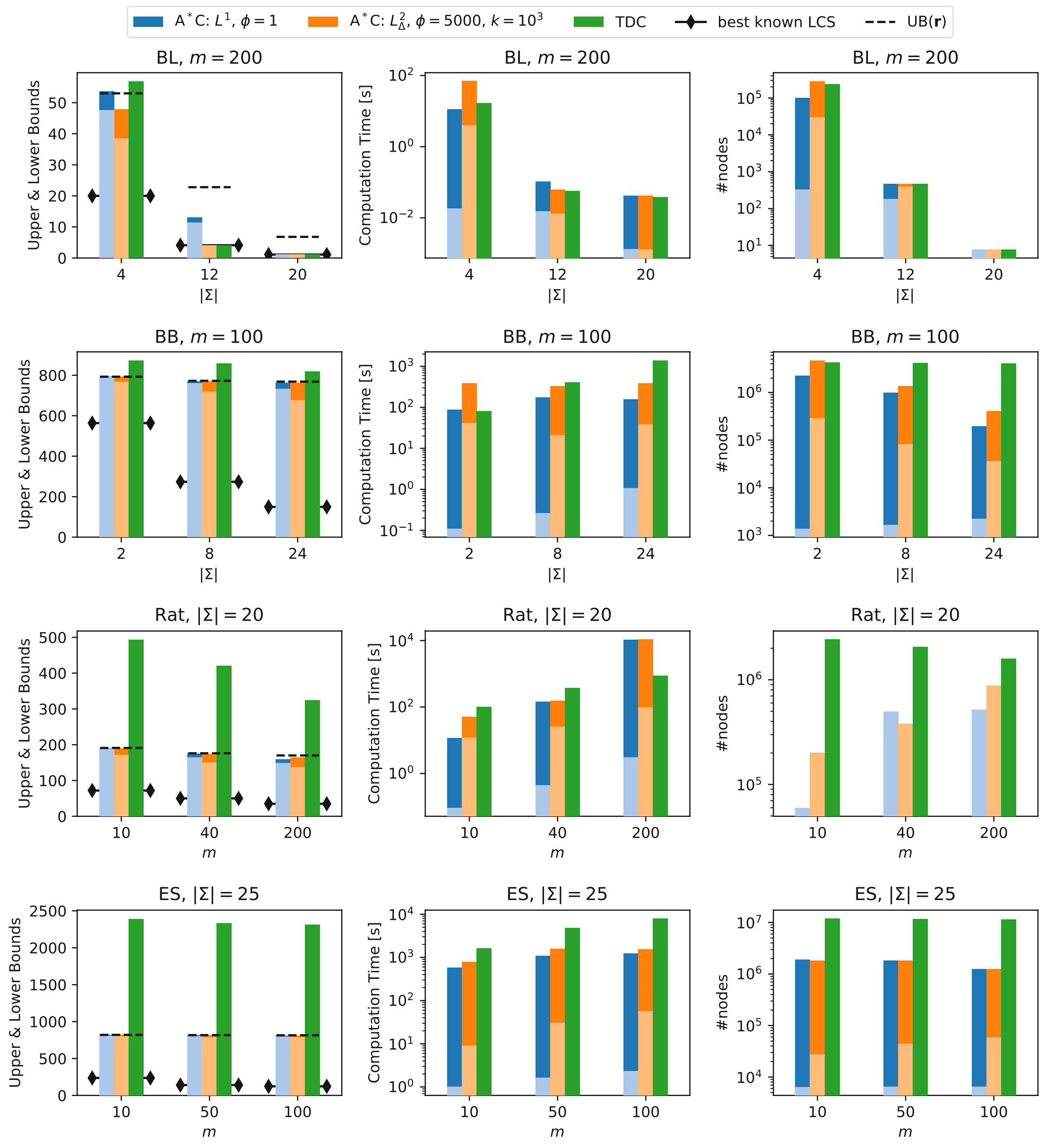


Figure 4.12: Lower and upper bounds, respective compilation times, and sizes of obtained relaxed MDDs for selected benchmark sets.

4.5.8 Main Comparison of A*C and TDC

We start with a graphical comparison for a selected subset of instance classes in Figure 4.12. Shown are upper bounds obtained from relaxed MDDs compiled with A*C and TDC, respectively, corresponding compilation times, and the sizes of the obtained MDDs. Each group of bars corresponds to a specific instance class and shows average results, except for instance class Rat which contains only one instance per instance class. The first two bars from the left to right always correspond to relaxed MDDs obtained from A*C with parameters $\{L^1,\ \phi = 1\}$ and $\{L^2_\Delta,\ \phi = 5000,\ k = 10^3\}$, respectively. The first parameter setting is the case where A*C merges nodes most aggressively whereas the latter setting lets A*C select nodes for merging more carefully, but still with a reasonable total compilation time. The third bar corresponds to relaxed MDDs obtained from TDC with $\beta = 5000$. The brighter parts of the bars indicate the results for the in general incomplete relaxed MDDs obtained when A*C terminates as soon as $\mathbf{t}$ is selected for expansion whereas the darker parts show the results for the completed relaxed MDDs. For instance, the brighter part of the bars in the diagrams on the left side show average $Z^{\text{ub}}_{\min}$ values. Diamond markers indicate the average lengths of the best known LCSs from the literature obtained from [51]. The black dashed lines show the independent upper bounds $\text{UB}(\mathbf{r})$.

We can see that if A*C terminates as soon as $\mathbf{t}$ is selected for expansion then we obtain in all considered cases MDDs yielding significantly tighter bounds than the MDDs obtained from TDC. Moreover, compilation times are shorter and the obtained MDDs are smaller in case of A*C. Note that although these relaxed MDDs are incomplete in the sense that not all feasible solutions are covered, they can still be further used, e.g., for the DD-based branch-and-bound approach as described by Bergman et al. [15]. It is still possible to derive an exact cut set of nodes to branch on by considering nodes that are not expanded yet, too. If we consider complete relaxed MDDs from A*C then the obtained upper bounds are still tighter or equal than those from relaxed MDDs obtained from TDC, however the compilation with A*C is not faster anymore. Note that TDC was not able to compile relaxed MDDs with $\beta = 5000$ within the time limit of three hours for instances from set ES with $n = 5000$. Also, the A*C approach could not compile a complete relaxed MDD for instances of set Rat with $m = 200$, $|\Sigma| = 20$, and $n = 600$ within the three hours time limit. However, with the stopping condition of selecting $\mathbf{t}$ for expansion, A*C terminated much earlier. As the length of the longest path of the incomplete relaxed MDD when A*C aborts after three hours is also a feasible upper bound, we show these values in these cases, too.

Finally, Table 4.4 presents more detailed main results of our computational experiments. Here, A*C is always terminated when $\mathbf{t}$ is selected for expansion. Each row contains aggregated results of one instance class. The characteristics of the instance classes can be seen in the first four columns whereas column $\text{UB}(\mathbf{r})$ shows the average independent upper bound. The next eight columns belong to results obtained from relaxed MDDs compiled with A*C and TDC, respectively. Hereby, columns $\overline{Z^{\text{ub}}_{\min}}$ and $\overline{Z^{\text{lp}}(\mathbf{t})}$ state the average lengths of the longest paths obtained from the compiled MDDs. Columns $\sigma(\cdot)$ report corresponding standard deviations. Average compilation times in seconds are listed in columns t. Finally, columns gap report the remaining optimality gaps $(\text{ub} - \text{obj})/\text{ub} \cdot 100\%$ in relation to the objective values of so far best known solutions obtained from [51] and listed in column obj; value ub refers to the upper bound obtained from the considered approach, i.e., $Z^{\text{ub}}_{\min}$ or Z^{lp}. We remark that [51] shows experimental

Table 4.4: Main results for A*C and TDC and comparison to the anytime A* search from [51].

					A*C				TDC				lit. best [51]	
	n	$\|\Sigma\|$	m	UB($\mathbf{r}$)	$\overline{Z^{\text{ub}}_{\text{min}}}$	$\sigma(\overline{Z^{\text{ub}}_{\text{min}}})$	t[s]	gap[%]	$\overline{Z^{\text{lp}}(\mathbf{t})}$	$\sigma(\overline{Z^{\text{lp}}(\mathbf{t})})$	t[s]	gap[%]	obj	gap[%]
BB	1000	2	10	807.4	**781.6**	9.1	8.9	**13.4**	882.7	4.4	18.7	23.3	676.7	16.2
			100	792.7	**767.3**	4.5	43.2	**26.5**	871.8	4.3	74.2	35.4	563.6	30.6
		4	10	796.5	**759.5**	6.7	6.4	**28.2**	879.5	4.1	27.7	38.0	545.5	29.4
			100	779.0	**739.4**	8.2	22.5	**47.2**	868.2	4.7	181.3	55.1	390.2	50.9
		8	10	794.8	**732.8**	11.3	7.7	**36.9**	874.9	5.7	45.5	47.1	462.7	38.0
			100	772.3	**708.2**	5.3	23.1	**61.4**	857.6	3.3	386.4	68.1	273.4	65.0
		24	10	786.1	**689.1**	14.5	12.5	44.0	846.9	3.5	131.8	54.5	385.6	**40.5**
			100	768.4	**669.8**	9.9	42.0	**77.7**	818.3	1.8	1261.4	81.7	149.5	79.5
Rat	600	4	10	345.0	**319.0**	-	4.8	**35.4**	570.0	-	27.7	63.9	206.0	38.0
			15	347.0	**331.0**	-	5.2	**42.9**	564.0	-	20.6	66.5	189.0	44.5
			20	293.0	**277.0**	-	9.4	**37.2**	494.0	-	34.4	64.8	174.0	39.5
			25	344.0	**327.0**	-	5.4	**47.1**	557.0	-	44.1	68.9	173.0	47.4
			40	315.0	**300.0**	-	10.6	48.7	455.0	-	26.2	66.2	154.0	**48.1**
			60	343.0	**323.0**	-	5.1	**52.3**	548.0	-	62.5	71.9	154.0	53.1
			80	281.0	**261.0**	-	12.6	**44.8**	466.0	-	36.6	69.1	144.0	47.6
			100	279.0	**263.0**	-	5.5	**47.1**	497.0	-	118.5	72.0	139.0	49.6
			150	**222.0**	222.0	-	13.2	41.0	443.0	-	142.3	70.4	131.0	**40.2**
			200	231.0	**228.0**	-	40.3	**44.7**	436.0	-	223.7	71.1	126.0	44.9
		20	10	191.0	**167.0**	-	19.6	**56.9**	493.0	-	101.3	85.4	72.0	58.7
			15	198.0	**169.0**	-	45.1	**62.7**	467.0	-	268.2	86.5	63.0	62.9
			20	190.0	**159.0**	-	101.7	65.4	456.0	-	278.2	87.9	55.0	**65.2**
			25	173.0	**145.0**	-	18.5	**64.1**	417.0	-	158.6	87.5	52.0	68.1
			40	176.0	**143.0**	-	53.0	**65.0**	421.0	-	379.6	88.1	50.0	70.3
			60	195.0	**161.0**	-	439.7	70.8	431.0	-	284.2	89.1	47.0	**70.3**
			80	180.0	**145.0**	-	518.7	69.7	376.0	-	269.5	88.3	44.0	**69.1**
			100	173.0	**138.0**	-	103.5	**71.0**	359.0	-	545.3	88.9	40.0	71.8
			150	172.0	**145.0**	-	128.0	73.8	323.0	-	609.6	88.2	38.0	**71.5**
			200	170.0	**133.0**	-	195.7	73.7	324.0	-	897.6	89.2	35.0	**70.2**
ES	1000	2	10	795.3	**783.6**	4.3	5.6	**21.0**	987.5	1.3	19.7	37.3	618.9	21.2
			50	791.0	**779.4**	3.0	12.8	**30.6**	982.7	1.2	40.8	45.0	540.9	**30.6**
			100	788.7	**777.3**	3.0	18.4	**32.8**	980.8	0.9	77.6	46.8	522.1	32.9
		10	10	477.6	**462.2**	2.9	4.9	55.6	951.8	2.7	138.4	78.5	205.0	**54.9**
			50	473.7	**455.7**	1.8	15.4	69.8	928.7	2.1	339.4	85.2	137.5	**69.1**
			100	472.2	**454.0**	2.0	28.9	72.7	919.5	2.1	591.8	86.5	124.1	**71.9**
	2500	25	10	820.1	**800.1**	2.4	11.5	70.4	2389.2	4.3	1453.7	90.1	236.6	**70.1**
			50	816.5	**791.0**	1.7	39.1	82.3	2332.4	4.5	4367.0	94.0	140.4	**81.9**
			100	814.4	**788.3**	1.4	74.2	84.3	2309.5	3.6	7514.3	94.7	123.4	**84.0**
	5000	100	10	888.3	**853.9**	2.6	62.7	**82.9**	-	-	-	-	145.7	**82.9**
			50	883.5	**835.9**	1.7	152.1	91.4	-	-	-	-	72.0	**91.3**
			100	882.3	**829.5**	1.6	373.3	92.7	-	-	-	-	60.8	**92.6**
BL	100	4	10	58.8	**47.5**	1.6	0.5	28.2	75.6	2.0	3.0	54.9	34.1	**10.8**
			50	56.2	**41.7**	1.4	2.1	42.0	65.0	1.2	6.1	62.8	24.2	**18.7**
			100	54.7	**40.6**	1.1	3.2	45.8	61.0	1.8	9.6	63.9	22.0	**20.4**
			150	53.8	**38.7**	1.2	3.9	46.8	58.0	1.4	11.6	64.5	20.6	**18.1**
			200	53.0	**38.3**	0.8	5.0	47.8	56.8	1.8	15.9	64.8	20.0	**20.2**
		12	10	37.4	**21.2**	1.7	0.2	40.1	36.3	4.1	3.7	65.0	12.7	**0.0**
			50	34.4	**8.7**	2.1	0.2	20.7	9.6	3.0	0.3	28.1	6.9	**0.0**
			100	28.8	**5.2**	0.4	<0.1	**0.0**	**5.2**	0.4	<0.1	**0.0**	5.2	**0.0**
			150	23.8	**4.7**	0.5	<0.1	**0.0**	**4.7**	0.5	<0.1	**0.0**	4.7	**0.0**
			200	22.8	**4.1**	0.3	<0.1	**0.0**	**4.1**	0.3	<0.1	**0.0**	4.1	**0.0**
		20	10	29.2	**9.5**	1.0	<0.1	16.8	10.5	2.2	0.3	24.8	7.9	**0.0**
			50	17.5	**3.0**	0.0	<0.1	**0.0**	**3.0**	0.0	<0.1	**0.0**	3.0	**0.0**
			100	12.1	**2.1**	0.3	<0.1	**0.0**	**2.1**	0.3	<0.1	**0.0**	2.1	**0.0**
			150	7.2	**1.9**	0.3	<0.1	**0.0**	**1.9**	0.3	<0.1	**0.0**	1.9	**0.0**
			200	6.8	**1.1**	0.3	<0.1	**0.0**	**1.1**	0.3	<0.1	**0.0**	1.1	**0.0**

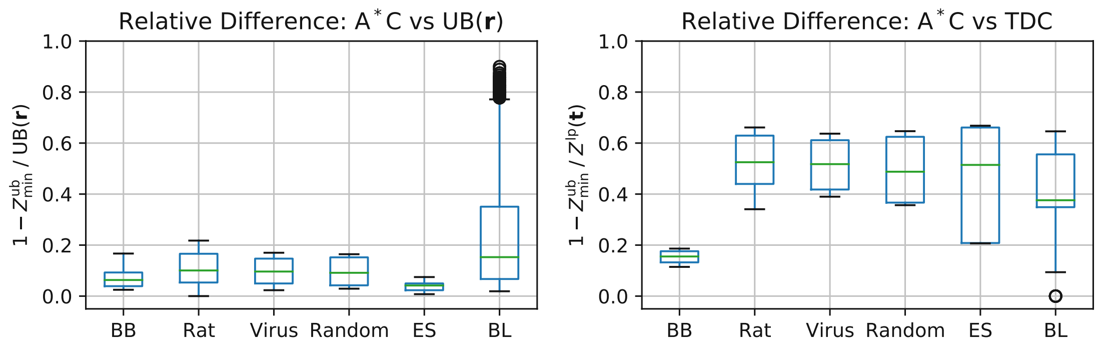


Figure 4.13: Relative differences of upper bounds between $Z^{\mathrm{ub}}_{\min}$ and $\mathrm{UB}(\mathbf{r})$ as well as between $Z^{\mathrm{ub}}_{\min}$ and $Z^{\mathrm{lp}}(\mathbf{t})$.

results for two parameter settings, one tailored to obtain as good as possible heuristic solutions, and one targeted towards smallest possible remaining optimality gaps. While we use the better objective values from the former results, the gaps listed in our table for [51] are those of the latter.

For the compilation of MDDs we set $\beta = 5000$ for TDC and $\phi = 5000$ for A*C with labeling function L^2_Δ and $k = 10^4$ for all instance except for benchmark set ES where k is set to 10^3.

We observe that in all considered cases the obtained upper bounds $Z^{\mathrm{ub}}_{\min}$ are tighter than $\mathrm{UB}(\mathbf{r})$ as well as the upper bounds obtained from relaxed MDDs compiled with TDC. Only in one single case, for benchmark set Rat with $|\Sigma| = 4$, $m = 150$, $n = 600$, the upper bound $\mathrm{UB}(\mathbf{r})$ is equal to $Z^{\mathrm{ub}}_{\min}$. We notice an average relative difference between $Z^{\mathrm{ub}}_{\min}$ and $\mathrm{UB}(\mathbf{r})$ of 14.8% over all instances. Considering $Z^{\mathrm{ub}}_{\min}$ and $Z^{\mathrm{lp}}(\mathbf{t})$ from relaxed MDDs compiled with TDC we get an average relative difference of 43.7%. The boxplots shown in Figure 4.13 give deeper insight on the relative differences between $Z^{\mathrm{ub}}_{\min}$ and $\mathrm{UB}(\mathbf{r})$ as well as the differences between $Z^{\mathrm{ub}}_{\min}$ and $Z^{\mathrm{lp}}(\mathbf{t})$. The largest relative difference between upper bounds obtained from relaxed MDDs compiled by A*C and TDC occurs for instance sets Rat, Virus, Random, and ES. For these benchmark sets the median of the obtained relative differences is about 50%. Regarding instances of the BB benchmark set, substantially smaller relative differences are obtained. The fact that BB instances are created in a way s.t. input strings have a large similarity to each other seems to be an explanation for this discrepancy. The median of the relative differences between upper bounds $Z^{\mathrm{ub}}_{\min}$ and $\mathrm{UB}(\mathbf{r})$ is about 10% for all benchmark sets. Only results from benchmark set ES exhibit a median relative difference of about 4%, which can be explained by the longer input strings of ES instances, e.g., $n = 5000$. Finally, BL instances exhibit some outliers, e.g., instances with a relative difference between $Z^{\mathrm{ub}}_{\min}$ and $\mathrm{UB}(\mathbf{r})$ of 80% and differences between $Z^{\mathrm{ub}}_{\min}$ and $Z^{\mathrm{lp}}(\mathbf{t})$ (TDC) of 0%. This is not surprising, since benchmark set BL contains small instances that could be solved to proven optimality by exact methods, and both construction methods, A*C and TDC, are able to compile relaxed MDDs that yield the optimal solution values as upper bounds. This is also documented in Table 4.4 for instance classes of set BL with $n = 100$ and $|\Sigma| \in \{12, 20\}$ where the average optimality gap is 0%. In comparison to [51], we can observe that A*C is able to obtain even smaller optimality gaps in 315 cases and equal optimality gaps in 73 cases. Most of the gaps from [51] were only obtained after a time limit of 15 minutes, while A*C created the MDDs in much shorter time.

4.6 Conclusion

In this chapter we considered two NP-hard optimization problems. The first problem is the PCJSOCMSR, a prize-collecting scheduling problem, where a subset of jobs must be selected from a ground set of jobs and sequenced to form a feasible solution. The second considered problem is the LCS problem, where a longest subsequence must be derived that is common to a set of input strings over a finite alphabet. By a simple extension, MDDs that are traditionally used for sequencing become suitable to represent the search space of the PCJSOCMSR as well as the LCS problem, where the solutions are of variable length.

By applying the principles of A* search, we proposed a new way of compiling relaxed MDDs for large instances of the PCJSOCMSR and the LCS problem that are challenging to solve to proven optimality. The suggested method has the advantage that it does not rely on a layer-to-variable correspondence, and consequently, multiple nodes for the same states at different layers are efficiently avoided. In contrast, traditional layer-oriented TDC and IR approaches would, for the PCJSOCMSR, typically lead to relaxed MDDs with a substantial amount of redundant isomorphic substructures. The same holds for relaxed MDDs for the LCS problem. Note further that also the merging of nodes is done across layers.

Moreover, our A*-based method utilizes an auxiliary heuristic function to estimate the cost-to-go from each reached node. This function guides the A* search, and thus the relaxed MDD may be constructed in a more meaningful way. As in any A* search, the better this heuristic function estimates the real cost-to-go, the more efficient the approach becomes.

We propose to restrict the number of nodes in the open list instead of restricting the width at each layer. If merging becomes necessary, a node that appears less promising to be part of a finally longest path is selected first and a similar partner node is efficiently determined by the proposed collector node concept. To this end, not yet expanded nodes are labeled in a state-space-relaxation fashion and maintained in a dictionary for efficient lookup. Choosing a proper labeling is important both to obtain a strong relaxation but also to prevent cycles in the construction of the relaxed MDD and to ensure termination of the construction. Our experiments confirmed that substantially smaller and stronger relaxed MDDs could be obtained in the same or shorter times than with traditional compilation methods.

While a relaxed MDD yields an upper bound on the optimal solution value for a maximization problem and encodes much useful information, it does in general not directly yield a promising heuristic solution and lower bound. For obtaining heuristic solutions, restricted MDDs are suitable. In previous works, they have been constructed independently of the relaxed MDD. We showed for the PCJSOCMSR how the construction of a restricted MDD can be improved by constructing a relaxed MDD first and then exploiting the encoded knowledge. Again, our experiments for the PCJSOCMSR confirmed the advantages: The main benefit is a substantial speedup in the construction of the restricted MDD. We even showed that the total time for constructing the relaxed MDD, filtering it, and deriving a restricted MDD of a certain size based on the relaxed MDD can take less time than the classical independent construction of a restricted MDD of the same size. Thus, one might say that in our combined approach, one gets the upper bound from the relaxed MDD and thus a quality guarantee in addition to a promising heuristic solution "for free".

We compared this overall approach for the PCJSOCMSR to an order-based MIP model solved by Gurobi, to a GVNS metaheuristic, and to a basic CP approach solved by MiniZinc. The MIP model only produced rather weak lower and upper bounds for all instances except the smallest. For most cases, our approach yielded the best solutions. Exceptions are the larger instances of the particle therapy benchmark set with three secondary resources, where the GVNS outperformed the other methods.

For the LCS problem we were mainly interested to obtain strong upper bounds by compiling relaxed MDDs with A*C. As auxiliary heuristic function we suggested using a combination of two fast-to-calculate bounds from the literature and the new variant UB_3 that is approximately equally fast to compute but occasionally stronger than the former bounds. We investigated also a different LCS-specific dynamic labeling function that adapts the domain dynamically during the compilation process such that depending on the current situation nodes are merged more or less aggressively. When rigorously comparing A*C with a classical TDC on several LCS-specific benchmark instance sets from the literature. We observed again that A*C is able to provide more compact relaxed MDDs that are significantly stronger than relaxed MDDs obtained from TDC in shorter time. For several instance classes relaxed MDDs compiled with A*C yielded stronger bounds than the best known upper bounds from the literature.

