SAT Techniques for Multiprocessor Scheduling on Uniform Machines

Master Thesis of

Matthew Akram

At the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. rer. nat. Peter Sanders
Advisor:	Dominik Schreiber

Time Period: 1st October 2023 – 31st March 2024

Statement of Authorship

Karlsruhe, March 25, 2024

Abstract

In this thesis, we address optimal makespan-minimising identical parallel machine scheduling, commonly known as the $P||C_{max}$ problem. We focus primarily on SAT solving techniques, adapting multiple existing SAT encodings for pseudo-Boolean constraints to create efficient and performant encodings. These encodings expose novel properties of the $P||C_{max}$ problem, which we can exploit to significantly reduce the search space of almost any search technique. To demonstrate this, we incorporate our techniques into two branch-and-bound algorithms and an ILP encoding. We improve upon the state of the art of branch-and-bound schemes for $P||C_{max}$ and shed light unto new and promising research avenues that may build upon our techniques. Finally, we improve upon state-of-the-art bounding schemes to tightly bound roughly 85% of tested instances from below, providing major speed-ups to any $P||C_{max}$ solving algorithm.

Deutsche Zusammenfassung

In dieser Arbeit beschäftigen wir uns mit der optimalen Planung auf identischen parallelen Maschinen zur Minimierung der maximalen Rechenzeit, allgemein bekannt als das $P||C_{max}$-Problem. Wir konzentrieren uns hauptsächlich auf SAT-basierte Lösungsansätze und passen mehrere existierende SAT-Kodierungen für *Pseudo-Boolean Constraints (PBC)* an, um effiziente und leistungsfähige Kodierungen zu erzeugen. Diese Kodierungen verdeutlichen neue Eigenschaften des $P||C_{max}$-Problems, die wir ausnutzen können, um den Suchraum fast aller Suchtechniken erheblich zu reduzieren. Um dies zu demonstrieren, integrieren wir unsere Techniken in zwei Branch-and-Bound-Algorithmen und eine ILP-Kodierung. Wir verbessern den Stand der Technik von Branch-and-Bound-Verfahren für $P||C_{max}$ und identifizieren neue und vielversprechende Forschungsansätze, die auf unseren Techniken aufbauen können. Schließlich entwickeln wir neue Heuristiken zur Berechnung von oberen und unteren Schranken für Instanzen des $P||C_{max}$-Problems. Unsere Heuristiken begrenzen ca. 85% der getesteten Instanzen exakt von unten, was eine erhebliche Beschleunigung für jeden $P||C_{max}$-Lösungsalgorithmus bedeutet.

Contents

1. Introduction

In this work, we address unconstrained task scheduling on identical parallel machines. This problem is often denoted by its Graham notation $P||C_{max}$ [GLLK79]—a three-part definition where P represents identical parallel machines, no side constraints are present, and $C_{\max}$ is the maximum makespan (completion time) of any machine, which we wish to minimise. Since this problem is strongly NP-hard [GJ78], it is necessary to consider heuristic approaches and/or approximation schemes [BDJR22]. We are concerned with finding exact solutions and thus consider multiple heuristic approaches for discrete mathematical optimisation, with a focus on SAT solving.

1.1 An Introduction to the $P||C_{max}$ Problem

Informally, the $P||C_{max}$ problem is defined by a series of jobs to be scheduled on a series of identical processors. We are given n single-threaded jobs and m processors. The total runtime of each job is known in advance, and all jobs are completely independent. Once a job is started, its execution cannot be stopped until it is finished, meaning that each job is to be executed in its entirety on exactly one processor. The processors are all identical in performance and capabilities. The $P||C_{max}$ problem poses the question, how can we schedule the n jobs on the m processors, such that we are done in the shortest amount of total time, referred to as the *optimal makespan.*

Let us consider an example. Given, are 3 processors and 8 jobs. The runtimes of the jobs are 5,4,3,3,2,2,2, and 1 time units. How can we schedule the jobs on the three processors, such that all computation is done as soon as possible? If we sum up the weights of the jobs and divide by the number of machines, we get roughly 7.3. Thus, via the pigeonhole principle, the optimal makespan is at least 8. In fact, we can schedule these jobs on these machines such that the longest running processor finishes in 8 time units. To present this, we introduce Figure 1.1 which shows multiple satisfying assignments to this $P||C_{max}$ instance. Each job is numbered with its index, and is represented by a rectangular box with width 1 and length proportional to its weight. The assignments to each processor are represented as a stack of boxes. The entire assignment is thus represented as three stacks of boxes; the height of each stack represents the makespan of a processor.

What we see from Figure 1.1, is that when trying to find a correct assignment, we do not need to consider the order of the execution of the jobs. Also, since all the processors are identical, we have to be careful to ignore the permutation of the processors. This, in essence, is what makes the $P||C_{max}$ problem so difficult, that there are numerous symmetries between different “equivalent” assignments.

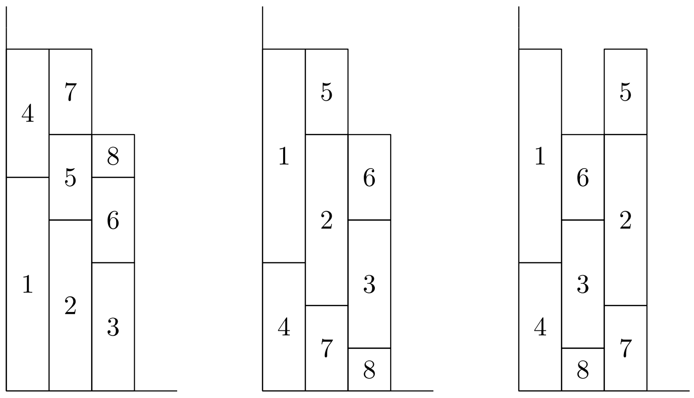


Figure 1.1: A representation of three possible assignments with optimal makespan to the $P||C_{max}$ instance with 3 processors and 8 jobs of sizes 5,4,3,3,2,2,2, and 1. The index on each box corresponds to the index of the corresponding job, and the size of each box to its size.

1.2 Motivation

$P||C_{max}$ is probably still "by far the most studied scheduling model from the viewpoint of approximation algorithms" [LLRS93]. This is in part due to its many applications, ranging from the scheduling of portfolio algorithms, to production planning [DSV93, Law17], and also partially due to its apparent simplicity.

Despite its apparent simplicity, as well as the astonishing effectiveness of approximation algorithms, progress on exact solving techniques has moved slowly. Nevertheless, there exist a variety of fundamentally different techniques that show considerable promise. Our goal, throughout this thesis, is to present a variety of speed-up techniques for exact solving algorithms that are transferable to almost any type of $P||C_{max}$ solving procedure.

Our interest in SAT solving is also not arbitrary. When considering encodings for generic solvers, SAT solvers might not be the obvious choice for the $P||C_{max}$ problem. The lack of integer representations makes SAT structurally different from $P||C_{max}$. This is also reflected in the literature, where there exist multiple *integer linear programming (ILP)*, hybrid-LP, and hybrid-ILP approaches, but no successful SAT based approach. Nonetheless, SAT and related *satisfiability modulo theory (SMT)* based approaches have been successful for other scheduling problems [Heb12, BCSV20]. Thus, it is entirely possible, albeit unlikely, that SAT solving is more appropriate for $P||C_{max}$ scheduling than ILP or other constraint programming solvers.

Furthermore, the field of SAT solving is rapidly growing due to its open-source community and yearly SAT competition. Great strides have been made towards bringing SAT solving to the cloud [SS21], allowing for the solving of much larger instances. On the other end of the spectrum, the re-emerging field of analogue computing promises speed-ups of multiple orders of magnitude, but is currently restricted to small scale simulations [YSV$^+$18, TGP$^+$09, MKHT20]. Thus, well performing SAT encodings would also be able to piggyback on the advancements in SAT solving in the future.

Last but not least, the stark contrast between propositional logic and $P||C_{max}$ can be a benefit, as opposed to a hurdle. Propositional logic, as the most basic form of knowledge representation, may help us uncover intrinsic properties of $P||C_{max}$, which in turn may also benefit other, non-SAT based approaches.

1.3 Contribution

The contributions of this thesis span a number of different topics. First, we improve upon existing upper and lower bounding methods to provide measurably significant improvements in their accuracy. We do this by improving upon the bounding techniques presented by Haouari et al. [HGJ06, HJ08]. Our techniques tightly bound roughly 85% of tested instances from below, and roughly 85% of tested instances from above. Our improvements to the lower-bounding algorithms provide an improvement over the state of the art.

Secondly, we present and evaluate different SAT encodings for $P||C_{max}$. This provides a novel perspective on the $P||C_{max}$ problem, and greatly improves upon the naive SAT encoding of $P||C_{max}$. The most advanced of these SAT encodings requires us to construct large reduced ordered binary decision diagrams for pseudo-Boolean constraints. As a tangential result, we develop a novel algorithm to efficiently construct these diagrams in $O(n \cdot U)$ (as opposed to $O(n \cdot U \log U)$ from the literature [Beh07]), where n is the number of variables and U is the upper limit of the given constraint.