Naturally, it is interesting to test the proposed methods in future work also on other problems that include both the selection and sequencing aspects, like those referred to in Section 4.1. Although not a strong limitation, an important property of a suitable problem may be the order-invariance of the objective function, which is exploited by the proposed approach. Moreover, note that the idea of exploiting relaxed DDs in the construction of a successive restricted DD is more generally applicable. For some problems, the proposed way of finding similar partner nodes for merging may also be useful in the context of a classical layer-wise TDC of relaxed MDDs, where so far a simpler bulk merging is primarily used.

For some applications of relaxed DDs, an important aspect is incrementability, i.e., that a once obtained complete DD can be further refined, for example, to strengthen the obtained bound. Naturally, known iterative refinement methods based on node splitting and filtering can also directly be applied to relaxed DDs obtained from the A*C. Moreover, the A*C may be iteratively applied with increasing open list size limits, yielding stronger and stronger DDs over the time. Hereby, information contained in one DD can always be exploited to speed up the construction of a successive DD in a similar way as we derived a larger restricted DD on the basis of a relaxed DD. An interesting research question is if a completely constructed DD can also be effectively updated in-place by a following A*-based refinement pass.

Last but not least, it is of relevance to investigate the proposed A*-based MDD construction also from a more theoretical side. Unfortunately, a constant open list size limit ϕ does in general not necessarily imply that the obtained relaxed MDD has polynomial size, and therefore also the algorithm's runtime is not necessarily polynomially bounded. Note, however, that similarly no better performance guarantees can be given for classical A* search without considering a more specific problem setting and a concrete heuristic function. In fact, the resulting MDD's actual size strongly depends on the interplay of ϕ, the problem-specific heuristic function for the cost-to-go, the labeling function for the merging, and how the merging is performed.

One extension to guarantee a termination with a complete relaxed MDD when reaching a certain time limit or MDD size is to reduce ϕ to a very small value from this point onward. However, this naive completion may in general degrade the strength of the obtained MDD substantially. Studying more advanced methods, possibly by adaptively adjusting ϕ over the whole run, or developing an entirely different way of deciding when to merge which nodes, is desirable.

CHAPTER 5

Decision Diagram Based Limited Discrepancy Search

In Chapter 4 a relaxed decision diagram (DD) has been used to accelerate the compilation process of a restricted DD with the traditional top-down compilation (TDC) method. This chapter extends this approach by using structural information of previously compiled relaxed DDs to significantly speed-up a heuristic search for solutions of a combinatorial optimization problem (COP). More precisely, we will use relaxed DDs to improve the performance of a hybrid of a limited discrepancy search (LDS) and a beam search (BS) approach. In order to evaluate our algorithm we tackle an extension of the prize-collecting job sequencing problem with one common and multiple secondary resources (PCJSOCMSR) in Section 4.4 of Chapter 4 by considering in addition precedence constraints.

This chapter covers the work presented at the *17th International Conference of Computer Aided Systems Theory* (EUROCAST 2019) [80]. The remaining part of the chapter is structured as follows. First, Section 5.1 gives a detailed introduction of the considered topic. The following Section 5.2 introduces the considered PCJSOCMSR with precedence constraints and Section 5.3 describes the used compilation process to construct relaxed DDs for the PCJSOCMSR with precedence constraints. The hybrid LDS/BS approach to find heuristic solutions is described in Section 5.4. Experimental results, which demonstrate a substantial speed-up of the computation times of our hybrid approach when using a relaxed DD are presented in Section 5.5. Furthermore, the obtained results are compared with a mixed integer programming (MIP) approach and a constraint programming (CP) approach. Finally, Section 5.6 concludes this chapter.

5.1 Introduction

The PCJSOCMSR without precedence constraints was introduced in Section 4.4 of Chapter 4 and consists of a set of jobs, one common resource, and a set of secondary resources. The common resource is shared by all jobs whereas a secondary resource is shared only by a subset of jobs. Each job has at least one time window and is associated with a prize. A feasible schedule requires that there is no resource used by more than one job at the same time and each job is scheduled within one of its time windows. Due to the time windows it may not be possible to schedule all jobs. The task is to find a subset of jobs that can be feasible scheduled and maximizes the total prize. As already mentioned in Chapter 4, there are at least two applications. The first is in the field of the daily scheduling of particle therapies for cancer treatments. The second application can be found in the field of hard real time scheduling of electronics within an aircraft, called avionics, where the PCJSOCMSR appears as a subproblem. See Section 4.4 for more details.

In particular in the avionic system scenario it often appears that some jobs need to be finished before other jobs may start. To address this aspect, we consider in this work also *precedence constraints*. Thus, there are given relationships between pairs of jobs as additional input such that one job can only be scheduled if the other job is already completely scheduled earlier. These new constraints require an adaption on the algorithmic side of Section 4.4 to incorporate the new precedence constraints. The goal is to solve large problem instances of the PCJSOCMSR with precedence constraints heuristically. Our solution approach builds upon the ideas from Section 4.4 but extended them to a LDS combined with BS that exploits structural information contained in a relaxed DD. The usage of the relaxed DD is two-folded: (1) to reduce computation time of the LDS and (2) to provide besides a heuristic solution also an upper bound on the total prize objective.

5.2 Prize-Collecting Job Sequencing with One Common and Multiple Secondary Resources Problem with Precedence Constraints

The PCJSOCMSR with precedence constraints is formally defined as follows. Given is a set of n jobs $J = \{1, \ldots, n\}$, a common resource 0 and a set of m secondary resources $R = \{1, \ldots, m\}$. Let $R_0 = \{0\} \cup R$ be the complete set of resources. Each job $j \in J$ needs during its whole execution time $p_j > 0$ one secondary resource $q_j \in R$ and, in addition, after some preprocessing time $p_j^{\mathrm{pre}} \geq 0$ also the common resource 0 for some time $p_j^0 > 0$. Furthermore, each job has associated (1) ω_j time windows $W_j = \bigcup_{\omega=1,\ldots,\omega_j} [w_{j\omega}^{\mathrm{start}}, w_{j\omega}^{\mathrm{end}}]$, where $w_{j\omega}^{\mathrm{end}} - w_{j\omega}^{\mathrm{start}} \geq p_j$, $\omega = 1, \ldots, \omega_j$, (2) a set of preceding jobs Γ_j, which must be scheduled before job j can be scheduled w.r.t. the common resource 0, and (3) a prize $z_j > 0$. The task is to find a subset of jobs $S \subseteq J$ which can be feasible scheduled such that the total prize of these jobs is maximized:

$$Z^* = \max_{S \subseteq J} Z(S) = \max_{S \subseteq J} \sum_{j \in S} z_j. \tag{5.1}$$

A feasible schedule assigns each job in S a starting time in such a way that all constraints are satisfied. See Figure 4.3 in Section 4.4.1 for an example of a PCJSOCMSR instance and a corresponding optimal solution. Note that a unique ordered sequence $\pi = (\pi)_{i=1,\dots,|S|}$ of jobs is implied by each feasible schedule of jobs $S \subseteq J$, since the common resource is required by each job and only one job can use this resource at a time. For each given sequence π of jobs S that can be associated with a feasible schedule, a *normalized schedule* without unnecessary waiting times can be computed greedily (see Section 4.4.1 for further details).

For related work of the PCJSOCMSR we refer to Section 3.2 and 4.4.2 of Chapters 3 and 4, respectively.

5.3 Relaxed Decision Diagrams and Filtering

In order to describe our approach in Section 5.4 we have to introduce some definition and structures beforehand. In our context a multivalued decision diagram (MDD) for the PCJSOCMSR is a weighted directed acyclic graph $\mathscr{D} = (V, A)$ with one root node $\mathbf{r} \in V(\mathscr{D})$, corresponding to the empty schedule and one target node $\mathbf{t} \in V(\mathscr{D})$ corresponding to all feasible schedules that cannot be further extended by any job. Each arc $\alpha = (u, v) \in A(\mathscr{D})$ corresponds to adding a specific job, denoted by $\mathrm{j}(\alpha) \in J$, as the next job after the ones already scheduled up to node u. The length of an arc $\alpha \in A(\mathscr{D})$ is associated with the prize $z_{\mathrm{j}}(\alpha)$. Hence, each path from $\mathbf{r}$ to any node $u \in V(\mathscr{D})$ corresponds to a specific sequence of jobs π and the length of the path is equal to the sum of prizes of jobs in π.

In an exact MDD each feasible normalized schedule $S \subseteq J$ has a corresponding path in the exact MDD originating from $\mathbf{r}$ and vice versa. The length of such a path corresponds exactly to the total prize $Z(S)$. Therefore, a longest path from $\mathbf{r}$ to $\mathbf{t}$ corresponds to an optimal solution of the PCJSOCMSR. Furthermore, each node $u \in V(\mathscr{D})$ is associated to a state $(P(u), t(u))$, where set $P(u)$ contains all jobs that can be feasibly scheduled next, and vector $t(u) = (t_r(u))_{r \in R_0}$ contains the earliest times from which on each of the resources are available for performing a next job. The transition function to obtain the successor state $(P(v), t(v))$ of state $(P(u), t(u))$ when scheduling job $j \in P(u)$ is

$$\tau\left((P(u), t(u)), j\right) = \begin{cases} (P(u) \setminus \{j\}, t(v)), & \text{if } \mathrm{s}((P(u), t(u)), j) < T^{\max} \land \\ & \qquad P(u) \cap \Gamma_j \neq \emptyset, \\ \hat{0}, & \text{else,} \end{cases} \tag{5.2}$$

with

$$t_0(v) = \mathrm{s}((P(u), t(u)), j) + p_j^{\mathrm{pre}} + p_j^0, \tag{5.3}$$
$$t_r(v) = \mathrm{s}((P(u), t(u)), j) + p_j, \qquad \text{for } r = q_j, \tag{5.4}$$
$$t_r(v) = t_r(u), \qquad \text{for } r \in R \setminus \{q_j\}, \tag{5.5}$$

where $\hat{0}$ represents the infeasible state and $s((P(u), t(u)), j)$ corresponds to the earliest start time of job j w.r.t. state $(P(u), t(u))$ and job j's time windows. If it is not possible to feasibly schedule job j then function $s(\cdot, \cdot)$ will return $T^{\max}$. See Equation 4.4 in Section 4.4.3 for a formal definition of function $s(\cdot, \cdot)$. A state is denoted as *exact* state if a longest path from the root node $\mathbf{r}$ to the node associated with the corresponding state represents a feasible solution. Hence, a node of an exact state is guaranteed to have a feasible solution that corresponds to this longest path.

Exact MDDs contain only exact states, which is not necessarily true for relaxed MDDs since states are merged in order to get a more compact MDD. Thereby new paths will emerge which correspond to infeasible schedules, denoted as *infeasible* paths. Let us merge two nodes $u, v \in V$. The merged state is

$$(P(u), t(u)) \oplus (P(v), t(v)) = (P(u) \cup P(v), (\min(t_r(u), t_r(v)))_{r \in R_0}). \tag{5.6}$$

We compile relaxed MDDs with the A*-based construction (A*C) method from previous Section 4.4.8 of Chapter 4 since it could be shown that at least for the PCJSOCMSR without precedence constraints the A*C can produce smaller relaxed MDDs in shorter time that represent stronger relaxations than relaxed MDDs compiled with standard methods from the literature. See Figure 4.5 in Section 4.4.8 for a graphical example of an exact MDD as well as a relaxed MDD for an instance of the PCJSOCMSR. Note that we initially ignore the precedence constraints in this compilation of a relaxed MDD. Otherwise, we would need to extend the states of the nodes with additional information in order to define a feasible merging rule for two nodes. Preliminary experiments had shown that those larger states cause substantially longer compilation times, which we want to avoid. However, we consider the precedence constraint after the initial construction by applying a respective filtering on the compiled relaxed MDD. We try to identify arcs which belong only to infeasible paths. Those arcs can be safely removed from the relaxed MDD to reduce the number of infeasible paths without removing paths that correspond to feasible schedules. To identify arcs that violate precedence constraints we adopted the corresponding filter operation suggested by Cire and van Hoeve [40]. Moreover, if we already got a primal solution then we can in addition filter arcs which only belong to paths corresponding to solutions that are worse than this known primal solution. Hence, paths that encode sub-optimal solutions will be removed from the relaxed MDD. These cost-based filter operations are adopted from Section 4.4.

Algorithm 5.1: LDSprobe

```
Input: node set $N'$, relaxed DD $\mathscr{D} = (V, A)$, allowed discrepancies $k$, beam
       width $\beta$
Output: sequence of jobs $\pi$
1  if $N' = \emptyset$ then return empty sequence;
2  node set $W' \leftarrow \emptyset$; job sequence $\pi_{\text{best}} \leftarrow \emptyset$;
3  foreach $u' \in N'$ do
4      let $u \in V(\mathscr{D})$ be the node corresponding to $u'$ w.r.t. the path from the root;
5      foreach outgoing arc $\alpha = (u, v)$ of node $u$ do
6          if $|W'| = (k+1)\,\beta \wedge$ node $v$ would be removed from $W' \cup \{v\}$ then
7              continue with next arc;
8          end
9          if $\tau((P(u'), t(u'), \mathrm{j}(\alpha)) = \emptyset$ then continue with next arc;
10         add new node $v'$ to $W'$ and set $(P(v'), t(v')) \leftarrow \tau((P(u'), t(u')), \mathrm{j}(\alpha))$;
11         if $|W'| > (k+1)\beta$ then remove worst node from $W'$ according to $Z^{\mathrm{h}}(\cdot)$;
12     end
13 end
14 if $W' = \emptyset$ then return $\arg\max_{\pi(u')|u' \in N'} Z(\pi(u'))$;
15 sort $W'$ according to $Z^{\mathrm{h}}(\cdot)$ and split $W'$ into $k+1$ slices $W'[i]$, $i = 0, \ldots, k$;
16 foreach $i = k, \ldots, 0$ do
17     $\pi = \text{LDSprobe}(W'[i], M, k-i, \beta)$;
18     if $Z(\pi_{\text{best}}) < Z(\pi)$ then $\pi_{\text{best}} \leftarrow \pi$;
19 end
20 return $\pi_{\text{best}}$;
```

5.4 Limited Discrepancy Search

Section 13 of Chapter 2 already introduces LDS and its extensions. However, for the sake of clarity, we will briefly repeat here the basic LDS approach. Limited discrepancy search was originally proposed by Harvey and Ginsberg [72] for heuristic binary searches where at each decision point a heuristic $Z^{\mathrm{h}}(\cdot)$ decides between two possibilities to extend the current partial solution. If $Z^{\mathrm{h}}(\cdot)$ is a perfect heuristic then the algorithm would return the optimal solution as soon as a complete solution is encountered during the search. However, in most cases $Z^{\mathrm{h}}(\cdot)$ will fail at some point and only a non-optimal solution can be returned. To overcome this, LDS allows in a systematic way discrepancies during the search. A *discrepancy* means that at some decision point the algorithm decides against $Z^{\mathrm{h}}(\cdot)$. Hence, if k discrepancies are allowed then LDS will encounter all paths in the search tree where the algorithm exactly decides k times against $Z^{\mathrm{h}}(\cdot)$. To apply LDS on the PCJSOCMSR we have to consider in general multiple possibilities at each decision point instead of just two, and we do this by counting $i-1$ discrepancies if we take the i-th best decision according to $Z^{\mathrm{h}}(\cdot)$.

Algorithm 5.2: LDS+BS

```
  Output: sequence of jobs $\pi$
1 compile relaxed MDD $\mathscr{D} = (V, A)$ by A*C, ignoring precedence constraints;
2 $\pi_{\text{best}} \leftarrow$ LDSprobe($\mathbf{r}, \mathscr{D}, 0, 10$);
3 for $k \leftarrow 0$; $k \leq k_{\max} \wedge$ time limit not exceeded; $k \leftarrow k + 1$ do
4     apply filtering on $\mathscr{D}$;
5     $\pi \leftarrow$ LDSprobe($\mathbf{r}, \mathscr{D}, k, \beta$);
6     if $Z(\pi_{\text{best}}) < Z(\pi)$ then $\pi_{\text{best}} \leftarrow \pi$;
7 end
8 return $\pi_{\text{best}}$;
```

Algorithm 5.1 shows our LDS-based approach using the transition function (5.2) from Section 5.3 to generate exact states. Note that we do not build an exact DD, but we rather keep all not yet expanded nodes in memory and assign to each node v' the so far best encountered partial solution $\pi(v')$. Furthermore, we extend LDS in similar ways as Furcy and Koening [58] by incorporating a BS like approach at each level into LDS. Instead of expanding always one node at each step Algorithm 5.1 expands at each step β nodes and keeps the $(k+1)\beta$-best successor nodes according to $Z^{\text{h}}(\cdot)$. As heuristic decision function $Z^{\text{h}}(v')$ for node v' we use the ratio $Z(\pi(v'))/t_0(v')$. In order to quickly identify those $(k+1)\beta$-best successor nodes we use the structural information contained in the relaxed MDD $\mathscr{D} = (V, A)$ similar as in Section 4.4.9 in Chapter 4. For node $u' \in N$ a corresponding node $u \in V(\mathscr{D})$ from $\mathscr{D}$ can be determined by following the job sequence $\pi(u')$ from $\mathbf{r}$ in $\mathscr{D}$. We do not consider transitions to successor nodes of u' where the corresponding arcs were removed from the relaxed DD during the filtering step. Furthermore, we can estimate $Z^{\text{h}}(\cdot)$ without creating the successor nodes of u' by using the corresponding nodes in the relaxed MDD. Based on this estimation we can decide quickly if a successor node is a candidate to be one of the β-best successor nodes or not. Note that for simplification reasons Algorithm 5.1 shows a recursive version of LDS, our implementation however, is implemented in an iterative way.

Algorithm 5.2 gives an overview of the overall approach to tackle the PCJSOCMSR with precedence constraints. First a relaxed MDD $\mathscr{D}$ is compiled with A*C with the same parameter settings as in Section 4.4.10 in Chapter 4 and by ignoring the precedence constraints. In order to get quickly an initial primal solution for filtering, Algorithm 5.1 is applied with the small beam with $\beta = 10$ and no allowed discrepancies. In the main loop we apply first the filtering for sup-optimal paths according to our current best primal solution and precedence constraints violations. Then we apply Algorithm 5.1 with beam width β and the number of current maximum allowed discrepancies k. After updating the incumbent solution π_{best}, k is increased by one. The algorithm terminates if the maximum allowed discrepancies $k_{\max}$ is reached or a certain time limit is exceeded.

5.5 Experimental Results

The LDS-based algorithm for the PCJSOCMSR with precedence constraints was implemented in C++ using GNU g++ 5.4.1. All tests were performed on a cluster of machines with Intel Xeon E5-2640 v4 processors with 2.40 GHz in single-threaded mode with a memory limit of 16 GB per run. We extended the two sets of benchmark instances for the particle therapy application scenario (denoted as P) and for the avionic system scheduling scenario (denoted as A) from Section 4.4.10 by adding randomly precedence constraints between n pairs of jobs such that circular dependencies between jobs are avoided. The instance sets contain 30 instances for each combination of different values of n with up to 500 jobs and m with up to 4 secondary resources. For further details on the benchmark characteristics we refer to Section 4.4.10 of Chapter 4. During the LDS+BS approach, relaxed MDDs are compiled with A*C by using the labeling function L^3 and limiting the open list to $\phi = 1000$ for instances of set P. For instances of set A we use the labeling function L^4 and $\phi = 20000$.

Figure 5.1 compares the obtained average total prizes and median computation times between LDS+BS and a standalone variant of LDS+BS without using a relaxed MDD. Both algorithms use different values for the beam width β and different numbers of maximum allowed discrepancies $k_{\max}$. The diagrams on the top visualize the obtained average total prizes. There are two main observations regarding the solution quality: First, as expected the solution quality tends to increase with increasing β and/or $k_{\max}$; second, similar results could be obtained from both LDS+BS variants. Therefore, we conclude that for the considered instances the solution quality does on average not depend on using a relaxed MDD to accelerate the LDS+BS when using the same parameter settings. However, regarding computation times, the LDS+BS approach using the relaxed MDD is in almost all cases except for $k_{\max} = 0$ and smaller β substantially faster. Note that we do not show the obtained results from standalone LDS+BS for $k_{\max} = 2$, since the approach exceeded in most cases the time limit of two hours.

Figure 5.2 compares the LDS+BS approach against a mixed integer programming (MIP) approach and a constraint programming (CP) approach. The MIP formulation as well as the CP formulation from Sections 4.4.6 and 4.4.7, respectively, were adapted to additionally consider the precedence constraints. The MIP model is solved with Gurobi Optimizer 7.5.1 whereas the CP model is solved with MiniZinc 2.1.7 using backbone solver Chuffed. All tested approaches use a time limit of 900 seconds. For LDS+BS the maximum allowed discrepancies $k_{\max}$ are set to infinity and β is set to 1000 and 10000 for instance sets of type P and A, respectively. The first bar of each group of bars shows the obtained average longest path length of the compiled relaxed MDD during the LDS+BS approach and the darker block at the bottom shows the obtained average primal bounds. In the same manner, the second bar shows the obtained upper- and primal bounds from the MIP approach. The third bar shows the obtained average primal bounds obtained from the CP approach. Note that this rather standard CP approach is not able to obtain a dual bound. On average the LDS+BS approach finds in all considered cases better or equally good solutions than the MIP or the CP solvers. In almost all cases the CP solver

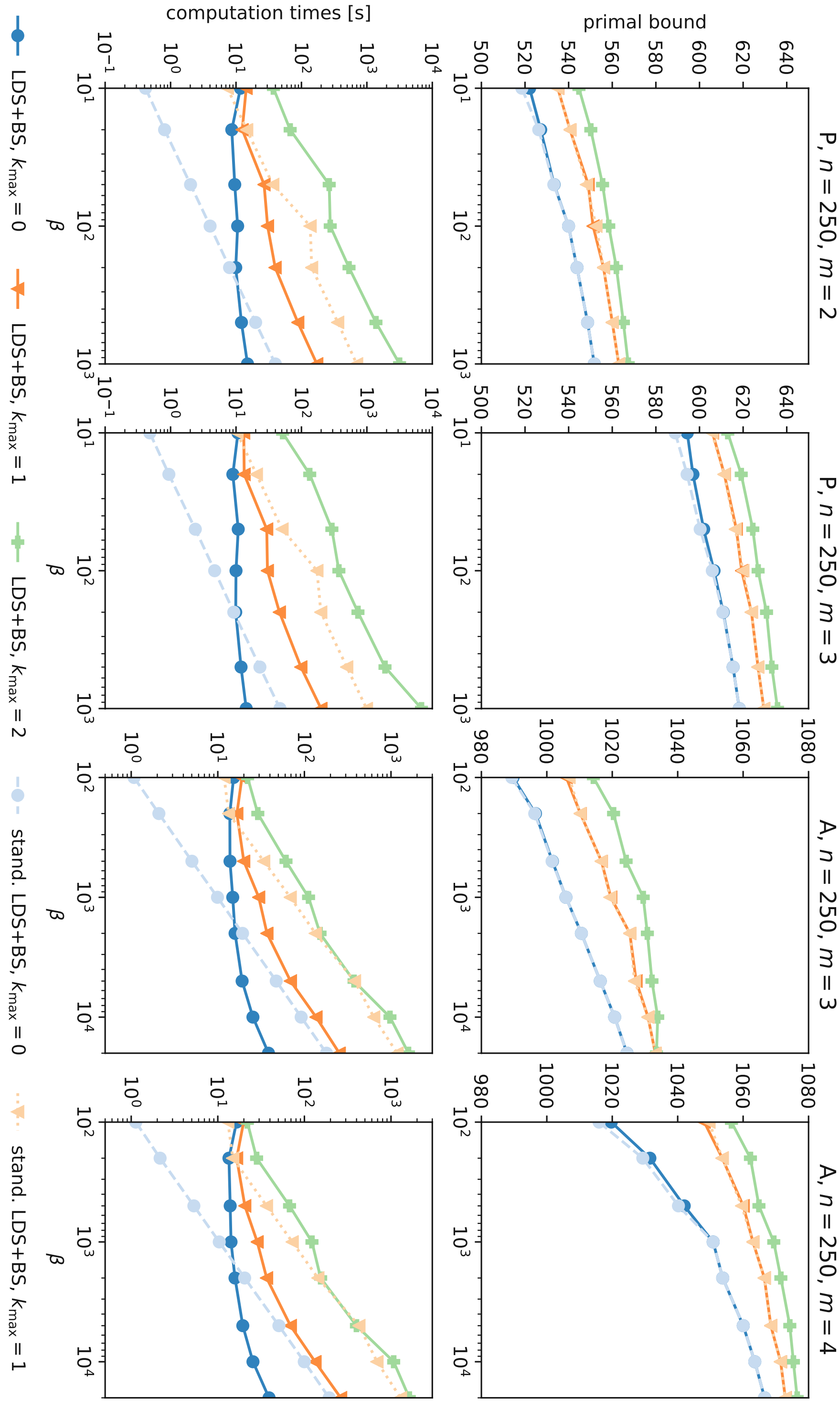


Figure 5.1: Comparison between LDS+BS and standalone LDS+BS for middle-sized instances with 250 jobs.