These SAT encodings also emphasise novel properties of the $P||C_{max}$ decision problem, which we can exploit for the improvement of our SAT encodings and beyond. To demonstrate this, we integrate these rules into the *branch-and-bound (BnB)* algorithms of Dell'Amico et al. [DM95] and Haouari et al. [HJ08], and into the ILP encoding presented by Mrad et al. [MS18]. This results in an improvement upon the state of the art of BnB schemes for $P||C_{max}$ and sheds light unto new and promising research avenues that may build upon our techniques.

1.4 Outline

The remainder of this thesis is structured as follows. In Section 2 we introduce the basic concepts and notation required for an intuitive understanding of the topics discussed in the rest of this work. This section is intended to provide the reader with a general understanding of relevant topics, such as practical SAT solving, PBCs, BDDs, and related NP-complete problems. There, we also present existing upper and lower bounding methods for $P||C_{max}$ from the literature, as well as pruning rules from the literature. In Section 3 we discuss all non-SAT based computation required for solving $P||C_{max}$ using SAT solving. This includes, but is not limited to, novel upper and lower bounding methods. In Section 4 we introduce different SAT encodings for the $P||C_{max}$ decision problem based on existing SAT encodings for pseudo-Boolean constraints. With the aim of using the pruning rules from the literature to decrease the size of the search space of our SAT encodings, we discuss the different properties of our SAT encodings that allow us to accomplish this efficiently. This reveals novel properties of the $P||C_{max}$ decision problem that can be exploited to create stronger pruning rules; we formalise this in Section 5 and incorporate it into our SAT encodings. At first glance, these properties seem to require immense computational power, so we discuss how these properties can be efficiently computed and, as a side effect of this, how we can efficiently compute BDDs much more efficiently. This algorithm works especially well for BDDs of the underlying PBCs of $P||C_{max}$. This is done in Section 6. To demonstrate the effectiveness of our new pruning rules, we incorporate them into two BnB algorithms and an ILP encoding in Section 7. In Section 8 we evaluate all of our presented techniques against each other, and against techniques from the literature, showing that our pruning rules are effective at limiting the search space in many different settings, and that their comparably cheap computational requirements and aggressive search space reduction makes them valuable in all tested scenarios. Lastly, in Section 9 we discuss how our findings can and should be improved upon to achieve even stronger speed-ups in different settings and conclude with a brief summary of our results.

2. Preliminaries

Before we start with the content of this thesis, we first need to introduce the most relevant concepts we use. In order to have an intuitive understanding of all of our techniques, we first give an introduction to SAT solving, which serves as the basis for understanding the differences between our different SAT encodings. We also introduce other crucial concepts such as PBCs, BDDs, as well as related NP-complete problems.

Beyond SAT solving, we also introduce BnB algorithms and, naturally, the pruning rules from the literature that we use throughout this thesis as an underlying base for cross-technique optimisations. The upper and lower bounds from the literature that we introduce serve as an introduction to the more subtle properties of $P||C_{max}$; we also use these bounds as a springboard towards the development of improved bounding techniques.

2.1 The $P||C_{max}$ Problem

Formally, a $P||C_{max}$ *optimisation instance* (W, m) is defined by n durations $W = \{w_1, \ldots, w_n\}$ of n corresponding jobs $J = \{j_1, \ldots, j_n\}$ and by the number m of identical processors $P = \{p_1, \ldots, p_m\}$. An instance (W, m) asks for an *assignment* $A = \{a_1, \ldots, a_n\}$ of jobs to machines $(1 \leq a_i \leq m)$ such that the *maximum completion time* $C_{\max} := \max_i\{\sum_{\alpha \,|\, a_\alpha = i} w_\alpha\}$ is minimised. By contrast, a *decision instance* (W, m, U) additionally imposes an *upper bound* U for $C_{\max}$ and poses whether a feasible solution exists. Throughout this thesis, we assume that the jobs are sorted by duration in decreasing order, i.e., $(w_1 \geq w_2 \geq \ldots \geq w_n)$.

2.2 Notation

First, we introduce the notation we use for relevant topics. We separate our notation into different paragraphs for easier lookup.

Partial Assignments

A *partial assignment* A on a problem with variables $a_1, \ldots, a_n$ is a set of mappings for some (or all) of the variables to distinct values. The assignment $a_i = x$ represents the assignment of a_i to x. Thus, the term $A \leftarrow A \cup \{a_i = x\}$ denotes the action of assigning the value x to a_i. The term $a_i = x \in A$ thus denotes that j_i is scheduled on p_x by A.

Decision Level

The current *decision level* denotes the size of A.

Currently Assigned Workload C_x^A

In the context of $P||C_{max}$ we often need to consider the makespan of a processor given a partial assignment A. We use C_x^A to denote the workload assigned to processor p_x by A, i.e. $\sum_{i|a_i=x\in A} w_i$. We also use C_x when A is clear from the context.