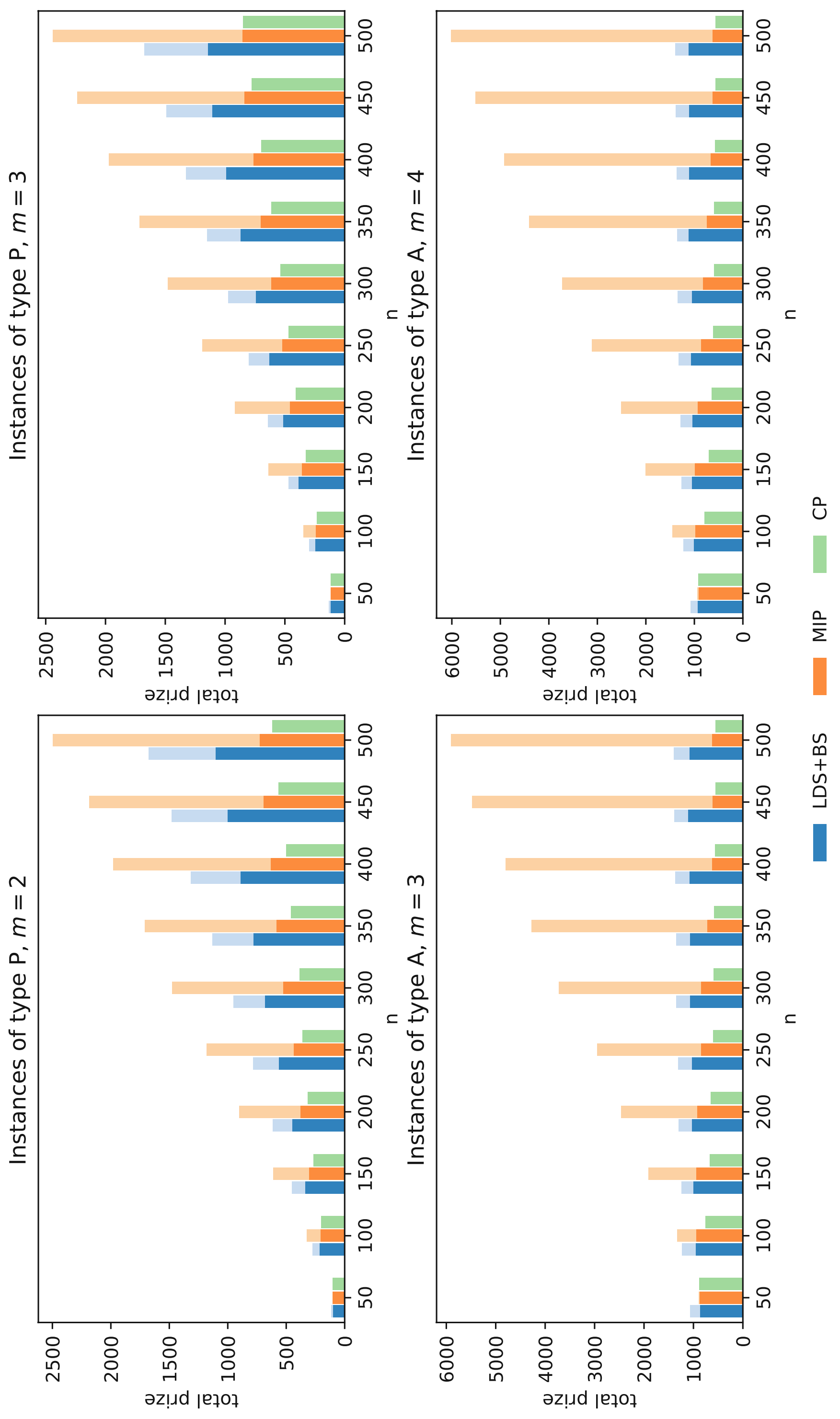


Figure 5.2: Primal and Dual Bounds obtained from LDS+BS, MIP and CP.

is not able to provide any better solutions than the MIP solver or the LDS+BS approach. Moreover, LDS+BS is able to return in most cases on average stronger upper bounds than the MIP solver.

5.6 Conclusion

In this chapter we solved the PCJSOCMSR with precedence constraints by an hybrid approach of LDS and BS. The PCJSOCMSR is a prize-collecting scheduling problem, where a subset of jobs must be selected from a ground set of jobs and sequenced to form a feasible solution. The considered precedence constraints make the problem more suitable for the avionic system scenario. Furthermore, we exploited the structural information of relaxed MDDs within the LDS+BS which has following advantages: (1) a substantial speed up of the heuristic search allows scanning larger regions of the search space compared to a standalone LDS+BS approach and (2) a dual bound can be obtained from the relaxed MDD. An exhausted experimental evaluation on two different benchmark sets with up to 500 jobs and up to 4 secondary resources confirms that LDS+BS is substantially faster than a standalone LDS+BS approach. Furthermore, we compared our hybrid approach to a MIP model and a CP model. Both models are inferior to our LDS+BS approach for large instance sizes.

Although we demonstrate these advantages specifically for the PCJSOCMSR, the general approach also appears promising for other combinatorial optimization problems. Next steps would be to incorporate other filtering techniques to further strengthen the relaxed MDD by removing more arcs to speed-up the computation times even more. Another promising research direction would be to apply the general idea of using the structural information of relaxed MDDs on further search heuristics and metaheuristics.

CHAPTER 6

Decision Diagrams for Finding Repetition-Free Longest Common Subsequences

This chapter considers the repetition-free longest common susequence (RFLCS) problem, where the goal is to find a longest sequence that appears as a subsequence in two input strings and in which each character appears at most once. Our approach is to transform a RFLCS instance to an instance of the maximum independent set (MIS) problem which is subsequently solved by a mixed integer linear programming (MILP) solver. To reduce the size of the underlying conflict graph of the MIS problem, a relaxed decision diagram is utilized. An experimental evaluation on two benchmark instance sets shows the advantages of the reduction of the conflict graphs in terms of shorter total computation times and the number of instances solved to proven optimality. A further advantage of the created relaxed decision diagrams is that heuristic solutions can be effectively derived. For some instances that could not be solved to proven optimality, new state-of-the-art results were obtained in this way. This chapter covers work presented at the *XI International Conference of Optimization and Applications* (OPTIMA 2020) [77].

The remaining part of the chapter focuses first on a more detailed introduction in Section 6.1. Afterwards we give an overview of related work in Section 6.2 and a formal problem definition in Section 6.3. The MIS problem and a corresponding integer linear programming (ILP) model are described in Section 6.4. Decision diagrams for the RFLCS are introduced in Section 6.5, and Section 6.6 describes the incremental refinement of relaxed multivalued decision diagrams (MDDs). Section 6.7 provides experimental results, showing that the suggested approach yields a performance improvement in terms of average computation times, number of instances solved to optimality, and average solution quality.

6.1 Introduction

The longest common subsequence (LCS) problem asks for the longest string which is a subsequence of a set of input strings. A subsequence is a string that can be obtained from another string by possibly deleting characters. For instance a longest common subsequence of the two input strings ABCDBA and ACBDBA is ABDBA. The LCS problem has applications in bioinformatics, where strings often represent segments of RNA or DNA [65, 87, 145]. Other fields where the LCS problem appears are text editing [104], data compression, file comparison [2, 150], and the production of circuits in field programmable gate arrays [32]. If the number of input strings m is constant, the problem is solvable by dynamic programming (DP) in $O(n^m)$ time, where n is the length of the longest input string [65]. Otherwise, if the number of input strings is arbitrary, the problem is $\mathcal{NP}$-hard. An additional constraint which arises in the context of gene duplication in the domain of genome rearrangement and which we consider in this work is that each character may appear in a common subsequence at most once. This problem, first introduced by Adi at el. [1] and denoted as the repetition-free LCS (RFLCS) problem, is usually considered for two input strings and is even then APX-hard [1].

The work presented in this chapter builds upon the work of Blum et al. [26], where instances of the RFLCS problem are transformed to instances of the maximum independent set (MIS) problem. Hereby, an independent set of the underlying conflict graph of the MIS problem corresponds to a repetition-free common subsequence of the RFLCS instance. To solve the MIS problem the ILP solver CPLEX is applied. The performance of the ILP solver depends to a large extent on the size of the conflict graph. Therefore, in [26] the size of the conflict graph is reduced by filtering redundant nodes based on lower and upper bounds. This boosts the range of instances that can be solved to optimality as well as the quality of heuristic solutions obtained for larger instances. In this way, numerous new state-of-the-art results were obtained.

6.1.1 Contributions

To reduce the size of the conflict graph even further we compile a relaxed MDD for the RFLCS problem, yielding a performance improvement of the subsequently applied ILP solver. In the last decade, decision diagrams (DDs) have been recognized as a powerful tool for combinatorial optimization problems; see Section 2.5 of Chapter 2 for more details or the text book [14] for a comprehensive survey. In particular, *relaxed DDs* may provide compact representations of discrete relaxations. Besides allowing for new inference techniques in constraint programming and novel branching schemes, they may also provide tight dual bounds. In case of the RFLCS problem it is further possible to effectively derive heuristic solutions directly from the relaxed MDD. This has the advantage that if the ILP solver is not able to solve an instance to proven optimality within a given time limit then the compiled relaxed MDD may be able to provide a tighter upper bound as the ILP solver does and/or may be able to deliver a better heuristic solution.

Furthermore, since our approach is able to derive both a dual and a primal bound by compiling a relaxed MDD, it seems natural to devise an exact branch and bound (BB) approach which is based on MDDs similar to the BB approach from Bergman et al. [15]. The BB scheme may branch over an appropriate selected subset of nodes from the relaxed MDD. In contrast to [15], however, it is not necessary to compile an additional restricted MDD for each subproblem in order to derive a primal bound, since our approach is able to directly derive a primal bound from the previously compiled relaxed MDD. This BB approach is left open for future work, since such method would be beyond the scope of this dissertation.

6.2 Related Work

As already mentioned, the current work builds upon the approach of Blum et al. [26]. Besides the RFLCS, Blum et al. also consider the longest arc-preserving common subsequence (LAPCS) problem [88], where additional dependencies among characters must be respected in a solution, as well as the longest common palindromic subsequence (LCPS) problem [38], where the resulting sequence must also be a palindrome. All these LCS variants were solved by transforming instances to instances of the MIS problem. Moreover, the equivalent maximum clique problem of the complement of the conflict graph is solved heuristically by the LSCC-BMS solver as well as exactly by the LMC solver. Both solvers are currently among the leading solvers for the maximum clique problem.

In the literature LCS related problems with additional constraints are well known for almost 40 years and research in that field is still active due the practical relevance and computational difficulties. Besides RFLCS and the already mentioned LAPCS and LCPS problems other considered variants are, for instance, the constrained longest common subsequence problem [154] or the generalized constrained longest common subsequence problem [37]. For further problem variants we refer to survey papers such as [31].

The RFLCS problem in particular was tackled by several heuristic approaches [1, 23, 35]. The best heuristic so far is a construct, merge, solve and adapt (CMSA) metaheuristic combined with beam search as proposed by Blum and Blesa [24]. The authors showed that this approach can outperform other heuristics as well as the CPLEX solver applied to an ILP model of the RFLCS problem.

Furthermore, the MIS problem has been well studied in the context of DDs, e.g. [13, 15, 17, 34, 63]. However, the existing approaches cannot be applied directly in order to compile DDs for the RFLCS problem, since the problem exhibits not only a selection aspect, but also a sequential structure as well as repetition-free constraints. Therefore we compile relaxed MDDs by explicitly using these additional properties of the RFLCS problem. For instance, by applying an adapted version of the powerful incremental refinement algorithm for sequencing problems, proposed by Cire and van Hoeve [40].

6.3 Problem Definition

The RFLCS problem considers a set of two input strings $S = \{s_1, s_2\}$ over a finite alphabet Σ. The goal is to find the longest subsequence which is common for both input strings s_1 and s_2 such that there is no character which occurs more than once. The character at position i is denoted by $s[i]$. A matching $\mathbf{m} = (m_1, m_2)$ is a pair of positions such that $s_1[m_1] = s_2[m_2]$ and the corresponding character is denoted by $c(\mathbf{m}) = s_1[m_1]$. Hence, the character $c(\mathbf{m})$ of a matching $\mathbf{m}$ is a possible candidate to appear in a common subsequence. This gives rise to define a domination relation among matchings.

Definition 6.3.1 (Domination relation among matchings)
A matching $\mathbf{m}$ *dominates* a matching $\mathbf{n}$, denoted as $\mathbf{m} \succeq \mathbf{n}$, if $m_1 \leq n_1 \wedge m_2 \leq n_2$, meaning that in a possible common subsequence $c(\mathbf{m})$ may appear before $c(\mathbf{n})$.

Therefore, a common subsequence can be represented by a sequence of matchings $(\mathbf{m}_1, \mathbf{m}_2, \ldots)$ such that $c(\mathbf{m}_1), c(\mathbf{m}_2), \ldots$ maps to the common subsequence and each matching of the sequence dominates each subsequent matching of the sequence. This observation is important since relaxed MDDs for the RFLCS problem will encode such sequences of matchings.

Another important property for the upcoming mixed integer programming (MIP) formulation is the *conflict relation* amoung matchings. If for two matchings $\mathbf{m}$ and $\mathbf{n}$ neither $\mathbf{m} \succeq \mathbf{n}$ nor $\mathbf{n} \succeq \mathbf{m}$ holds then $c(\mathbf{m})$ and $c(\mathbf{n})$ cannot appear together in a common subsequence which will be henceforth referred to as $\mathbf{m}$ and $\mathbf{n}$ are *in conflict*, denoted as $\mathbf{n} \curlyvee \mathbf{m}$. Figure 6.1a shows an example of a RFLCS instance with input strings $s_1 =$ `ABCDBA` and $s_2 =$ `ACBDBA` and an optimal solution of `ACDB`. In this example, matching $\mathbf{m}_1$ dominates matching $\mathbf{m}_2$ and $\mathbf{m}_3$ whereas matching $\mathbf{m}_2$ is in conflict with matching $\mathbf{m}_3$.

6.4 Integer Linear Program and Independent Set Model

An instance of the RFLCS problem can be solved by transforming it into an instance of the MIS problem. Thereby, each matching corresponds to a node of the underlying conflict graph of the MIS problem. An edge is added between two nodes if the corresponding matchings are in conflict or they refer to the same character; see Figure 6.1b for an example. A solution of the MIS instance corresponds to a solution of the RFLCS instance and vice versa, since only matchings are selected that are not in conflict with each other and can therefore appear in the same common subsequence and for each character there is at most one matching selected. The resulting common subsequence can be derived from the set of selected matchings by a topological sort considering the domination relationship.

We solve the MIS instance by a corresponding ILP model. Let M be the set of all matchings of the RFLCS instance and thus nodes of the MIS instance. We use a binary decision variable $x_{\mathbf{m}}$ for each matching $\mathbf{m} \in M$ indicating whether the matching is

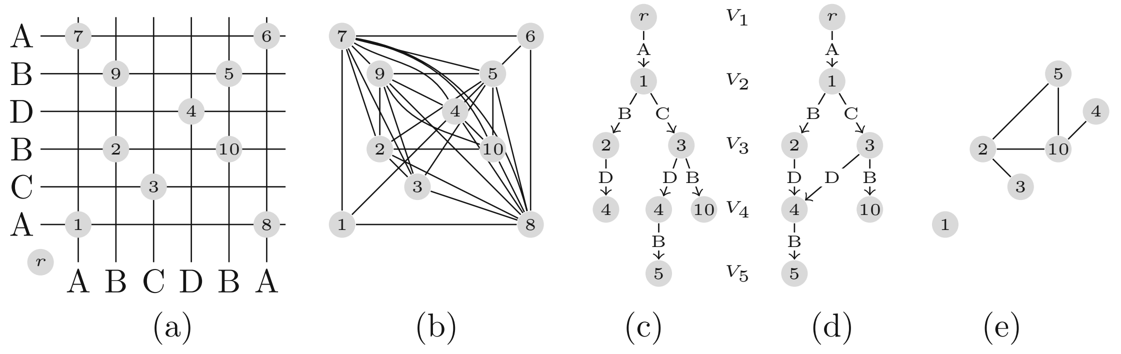


Figure 6.1: (a) Example of a RFLCS instance with input strings s_1 = ABCDBA and s_2 = ACBDBA. Gray circles correspond to the matchings $M = \{\mathbf{m}_1, \ldots, \mathbf{m}_{10}\}$ of the instance. (b) Corresponding MIS instance. (c) Exact MDD $\mathscr{D}_M$ with $\mathrm{mat}(\mathscr{D}_M) = \{\mathbf{m}_1, \ldots, \mathbf{m}_5, \mathbf{m}_{10}\}$. The state of each node u is partially indicated by $\mathbf{m}(u)$. (d) Relaxed MDD where nodes associated with matching $\mathbf{m}_4$ in layer V_3 are merged. (e) MIS instance obtained from matchings $\mathrm{mat}(\mathscr{D}_M)$.

selected (=1) for the solution or not (=0). The model is:

$$\mathrm{ILP}(M) = \max \sum_{x_{\mathbf{m}} \in M} x_{\mathbf{m}} \tag{6.1a}$$

$$\text{s.t.} \quad x_{\mathbf{m}} + x_{\mathbf{n}} \leq 1 \qquad \mathbf{m}, \mathbf{n} \in M : \mathbf{m} \curlyvee \mathbf{n} \tag{6.1b}$$

$$\sum_{x_{\mathbf{m}} \in M_a} x_{\mathbf{m}} \leq 1 \qquad a \in \Sigma \tag{6.1c}$$

$$x_{\mathbf{m}} \in \{0, 1\} \qquad \mathbf{m} \in M \tag{6.1d}$$

The number of selected matchings is maximized. Inequalities (6.1b) ensure that the *common subsequence* constraints are satisfied, i.e., no conflicting matchings are selected together. The *repetition-free* constraints are realized by Inequalities (6.1c), where set $M_a = \{\mathbf{m} \in M \mid c(\mathbf{m}) = a\}$ contains all matchings corresponding to the same character $a \in \Sigma$.

6.5 Relaxed Decision Diagrams for the RFLCS

We use a relaxed MDD to derive a reduced set of matchings $M' \subseteq M$ to subsequently solve the model ILP(M'). Our approach compiles relaxed MDDs in an iterative way s.t. if set M' is derived from a relaxed MDD then another relaxed MDD is compiled w.r.t. set M' to possible derive an even smaller set $M'' \subseteq M'$. This procedure is repeated until some termination criterion is fulfilled.

A MDD w.r.t. a set of matchings M is a directed acyclic multi-graph $\mathscr{D}_M = (V, A)$ with one root node $\mathbf{r}$. All nodes are partitioned into at most $|\Sigma|+1$ layers $V_1(\mathscr{D}_M), \ldots, V_{|\Sigma|+1}(\mathscr{D}_M)$, where $V_i(\mathscr{D}_M)$, $i > 0$ contains only nodes that are reachable from $\mathbf{r}$ over exactly $i - 1$

arcs and $V_1(\mathscr{D}_M)$ is a singleton containing only $\mathbf{r}$. An arc $\alpha = (u, v) \in A(\mathscr{D}_M)$ is always directed from a source node u in some layer $V_i(\mathscr{D}_M)$ to a target node v in a subsequent layer $V_{i+1}(\mathscr{D}_M)$. Each arc α is associated with a matching $\mathrm{mat}(\alpha) \in M$ that represents the assignment of character $c(\mathrm{mat}(\alpha)) \in \Sigma$ to the i-th position of a common subsequence. For convenience, we write $c(\alpha)$ for $c(\mathrm{mat}(\alpha))$. Any directed path $p = (\alpha_1, \alpha_2, \ldots)$ originating from $\mathbf{r}$ identifies a sequence of characters $(c(\alpha_1), c(\alpha_2), \ldots)$ and thus a (partial) solution. A node without any further outgoing arcs is a *sink node*. An *exact* MDD encodes precisely the set of all feasible solutions. Due to the NP-hardness of the RFLCS problem such exact MDDs tend to have exponential size.

Therefore we consider more compact *relaxed* MDDs which encode supersets of all feasible solutions. In such a relaxed MDD nodes of an exact MDD are superimposed (*merged*) so that at each layer a maximum allowed number of nodes, called width, is not exceeded. We do this merging in such a way that any path from the root still represents a common subsequence, but repetition-free constraints may be violated. The length of the longest path in such a relaxed MDD then represents an upper bound to the length of a RFLCS.

To compile a MDD, a DP formulation of the considered problem is usually the starting point [74]. Each node $u \in V(\mathscr{D}_M)$ is associated to a state of the DP formulation. For the RFLCS problem, the DP formulation is defined as follows. Consider for a matching $\mathbf{m} \in M$ the set $\mathrm{D}_M(\mathbf{m}) = \{\mathbf{m}' \in M \setminus \{\mathbf{m}\} \mid \mathbf{m} \succeq \mathbf{m}'\}$ of possible successor matchings of $\mathbf{m}$ that may appear in the same common subsequence after $\mathbf{m}$. Note that set $\mathrm{D}_M(\mathbf{m})$ can be efficiently pre-computed for each $\mathbf{m} \in M$. Then a state $(\mathbf{m}(u), P(u), S(u))$ associated to node u consists of

- a matching $\mathbf{m}(u)$ whose successor matchings $\mathrm{D}_M(\mathbf{m}(u))$ represent the remaining matchings to consider further,
- set $P(u) \subseteq \Sigma$ containing all letters that may still be appended to the common subsequence, and
- set $S(u) \subseteq \Sigma$ containing all letters that appear on some paths from $\mathbf{r}$ to u.