Graph Theory

Throughout this thesis, we consider graphs in multiple contexts. We consider directed multi-graphs and give each edge a distinct name whenever necessary. An edge is denoted $e := ab$ where a and b are nodes in the graph and e travels from a to b.

Boolean Assignments

In the context of Boolean assignments, we use $\perp$ or 0 to represent false, and $\top$ or 1 to represent true.

2.3 An Introduction to SAT Solving

For a majority of this thesis, we aim to solve a $P||C_{max}$ instance using a SAT solver. Since SAT is inherently a decision problem, a direct encoding of the $P||C_{max}$ optimisation problem into SAT is not possible. Instead, we identify appropriate instances of the $P||C_{max}$ decision problem. Using a chosen Karp-reduction (*encoding*), we encode these instances as SAT formulas.

However, not every encoding is equivalent in practice. Due to the structure of modern SAT solvers, the performance of different encodings can vary significantly. As modern SAT solvers rely on heuristics, the properties that make some encodings better than others often rely on educated guesses and empirical evidence. In this section, we provide an explanation of some basic inner workings of modern SAT solvers to give the reader a basic intuition regarding the properties that make a good encoding.

As input, modern SAT solvers take SAT formulas in *Conjunctive Normal Form (CNF)* [Pfa10], represented as:

$$(x_1 \vee \overline{x_2} \vee x_3 \ldots) \wedge (\overline{x_4} \vee x_5 \vee x_6 \ldots) \ldots$$

These formulas consist of a logical conjunction of logical disjunctions (called *clauses*) of Boolean variables. Negations are only allowed directly in front of variables. The solvers return either "SATISFIABLE" along with a satisfying assignment of the formula, or "UNSATISFIABLE".

To solve these formulas, the most prominent algorithm used in almost all modern solvers is based on *conflict-driven clause learning (CDCL)* [MSS03, BS97]. The CDCL algorithm is a heuristic search algorithm, based on the DPLL algorithm [DP60, DLL62], and starts with an assignment $A = \emptyset$ of the variables. The algorithm performs *unit propagation* [DG84], where a *unit clause* is an unsatisfied clause with exactly one unassigned variable. Unit propagation identifies unit clauses and infers the value of the last remaining variable in the clause, potentially creating more unit clauses (which are also propagated). After that, a variable is chosen, using some heuristic, and its assignment is branched. When an illogical situation is reached (i.e. an unsatisfied clause with no unassigned variables), this is referred to as a conflict. CDCL finds a reason for this conflict, learns a new enforceable clause, and then backtracks.

An important property to note is that it is easier to learn the values of variables by unit propagation than by branching. Thus, short clauses are more valuable since they are "closer" to being propagated. Modern SAT solvers spend most of their time doing unit propagation [MMZ+01], making the *length of propagation chains* crucial. The concept of length for propagation chains is similar to the concept of depth in a circuit or level in a BDD. A rough explanation goes as follows. Given two variables, where the assignment of one directly affects the assignment of the other (via unit propagation), if we set one, how many other variables must be propagated before inferring the value of the other. This will be made clearer later in the thesis.

There are also more complex properties that greatly affect the performance of an encoding. Let us consider an example that will come up again later in this thesis.

Example 2.3.1. Let us define 4 variables a, b, c, and d. Now let us assert that exactly one of a, b, c, d must be $\top$, by adding the clauses

$$\begin{gathered} a \vee b \vee c \vee d \\ \overline{a} \vee \overline{b} \\ \overline{a} \vee \overline{c} \\ \overline{a} \vee \overline{d} \\ \overline{b} \vee \overline{c} \\ \dots \end{gathered}$$

Now let us also represent a fictitious condition *cond* that depends on the assignments of other variables in the formula. If *cond* is assigned to $\top$, then either a or b is true.

There are two ways of representing this. The first, asserts that either a or b is true

$$\overline{cond} \vee a \vee b.$$

The second asserts that c and d are false

$$\begin{gathered} \overline{cond} \vee \overline{c}, \\ \overline{cond} \vee \overline{d}. \end{gathered}$$

With the second encoding, if *cond* is true, then we propagate $c = d = \bot$ and obtain the clause $a \vee b$. With the first encoding, if *cond* is false, we obtain the clause $a \vee b$, which initially does not tell us anything. If we decide to branch c or d to $\top$ then we obtain a conflict via propagation. If we decide to branch a or b to $\top$ we obtain $c = d = \bot$ via propagation. When backtracking and assigning the other variable to $\top$, we might have to propagate $c = d = \bot$ again.

Even for this simple example, the work done by the solver is much greater with the first encoding in comparison to the second encoding. If instead of 4 we had 20 variables, this problem would only be exacerbated. It is therefore not only important what information we provide to the solver, but how we provide it as well.