The root state is $(\mathbf{m}_r, \Sigma, \emptyset)$ with the artificial matching $\mathbf{m}_r = (-1, -1)$, and all characters may be appended to it. Note that $\mathrm{D}_M(\mathbf{m}_r) = M$. An arc $\alpha = (u, v)$ corresponds to a transition from state $(\mathbf{m}(u), P(u), S(u))$ to state $(\mathbf{m}(v), P(v), S(v))$ that is achieved by appending character $c(\alpha)$ to the common subsequence w.r.t. to the remaining matchings $\mathrm{D}_M(\mathbf{m}(u))$. Instead of considering all matchings from $\mathrm{D}_M(\mathbf{m}(u))$ as possible outgoing transitions, we consider only matchings that can appear *directly* after $\mathbf{m}(u)$ in a longest common subsequence, i.e., matchings from the subset

$$\begin{aligned}\mathrm{ND}(u) = \{\mathbf{m}' \in \mathrm{D}_M(\mathbf{m}(u)) \mid \\ \nexists \mathbf{m}'' \in \mathrm{D}_M(\mathbf{m}(u)) \setminus \{\mathbf{m}'\} : c(\mathbf{m}'') \notin S(u) \wedge \mathbf{m}'' \succeq \mathbf{m}'\}\end{aligned} \tag{6.2}$$

which are not dominated by any other matching in $\mathrm{D}_M(\mathbf{m}(u))$. The transition function to obtain the successor state $(\mathbf{m}(v), P(v), S(v))$ by considering matching $\mathrm{mat}(\alpha) \in \mathrm{D}_M(u)$

is defined as

$$\tau((\mathbf{m}(u), P(u), S(u)), \mathrm{mat}(\alpha)) = \tag{6.3}$$
$$\begin{cases} (\mathrm{mat}(\alpha), P(u) \setminus \{c(\alpha)\}, S(u) \cup \{c(\alpha)\}) & \text{if } c(\alpha) \in P(u) \wedge \mathrm{mat}(\alpha) \in \mathrm{ND}(u) \\ \hat{0} & \text{otherwise} \end{cases}$$

where $\hat{0}$ represents the infeasible state. Note that no node $\hat{0}$ is created in $\mathscr{D}_M$ and the respective arcs are also skipped.

A state $(\mathbf{m}(u), P(u), S(u))$ may be replaced by a *strengthened state* $(\mathbf{m}(u), P'(u), S(u))$, where $P'(u) = \{a \in P(u) \mid \exists \mathbf{m'} \in \mathrm{D}_M(\mathbf{m}(u)) : a = c(\mathbf{m'})\} \subset P(u)$ without excluding any feasible solutions.

So far, we considered exact MDDs. For relaxed MDDs we have to define a state merger which computes the state of merged nodes. To still encode all feasible common subsequences in the relaxed MDD, only nodes of the same layer and with the same associated matching are merged. Let U be a subset of nodes such that all nodes are associated to matching $\mathbf{n}$, i.e. $\forall u \in U : \mathbf{m}(u) = \mathbf{n}$, then an appropriate state merger is

$$\oplus(U) = \left(\mathbf{n}, \bigcup_{u \in U} P(u), \bigcup_{u \in U} S(u)\right). \tag{6.4}$$

Since we restrict the state merger to nodes with the same associated matching, the possibilities to reduce the size of the relaxed MDD are also limited. However, since $|M|$ is at most the product $|s_1|\,|s_2|$ of the lengths of the two input strings s_1 and s_2, the size of each layer is still polynomially bounded by $O(|s_1|\,|s_2|)$.

Let

$$\mathrm{mat}(\mathscr{D}_M) = \{\mathbf{m}(u) \mid u \in V(\mathscr{D}_M) \setminus \{\mathbf{r}\}\} \subseteq M \tag{6.5}$$

be the set of matchings derived from $\mathscr{D}_M$. To see that $\mathrm{mat}(\mathscr{D}_M)$ is indeed a feasible set of matchings to solve the model ILP($\mathrm{mat}(\mathscr{D}_M)$) from Section 6.4, remember that each path from $\mathbf{r}$ in $\mathscr{D}_M$ encodes a feasible common subsequence. Hence, each such path can also be described as a sequence of matchings from M. In particular this is true for the matchings of a RFLCS, which must be therefore also contained in $\mathrm{mat}(\mathscr{D}_M)$.

6.5.1 Problem specific upper bounds

To reduce $\mathrm{mat}(\mathscr{D}_M)$ further we filter arcs and nodes based on sub-optimality. The idea is to compute for each node u an upper bound $Z^{\mathrm{ub}}(u)$ on the number of characters that can appear in a common subsequence after the character $c(\mathbf{m}(u))$ of matching $\mathbf{m}(u)$. Then we can prune each node u in the relaxed MDD where $Z^{\mathrm{lp}}(u) + Z^{\mathrm{ub}}(u) < lb$ holds, where lb is a known lower bound on the length of the RFLCS and $Z^{\mathrm{lp}}(u)$ is the length of the longest path from $\mathbf{r}$ to u. We compute the upper bound for each node u by

$$Z^{\mathrm{ub}}(u) = \min\{|P(u)|, \mathrm{UB}^{\mathrm{lcs}}(\mathbf{m}(u)), \max_{\alpha=(w,u)}\{Z^{\mathrm{ub}}(w) - 1\}, Z^{\mathrm{lp}\uparrow}(u)\}. \tag{6.6}$$

The first term takes the number of characters into account that can still be appended to the common subsequence after matching $\mathbf{m}(u)$. The second term $\mathrm{UB}^{\mathrm{lcs}}(\mathbf{m}(u)) = \mathrm{LCS}(m_1, m_2)$ is based on DP and computes the length of the longest common subsequence from matching $\mathbf{m}(u)$ onward. Note that this bound can be obtained in constant time by using a data structure known as scoring matrix, which can be computed during preprocessing for two input strings in $O(|s_1|\,|s_2|)$ time [26]. The third term takes the upper bounds from the parent nodes of u into account. Finally, the last term corresponds to the length of the longest path from u to any sink node in the relaxed MDD. Note that this term is only available if the whole relaxed MDD is already compiled.

6.6 Incremental Refinement

Our approach to compile a relaxed MDD $\mathscr{D}_M$ w.r.t. matching set M for the RFLCS problem is based on the incremental refinement (IR) algorithm from Cire and van Hoeve [40] for sequencing problems. For a general description of IR-based approaches we refer to Section 2.5.2 of Chapter 2. Since $\mathscr{D}_M$ considers the common subsequence constraints exactly and only relaxes the repetition-free constraint, paths in $\mathscr{D}_M$ originating from $\mathbf{r}$ will correspond to common subsequences where characters may appear more than once. We use the ideas from [40] to ensure at least for some characters that they occur at most once at each path for refining $\mathscr{D}_M$. Cire and van Hoeve [40] showed that the size of a given relaxed MDD will be at most doubled to establish this property for one more character.

The algorithm applies repeatedly two major steps—filtering and refinement—until some termination condition is fulfilled. Let $a_1^*, a_2^*, \ldots, a_{|\Sigma|}^*$ be a ranking of the characters in Σ such that a_1^* is the most important character to appear at most once at each path in $\mathscr{D}_M$ to get a strong relaxation. The following refinement step is applied layer by layer starting with $V_1(\mathscr{D}_M)$: For each character $a^* = a_1^*, a_2^*, \ldots, a_{|\Sigma|}^*$ we identify nodes u such that $a^* \in P(u) \cap S(u)$ and split them into two new nodes u_1 and u_2 where an incoming arc $\alpha = (v, u)$ is redirected to u_1 if $a^* \in P(u) \setminus \{c(\alpha)\}$ and to u_2 otherwise. All outgoing arcs are replicated for both nodes u_1 and u_2. We do this as long as the size of the layer is below a maximum width threshold W. For more details and a correctness proof in the context of sequencing problems see [40]. Due to the splitting of nodes the corresponding states may be changed and some of the outgoing arcs from the current layer to the next layer may become infeasible. Those arcs are filtered for each layer after the refinement step finishes. Algorithm 6.1 shows this at lines 13 and 14. The algorithm terminates if set $\mathrm{mat}(\mathscr{D}_M)$ could not be further reduced by the previously applied refinement/filtering round. The other main parts of the algorithm are described in the following subsections.

6.6.1 Initial relaxed MDD

The IR algorithm starts with an initial relaxed MDD. Usually, this initial relaxed MDD is a naive one of width one, i.e., a relaxed MDD with just a single node at each layer. However, in our case we want to respect the common subsequence constraints and only

Algorithm 6.1: Incremental Refinement

```
   Input: set of matchings $M$, lower bound $lb$, maximum width threshold $W$
1  $s^{\text{best}} \leftarrow \varepsilon$;
2  construct initial relaxed decision diagram $\mathscr{D}_M$;
3  do
4      filter-bottom-up($\mathscr{D}_M$, max($lb$, $|s^{\text{best}}|$));
5      if $|s^{\text{best}}| < |s^{\text{rfcs}}|$ then
6          $s^{\text{rfcs}} \leftarrow$ derive-primal-solution($\mathscr{D}_M$) and update $s^{\text{best}}$
7      end
8  while new best solution $s^{\text{best}}$ found;
9  determine priority ranking $a_1^*, \ldots, a_{|\Sigma|}^*$ of all characters;
10 do
11     $M \leftarrow$ mat($\mathscr{D}_M$);
12     for $i \leftarrow 1$ to $|\Sigma| + 1$ do
13         refine($V_i(\mathscr{D}_M)$, $a_1^*, \ldots, a_{|\Sigma|}^*$, $W$);
14         filter arcs between $V_i$ and $V_{i+1}$;
15     end
16     do
17         filter-bottom-up($\mathscr{D}_M$, max($lb$, $|s^{\text{best}}|$));
18         if $|s^{\text{best}}| < |s^{\text{rfcs}}|$ then
19             $s^{\text{rfcs}} \leftarrow$ derive-primal-solution($\mathscr{D}_M$) and update $s^{\text{best}}$
20         end
21     while new best solution $s^{\text{best}}$ found;
22 while $|\text{mat}(\mathscr{D}_M)| < |M|$;
23 return ($\mathscr{D}_M$, $s^{\text{best}}$)
```

superimpose states that correspond to the same matching. Therefore we compile the initial $\mathscr{D}_M$ layer-by-layer in a top-down approach. At each layer $V_i(\mathscr{D}_M)$, $i \geq 1$, we expand all nodes using the transition function (6.3), thus creating for each feasible transition a corresponding node in $V_{i+1}(\mathscr{D}_M)$ and adding the corresponding arc if the node is not sub-optimal according to Equation (6.6). Then all nodes in $V_{i+1}(\mathscr{D}_M)$ with the same corresponding matching are merged. Since no feasible common subsequence can be longer than the upper bound $Z^{\text{ub}}(\mathbf{r})$, the compilation of $\mathscr{D}_M$ stops at the $(Z^{\text{ub}}(\mathbf{r}) + 1)$-th layer.

Note that another, possible promising, way to compile an appropriate initial relaxed MDD is to use the A*-based construction (A*C) algorithm from Section 4.5, which is able to produce strong relaxed MDDs for the LCS problem with multiple input strings (cf. numerical results in Section 4.5.6). However, such relaxed MDDs are not based on layers. As a consequence, the IR algorithm from [40], which is based on layers, needs a suitable adaptation. Although this is possible, we decided to use the top-down approach and apply the IR algorithm in a straightforward way for the time being. Using A*C to compile the initial relaxed MDDs is left open for further research.

6.6.2 Character ranking for refinement

To determine priorities for the characters we use some structural information obtained from the initial MDD. For this purpose let $\mathrm{All}^{\uparrow}(u)$ for each node $u \in V(\mathscr{D}_M)$ be the set of characters that appear on all paths from node u to a sink node. Note that set $\mathrm{All}^{\uparrow}(u)$ can be efficiently computed in a recursive way by a single bottom-up pass. If there exists a node v with an incoming arc $\alpha = (u, v)$ such that $c(\alpha) \in \mathrm{All}^{\uparrow}(v)$ holds, then each path originating from $\mathbf{r}$ and leading to any sink node will be infeasible if the path traverses α since character $c(\alpha)$ will appear more than once in a corresponding common subsequence, i.e., the repetition-free constraint will be violated. In [40] such arcs could be safely removed without also removing any feasible solution from the relaxed MDD. In our case this is not possible since solutions have arbitrary length and the path from $\mathbf{r}$ to v could still correspond to a complete feasible solution. However, we can use these violations to determine for which character it is most important to appear on all paths at most once to get a strong relaxation. Hence, we count for each character how often such a violation occurs in $\mathscr{D}_M$ and sort the characters according to non-increasing numbers of violations. Ties are resolved by preferring characters that appear in more matchings.

6.6.3 Filtering and deriving new primal solutions

Lines 3-8 and 16-21 from Algorithm 6.1 perform the following steps. First the function filter-bottom-up performs a single bottom up pass where for each node u the length of the longest path $Z^{\mathrm{lp}\uparrow}(u)$ from u to any sink node is computed and the upper bound $Z^{\mathrm{ub}}(u)$ is updated accordingly. If $Z^{\mathrm{lp}}(u) + Z^{\mathrm{ub}}(u) < lb$ then node u and all incident arcs are removed from $\mathscr{D}_M$.

After filtering we try to derive from $\mathscr{D}_M$ a new best heuristic solution. Since each path in $\mathscr{D}_M$ originating from $\mathbf{r}$ corresponds to a CS, we can derive a repetition-free common subsequence by removing duplicate letters. This is done in two steps. First, a bottom-up pass is performed where primal bounds are computed: For each node $u \in V(\mathscr{D}_M)$ we recursively determine set $B^{\uparrow}(u) = B^{\uparrow}(v) \cup \{c(\alpha')\}$ where outgoing arc $\alpha' = (u, v)$ maximizes the term $|B^{\uparrow}(v) \cup \{c(\alpha)\}|$ over all outgoing arcs $\alpha = (u, v) \in A(\mathscr{D}_M)$. Ties are resolved by sticking at the first arc that maximizes the expression. If u has no outgoing arcs then $B^{\uparrow}(u) = \emptyset$. Note that $|B^{\uparrow}(\mathbf{r})|$ is a valid primal bound on the RFLCS problem, since only the union is taken to compute $B^{\uparrow}(.)$. To improve this bound further, the second step performs a top-down pass where set $B^{\downarrow}(v)$ is recursively computed for each node v using the information of the precisely computed set $B^{\uparrow}(v)$. Hence, $B^{\downarrow}(v) = B^{\downarrow}(u) \cup \{c(\alpha')\}$ where incoming arc $\alpha' = (u, v)$ maximizes the term $|B^{\downarrow}(u) \cup \{c(\alpha)\} \cup B^{\uparrow}(v)|$ over all incoming arcs $\alpha = (u, v) \in A(\mathscr{D}_M)$ using $|B^{\downarrow}(v) \cup \{c(\alpha)\}|$ as tie breaking criterion. A sink node v' that maximizes $|B^{\downarrow}(v')|$ then provides the strongest primal bound. A respective repetition-free common subsequence is derived by going from v' backwards to $\mathbf{r}$, skipping any character that already occurred along the path.

If a new best heuristic solution could be obtained in this way then the filter-bottom-up step is repeated and we try again to obtain a new best heuristic solution.

6.6.4 Main Procedure

Algorithm 6.2 shows the main procedure to solve an instance of the RFLCS problem. As input the algorithm takes the set of input strings S, a possibly known lower bound on the RFLCS length or zero, and the maximum width threshold W for the relaxed MDDs. The original set of matchings M is reduced by performing iteratively the following steps. The first step processes the input strings s_1 and s_2 by removing characters that have no associated matching in M and characters that appear immediately one after the other in the input strings. For example, if character $a \in \Sigma$ appears in an input string at both position i and $i+1$ then a can be removed from $i+1$ without removing any feasible solution. Furthermore, if the pattern $abab$ with $a, b \in \Sigma$ has been discovered in one of the input strings then the last b can be removed from the input string due to the repetition-free constraint. Next, M is reduced by removing matchings $\mathbf{m} \in M$ where the upper bound used in [26], denoted by $\mathrm{UB}^{\mathrm{BLUM}}(\mathbf{m})$ is lower than our currently best primal bound. This upper bound is based on the first two terms in Equation (6.6), i.e., on the number of characters that can appear in a repetition-free common subsequence that contains $\mathbf{m} \in M$ and on the length of the LCS that contains $\mathbf{m}$. Note that the difference to Equation (6.6) is that $\mathrm{UB}^{\mathrm{BLUM}}(\mathbf{m})$ is an upper bound on the length of a complete repetition-free common subsequence containing $\mathbf{m}$ whereas Equation (6.6) describes an upper bound on the remaining part from $\mathbf{m}$ onward. With this reduced set M we compile a relaxed MDD $\mathscr{D}_M$. If the length of the hereby derived repetition-free common subsequence s^{rfcs} is equal to the longest path in $\mathscr{D}_M$ then s^{rfcs} is an optimal solution and the algorithm terminates. Otherwise, if due to the reduced set $\mathrm{mat}(\mathscr{D}_M)$

Algorithm 6.2: Main Procedure for solving the RFLCS problem

```
   Input: input strings $S$, lower bound $lb$, maximum width threshold $W$
1  $s^{\mathrm{best}} \leftarrow \varepsilon$;
2  derive original $M$ w.r.t. $S$;
3  do
4      process $S$ w.r.t. $M$;
5      $M \leftarrow \{\mathbf{m} \in M \mid \mathrm{UB}^{\mathrm{BLUM}}(\mathbf{m}) \geq \max(lb, |s^{\mathrm{best}}|)\}$;
6      $(\mathscr{D}_M, s^{\mathrm{rfcs}}) \leftarrow \mathrm{IR}(M, \max(lb, |s^{\mathrm{best}}|), W)$;
7      if $|s^{\mathrm{rfcs}}| > |s^{\mathrm{best}}|$ then
8          $M \leftarrow \mathrm{mat}(\mathscr{D}_M)$ and update $s^{\mathrm{best}}$
9      end
10     if $s^{\mathrm{best}} = Z^{\mathrm{lp}\uparrow}(\mathbf{r})$ then
11         return $s^{\mathrm{best}}$
12     end
13 while further characters can be removed from input strings;
14 $s^{\mathrm{ilp}} \leftarrow$ solve ILP$(M)$;
15 update $s^{\mathrm{best}}$ if $|s^{\mathrm{ilp}}| > |s^{\mathrm{best}}|$;
16 return $s^{\mathrm{best}}$
```

further characters can be removed from s_1 and s_2 then we repeat the procedure until no further characters can be removed. Note that since the size of the input strings are reduced at each iteration also $\mathrm{UB}^{\mathrm{BLUM}}(\mathbf{m})$ changes, which may further reduce set M. Finally the ILP model from Section 6.4 is solved for set M.

6.7 Experimental Results

To test and compare our approach we used two benchmark sets from Blum and Blesa [23]. The first set, Set1, consists of 1680 randomly generated instances. For each combination of the input string lengths $n \in \{32, 64, 128, 256, 512, 1024, 2048, 4096\}$ and the alphabet sizes $|\Sigma| \in \{\frac{n}{8}, \frac{n}{4}, \frac{3n}{8}, \frac{n}{2}, \frac{5n}{8}, \frac{3n}{4}, \frac{7n}{8}\}$ there are 30 instances. The second set, Set2, consists of 30 randomly generated instances for each combination of the alphabet size $|\Sigma| \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and the maximal repetition of each character, reps $\in \{3, 4, 5, 6, 7, 8\}$. This set has a total of 1440 instances.

The algorithms were implemented using GNU C++ 5.4.1. All tests were executed on a single core of an Intel Xeon E5649 with 2.53 GHz and 16 GB RAM. The ILP model from Section 6.4 was solved with CPLEX 12.7 with a CPU-time limit of 3600 seconds. For Algorithm 6.2, henceforth denoted as MDD+CPLEX, the maximum width threshold was set to $W = 5000$. This value was determined in preliminary experiments such that set M could be reduced as much as possible and as many instances as possible can be solved to optimality within the memory limit of 16 GB. MDD+CPLEX is compared to the approach from Blum et al. [26], henceforth denoted as UB+CPLEX, where the ILP(M') model is solved with the reduced set of matchings $M' = \{\mathbf{m} \in M \mid \mathrm{UB}^{\mathrm{BLUM}}(\mathbf{m}) < lb\}$. Both approaches use the lengths of the currently best known solutions from the literature as initial lower bound lb. Note that the compiled relaxed MDDs from Algorithm 6.1 are not strictly limited to W since the initial relaxed MDD could already contain layers that contain more nodes than W. However, such layers are not further refined during the compilation.

6.7.1 Reduction Rates

The average reduction rate of matchings from M is shown in Figure 6.2 for 18 instance classes by means of bar plots. The first bar always corresponds to the UB+CPLEX approach whereas the next three bars corresponds to the MDD+CPLEX approach with different values for $W \in \{1, 1000, 5000\}$. Note that $W = 1$ means that only the initial relaxed MDD is compiled and no further refinement will take place. As expected, the obtained reduction rate increases with W. In each case the MDD+CPLEX approach is able to reduce the set of matchings M to a larger extend than the UB+CPLEX approach. Furthermore, it seems that the set of matchings M is harder to reduce for instances from Set1 with a rather small alphabet size / input string length ratio than for instances with a larger ratio. This seems to be true for both tested approaches. For instance, consider instances from Set1 with $|\Sigma| = n/8$ and $n = 512$, where an average reduction of about $\sim 20\%$ can be achieved whereas an average reduction of $\sim 80\%$ could be obtained

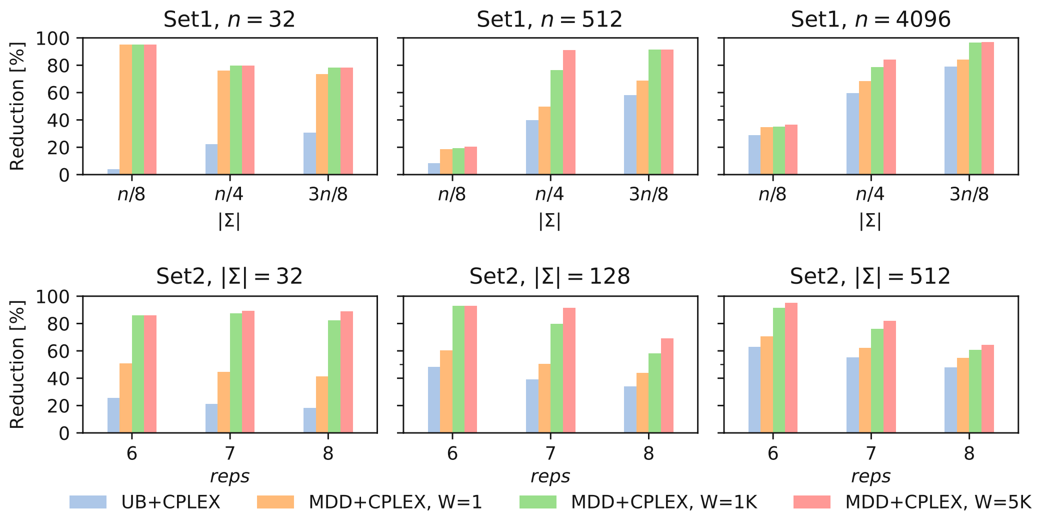


Figure 6.2: Average reduction rates of matchings obtained from UB+CPLEX and MDD+CPLEX with different maximum width thresholds W.

for instances with the same input string length and $|\Sigma| = 3n/8$. For instances from Set2 it seems that set M gets more difficult to reduce for instances with larger alphabet size.

Figure 6.3 shows a reduction example of one single instance from Set2 with $|\Sigma| = 16$ and reps$= 6$. The original instance consists of 221 matchings and is drawn on the left side of the figure, where black and red dots represent the corresponding set of matchings M. Red dots connected by arcs indicate matchings that belong to an optimal solution sequence of matchings. After applying Algorithm 6.2 the reduced set of matchings M' consists of only 20 matchings after the main reduction loop at Line 13. Relaxed DDs are created with a maximum width threshold of $W = 5000$. The reduced instance after the reduction is drawn on the right side of Figure 6.3. Note that the optimal solution sequence of matchings is still contained in M'. Furthermore, we can derive reduced input strings from M' by deleting all characters from the input strings that do not belong to any matching in M'.

The boxplots in Figure 6.4 report the average difference $\mathrm{red}_{\mathrm{MDD}} - \mathrm{red}_{\mathrm{UB}}$ between the average reduction rate $\mathrm{red}_{\mathrm{UB}}$ obtained from UB+CPLEX and $\mathrm{red}_{\mathrm{MDD}}$ obtained from MDD+CPLEX in percentage points aggregated over the ratio between n and $|\Sigma|$ in case of Set1 and over $|\Sigma|$ in case of Set2. On average the MDD+CPLEX approach is able to reduce the original set of matchings by more than 25.79% and 41.28% as UB+CPLEX does, for Set1 and Set2, respectively.

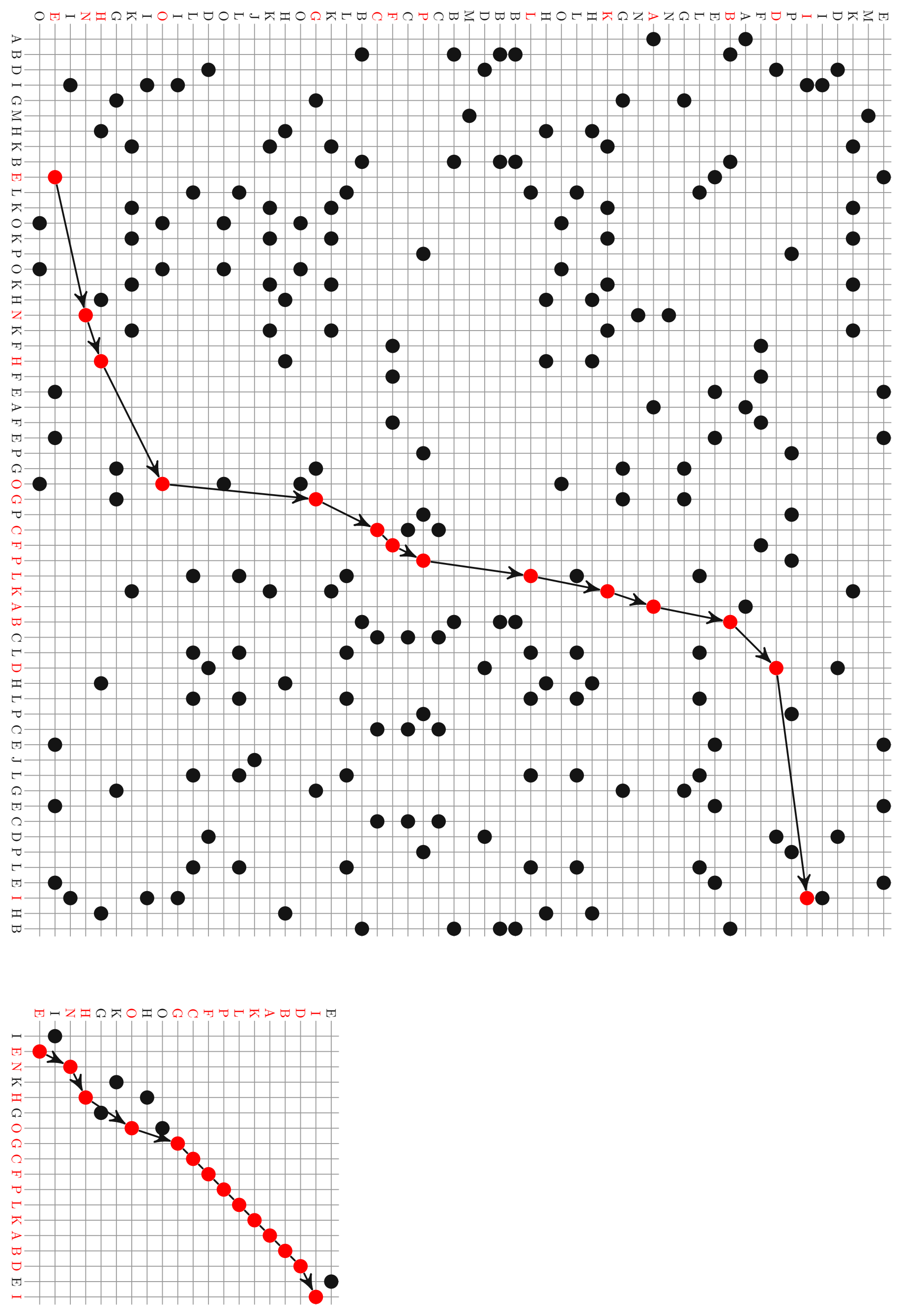


Figure 6.3: Reduction example of an instance from Set2 with $|\Sigma| = 16$ and reps$= 6$. The original instance is shown on the left side whereas the reduced instance is drawn on the right side. Both instances contain the optimal solution ENHOGCFPLKABDI.

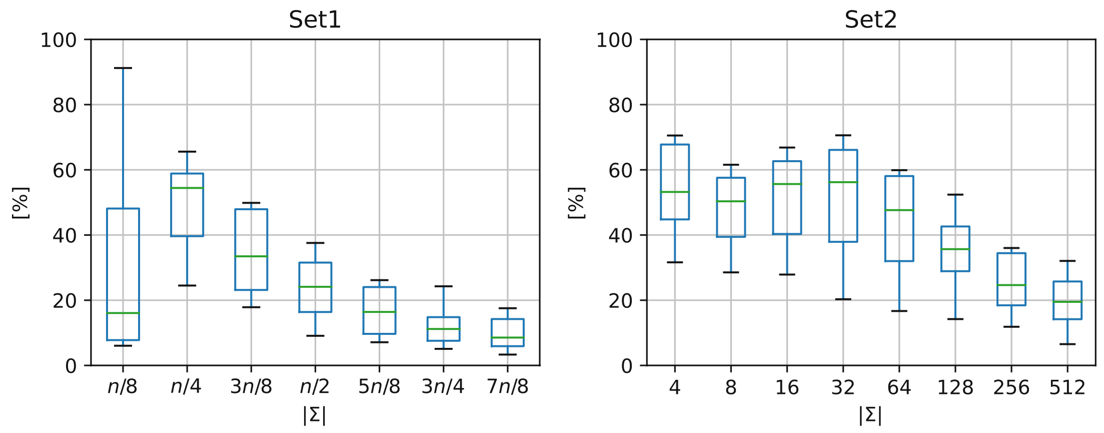


Figure 6.4: Average difference between the obtained reduction rates of UB+CPLEX and MDD+CPLEX with $W = 5000$.

6.7.2 Main Results

Detailed aggregated results are presented in Tables 6.1 and 6.2 where the first two columns show the instance characteristics and the third column shows the average length of the so far best known solution from the literature. Columns obj report for each tested approach the average length of the best obtained solutions. In case of MDD+CLPEX these solutions are either those obtained from the ILP model or the ones found during the compilation of the relaxed MDDs. In case of UB+CPLEX a "-" symbol indicates that CPLEX was not able at all to derive a primal solution within the time and memory limits. Average optimality gaps, shown in columns gap, are calculated by $100\% \times (\text{ub} - \text{obj})/\text{ub}$ where ub is for each approach the best obtained upper bound. In case of MDD+CPLEX this is either the upper bound obtained from the ILP model or the length of the longest path from a compiled relaxed MDD. Columns t_{prep} list average preprocessing times in CPU seconds including the computation of the reduced set of matchings M (see Algorithm 6.2, Line 13). Columns t_{tot} list average total computation times in CPU seconds until the algorithm terminates including t_{prep} plus the time CPLEX needs and columns #opt report the total numbers of instances solved to optimality. In case of MDD+CPLEX the second number corresponds to the number of instances where optimality could already be proven by the compiled relaxed MDD at Line 11 in Algorithm 6.2. Hence, the number of times, where it was not required to solve the ILP model at all. Average reduction rates of the original set of matchings are reported by columns red.

Regarding the number of instances solved to proven optimality, note that already UB+CPLEX was quite successful with a total of 90.26%. More precisely, 1489 out of 1680 instances from Set1 and 1327 out of 1440 instances of Set2 could be solved to proven optimality by UB+CPLEX. Nevertheless, MDD+CPLEX is able to solve significantly more instances to proven optimality: 1541 instances from Set1 and 1381 instances of Set2, and thus a total of 93.65%. Moreover, in 90.90% of all instances it

Table 6.1: Results on Set1 instances.

		so far	UB+CPLEX					MDD+CPLEX					
$\|\Sigma\|$	n	best.	obj	gap [%]	t [s]	#opt	red [%]	obj	gap [%]	t_{prep} [s]	t_{tot} [s]	#opt	red [%]
$n/8$	32	**4.00**	**4.00**	**0.00**	0.16	**30**	3.72	**4.00**	**0.00**	<0.01	**<0.01**	**30/30**	94.94
	64	**8.00**	**8.00**	**0.00**	1.69	**30**	4.67	**8.00**	**0.00**	<0.01	**<0.01**	**30/30**	78.23
	128	**16.00**	**16.00**	**0.00**	11.46	**30**	4.80	**16.00**	**0.00**	0.04	**0.04**	**30/30**	44.45
	256	**31.97**	**31.97**	**0.00**	298.59	**30**	5.47	**31.97**	**0.00**	0.42	**0.42**	**30/30**	25.47
	512	**63.90**	32.30	49.50	3615.63	0	8.16	62.80	**1.82**	60.82	**3434.53**	**2/2**	20.29
	1024	**116.30**	0.00	100.00	**3676.46**	**0**	15.89	113.47	**11.22**	271.15	3946.52	**0/0**	23.34
	2048	185.07	-	-	**0.48**	**0**	22.45	**186.23**	**16.29**	1134.30	4135.18	**0/0**	28.49
	4096	284.80	-	-	**2.04**	**0**	28.71	**292.03**	**11.05**	4297.07	4297.07	**0/0**	36.59
$n/4$	32	**7.83**	**7.83**	**0.00**	0.03	**30**	22.13	**7.83**	**0.00**	<0.01	**<0.01**	**30/30**	79.54
	64	**14.67**	**14.67**	**0.00**	0.21	**30**	21.08	**14.67**	**0.00**	<0.01	**<0.01**	**30/30**	84.06
	128	**25.93**	**25.93**	**0.00**	4.82	**30**	21.76	**25.93**	**0.00**	0.47	**0.47**	**30/30**	87.34
	256	**43.97**	**43.97**	**0.00**	22.93	**30**	33.17	**43.97**	**0.00**	4.61	**4.61**	**30/30**	90.67
	512	**68.57**	**68.57**	**0.00**	389.04	**30**	39.54	**68.57**	**0.00**	28.43	**38.93**	**30/28**	91.01
	1024	**105.07**	104.97	0.96	2064.94	21	48.42	**105.07**	**0.00**	122.81	**188.27**	**30/24**	91.56
	2048	155.73	120.47	26.52	3314.28	4	56.72	**156.87**	**0.37**	576.39	**1571.46**	**24/15**	85.91
	4096	227.23	12.77	95.05	**3650.80**	0	59.48	**230.37**	**0.73**	1996.26	4036.48	**15/6**	83.99
$3n/8$	32	**8.77**	**8.77**	**0.00**	0.02	**30**	30.44	**8.77**	**0.00**	<0.01	**<0.01**	**30/30**	78.03
	64	**15.53**	**15.53**	**0.00**	0.06	**30**	32.30	**15.53**	**0.00**	<0.01	**<0.01**	**30/30**	81.16
	128	**24.90**	**24.90**	**0.00**	0.70	**30**	36.58	**24.90**	**0.00**	0.06	**0.06**	**30/30**	86.44
	256	**39.97**	**39.97**	**0.00**	1.52	**30**	55.50	**39.97**	**0.00**	0.50	**0.50**	**30/30**	89.20
	512	**59.97**	**59.97**	**0.00**	17.62	**30**	57.92	**59.97**	**0.00**	4.26	**4.26**	**30/30**	91.16
	1024	**90.73**	**90.73**	**0.00**	29.12	**30**	70.02	**90.73**	**0.00**	13.59	**13.59**	**30/30**	93.59
	2048	131.13	**131.17**	0.05	476.47	29	72.92	**131.17**	**0.00**	50.39	**50.39**	**30/30**	94.79
	4096	193.20	192.77	0.50	1030.16	25	78.79	**193.37**	**0.00**	163.90	**163.99**	**30/29**	96.65
$n/2$	32	**8.87**	**8.87**	**0.00**	0.01	**30**	37.51	**8.87**	**0.00**	<0.01	**<0.01**	**30/30**	75.08
	64	**14.80**	**14.80**	**0.00**	0.02	**30**	46.60	**14.80**	**0.00**	<0.01	**<0.01**	**30/30**	81.22
	128	**22.93**	**22.93**	**0.00**	0.08	**30**	52.04	**22.93**	**0.00**	0.01	**0.01**	**30/30**	82.57
	256	**35.20**	**35.20**	**0.00**	0.46	**30**	60.22	**35.20**	**0.00**	0.07	**0.07**	**30/30**	87.37
	512	**53.13**	**53.13**	**0.00**	2.72	**30**	69.40	**53.13**	**0.00**	0.92	**0.92**	**30/30**	90.49
	1024	**79.13**	**79.13**	**0.00**	7.38	**30**	75.93	**79.13**	**0.00**	3.45	**3.45**	**30/30**	93.05
	2048	**115.70**	**115.70**	**0.00**	21.32	**30**	80.40	**115.70**	**0.00**	19.65	**19.65**	**30/30**	94.59
	4096	**167.97**	**167.97**	**0.00**	93.68	**30**	86.74	**167.97**	**0.00**	26.89	**26.89**	**30/30**	95.84
$5n/8$	32	**8.60**	**8.60**	**0.00**	0.01	**30**	46.29	**8.60**	**0.00**	<0.01	**<0.01**	**30/30**	72.45
	64	**13.30**	**13.30**	**0.00**	0.01	**30**	52.75	**13.30**	**0.00**	<0.01	**<0.01**	**30/30**	78.35
	128	**21.20**	**21.20**	**0.00**	0.03	**30**	59.95	**21.20**	**0.00**	<0.01	**<0.01**	**30/30**	83.47
	256	**32.53**	**32.53**	**0.00**	0.11	**30**	67.54	**32.53**	**0.00**	0.02	**0.02**	**30/30**	86.36
	512	**47.83**	**47.83**	**0.00**	0.61	**30**	74.69	**47.83**	**0.00**	0.10	**0.10**	**30/30**	88.68
	1024	**70.20**	**70.20**	**0.00**	1.12	**30**	81.45	**70.20**	**0.00**	0.76	**0.76**	**30/30**	91.35
	2048	**103.97**	**103.97**	**0.00**	4.66	**30**	84.96	**103.97**	**0.00**	3.27	**3.27**	**30/30**	93.98
	4096	**150.57**	**150.57**	**0.00**	16.11	**30**	88.65	**150.57**	**0.00**	10.26	**10.26**	**30/30**	95.77
$3n/4$	32	**8.17**	**8.17**	**0.00**	0.01	**30**	47.56	**8.17**	**0.00**	<0.01	**<0.01**	**30/30**	71.83
	64	**12.53**	**12.53**	**0.00**	0.01	**30**	53.92	**12.53**	**0.00**	<0.01	**<0.01**	**30/30**	71.72
	128	**19.70**	**19.70**	**0.00**	0.02	**30**	65.68	**19.70**	**0.00**	<0.01	**<0.01**	**30/30**	79.50
	256	**29.97**	**29.97**	**0.00**	0.04	**30**	72.89	**29.97**	**0.00**	0.01	**0.01**	**30/30**	84.41
	512	**44.57**	**44.57**	**0.00**	0.26	**30**	77.45	**44.57**	**0.00**	0.03	**0.03**	**30/30**	88.28
	1024	**65.20**	**65.20**	**0.00**	0.53	**30**	83.86	**65.20**	**0.00**	0.26	**0.26**	**30/30**	92.07
	2048	**94.67**	**94.67**	**0.00**	1.45	**30**	88.57	**94.67**	**0.00**	0.51	**0.51**	**30/30**	94.18
	4096	**136.77**	**136.77**	**0.00**	6.06	**30**	90.14	**136.77**	**0.00**	4.19	**4.19**	**30/30**	95.22
$7n/8$	32	**7.67**	**7.67**	**0.00**	0.01	**30**	47.37	**7.67**	**0.00**	<0.01	**<0.01**	**30/30**	64.92
	64	**11.57**	**11.57**	**0.00**	0.01	**30**	56.15	**11.57**	**0.00**	<0.01	**<0.01**	**30/30**	73.63
	128	**18.40**	**18.40**	**0.00**	0.02	**30**	63.40	**18.40**	**0.00**	<0.01	**<0.01**	**30/30**	76.54
	256	**27.80**	**27.80**	**0.00**	0.03	**30**	74.05	**27.80**	**0.00**	0.01	**0.01**	**30/30**	84.25
	512	**40.60**	**40.60**	**0.00**	0.09	**30**	80.25	**40.60**	**0.00**	0.03	**0.03**	**30/30**	87.14
	1024	**60.57**	**60.57**	**0.00**	0.47	**30**	85.41	**60.57**	**0.00**	0.11	**0.11**	**30/30**	91.16
	2048	**88.00**	**88.00**	**0.00**	2.54	**30**	85.93	**88.00**	**0.00**	0.63	**0.63**	**30/30**	91.89
	4096	**127.37**	**127.37**	**0.00**	4.76	**30**	91.37	**127.37**	**0.00**	2.17	**2.17**	**30/30**	94.71

Table 6.2: Results on SET2 instances.

		so far	UB+CPLEX					MDD+CPLEX					
$\lvert\Sigma\rvert$	reps	best	obj	gap [%]	t [s]	#opt	red [%]	obj	gap [%]	t_{prep} [s]	t_{tot} [s]	#opt	red [%]
4	3	**3.47**	**3.47**	**0.00**	**<0.01**	**30**	34.16	**3.47**	**0.00**	<0.01	**<0.01**	**30/30**	65.78
	4	**3.77**	**3.77**	**0.00**	**<0.01**	**30**	32.24	**3.77**	**0.00**	<0.01	**<0.01**	**30/30**	76.59
	5	**3.83**	**3.83**	**0.00**	**<0.01**	**30**	35.31	**3.83**	**0.00**	<0.01	**<0.01**	**30/30**	81.44
	6	**3.90**	**3.90**	**0.00**	0.01	**30**	25.62	**3.90**	**0.00**	<0.01	**<0.01**	**30/30**	85.92
	7	**3.97**	**3.97**	**0.00**	0.02	**30**	18.34	**3.97**	**0.00**	<0.01	**<0.01**	**30/30**	88.59
	8	**3.97**	**3.97**	**0.00**	0.02	**30**	19.01	**3.97**	**0.00**	<0.01	**<0.01**	**30/30**	89.54
8	3	**6.23**	**6.23**	**0.00**	**<0.01**	**30**	38.40	**6.23**	**0.00**	<0.01	**<0.01**	**30/30**	66.94
	4	**6.87**	**6.87**	**0.00**	0.01	**30**	34.77	**6.87**	**0.00**	<0.01	**<0.01**	**30/30**	71.70
	5	**7.40**	**7.40**	**0.00**	0.02	**30**	33.12	**7.40**	**0.00**	<0.01	**<0.01**	**30/30**	80.10
	6	**7.53**	**7.53**	**0.00**	0.02	**30**	25.51	**7.53**	**0.00**	<0.01	**<0.01**	**30/30**	79.22
	7	**7.70**	**7.70**	**0.00**	0.03	**30**	21.51	**7.70**	**0.00**	<0.01	**<0.01**	**30/30**	80.36
	8	**7.77**	**7.77**	**0.00**	0.07	**30**	19.09	**7.77**	**0.00**	<0.01	**<0.01**	**30/30**	80.65
16	3	**9.70**	**9.70**	**0.00**	0.01	**30**	43.91	**9.70**	**0.00**	<0.01	**<0.01**	**30/30**	71.78
	4	**11.57**	**11.57**	**0.00**	0.02	**30**	42.73	**11.57**	**0.00**	<0.01	**<0.01**	**30/30**	79.44
	5	**12.93**	**12.93**	**0.00**	0.04	**30**	28.84	**12.93**	**0.00**	<0.01	**<0.01**	**30/30**	79.97
	6	**14.00**	**14.00**	**0.00**	0.12	**30**	23.82	**14.00**	**0.00**	<0.01	**<0.01**	**30/30**	83.96
	7	**14.93**	**14.93**	**0.00**	0.29	**30**	21.32	**14.93**	**0.00**	0.01	**0.01**	**30/30**	84.81
	8	**14.80**	**14.80**	**0.00**	0.46	**30**	19.26	**14.80**	**0.00**	0.01	**0.01**	**30/30**	86.09
32	3	**16.13**	**16.13**	**0.00**	0.02	**30**	57.94	**16.13**	**0.00**	<0.01	**<0.01**	**30/30**	78.26
	4	**19.00**	**19.00**	**0.00**	0.05	**30**	48.26	**19.00**	**0.00**	<0.01	**<0.01**	**30/30**	81.46
	5	**21.63**	**21.63**	**0.00**	0.52	**30**	33.72	**21.63**	**0.00**	0.04	**0.04**	**30/30**	85.76
	6	**23.73**	**23.73**	**0.00**	1.39	**30**	25.52	**23.73**	**0.00**	0.10	**0.10**	**30/30**	85.91
	7	**25.57**	**25.57**	**0.00**	3.65	**30**	21.18	**25.57**	**0.00**	0.61	**0.61**	**30/30**	89.21
	8	**27.50**	**27.50**	**0.00**	5.19	**30**	18.22	**27.50**	**0.00**	0.99	**0.99**	**30/30**	88.80
64	3	**25.43**	**25.43**	**0.00**	0.04	**30**	65.65	**25.43**	**0.00**	<0.01	**<0.01**	**30/30**	82.34
	4	**30.37**	**30.37**	**0.00**	0.22	**30**	57.79	**30.37**	**0.00**	0.04	**0.04**	**30/30**	86.45
	5	**34.93**	**34.93**	**0.00**	3.36	**30**	44.23	**34.93**	**0.00**	0.74	**0.74**	**30/30**	86.26
	6	**39.13**	**39.13**	**0.00**	12.93	**30**	37.10	**39.13**	**0.00**	2.03	**2.03**	**30/30**	90.35
	7	**43.63**	**43.63**	**0.00**	28.43	**30**	29.84	**43.63**	**0.00**	7.51	**7.92**	**30/29**	89.54
	8	**45.53**	**45.53**	**0.00**	84.45	**30**	24.83	**45.53**	**0.00**	14.14	**27.62**	**30/25**	84.73
128	3	**36.77**	**36.77**	**0.00**	0.25	**30**	70.96	**36.77**	**0.00**	0.02	**0.02**	**30/30**	85.18
	4	**45.03**	**45.03**	**0.00**	3.20	**30**	60.94	**45.03**	**0.00**	0.37	**0.37**	**30/30**	87.78
	5	**53.43**	**53.43**	**0.00**	13.48	**30**	54.18	**53.43**	**0.00**	3.30	**3.30**	**30/30**	90.42
	6	**61.53**	**61.53**	**0.00**	47.21	**30**	48.04	**61.53**	**0.00**	8.17	**8.17**	**30/30**	92.79
	7	**68.47**	**68.47**	**0.00**	337.66	**30**	39.01	**68.47**	**0.00**	29.22	**36.33**	**30/28**	91.40
	8	**74.60**	74.43	1.43	1941.38	20	33.90	**74.60**	**0.11**	74.85	**1010.01**	**28/12**	68.99
256	3	**55.03**	**55.03**	**0.00**	0.66	**30**	77.45	**55.03**	**0.00**	0.07	**0.07**	**30/30**	89.31
	4	**68.93**	**68.93**	**0.00**	4.99	**30**	74.05	**68.93**	**0.00**	1.82	**1.82**	**30/30**	92.27
	5	**81.43**	**81.43**	**0.00**	41.75	**30**	63.63	**81.43**	**0.00**	12.95	**12.95**	**30/30**	93.83
	6	**93.60**	**93.60**	**0.00**	406.78	**30**	56.99	**93.60**	**0.00**	45.63	**48.11**	**30/28**	93.00
	7	**104.50**	104.40	0.80	1764.49	24	52.06	**104.50**	**0.00**	129.27	**369.43**	**30/20**	87.91
	8	**115.07**	110.77	8.91	3562.90	1	43.12	115.03	**2.70**	375.68	**3167.78**	**10/2**	62.22
512	3	**81.63**	**81.63**	**0.00**	0.83	**30**	86.31	**81.63**	**0.00**	0.58	**0.58**	**30/30**	92.84
	4	**101.13**	**101.13**	**0.00**	10.56	**30**	80.23	**101.13**	**0.00**	9.01	**9.01**	**30/30**	93.71
	5	**121.03**	**121.03**	**0.00**	162.42	**30**	72.44	**121.03**	**0.00**	37.06	**37.06**	**30/30**	95.16
	6	138.40	137.13	1.81	2040.66	21	62.80	**138.60**	**0.00**	143.51	**148.66**	**30/27**	94.86
	7	155.17	126.53	23.97	3570.00	1	55.24	**156.00**	**0.70**	679.41	**2115.36**	**20/10**	82.00
	8	173.07	19.67	90.03	**3633.19**	0	47.95	**174.23**	**3.25**	1221.36	4499.14	**3/1**	64.24

was not necessary to solve the ILP model at all, since Algorithm 6.2 terminated early at Line 11. Hence, the obtained upper bound from the compiled relaxed MDD was equal to the length of the currently best found solution in these cases. Concerning the computation times, the UB+CPLEX approach was on average in only two cases faster than the MDD+CPLEX approach regarding benchmark set Set1 and only in one case regarding benchmark set Set2. Finally, the MDD+CPLEX approach is able to obtain in 135 cases better results than the currently best-known-solutions from the literature. For each considered problem class, MDD+CPLEX is able to provide on average better results than UB+CPLEX.