There are two critical properties to consider when discussing SAT encodings: *generalised arc consistency (GAC)* and *inconsistency detection*. Both are important inference rules that can significantly reduce the search space [RvBW06, PS15]. Inconsistency detection states that, given a partial assignment A without a valid continuation, unit propagation will find a collision. For GAC, given a partial assignment A where, in every valid continuation, a variable x must be assigned to $\diamond$ ($\diamond \in \{\top, \bot\}$), unit propagation must derive the assignment $x = \diamond$.

Building upon Example 2.3.1, we see that the first encoding clearly does not maintain GAC, since setting $cond = \top$ forces us to set $c = d = \bot$ but we cannot detect that via unit propagation. The second encoding, does allow us to detect $c = d = \bot$ via unit propagation since it does maintain GAC.

It's important to note that, unless $P = NP$, constructing a SAT formula of polynomial size with GAC or inconsistency detection is not possible for NP problems, as it would imply solving them in polynomial time. However, these properties are still valuable even if maintained within certain sub-formulas. This will be most relevant in the context of SAT encodings for pseudo-Boolean constraints.

2.4 Pseudo-Boolean Constraints

Pseudo-Boolean constraints (PBC) are a highly studied topic in constraint programming and SAT solving [ES06, PS15], and are also closely related to the $P||C_{max}$ problem. As such, they appear in many places throughout this thesis.

A PBC is defined by a series of variables $a_1, \ldots, a_n \in \{0, 1\}$, a series of corresponding weights $w_1, \ldots, w_n \in \mathbb{N}$ and a limit value $U \in \mathbb{N}$. A PBC is defined as

$$a_1 \cdot w_1 + \ldots + a_n \cdot w_n \leq U$$

and constrains which variables can be set to 1 together.[1] Since this thesis mostly revolves around SAT solving, we also use Boolean variables as the variables of the PBC, and map $\top$ to 1 and $\bot$ to 0.

As early as possible, we would like to draw attention to the relation between the state-transition graph (from dynamic programming literature [Ber05]) of a PBC and its binary decision diagram.

2.5 (Reduced Ordered) Binary Decision Diagrams for PBCs

A *reduced ordered binary decision diagram (BDD)* is a representation of a Boolean function as a directed acyclic graph. The formal definition of this simple data-structure can be unintuitive (see pg. 4-5 [Bry86]), so we provide a constructive definition here. A (non-reduced) ordered binary decision diagram for a Boolean function on the variables $x_1, \ldots, x_n$ is constructed as follows.

First, define an arbitrary ordering on the variables, which we here will assume is $x_1, \ldots, x_n$. Second, create a complete directed tree with $n + 1$ levels (here the root is on level 1). Each inner node of the tree has two outward edges, the 0 / *left* / $\bot$ edge and the 1 / *right* / $\top$ edge. A 0 (1) edge going from level i to level $i + 1$ represents the assignment of the i'th variable to 0 (1). The leaves of the tree are labelled with either 1 or 0 and represent the output of the function. Thus, each input to the function is represented by a distinct and unique path in the tree, ending at a leaf, whose label matches the output of the function. A *subgraph* of a binary decision diagram is defined as a node and all nodes below it.

In order to obtain the reduced ordered binary decision diagram, we must also exhaustively apply the following two rules.

[1] In the literature, PBCs are often defined with weights in $\mathbb{Z}$. We choose this simpler definition for the sake of simplicity, since we use PBCs in the context of $P||C_{max}$ where weights are strictly positive. Any work we do on PBCs can also be done with weights in $\mathbb{Z}$ via normalisation.

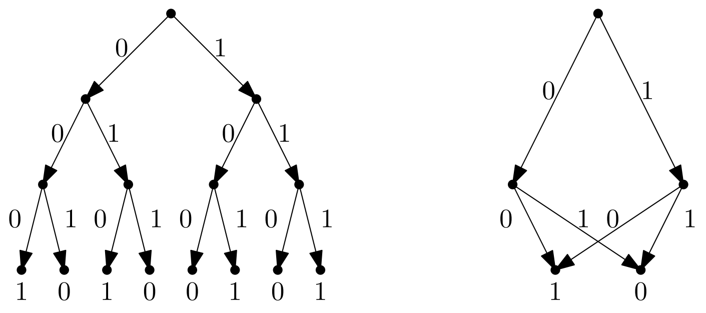


Figure 2.1: The non-reduced (left) and reduced ordered binary decision diagram of the palindrome function with three digits. Note that the middle digit is irrelevant to the outcome and, as such, gets eliminated by the BDD reduction rules.

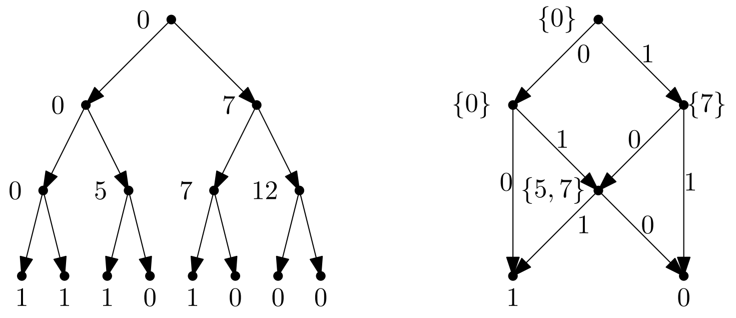


Figure 2.2: The non-reduced decision diagram of the PBC $a_1 \cdot 7 + a_2 \cdot 5 + a_3 \cdot 3 \leq 7$ (left) with nodes labelled with the partial sum they represent, and the corresponding BDD, with nodes labelled with their $MERGED$ sets.

1. Merge any two isomorphic subgraphs.

2. Contract any node where the left and right edges both go to the same node.

Figure 2.1 gives an example of a non-reduced binary decision diagram, and its reduced counterpart.

Throughout this paper, we build BDDs for PBCs. These BDDs have a very specific structure, which we will discuss later. In order to construct the BDD for a PBC, we first need to order the variables. The ordering we choose can greatly affect the size of the BDD, and finding an optimal ordering is itself co-NP complete [Bry86]. Fortunately, as with many problems, “a human with some understanding of the problem domain can generally choose an appropriate ordering without great difficulty” [Bry86]. In most works, including this one, the variables are ordered in descending order of corresponding weight, and this does logically provide appropriately sized BDDs.

Next, we construct the non-reduced binary decision diagram of the PBC. While constructing it, we notice that each node in the decision diagram represents the sum of a subset of the weights in the PBC (see [CKE08, Beh07]). For example, given the PBC $a_1{\cdot}7{+}a_2{\cdot}5{+}a_3{\cdot}3 \leq 7$, the root node represents the value 0, the nodes on the second level represent the values 0 and 7, and the nodes on the third level represent the values $0, 5, 7$, and 12. We refer to the node on level i that represents the value u as $n_{i,u}$.[2] When reducing the decision diagram, we can remember the values that a node n represents in a set, which we refer to as $MERGED(n)$. Figure 2.2 represents a BDD for a PBC, and the $MERGED$ set of each node.

[2]When constructing the decision diagram, there can be multiple such nodes on a given level. However, since they are all equivalent, we use $n_{i,u}$ to refer to any one of them.

2.6 Related NP-Complete Problems

There are multiple NP-complete problems that are closely related to $P||C_{max}$. In this section, we introduce some of these problems, and highlight the structural connection between each one of them and the $P||C_{max}$ problem.

The Bin Packing Problem (BPP)

In a BPP (see [MT90, FS01]) instance (U, W), we are given n objects with integer sizes $W = \{w_1, \ldots, w_n\}$ and boxes of size $U \in \mathbb{N}$. We are asked to pack the elements into the minimum number of boxes m. In a way, BPP is the "transpose" of $P||C_{max}$, instead of minimising U and having m constant, we minimise m and have U constant. Accurate bounds on BPP instances can thus also be used for the respective $P||C_{max}$ instances.

The Subset-Sum Problem (SSP)

We also have many use cases for the SSP optimisation problem [AA20]. A SSP instance (U, W) asks, given the elements $W = \{w_1, \ldots, w_n\}$ $(w_i \in \mathbb{N})$ and $U \in \mathbb{N}$, find

$$\max_{W' \subseteq W} \sum_{w_i \in W'} w_i \text{ s.t. } \sum_{w_i \in W'} w_i \leq U.$$

This problem is very closely related to the $P2||C_{max}$ problem [HGJ06] ($P||C_{max}$ with $m = 2$).

One can easily solve an instance of $P2||C_{max}$ by solving an SSP with $U \leftarrow \left\lfloor \sum_{w_i \in W} w_i/2 \right\rfloor$. This is highly practical since, unlike $P||C_{max}$ SSP, is not strongly NP-complete, meaning it can be solved in pseudo-polynomial time. For practical scenarios, this is as useful as it being in P.

Integer Linear Programming (ILP)

Another popular problem for generic solving is ILP [Sch86]. In an ILP instance, we are provided an $m \times n$ matrix $A \in \mathbb{R}^{m \times n}$ and an m-dimensional vector $b \in \mathbb{R}^m$. We ask, is there a vector $x \in \mathbb{N}^n$ s.t. $Ax \leq b$.

Due to the numerical nature of the $P||C_{max}$ problem, it is tempting to use ILP translations to solve $P||C_{max}$. However, since SAT is structurally different to $P||C_{max}$, a SAT translation could give us more insight into the structure of the $P||C_{max}$ problem. We do however revisit ILP encodings at a later point, where we discuss integrating pruning rules for BnB algorithms into existing state-of-the-art ILP encodings.