6.8 Conclusion

In this chapter we approached the RFLCS problem by transforming an instance into a maximum independent set (MIS) problem instance as this is done by Blum et al. [26]. The MIS problem is subsequently solved by the ILP solver CPLEX. Our major contribution is to heavily reduce the conflict graph of the MIS problem by means of relaxed MDDs. This has multiple advantages: (1) reducing the conflict graph leads to an improved performance of CPLEX such that more instances could be solved faster to proven optimality, (2) the compiled relaxed MDDs present a discrete relaxation of the RFLCS problem meaning that upper bounds can be additionally derived and (3) it is also possible to quickly derive heuristic solutions from the MDDs. In many cases it was not necessary anymore to solve the ILP for the MIS problem since the upper bound from the MDD corresponded to the length of the derived heuristic solution and thus optimality was already proven. Overall, for many benchmark instances new state-of-the-art results could be obtained.

In the literature there are works where relaxed decision diagrams are successfully embedded into a branch-and-bound algorithm such that branching is done over nodes in the relaxed decision diagram. Since relaxed MDDs provide also strong upper bonds for the RFLCS problem it may be promising future work to develop such a branch-and-bound algorithm for the RFLCS problem to solve even larger instances to optimality. Moreover, it seems promising to apply relaxed decision diagrams also on other LCS-related problems.

Another promising research direction is to apply the A*-based construction algorithm from Chapter 4 in order to compile the initial relaxed MDD for the IR algorithm in Section 6.6. This implies to adapt the proposed IR algorithm, since the A*-based construction algorithm compiles layer-free relaxed MDDs.

CHAPTER 7

Conclusions

In this thesis we considered different graph search strategies to solve a variety of combinatorial optimization problems (COPs). In particular, we focused on the state space representation of problems where a state represents a set of partial solutions of the considered COP. A transition from one state to another state represents the feasible extension of those partial solutions. Hence, each state contains all necessary information to perform any feasible extension to a successor state and to check if the state represents a complete feasible solution of the COP. This state space is usually represented as a state graph where each node is associated to a state and an outgoing arc represents the transition to a successor state. We applied diverse search techniques on such state graphs to obtain proven optimal solutions as well as heuristic solutions and/or dual bounds.

Chapter 3 focused on a novel anytime A* search algorithm to tackle the job sequencing problem with one common and multiple secondary resources (JSOCMSR). There a set of non-preemptive jobs needs to be scheduled, where each job requires two kinds of resources: A common resource is required for a part of the job's processing time and a secondary resource is required for the whole duration of the job. While the common resource is shared by all jobs, the secondary resources are shared with only a subset of the other jobs. No resource can be claimed by more than one job simultaneously. Hence, the common resource acts like a bottleneck resource since it is claimed by all jobs. The objective is to minimize the makespan over all jobs. The JSOCMSR has its application in the production of certain goods on a single machine and in the field of particle therapy treatment scheduling for cancer patients. Besides an NP-hardness proof of the JSOCMSR, we considered different kinds of lower bounds on the makespan objective. We utilized these lower bound functions in a greedy construction algorithm. Moreover, an A* search algorithm was designed which uses the lower bound functions as search guidance. The A* algorithm is applied on a special state graph structure with nodes holding sets of non-dominated time-records in order to exploit symmetries and to keep the memory consumption reasonable. In order to obtain also primal solutions during the search, the A* search algorithm was turned into an anytime algorithm by using an advanced diving mechanism, i.e., by switching in regular intervals from best-first search to an advanced depth-first search. This diving mechanism uses in essence beam search (BS), which is based on the devised greedy construction algorithm. Furthermore, each obtained primal solution is improved by a local search (LS) procedure. Finally, the proposed anytime A* search prunes nodes that can not lead to a better primal solution than the current best incumbent solution. Our experimental evaluation showed that the anytime A* algorithm is able to obtain high-quality solutions early and exhibits a good anytime behavior. For this purpose we tested the algorithm on very large non-trivial instances with up to 2000 jobs and compared the anytime A* algorithm to the A* based anytime variants anytime pack search (APS) and anytime repairing A* (ARA*), a pure A* algorithm, a position-based mixed integer programming (MIP) model solved by CPLEX, a basic constraint programming (CP) model solved by ILOG CP solver, and an advanced CP model using the same search guidance as A*. For large instances the anytime A* search significantly outperforms the ILOG CP solver; in cases where the ILOG CP solver provides smaller average optimality gaps our approach is usually able to solve more instances to optimality. Overall, the proposed anytime A* search was able to either solve instances with up to 2000 jobs to proven optimality or to obtain solutions with usually small remaining gaps of less than one percent. Moreover, we designed a general variable neighborhood search (GVNS) to obtain even better solutions from hard-to-solve instances. The GVNS features an efficient evaluation scheme to quickly scan insertion and exchange neighborhoods. In this way, we obtained high-quality solutions with an average optimality gap of below 0.288%.

There are still some open research questions on this topic. For instance, the anytime A* algorithm uses a rather simple start strategy to switch from best-first search to BS by following a regular interval. It may be worthwhile to consider different start strategies that depend also on the states stored in the open list of A*. Another interesting investigation is to consider adaptive tuning strategies of the beam width during the BS. Furthermore, the proposed anytime A* algorithm is rather problem-independent, and its applications in other problem domains appears promising.

In Chapter 4 we considered another technique to address COPs that is strongly related to state graphs: decision diagrams (DDs). They provide a graphical representation of the solution space of a COP. In particular, relaxed DDs provide a discrete relaxation of the solution space by superimposing, i.e. merging, nodes of a state graph. This thesis contributed by proposing a novel construction algorithm for non-layered relaxed DDs that is based on the principles of A* search. While traditional construction methods for relaxed DDs are based on layers, e.g., the top-down compilation (TDC), our A*-based construction (A*C) method uses a problem specific fast-to-calculate dual bound function that guides the compilation process such that no layers are necessary. We propose to restrict the number of nodes in the open list instead of restricting the width of each layer. In this way our construction method is able to effectively avoid multiple nodes for identical states at different layers and similar nodes can be merged across different layers. The A*C is especially well suited for problems that include both selection and sequencing aspects. We tested the A*C method on two different NP-hard COPs which have such characteristics. The first problem adapts the JSOCMSR by considering a selection of the jobs rather than minimizing the makespan: The prize-collecting job sequencing problem with one common and multiple secondary resources (PCJSOCMSR) additionally equips each job with a prize and a set of time windows where the job is allowed to be scheduled. The objective is to select a subset of jobs that can be feasibly scheduled and maximizes the total prize over the selected jobs. The aim of this extension was two-folded. First, it is a more accurate model for the real world patient scheduling problem. Second, it becomes suitable for a further application: pre-runtime scheduling of electronic systems within aircraft, called avionic systems. We first proved the NP-hardness of the new PCJSOCMSR and considered different upper bound functions on the total prize, which are based on linear programming and Lagrangian relaxations of a multidimensional knapsack problem formulation. The upper bound functions are evaluated by a classical A* search on the PCJSOCMSR for small-sized instances. Afterwards, the best combination of these upper bounds are used to compile relaxed DDs with A*C for large-sized instances. The second considered problem is the longest common subsequence (LCS) problem consisting on multiple input strings and a finite alphabet. The task is to find a longest subsequence that is common to all input strings. For the A*C we considered two different problem specific upper bound functions from the literature on the LCS as well as a third upper bound which is an adaption of one of the former.

For both problems we could show that the proposed A*C is able to produce relaxed DDs that are stronger relaxations and at the same time are more compact than relaxed DDs created with traditional compilation methods. More specifically, relaxed DDs are created for the PCJSOCMSR with up to 500 jobs with the A*C method as well as with the traditional TDC and incremental refinement (IR) methods. In order to obtain heuristic solutions we create also restricted DDs that provide a graphical representation of only a subset of a COP's solution space. To compile restricted DDs for the PCJSOCMSR we use the traditional TDC approach and significantly speed up the compilation process by using some structural information of a previously compiled relaxed DD with A*C. We compared this overall approach to a GVNS, an order-based MIP model solved by Gurobi Optimizer, and to a basic CP model solved by MiniZinc with backbone solver Chuffed. In general, our approach yielded the best solutions, except for larger instances of the particle therapy benchmark set with three secondary resources, where the GVNS outperforms all other considered methods. For the LCS problem we created relaxed DDs with A*C as well as with TDC on several standard LCS benchmark sets from the literature. Experimental evaluation documented again that A*C is able to compile stronger and more compact DDs than TDC. In particular, the obtained upper bounds from relaxed DDs compiled with A*C are stronger than the LCS-specific upper bound functions from the literature and in some cases even stronger than the currently best known upper bounds for certain problem instances.

In future work it will be interesting to test the proposed compilation method also on other problems that include selection as well as sequencing aspects. One limitation for suitable problems may be the order-invariance of the objective function, which is exploited by the proposed method. Another important aspect for some applications of relaxed DDs is that a once obtained complete DD can be further refined. This property is also known as incrementability and an interesting research question in this regard is if a completely constructed DD can be effectively updated in-place by a following A*-based refinement pass. Other relevant open research questions about A*-based compilation are from a more theoretical side. A constant open list size limit does in general not necessarily imply that the obtained relaxed DD has polynomial size, and consequently also the A*C's compilation time is in general not polynomially bounded. This is similar to the classical A* search, where no better performance guarantee can be given without considering a more specific problem setting and a concrete heuristic function. Studying more advanced methods, possible by developing an entirely different way of deciding when to merge which nodes, in order to obtain possible better performance guarantees is desirable.

Chapter 5 pursued the idea from Chapter 4 further, where a previously compiled relaxed DDs is used to speed-up the compilation process of restricted DDs. However, instead of obtaining heuristic solutions by compiling a restricted DDs, we used a limited discrepancy search (LDS) based approach combined with BS. The previously compiled relaxed DD was further strengthened by removing arcs that belong only to infeasible solutions or to sub-optimal solutions. In this way, a substantial number of possible extensions could be excluded during the LDS/BS hybrid, leaded to a substantial acceleration of the search. This acceleration allowed our approach to search through significantly larger promising parts of the search space than a standalone LDS/BS approach without using a relaxed DD. Another advantage of the proposed approach is that the relaxed DD is able to provide a feasible upper bound of the problem. The algorithm was tested on the PCJSOCMSR with precedence constraints, which is in particular for the avionic system application a reasonable extension of the PCJSOCMSR. The relaxed DD was compiled with the A*C method from Chapter 4 by ignoring the precedence constraints during the compilation step and consider them only in the following filtering step. Thereby obtained relaxed DDs are still a feasible relaxation of the considered problem. We tested our proposed approach on two benchmark sets with up to 500 jobs. Numerical experiments showed that our proposed LDS/BS hybrid with exhibiting a previously compiled and filtered relaxed DD is able to outperform a standalone LDS/BS approach without using a relaxed DD. Our approach is significantly faster and is therefore able to scan larger subspaces of the solution space. Consequently, better heuristic solution can be found. Furthermore, we compared our approach to an order-based MIP model solved by Gurobi Optimizer and a basic CP model solved by MiniZinc with backbone solver Chuffed. In general, the LDS/BS approach provided stronger upper bounds than the MIP approach and better heuristic solutions than both MIP and CP approaches. Next steps to further improve the quality of obtained solutions would be to incorporate other promising filtering techniques from the literature to even further strengthen the relaxed DD. In this way an even greater acceleration of the subsequently applied LDS/BS approach may be achieved. Furthermore, although the advantages of the considered approach was shown on a rather specific scheduling problem, we believe that the general approach also seems to be promising for other COPs. Another promising research direction would be to apply the general idea of using the structural information of relaxed DDs on further search heuristic and metaheuristics.

Finally, this thesis considered in Chapter 6 the repetition-free longest common susequence (RFLCS) problem which consists of two input strings over a finite alphabet. The RFLCS problem asks for the longest subsequence that is common to both input strings and is repetition free, i.e., characters from the alphabet are not allowed to appear more than once. We tackled this problem by transforming an instance of the RFLCS problem into a maximum independent set (MIS) instance, which is subsequently solved by the MIP solver CPLEX. The major contribution of this thesis in this regard is to heavily reduce the underlying conflict graph of the MIS problem by using relaxed DDs in order to accelerate the solving time of the MIP solver. The relaxed DDs for a RFLCS instance are created by an IR approach and exhibit an important property: Each path from the root node to any other node of such a relaxed DD encodes a feasible common subsequence that is not necessarily repetition free. Hence, paths in the relaxed DD do not violate common subsequence constraints but repetition-free constraints. Experimental evaluation showed that such relaxed DDs represent a strong relaxation of the RFLCS problem and can be used to heavily reduce the conflict graph of the corresponding MIS instance. Furthermore, such relaxed DDs can be used to derive high quality heuristic solutions, which allowed us to derive besides an upper bound from a relaxed DD also a primal bound. In 90% of the considered benchmark instances, we were able to prove optimality from the obtained upper- and primal bound from the relaxed DD. Hence, in these cases it was not necessary to solve the corresponding MIS problem at all. Overall, for many benchmark instances new state-of-the-art results could be obtained.

One possible interesting research direction is to embed the relaxed DDs for the RFLCS problem into a DD-based branch-and-bound approach as already done in the literature and branch over exact nodes of the relaxed DD. Since high-quality primal- and upper bounds could be obtained from one relaxed DD, a branch-and-bound approach seems promising to solve even more instances to proven optimality. Another open question in this regard is the construction of relaxed DDs with the A*C method from Chapter 4. Until now, we used an IR-based approach to create relaxed DDs that ensures that no common subsequence constraint is violated. It would be interesting to think of a possibility to adapt the A*C method such that this desired property remains. In this way it may be possible to create with A*C even stronger relaxed DDs, that are at the same time, more compact.

APPENDIX A

Proof of Theorem 3.4.2

Proof. We first show that $\text{MS}^{\text{LB0}} \leq \text{MS}^{\text{LB1}} \leq \text{MS}^{\text{LB2}}$. The definition of MS^{LB1} differs only by the additive terms h_r^1, $r \in R$ from the definition of MS^{LB0}. Since these terms are all non-negative it follows that $\text{MS}^{\text{LB0}} \leq \text{MS}^{\text{LB1}}$. For $\text{MS}^{\text{LB1}} \leq \text{MS}^{\text{LB2}}$ it is sufficient to show that $h_r^1 \leq h_r^2, \forall r \in R$, since the definitions of MS^{LB1} (3.7) and MS^{LB2} (3.8) differ only in the additional strengthened terms h_r^1 and h_r^2. Consider an arbitrary secondary resource $r \in R$. By the definitions of MS^{LB1} and MS^{LB2} both terms h_r^1 and h_r^2 become zero if $J = \emptyset$ or $J_r = \emptyset$ or $J \setminus J_r = \emptyset$. Therefore, let us assume that all three sets contain at least one element. Let $S = \{j \in J \setminus J_r \mid p_j^0 > p_{\max}^{\text{prepost}}(J_r)\}$ be the set of jobs which require the common resource 0 longer than $p_{\max}^{\text{prepost}}(J_r)$ and which do not require the secondary resource r. In this case it holds that $h_r^1 = \sum_{j \in S}(p_j^0 - p_{\max}^{\text{prepost}}(J_r))$. Next, we show that Algorithm 3.2 selects in Line 6 or in Line 11 also each job $j \in S$ for adding the corresponding p_j^0 time to variable h_r^2. At the start of the algorithm $S \subseteq J^0$, since J^0 is initially set to $J \setminus J_r$. At each iteration of the for-loop the job which requires resource 0 the most is selected from set J^0 and removed at the end of the loop. Consider the case that job $j \in S$ is not selected during the loop. There are only two possibilities in which the loop terminates. Either the if condition becomes true and the algorithm terminates, or $J^0 \neq \emptyset \wedge i \leq k$ does not hold anymore. In the former case assume that at iteration i job j_i is selected from set J^0 and $p_{j_i}^0 < g_i$. Since $j_i = \arg\max_{j' \in J^0} p_{j'}^0$ it must also hold that $g_i > p_{j_i}^0 \geq p_j^0$. But for job j it must further hold that $p_j^0 \geq p_{\max}^{\text{prepost}}(J_r) = g_1 \geq g_i$. Hence, we have a contradiction and the if-condition cannot be fulfilled before all jobs from S are selected. The latter case that $J^0 \neq \emptyset \wedge i \leq k$ becomes false remains. Since job j is not selected yet, $J^0 \neq \emptyset$ must hold. This implies that i must be larger than k for terminating the loop. But then job j is still in J^0 and the corresponding p_j^0 will be added to h_r^2 at the end of the algorithm. Hence, p_j^0, $j \in S$ will be added either during the execution of the for loop or afterwards at the end of the algorithm. Let G be the set of all time gaps $[g_i]_{i=1}^k$ that are subtracted from h_r^2 when a job from S is selected in the for-loop. Then, $h_r^2 \geq \sum_{j \in S} p_j^0 - \sum_{g \in G} g \geq \sum_{j \in J}(p_j^0 - p_{\max}^{\text{prepost}}(J_r)) = \sum_{j \in J} p_j^0 - |S| p_{\max}^{\text{prepost}}(J_r) = h_r^1$.

It remains to show that $\mathrm{MS}^{\mathrm{LB2}}$ is a lower bound for the makespan. Assume that there exists a normalized feasible solution s with $\mathrm{MS}(s) < \mathrm{MS}^{\mathrm{LB2}}$. We have to consider two cases:

- $\mathrm{MS}_0^{\mathrm{LB0}} \geq \max_{r \in R} \mathrm{MS}_r^{\mathrm{LB2}}$: The lower bound is determined by the basic lower bound $\mathrm{MS}_0^{\mathrm{LB0}}$. This case has already been shown to yield a contradiction in the proof of Theorem 3.4.1.
- $\mathrm{MS}_0^{\mathrm{LB0}} < \max_{r \in R} \mathrm{MS}_r^{\mathrm{LB2}}$: The lower bound is determined by the strengthened lower bounds. Let $r \in R$ be a secondary resource that determines $\max_{r \in R} \mathrm{MS}_r^{\mathrm{LB2}}$. We have to distinguish between two cases: (1) $\mathrm{MS}(s) < \mathrm{MS}_r^{\mathrm{LB0}}$ and (2) $\mathrm{MS}_r^{\mathrm{LB0}} \leq \mathrm{MS}(s) < \mathrm{MS}_r^{\mathrm{LB0}} + h_r^2$, where $h_r^2 > 0$. In the first case solution s cannot be feasible because the makespan of solution s would be smaller than $\mathrm{MS}_r^{\mathrm{LB0}} = \sum_{r \in J_r} p_j$, which would imply that resource r is used by more than one job at the same time. In the second case we assume that resource r is not used by more than one job at the same time. Since $h_r^2 > 0$ there must be jobs which are selected by Algorithm 3.2 to add the corresponding difference at Line 11 and 6 to h_r^2. Let S be the set of these jobs and $[\sigma_i]_{i=1}^{|S|}$ be the sequence of these jobs ordered according to decreasing times p^0, i.e., $p_{\sigma_i}^0 \geq p_{\sigma_{i+1}}^0$ for all $i = 1, \ldots, |S| - 1$. Furthermore, let $[g_i]_{i=1}^{k}$ be the corresponding sequence of time gaps for resource r. Then $h_r^2 = \sum_{i=1}^{|S|} p_{\sigma_i}^0 - g_i$ where $g_i = 0$ for $i > k$. Now, consider a normalized partial solution s' where exactly the jobs in J_r are scheduled in the exactly same order as in solution s. Then we can determine time gaps between two consecutive jobs in s' regarding the common resource 0. Let $[\bar{g}_i]_{i=1}^{|J_r|+1}$ be the decreasingly sorted sequence of these time gaps. Note, that each element of this sequence is the sum of one preprocessing and one postprocessing time of jobs in J_r. Except for two jobs, which corresponds to the preprocessing time of the first scheduled job from J_r in s' and the postprocessing time of the last scheduled job from J_r in s'. Since Algorithm 3.1 determines the sequence of time gaps $[g_i]_{i=1}^{k}$ by taking at each iteration the maximum pre- and postprocessing times which are not already consumed, it holds that $\bar{g}_i \leq g_i$, $\forall i : 1 \leq i \leq k$. This implies that $h_r^2 \leq \sum_{i=1}^{|S|} p_{\sigma_i}^0 - \bar{g}_i$ where $\bar{g}_i = 0$ for $i > |J_r| + 1$. Now let us insert the jobs from S into solution s' such that job σ_i is placed between the jobs which are associated with time gap $\bar{g}_i$ and causes a delay of $p_{\sigma_i}^0 - \bar{g}_i$, $\forall i : 1 \leq |J_r| + 1$. Jobs σ_i where $i > |J_r| + 1$ are appended to solution s'. Then $\mathrm{MS}_r^{\mathrm{LB0}} + h_r^2 \leq \mathrm{MS}_r^{\mathrm{LB0}} + \sum_{i=1}^{|S|} p_{\sigma_i}^0 - \bar{g}_i \leq \mathrm{MS}(s')$ where $\bar{g}_i = 0$ for $i > |J_r| + 1$. Note that any different schedule order of jobs in S will cause an increase of the delay and therefore of the makespan of solution s'. This holds in particular if we choose the same schedule order for the jobs in S as in solution s. Therefore $\mathrm{MS}(s') \leq \mathrm{MS}(s)$. This contradicts the assumption that $\mathrm{MS}(s) < \mathrm{MS}_r^{\mathrm{LB0}} + h_r^2$.

We conclude that $\mathrm{MS}^{\mathrm{LB2}}$ is indeed a lower bound for MS^*. Since $\mathrm{MS}^{\mathrm{LB1}} \leq \mathrm{MS}^{\mathrm{LB2}}$ it follows that also $\mathrm{MS}^{\mathrm{LB1}}$ is indeed a lower bound for MS^*. □

APPENDIX B

Further Results for the PCJSOCMSR

Figure B.1 shows additional results of the PCJSOCMSR from Chapter 4 for instances of sets P and A with three and four secondary resources, respectively. As in Section 4.4.10 the impact of the open list size limit ϕ and different labeling functions are analyzed and conclusions are similar: In general, with increasing ϕ the lengths of the longest paths of the obtained relaxed multivalued decision diagrams (MDDs) from A*C get smaller, while the computation times and the MDD sizes increase. The smallest relaxed MDDs with the weakest bounds could in general be obtained from labeling function L^1 whereas labeling function L^4 typically provides the largest MDDs with the strongest bounds.