2.7 Related Work

$P||C_{max}$ has been the subject of numerous publications due to both its difficulty and simplicity. For a thorough summary of the work done on $P||C_{max}$, we refer to the excellent related work sections by Lawrinenko [Law17] and Mrad et al. [MS18], that cover almost all related work up to 2017. Since then, research has mostly focused on other models with additional constraints on the jobs and the machines. Thus, we present only the most relevant related work here, and refer to the two aforementioned papers for a more broad overview.

Since $P||C_{max}$ is an optimisation problem, we can make use of upper and lower bounds on the optimal makespan of a given instance. These bounds tell us that the optimal makespan of the given instance is within a given range, such that we can search for assignments only within the given bounds. This can greatly affect the speed of the search, especially if at least one of the given bounds is tight. We revisit some of the following literature on upper and lower bounding techniques in greater detail in Section 2.9.

The most advanced lower bounding techniques are presented by Haouari et al. [HGJ06, HJ08]. In those two papers, they present a series of advanced lower bounding techniques, as well as two lifting procedures for the improvement of lower bounding algorithms.

They also present and improve an advanced upper bounding technique, namely *multi-start subset sum*, which is essentially an improvement scheme with restarts. Dell'Amico et al. [DIMM08] introduce a scatter-search based bounding technique that gives their exact solving scheme excellent performance. The literature on upper bounds for $P||C_{max}$ is vast. For an overview, we recommend referring to Lawrinenko's dissertation [Law17].

The literature on exact solving procedures is a little more sparse. In 1966, Rothkopf [Rot66] presented a dynamic programming approach in $O(n \cdot U^m)$. This algorithm is slow, and can only solve very small instances. Mokottof [Mok04] presents a cutting plane algorithm where valid inequalities are identified and added to an LP encoding, until the respective solution is integer. The first BnB algorithm for $P||C_{max}$ was introduced by Dell'Amico and Martello [DM95] who introduce novel pruning rules, used to decrease the size of the search space of the BnB algorithm. Haouari and Jemmali [HJ08] use a different branch order to construct a novel BnB algorithm which far exceeds the performance of the algorithm by Dell'Amico and Martello. During experimentation, they detect a set of difficult $P||C_{max}$ instances, instances with $n/m = 2.5$. They discuss the relevance of this set, and how future work should focus on such instances, since instances with larger n/m can usually be solved to optimality using simple bounding heuristics. Roughly at the same time, Dell'Amico et al. [DIMM08] present the DIMM heuristic. The DIMM heuristic consists of a combination of a scatter search algorithm, followed by an ILP translation of the $P||C_{max}$ decision problem, and a dedicated BnB algorithm to solve their ILP translation. This is currently considered state-of-the-art due to its excellent performance, even on very large instances. Lawrinenko [Law17] studies the structure of solutions of the $P||C_{max}$ problem, to develop new pruning rules for a BnB algorithm. The algorithm presented caters specially to the instances presented by Haouari and Jemmali, and has improved performance on those (and only those) instances. Finally, Mrad et al. [MS18] present a pseudo-polynomial ILP encoding of the $P||C_{max}$ problem with great performance on the instances proposed by Haouari and Jemmali.

2.8 Branch-and-Bound Algorithms

Branch-and-bound (BnB) is a common method for solving optimisation problems by recursively splitting them up into smaller problems until there are tiny, easily solvable instances.

2.8.1 An Introduction to BnB Algorithms

A BnB algorithm for $P||C_{max}$ is a tree-like search where we successively extend an initially empty partial assignment A of jobs to processors until $|A| = n$. At each decision level, we identify a set of decisions ("branches") which each extend A by one element. In addition, we maintain admissible bounds on $C_{\max}$ during search, which allows us to exclude decisions which inevitably lead to suboptimal solutions ("bound"). We recursively and heuristically search the remaining decisions.

2.8.2 Pruning Rules from the Literature

In current research, there are a few BnB based approaches to find exact solutions to $P||C_{max}$. Since the search-tree of possible assignments can be exponentially large, it is important to find rules with which large branches can be left unexplored. Consequently, we restrict the search space by imposing further rules on the structure of the solution, while maintaining at least one optimal assignment. We refer to these rules as *pruning rules*, but they are also commonly referred to as *dominance criteria* in the literature, or (in some cases) *symmetry breaking*. Due to the similarity between SAT solving and BnB algorithms, we study these pruning rules in the hopes of being able to encode them to the SAT solver. This should in turn allow us to massively decrease the size of the search space for the SAT solver.