Regarding the comparison of upper bounds obtained from different approaches, Figure B.2 shows in addition to the results presented in Section 4.4.10 corresponding results for instances of set P and A with three and four secondary resources, respectively. Average values of upper bounds, computation times, and the sizes of relaxed MDDs, obtained from A*C, the classical TDC, the IR, and the order based MIP approach are shown. Again, we observe remarkable differences between results obtained from instances of set P and A. Nevertheless, in each case the strongest average upper bounds could be obtained by A*C thereby creating as well the smallest obtained relaxed MDDs.

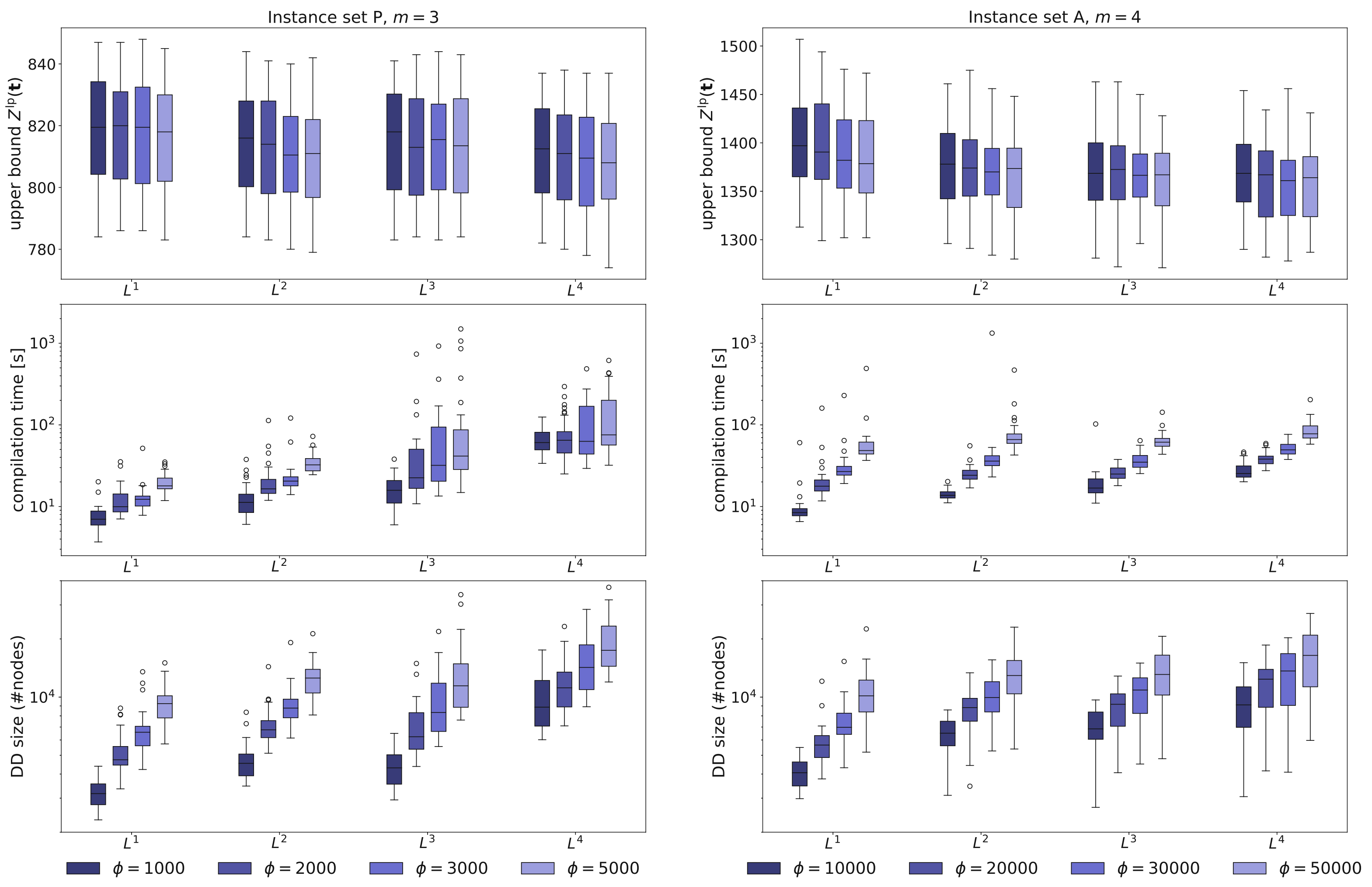


Figure B.1: Comparison of open list size limits ϕ and labeling functions L^i, $i = 1, \ldots, 4$, for instances of sets P and A with 250 jobs and $m = 3$ and $m = 4$ secondary resources, respectively.

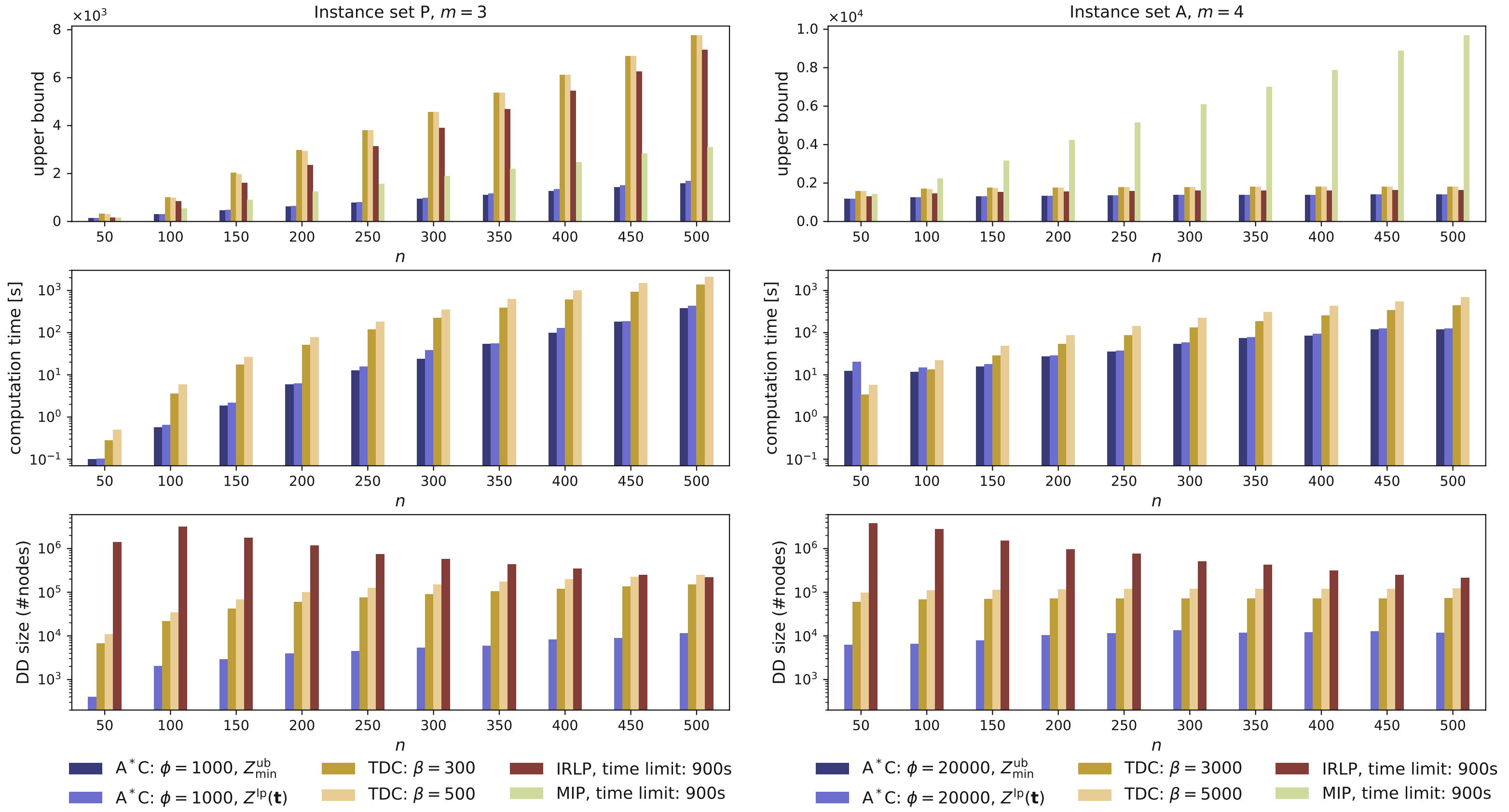


Figure B.2: Instance sets P and A with three and four secondary resources, respectively, average values of: upper bounds obtained from A*C, the classical TDC, the IR, and the order-based MIP approach; respective computation times; and the sizes of obtained relaxed MDDs.

Acronyms

A*C A*-based construction. 7, 8, 93, 94, 96, 102, 106, 108, 111, 120, 127, 132, 133, 135–142, 144, 145, 147, 148, 150–157, 162, 164, 165, 177, 188–191, 195, 197

ACO ant colony optimization. 32

ANA* anytime nonparametric A*. 49

APS anytime pack search. 50, 78, 79, 91, 188

APTS anytime potential search. 49

ARA* anytime repairing A*. 49, 74, 84, 85, 88, 91, 188

AWA* anytime window A*. 50

BB branch and bound. 23, 24, 27, 30, 40, 96, 102, 106, 171

BDD binary decision diagram. 38, 94–96

BFS breadth-first search. 14, 15, 20, 65

BS beam search. ix, x, 3–6, 8, 20, 22, 23, 33, 39, 45, 46, 50, 62–65, 67, 76–79, 82, 86–88, 92, 143, 144, 159, 160, 168, 188, 190

CMSA construct, merge, solve and adapt. 171

COP combinatorial optimization problem. ix, x, 2, 3, 9–12, 23–27, 29–31, 33, 38–40, 42, 43, 94, 159, 187–189, 191

CP constraint programming. 6–8, 28, 29, 37, 46, 48, 49, 68, 74, 84, 86, 89–92, 106, 119, 127, 132, 133, 137, 140, 141, 156, 159, 165, 167, 168, 188–190

CSP constraint satisfaction problem. 28, 29

DD decision diagram. ix, x, 3–9, 37–43, 94–102, 106, 120, 121, 124, 152, 156, 157, 159, 160, 170, 171, 181, 188–191

DFS depth-first search. 14–16

DP dynamic programming. 30, 31, 38–40, 43, 97, 101, 113, 143, 145, 170, 174, 176

GA genetic algorithm. 32

GRASP greedy randomized adaptive search procedure. 47

GVNS general variable neighborhood search. 6, 36, 45, 46, 69, 71, 73, 88–92, 110, 111, 137, 140, 141, 156, 188, 189

IG iterated greedy. 47, 48

ILDS improved limited discrepancy search. 21, 22

ILP integer linear programming. x, 7, 26, 27, 42, 43, 169–172, 180, 183, 186

IR incremental refinement. 40, 42, 43, 98, 100, 110, 135–137, 155, 176, 177, 186, 189, 191, 195, 197

JSOCMSR job sequencing problem with one common and multiple secondary resources. ix–xii, 4–6, 47–53, 56, 60, 63, 67–69, 93, 106, 107, 110, 119, 187, 189

LAPCS longest arc-preserving common subsequence. 171

LCPS longest common palindromic subsequence. 171

LCS longest common subsequence. x, xii, 5, 7, 94–97, 99, 100, 102, 123, 142–147, 152, 155, 156, 170, 171, 177, 179, 186, 189

LDS limited discrepancy search. x, 4, 5, 8, 21, 22, 33, 159, 160, 163–165, 168, 190

LLBH least lower bound heuristic. 6, 56, 58–62, 68, 74, 76–78, 84–87, 89, 91

LP linear programming. 10, 25–28, 39, 108, 113–115, 129, 130

LS local search. 6, 31–35, 45, 46, 64, 65, 67, 76–79, 81, 82, 86–88, 91, 188

MDD multivalued decision diagram. 6, 8, 38, 93–106, 108, 110–112, 120–127, 132–138, 140–142, 144–152, 154–157, 161, 162, 164, 165, 168–180, 183, 186, 195

MILP mixed integer linear programming. 6, 8, 25, 27, 28, 37, 46, 67, 68, 74, 84, 169

MIP mixed integer programming. 91, 98, 106, 108, 118, 119, 127, 132, 133, 135–137, 140, 141, 143, 156, 159, 165, 167, 168, 172, 188–191, 195, 197

MIS maximum independent set. x, xii, 4, 8, 169–173, 186, 191

NDT non-dominated times. 60–62, 64–66, 83, 84

OP orienteering problem. 111

PCJSOCMSR prize-collecting job sequencing problem with one common and multiple secondary resources. 5, 7, 8, 93–95, 97, 99, 100, 102, 106–111, 113, 116–122, 126, 127, 132, 137, 142, 148, 155, 156, 159–165, 168, 189, 190, 195

PP partition problem. 51, 52

PSO particle swarm optimization. 32

RCPSP resource constrained project scheduling problem. 48

RFLCS repetition-free longest common susequence. x, xii, 5, 6, 8, 169–176, 178, 179, 186, 191

RWA* restarting weighted A*. 49

SA simulating annealing. 31

SQT solution quality over time. 79–83

TDC top-down compilation. 39–41, 98, 100, 106, 108, 110, 127, 135–142, 148–156, 159, 188, 189, 195, 197

TS tabu search. 31, 32

TSP traveling salesperson problem. 37, 48, 95

VGSCO Vienna Graduate School on Computational Optimization. vii, 2

VND variable neighborhood descent. 35, 36, 69, 71, 73, 91

VNS variable neighborhood search. 31, 32, 36

VRP vehicle routing problem. 2

Bibliography

[1] S. S. Adi, M. D. Braga, C. G. Fernandes, C. E. Ferreira, F. V. Martinez, M.-F. Sagot, M. A. Stefanes, C. Tjandraatmadja, and Y. Wakabayashi. Repetition-free longest common subsequence. *Discrete Applied Mathematics*, 158(12):1315–1324, 2010.

[2] A. Aho, J. Hopcroft, and J. Ullman. *Data structures and algorithms*. Addison-Wesley, 1983.

[3] S. Aine, P. P. Chakrabarti, and R. Kumar. AWA* – a window constrained anytime heuristic search algorithm. In R. Sangal, H. Mehta, and R. K. Bagga, editors, *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*, pages 2250–2255, Hyderabad, India, 2007. Morgan Kaufmann Publishers.

[4] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[5] A. Allahverdi. A survey of scheduling problems with no-wait in process. *European Journal of Operational Research*, 255(3):665–686, 2016.

[6] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Principles and Practice of Constraint Programming, CP 2007*, volume 4741 of *LNCS*, pages 118–132. Springer, 2007.

[7] C. Ansótegui, M. Bofill, M. Palahı, J. Suy, and M. Villaret. Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem. In *Proceedings of the 9th Symposium on Abstraction, Reformulation and Approximation, SARA 2011*, pages 2–9. AAAI Press, 2011.

[8] R. Beal, T. Afrin, A. Farheen, and D. Adjeroh. A new algorithm for "the LCS problem" with application in compressing genome resequencing data. *BMC Genomics*, 17(4):544, 2016.

[9] J. C. Beck and L. Perron. Discrepancy-bounded depth first search. In *Proceedings of the Constraint Programming, Artificial Intelligence and Operations Research*, 2000.

[10] R. Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences, PNAS*, 38(8):716–719, 1952.

[11] R. Bellman. *Dynamic programming*. Princeton Univ. Press, Princeton, NJ, 1957.

[12] D. Bergman and A. A. Cire. On finding the optimal BDD relaxation. In D. Salvagnin and M. Lombardi, editors, *Proceedings of the Fourteenth International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, CPAIOR 2017*, volume 10335 of *LNCS*, pages 41–50. Springer, 2017.

[13] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Variable ordering for the application of bdds to the maximum independent set problem. In N. Beldiceanu, N. Jussien, and É. Pinson, editors, *Proceedings of the Ninth International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Contraint Programming for Combinatorial Optimzation Problems, CPAIOR 2012*, volume 7298 of *LNCS*, pages 34–49. Springer, 2012.

[14] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016.

[15] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.

[16] D. Bergman, A. A. Cire, W.-J. van Hoeve, and T. Yunes. BDD-based heuristics for binary optimization. *Journal of Heuristics*, 20(2):211–234, 2014.

[17] D. Bergman, A. A. Cire, W.-J. von Hoeve, and J. N. Hooker. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing*, 26(2):253–268, 2014.

[18] D. P. Bertsekas. *Dynamic programming and optimal control*, volume 1 and 2. Athena Scientific, 3 edition, 2001.

[19] D. Bertsimas and J. N. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1 edition, 1997.

[20] H.-G. Beyer and H.-P. Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[21] M. Blikstad, E. Karlsson, T. Lööw, and E. Rönnberg. An optimisation approach for pre-runtime scheduling of tasks and communication in an integrated modular avionic system. *Optimization and Engineering*, 19(4):977–1004, 2018.

[22] C. Blum and M. J. Blesa. Probabilistic beam search for the longest common subsequence problem. In T. Stützle, M. Birratari, and H. H. Hoos, editors, *Proceedings of SLS 2007 – The 1st International on Engineering Stochastic Local Search Algorithms*, volume 4638 of *LNCS*, pages 150–161. Springer, 2007.

[23] C. Blum and M. J. Blesa. Construct, merge, solve and adapt: application to the repetition-free longest common subsequence problem. In F. Chicano, B. Hu, and P. García-Sánchz, editors, *Proceedings of the 16th European Conference on Evolutionary Computation in Combinatorial Optimization, EvoCOP 2016*, volume 9595 of *LNCS*, pages 46–57. Springer, 2016.

[24] C. Blum and M. J. Blesa. A comprehensive comparison of metaheuristics for the repetition-free longest common subsequence problem. *Journal of Heuristics*, 24(3):551–579, 2018.

[25] C. Blum, M. J. Blesa, and M. López-Ibáñez. Beam search for the longest common subsequence problem. *Computers & Operations Research*, 36(12):3178–3186, 2009.

[26] C. Blum, M. Djukanovic, A. Santini, H. Jiang, C.-M. Li, F. Manya, and G. R. Raidl. Solving longest common subsequence problems via a transformation to the maximum clique problem. *Computers & Operations Research*, 125, 2021. doi.org/10.1016/j.cor.2020.105089.

[27] C. Blum and P. Festa. Longest common subsequence problems. In *Metaheuristics for String Problems in Bioinformatics*, chapter 3, pages 45–60. Iste Wiley, 2016.

[28] C. Blum and G. R. Raidl. *Hybrid Metaheuristics: Powerful Tools for Optimization*. Springer, 2016.

[29] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *Association for Computing Machinery (ACM) Computing Surveys*, 35(3):268–308, 2003.

[30] P. Bonizzoni, G. Della Vedova, and G. Mauri. Experimenting an approximation algorithm for the LCS. *Discrete Applied Mathematics*, 110(1):13–24, 2001.

[31] P. Bonizzoni, G. D. Vedova, R. Dondi, and Y. Pirola. Variants of constrained longest common subsequence. *Information Processing Letters*, 110(20):877–881, 2010.

[32] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of the 41st annual Design Automation Conference, DAC 2004*, pages 395–400. IEEE press, 2004.

[33] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[34] Q. Cappart, E. Goutierre, D. Bergman, and L.-M. Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451. AAAI Press, 2019.

[35] M. Castelli, S. Beretta, and L. Vanneschi. A hybrid genetic algorithm for the repetition free longest common subsequence problem. *Operations Research Letters*, 41(6):644–649, 2013.

[36] H.-T. Chan, C.-B. Yang, and Y.-H. Peng. The generalized definitions of the two-dimensional largest common substructure problems. In *Proceedings of the 33rd Workshop on Combinatorial Mathematics and Computation Theory*, pages 1–12. National Taiwan University, Department of Mathematics, 2016.

[37] Y.-C. Chen and K.-M. Chao. On the generalized constrained longest common subsequence problems. *Journal of Combinatorial Optimization*, 21(3):383–392, 2011.

[38] S. R. Chowdhury, M. M. Hasan, S. Iqbal, and M. S. Rahman. Computing a longest common palindromic subsequence. *Fundamenta Informaticae*, 129(4):329–340, 2014.

[39] N. Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981.

[40] A. A. Cire and W.-J. van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, 2013.

[41] D. Conforti, F. Guerriero, and R. Guido. Optimization models for radiotherapy patient scheduling. *4OR - A Quarterly Journal of Operations Research*, 6(3):263–278, 2008.

[42] M. Conforti, G. Cornuéjols, and G. Zambelli. *Integer programming*, volume 271. Springer, 2014.

[43] R. Cordone, P. Hosteins, and G. Righini. A branch-and-bound algorithm for the prize-collecting single-machine scheduling problem with deadlines and total tardiness minimization. *INFORMS Journal on Computing*, 30(1):168–180, 2018.

[44] G. B. Dantzig. *Origins of the Simplex Method*, pages 141—151. Association for Computing Machinery (ACM), New York, NY, USA, 1990.

[45] G. B. Dantzig. Linear programming. *INFORMS Operations research*, 50(1):42–47, 2002.

[46] J. M. Davis, H. Topaloglu, and D. P. Williamson. Assortment optimization over time. *Operations Research Letters*, 43(6):608–611, 2015.

[47] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[48] E. V. Denardo. *Dynamic programming: models and applications.* Dover Publications, 2003.

[49] M. Djukanovic, G. R. Raidl, and C. Blum. Exact and heuristic approaches for the longest common palindromic subsequence problem. In R. Battiti, M. Brunato, I. Kotsireas, and P. M. Pardalos, editors, *Proceedings of the 12th International Conference on Learning and Intelligent Optimization, LION 12*, volume 11353 of *LNCS*, pages 199–214. Springer, 2019.

[50] M. Djukanovic, G. R. Raidl, and C. Blum. A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation. In G. Nicosia, P. Pardalos, G. Giuffrida, R. Umeton, and V. Sciacca, editors, *Proceedings of LOD 2019 – The 5th International Conference on Machine Learning, Optimization and Data Science*, volume 11943 of *LNCS*, pages 154–167. Springer, 2020.

[51] M. Djukanovic, G. R. Raidl, and C. Blum. Finding longest common subsequences: New anytime A* search results. *Applied Soft Computing*, 95(106499), 2020.

[52] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[53] T. Easton and A. Singireddy. A large neighborhood search heuristic for the longest common subsequence problem. *Journal of Heuristics*, 14(3):271–283, 2008.

[54] I. Fister Jr, X.-S. Yang, I. Fister, J. Brest, and D. Fister. A brief review of nature-inspired algorithms for optimization. *arXiv preprint arXiv:1307.4186*, 2013.

[55] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the Association for Computing Machinery (ACM)*, 5(6):345, 1962.

[56] M. S. Fox. *Constraint-directed search: a case study of job-shop scheduling*. PhD thesis, Carnegie-Mellon University, 1983.

[57] C. B. Fraser. *Subsequences and supersequences of strings*. PhD thesis, University of Glasgow, UK, 1995.

[58] D. Furcy and S. Koenig. Limited discrepancy beam search. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI'05*, pages 125–131, Edinburgh, Scotland, 2005. Morgan-Kaufmann.

[59] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[60] Z. W. Geem, J. H. Kim, and G. V. Loganathan. A new heuristic optimization algorithm: harmony search. *Simulation*, 76(2):60–68, 2001.

[61] P. C. Gilmore and R. E. Gomory. Sequencing a one-state variable machine: A solvable case of the traveling salesman problem. *Operations Research*, 12(5):655–679, 1964.

[62] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research (COR)*, 13(5):533–549, 1986.

[63] J. E. González, A. A. Cire, A. Lodi, and L.-M. Rousseau. Integrated integer programming and decision diagram search tree with an application to the maximum independent set problem. *Constraints*, 25:1–24, 2020.

[64] A. Gunawan, H. C. Lau, and P. Vansteenwegen. Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, 255(2):315–332, 2016.

[65] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[66] E. A. Hansen and R. Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007.

[67] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research (EJOR)*, 130(3):449–467, 2001.

[68] P. Hansen, N. Mladenović, J. Brimberg, and J. A. M. Pérez. Variable neighborhood search. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, pages 61–86. Springer, Boston, MA, 2010.

[69] P. Hansen, N. Mladenović, R. Todosijević, and S. Hanafi. Variable neighborhood search: basics and variants. *EURO Journal on Computational Optimization*, 5(3):423–454, 2017.

[70] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[71] S. Hartmann and D. Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, 2010.

[72] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In C. S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI'95*, pages 607–615, Montreal, Que Canada, 1995. Morgan-Kaufmann.

[73] Y.-C. Ho and D. L. Pepyne. Simple explanation of the no-free-lunch theorem and its implications. *Journal of optimization theory and applications*, 115(3):549–570, 2002.

[74] J. N. Hooker. Decision diagrams and dynamic programming. In C. Gomes and M. Sellmann, editors, *Proceedings of the tenth International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in*

Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2013, LNCS, pages 94–110, Yorktown Heights, NY, USA, 2013. Springer.

[75] J. N. Hooker. Job sequencing bounds from decision diagrams. In *Proceedings of 23rd International Conference on the Principles and Practice of Constraint Programming, CP 2017*, volume 10416 of *LNCS*, pages 565–578. Springer, 2017.

[76] M. Horn. A heuristic framework for dynamic vehicle routing with site-dependent constraints. Master's thesis, TU Wien, 2017.

[77] M. Horn, M. Djukanovic, C. Blum, and G. R. Raidl. On the use of decision diagrams for finding repetition-free longest common subsequences. In N. Olenev, Y. Evtushenko, M. Khachay, and V. Malkova, editors, *Proceedings of the XI International Conference Optimization and Applications, OPTIMA 2020*, volume 12422 of *LNCS*, pages 134–149. Springer, 2020.

[78] M. Horn, J. Maschler, G. R. Raidl, and E. Rönnberg. A*-based construction of decision diagrams for a prize-collecting scheduling problem. *Computers & Operations Research (COR)*, 126:105125, 2021.

[79] M. Horn, G. Raidl, and C. Blum. Job sequencing with one common and multiple secondary resources: An A*/Beam Search based anytime algorithm. *Artificial Intelligence*, 277(103173), 2019.

[80] M. Horn and G. R. Raidl. Decision diagram based limited discrepancy search for a job sequencing problem. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *Proceedings of the 17th International Conference of Computer Aided Systems Theory, EUROCAST 2019*, volume 12013 of *LNCS*, pages 344–351. Springer, 2020.

[81] M. Horn and G. R. Raidl. A*-based compilation of relaxed decision diagrams for the longest common subsequence problem. volume 12735 of *LNCS*, 2021. To appear. Accepted for the 18th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR 2021.

[82] M. Horn, G. R. Raidl, and C. Blum. Job sequencing with one common and multiple secondary resources: A problem motivated from particle therapy for cancer treatment. In G. Giuffrida, G. Nicosia, P. Pardalos, and R. Umeton, editors, *MOD 2017: Machine Learning, Optimization, and Big Data – Third International Conference*, volume 10710 of *LNCS*, pages 506–518. Springer, 2017.

[83] M. Horn, G. R. Raidl, and E. Rönnberg. An A* algorithm for solving a prize-collecting sequencing problem with one common and multiple secondary resources and time windows. In E. K. Burke, L. Di Gaspero, B. McCollum, N. Musliu, and E. Özcan, editors, *Proceedings of the 12th International Conference of the Practice and Theory of Automated Timetabling, PATAT 2018*, pages 235–256, Vienna, Austria, 2018.

[84] M. Horn, G. R. Raidl, and E. Rönnberg. A* search for prize-collecting job sequencing with one common and multiple secondary resources. *Annals of Operations Research*, 2020.

[85] A. J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. PhD thesis, Stanford University, Stanford, CA, USA, 1995. CS-TR-95-1561.

[86] K. Huang, C. Yang, and K. Tseng. Fast algorithms for finding the common subsequences of multiple sequences. In *Proceedings of the IEEE International Computer Symposium*, pages 1006–1011. IEEE Press, 2004.

[87] T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2):371–388, 2002.

[88] T. Jiang, G.-H. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. In R. Giancarlo and D. Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, CPM 2000*, volume 1848 of *LNCS*, pages 154–165. Springer, 2000.

[89] T. Kapamara and D. Petrovic. A heuristics and steepest hill climbing method to scheduling radiotherapy patients. In *Proceedings of the International Conference on Operational Research Applied to Health Services*, pages 1–7, 2009.

[90] T. Kapamara, K. Sheibani, O. Haas, D. Petrovic, and C. Reeves. A review of scheduling problems in radiotherapy. In *Proceedings of the International Control Systems Engineering Conference*, pages 207–211. Coventry University Publishing, 2006.

[91] E. Karlsson, E. Rönnberg, A. Stenberg, and H. Uppman. A matheuristic approach to large-scale avionic scheduling. *Annals of Operations Research*, pages 1–35, 2020.

[92] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 302–311, New York, NY, USA, 1984. Association for Computing Machinery (ACM).

[93] W. Karoui, M.-J. Huguet, P. Lopez, and W. Naanaa. Yields: A yet improved limited discrepancy search for csps. In P. Van Hentenryck and L. Wolsey, editors, *Proceedings of the 4th International Conference of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2007:*, pages 99–111. Springer, 2007.

[94] T. Kaufmann. A variable neighborhood search for the job sequencing with one common and multiple secondary resources problem. Master's thesis, TU Wien, Vienna, Austria, 2019.

[95] T. Kaufmann, M. Horn, and G. R. Raidl. A variable neighborhood search for the job sequencing with one common and multiple secondary resources problem. In T. Bäck, M. Preuss, A. Deutz, H. Wang, C. Doerr, M. Emmerich, and H. Trautmann, editors, *Proceedings of PPSN XVI: Parallel Problem Solving from Nature*, volume 12270 of *LNCS*, pages 385–398. Springer, 2020.

[96] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[97] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of International Conference on Neural Networks, ICNN'95*, volume 4, pages 1942–1948. Institute of Electrical and Electronics Engineers (IEEE), 1995.

[98] L. Khachiyan. Polynomial algorithms in linear programming (english translation). *Soviet Mathematics Doklady*, 20(1):191–194, 1979.

[99] J. Kinable, A. A. Cire, and W. J. van Hoeve. Hybrid optimization methods for time-dependent sequencing problems. *European Journal of Operational Research*, 259(3):887–897, 2017.

[100] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[101] V. Klee and G. J. Minty. How good is the simplex algorithm? In Shisha, editor, *Inequalities III*, pages 159–175, New York, 1972. Academic Press.

[102] R. E. Korf. Improved limited discrepancy search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI'96, Vol. 1*, pages 286–291, Portland, Oregon, 1996.

[103] D. Kowalczyk and R. Leus. A branch-and-price algorithm for parallel machine scheduling using ZDDs and generic branching. *INFORMS Journal on Computing*, 30(4):768–782, 2018.

[104] J. B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, 1983.

[105] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[106] J. Y. Lee and Y. D. Kim. Minimizing the number of tardy jobs in a single-machine scheduling problem with periodic maintenance. *Computers and Operations Research*, 39:2196–2205, 2012.

[107] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma, and J. Huang. A novel fast and memory efficient parallel mlcs algorithm for long and large-scale sequences alignments. In *Proceedings of the 32nd International Conference on Data Engineering, ICDE 2016*, pages 1170–1181. IEEE Press, 2016.

[108] M. Likhachev, G. J. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Proceedings of the Conference Advances in Neural Information Processing Systems 16*, pages 767–774. MIT Press, 2004.

[109] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[110] B. T. Lowerre. *The HARPY Speech Recognition System.* PhD thesis, Carnegie-Mellon University, 1976.

[111] A. K. Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[112] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[113] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, USA, 1990.

[114] J. Maschler. *Patient scheduling in particle therapy.* PhD thesis, Institute of Logic and Computation, TU Wien, Vienna, Austria, 2019. supervised by Günther R. Raidl.

[115] J. Maschler, T. Hackl, M. Riedler, and G. R. Raidl. An enhanced iterated greedy metaheuristic for the particle therapy patient scheduling problem. In *Proceedings of the 12th Metaheuristics International Conference, MIC 2017*, pages 465–474, 2017.

[116] J. Maschler and G. R. Raidl. Multivalued decision diagrams for a prize-collecting sequencing problem. In *Proceedings of the 12th International Conference of the Practice and Theory of Automated Timetabling, PATAT 2018*, pages 375–397, Vienna, Austria, 2018.

[117] J. Maschler and G. R. Raidl. Multivalued decision diagrams for prize-collecting job sequencing with one common and multiple secondary resources. *Annals of Operations Research*, 2019.

[118] J. Maschler and G. R. Raidl. Particle therapy patient scheduling with limited starting time variations of daily treatments. *International Transactions in Operational Research*, 27(1):458–479, 2020.

[119] J. Maschler, M. Riedler, and G. R. Raidl. Particle therapy patient scheduling: Time estimation for scheduling sets of treatments. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *Proceedings of the 16th International Conference of Computer Aided Systems Theory, EUROCAST 2017*, volume 10671 of *LNCS*, pages 364–372. Springer, 2018.

[120] J. Maschler, M. Riedler, M. Stock, and G. R. Raidl. Particle therapy patient scheduling: First heuristic approaches. In *Proceedings of the 11th International Conference of the Practice and Theory of Automated Timetabling, PATAT 2016*, pages 223–244, 2016.

[121] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference, DAC'93*, pages 272–277. Association for Computing Machinery (ACM), 1993.

[122] S. Minato. πDD: A new decision diagram for efficient problem solving in permutation space. In *Proceedings of the 30th International Conference on Theory and Applications of Satisfiability Testing, SAT 2011*, volume 6695 of *LNCS*, pages 90–104. Springer, 2011.

[123] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research (COR)*, 24(11):1097–1100, 1997.

[124] J. M. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15:102–109, 1968.

[125] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[126] C. Oğuz, F. Sibel Salman, and Z. Bilgintürk Yalçin. Order acceptance and scheduling decisions in make-to-order systems. *International Journal of Production Economics*, 125(1):200–211, 2010.

[127] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover Publications, New York, 1998.

[128] Z. Peng and Y. Wang. A novel efficient graph model for the multiple longest common subsequences (MLCS) problem. *Frontiers in Genetics*, 8:104, 2017.

[129] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 5th edition, 2015.

[130] I. Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3):193–204, 1970.

[131] D. L. Poole and A. K. Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[132] P. Prosser and C. Unsworth. Limited discrepancy search revisited. *ACM Journal of Experimental Algorithmics*, 16, 2011.

[133] S. Richter, J. T. Thayer, and W. Ruml. The Joy of Forgetting: Faster Anytime Search via Restarting. In R. Brafman, H. Geffner, J. Hoffmann, and H. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning*

and Scheduling, ICAPS 2010, pages 137–144, Toronto, Ontario, Canada, 2010. AAAI Press.

[134] L. H. O. Rios and L. Chaimowicz. A Survey and Classification of A* Based Best-First Heuristic Search Algorithms. In A. C. da Rocha Costa, R. M. Vicari, and F. Tonidandel, editors, *Proceedings of Advances in Artificial Intelligence: 20th Brazilian Symposium on Artificial Intelligence*, volume 6404 of *LNCS*, pages 253–262. Springer, 2010.

[135] H. Röck. The three-machine no-wait flow shop is NP-complete. *Journal of the ACM*, 31(2):336–345, 1984.

[136] M. Römer, A. A. Cire, and L.-M. Rousseau. A local search framework for compiling relaxed decision diagrams. In *Proceedings of the 15th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR 2018*, volume 10848 of *LNCS*, pages 512–520. Springer, 2018.

[137] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Foundations of Artificial Intelligence. Elsevier, 2006.

[138] S. Rubin. *The ARGOS Image Understanding System*. PhD thesis, Carnegie-Mellon University, 1978.

[139] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice Hall, 2002.

[140] I. Sabuncuoglu and M. Bayiz. Job shop scheduling with beam search. *European Journal of Operational Research*, 118(2):390–412, 1999.

[141] T. Schiavinotto and T. Stützle. A review of metrics on permutations for search landscape analysis. *Computers & Operations Research (COR)*, 34(10):3143–3153, 2007.

[142] A. Schutt, T. Feydy, P. J. Stuckey, and M. G. Wallace. Solving RCPSP/max by lazy clause generation. *Journal of Scheduling*, 16(3):273–289, 2013.

[143] S. J. Shyu and C.-Y. Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Computers & Operations Research*, 36(1):73–91, 2009.

[144] Y. L. T. Silva, A. Subramanian, and A. A. Pessoa. Exact and heuristic algorithms for order acceptance and scheduling with sequence-dependent setup times. *Computers and Operations Research*, 90:142–160, 2018.

[145] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[146] P. F. Stadler and W. Schnabl. The landscape of the traveling salesman problem. *Physics Letters A*, 161(4):337–344, 1992.

[147] R. Stern, A. Felner, J. van den Berg, R. Puzis, R. Shah, and K. Goldberg. Potential-based bounded-cost search and anytime non-parametric A*. *Artificial Intelligence*, 214:1–25, 2014.

[148] R. Stern, R. Puzis, and A. Felner. Potential search: A new greedy anytime heuristic search. In N. Sturtevant and F. Ariel, editors, *Proceedings of the 3rd Annual Symposium on Combinatorial Search, SOCS 2010*, pages 119–120. AAAI Press, 2010.

[149] R. Stern, R. Puzis, and A. Felner. Potential search: A bounded-cost search algorithm. In F. Bacchus, C. Domshlak, S. Edelkamp, and M. Helmert, editors, *Proceedings of the 21th International Conference on Automated Planning and Scheduling, ICAPS 2011*, pages 234–241. AAAI Press, 2011.

[150] J. A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, Inc., 1987.

[151] K. Sörensen. Metaheuristics—the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[152] E.-G. Talbi. *Metaheuristics: from design to implementation*. John Wiley & Sons, Hobokon, New Yersey, USA, 2009.

[153] C. Tjandraatmadja and W.-J. van Hoeve. Incorporating bounds from decision diagrams into integer programming. *Mathematical Programming Computation*, pages 1–32, 2020.

[154] Y.-T. Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4):173–176, 2003.

[155] S. G. Vadlamudi, S. Aine, and P. P. Chakrabarti. Anytime pack search. *Natural Computing*, 15(3):395–414, 2016.

[156] J. Van Den Berg, R. Shah, A. Huang, and K. Goldberg. ANA*: anytime non-parametric A*. In *Proceedings of Twenty-fifth AAAI Conference on Artificial Intelligence, AAAI 2011*, pages 105–111, San Francisco, California, USA, 2011. AAAI Press.

[157] J. A. A. Van der Veen, G. J. Wöginger, and S. Zhang. Sequencing jobs that require common resources on a single machine: A solvable case of the TSP. *Mathematical Programming*, 82(1–2):235–254, 1998.

[158] P. Van Hentenryck. *Constraint Satsifaction in Logic Programming*. MIT Press, 1989.

[159] M. C. Vélez-Gallego, J. Maya, and J. R. Montoya-Torres. A beam search heuristic for scheduling a single machine with release dates and sequence dependent setup times to minimize the makespan. *Computers & Operations Research (COR)*, 73:132–140, 2016.

[160] T. Walsh. Depth-bounded discrepancy search. In M. E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI'97*, volume 2, pages 1388–1393, Nagoya, Japan, 1997. Morgan-Kaufmann.

[161] Q. Wang, D. Korkin, and Y. Shang. A fast multiple longest common subsequence (mlcs) algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):321–334, 2011.

[162] S. Warshall. A theorem on Boolean matrices. *Journal of the Association for Computing Machinery (ACM)*, 9(1):11–12, 1962.

[163] J.-P. Watson. *An Introduction to Fitness Landscape Analysis and Cost Models for Local Search*, pages 599–623. Springer US, Boston, MA, 2010.

[164] D. Weyland. A rigorous analysis of the harmony search algorithm: How the research community can be misled by a "novel" methodology. *International Journal of Applied Metaheuristic Computing (IJAMC)*, 1(2):50–60, 2010.

[165] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[166] L. A. Wolsey. *Integer programming.* John Wiley & Sons, 1998.

Wiesfeldstraße 28
3130 Herzogenburg, Austria
✉ *matthias.g.horn@gmail.com*
D.O.B: 25.08.1989
Birth place: St. Pölten
Nationality: Austria

Matthias Horn

Curriculum Vitae

Educational Background

2017

Doctoral programme in Engineering Science, *TU Wien*.

Field of Study	Combinatorial Optimization
Advisor	Günther Raidl

2014 2017

Master of Science, *TU Wien*.

Field of Study	Computer Engineering
Thesis	A heuristic framework for dynamic vehicle routing with site-dependent constraints
Advisor	Günther Raidl

2010 2014

Bachelor of Science, *TU Wien*.

Field of Study	Computer Engineering
Thesis	Modellierung und Regelung einer Hebebühne
Advisor	Andreas Kugi

Work Experiences

2017

Research Assistant,
TU Wien, Institute of Logic and Computation,
Algorithms and Complexity Group.
- Published several research articles with other authors
- Co-supervised several master theses
- Research presented at international conferences

2019 2020

Six Month Research Stay,
Universitat Autònoma de Barcelona, Artifical Intelligence Research Institute.

2014 2017

Software Developer,
Company Destion – IT Consulting & Software Solutions GmbH.
- Worked on heuristic optimization methods in the field of logistics
- Worked on heuristic optimization methods in the field of school timetabling
- Worked on the graphical user interface design

2010 2014

Technical CAD Drafter/Summer Internships,
Company KWI Engineers GmbH.
- Electrical drafting

2009 2010

Military Musician/Mandatory Military Service,
Austrian Armed Forces.

Peer-Reviewed Publications

Journal Articles

2021
Computers & Operations Research

M. Horn, J. Maschler, G. R. Raidl, and E. Rönnberg. A*-based construction of decision diagrams for a prize-collecting scheduling problem. *Computers & Operations Research (COR)*, 126:105125, 2021

2020
Annals of Operations Research

M. Horn, G. R. Raidl, and E. Rönnberg. A* search for prize-collecting job sequencing with one common and multiple secondary resources. *Annals of Operations Research*, 2020

2019
Artificial Intelligence

M. Horn, G. Raidl, and C. Blum. Job sequencing with one common and multiple secondary resources: An A*/Beam Search based anytime algorithm. *Artificial Intelligence*, 277(103173), 2019

Conference Proceedings

CPAIOR 2021

M. Horn and G. R. Raidl. A*-based compilation of relaxed decision diagrams for the longest common subsequence problem. volume 12735 of *LNCS*. Springer, 2021. To appear. Accepted for the 18th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2021)

ISM 2020

M. Horn, N. Frohner, and G. R. Raidl. Driver shift planning for an online store with short delivery times. *Procedia Computer Science*, 180:517–524, 2021. Proceedings of the 2nd International Conference on Industry 4.0 and Smart Manufacturing (ISM 2020)

ISM 2020

N. Frohner, M. Horn, and G. R. Raidl. Route duration prediction in a stochastic and dynamic vehicle routing problem with short delivery deadlines. *Procedia Computer Science*, 180:366–370, 2021. Proceedings of the 2nd International Conference on Industry 4.0 and Smart Manufacturing (ISM 2020)

OPTIMA 2020

M. Horn, M. Djukanovic, C. Blum, and G. R. Raidl. On the use of decision diagrams for finding repetition-free longest common subsequences. In N. Olenev, Y. Evtushenko, M. Khachay, and V. Malkova, editors, *Proceedings of the XI International Conference Optimization and Applications, OPTIMA 2020*, volume 12422 of *LNCS*, pages 134–149. Springer, 2020

PPSN 2020

T. Kaufmann, M. Horn, and G. R. Raidl. A variable neighborhood search for the job sequencing with one common and multiple secondary resources problem. In T. Bäck, M. Preuss, A. Deutz, H. Wang, C. Doerr, M. Emmerich, and H. Trautmann, editors, *Proceedings of PPSN XVI: Parallel Problem Solving from Nature*, volume 12270 of *LNCS*, pages 385–398. Springer, 2020

EUROCAST 2019 M. Horn and G. R. Raidl. Decision diagram based limited discrepancy search for a job sequencing problem. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *Proceedings of the 17th International Conference of Computer Aided Systems Theory, EUROCAST 2019*, volume 12013 of *LNCS*, pages 344–351. Springer, 2020

AI 2019 C. Blum, D. Thiruvady, A. T. Ernst, M. Horn, and G. R. Raidl. A biased random key genetic algorithm with rollout evaluations for the resource constraint job scheduling problem. In J. Liu and J. Bailey, editors, *Proceedings of the 32nd Australasian Joint Conference on Advances in Artificial Intelligence, AI 2019*, volume 11919 of *LNCS*, pages 549–560. Springer, 2019

PATAT 2018 M. Horn, G. R. Raidl, and E. Rönnberg. An A* algorithm for solving a prize-collecting sequencing problem with one common and multiple secondary resources and time windows. In E. K. Burke, L. Di Gaspero, B. McCollum, N. Musliu, and E. Özcan, editors, *Proceedings of the 12th International Conference of the Practice and Theory of Automated Timetabling, PATAT 2018*, pages 235–256, Vienna, Austria, 2018

MOD 2017 M. Horn, G. R. Raidl, and C. Blum. Job sequencing with one common and multiple secondary resources: A problem motivated from particle therapy for cancer treatment. In G. Giuffrida, G. Nicosia, P. Pardalos, and R. Umeton, editors, *MOD 2017: Machine Learning, Optimization, and Big Data – Third International Conference*, volume 10710 of *LNCS*, pages 506–518. Springer, 2017

Presentations

2020 **Conference**, *ISM 2020*.
- Title: Driver shift planning for an online store with short delivery times
- Place: Online Conference, Austria

2020 **Conference**, *OPTIMA 2020*.
- Title: On the use of decision diagrams for finding repetition-free longest common subsequences
- Place: Online Conference, Moscow, Russia

2020 **Conference**, *CPAIOR 2020*.
- Title: On the use of decision diagrams for finding repetition-free longest common subsequences
- Place: Online Conference, Vienna, Austria

2019 **Talk**, *Universitat Autònoma de Barcelona 2019*.
- Title: Job sequencing with one common and multiple secondary resources: an A*/beam search based anytime algorithm
- Place: Universitat Autònoma de Barcelona, Barcelona, Spain

2019

Conference, *EUROCAST 2019*.

- Title: Decision diagram based limited discrepancy search for a job sequencing problem
- Place: Las Palmas de Gran Canaria, Spain

2018

Symposium, *DDOPT 2018*.

- Title: Exploiting relaxed decision diagrams to obtain heuristic solutions for a prize-collecting scheduling problem
- Place: Pittsburgh, PA, USA

2018

Conference, *PATAT 2018*.

- Title: An A* algorithm for solving a prize-collecting sequencing problem with one common and multiple secondary resources and time windows
- Place: Vienna, Austria

2017

Conference, *MOD 2017*.

- Title: Job sequencing with one common and multiple secondary resources: A problem motivated from particle therapy for cancer treatment
- Place: Volterra, Tuscany, Italy