We now mention some relevant pruning rules from the literature. Since we aim to translate these rules into SAT clauses, we unfortunately cannot employ any complex dynamically calculated rules, i.e. rules that require large extra computations done at each decision node, s.a. rules that require the calculation of a lower bound on the remainder of the problem. Dell'Amico and Martello [DM95] present several pruning rules for their BnB algorithm. These rules are intended to break symmetries between equivalent solutions.

Pruning Rule 1. *If there are two jobs with the same weight* ($w_h = w_i$), *let* A_h, A_i *be the partial assignments directly before* j_h, j_i *are respectively assigned. We only need to consider assignments* a_h, a_i *that fulfil* $C_{a_h}^{A_h} \leq C_{a_i}^{A_i}$.

In words, if there are two jobs of the same size, the one with a lower index is assigned at an earlier point in time. This imposes an order onto equivalent jobs, and prevents us from enumerating solutions where these jobs are simply swapped.

Pruning Rule 2. *For a partial assignment* A ($|A| < n$), *where there are two processors* p_x, p_y ($x < y$), *with* $C_x^A = C_y^A$, *if we are looking to assign* a_i *we need not consider assigning* $a_i = y$.

This rule prevents us from trying to assign j_i to p_y. It is clear that $A \cup \{a_i = y\}$ has an optimal completion if and only if $A \cup \{a_i = x\}$ does as well.

Pruning Rule 3. *If there are exactly 3 unassigned jobs, with* $g > h > i$, *there are only two possible assignments to be considered:*

- *The jobs* j_g, j_h, j_i *are greedily assigned to the least-loaded processor*
- *The job* j_g *is assigned to the currently second-emptiest processor, and the two remaining jobs are greedily assigned to the least-loaded-processor.*

This rule functions as the end condition for their recursive algorithm. Dell'Amico and Martello consider all possible completions on the last three jobs and show that only these two cases need to be considered.

Pruning Rule 4. *If* $i < m$ *jobs remain unassigned, then only the* i *least loaded processors must be considered for the next decision.*

This rule prevents us from exploring suboptimal search paths. Dell'Amico and Martello prove that w.l.o.g. an optimal completion on the last $i < m$ jobs must only use the i least loaded processors.

These are by no means the only pruning rules from the literature. For more rules, we refer to the literature on the subject [DM95, Law17, HJ08]. Unfortunately, as it currently stands, these rules can be hard to encode into SAT. Therefore, in the remainder of this thesis, we work on extending these rules, to allow for more aggressive search space reduction, and for a better SAT encoding.

2.9 Upper and Lower Bounds from the Literature

Haouari et al. “raise the legitimate question of whether $P||C_{max}$ could be considered as rather easy from the practical computational point of view” [HJ08]. In practice, simple upper and lower bounding techniques are very effective at solving $P||C_{max}$. Here are some simple and commonly used bounding techniques from the literature.

2.9.1 Upper Bounds

For upper bounds, we introduce two simple heuristics from the literature. In order to generate an upper bound, we need to find a valid assignment of jobs to processors. The lower this assignment’s makespan is, the better the respective upper bound is.

LPT

The first and most simple method is the *longest-processing-time (LPT)* heuristic, based on *list-scheduling* [Gra66]. In list-scheduling, the jobs are first ordered arbitrarily. In the given order, the jobs are sequentially assigned to the processor with the smallest workload. In LPT, list-scheduling is executed with the jobs in decreasing order of job size. This heuristic has been shown to experimentally provide very accurate results [GRT01], and is the most efficient upper bounding technique used here. Furthermore, it provides very good guarantees, and can therefore also be used to provide lower bounds [Gra69, BCS74]. LPT has a competitive factor of $\frac{4}{3} - \frac{1}{3m}$.

Multi-Start-Subset-Sum

Haouari et al. [HGJ06, HJ08] introduce (and improve) a *multi-start subset-sum (MSS)* based heuristic. They exploit the equivalence between $P2||C_{max}$ and SSP. Initially, a random assignment of jobs to processors is generated. In a step of the MSS heuristic, two processors are chosen at random and the $P2||C_{max}$ instance defined on those two processors and the jobs assigned to them is solved using a solution to the respective SSP optimisation instance. This can be done exactly in pseudo-polynomial time using dynamic programming. This is repeated until a local minimum is reached, and the process is restarted with a different initial assignment. Throughout the procedure, the best assignment is remembered, and is returned when a stop criterion is reached. Since this is an iterative improvement heuristic, it can also be utilised to improve the solution obtained from a SAT call.

2.9.2 Lower Bounds

Here, we state the relevant lower bounds used from the literature. We do not prove these bounds, so we refer to the literature for a more detailed overview of how to find lower bounds for $P||C_{max}$.

The Trivial Lower Bound [McN59, DM95]

The simplest lower bound for a $P||C_{max}$ instance (W, m) is

$$L_{TV} = \max\{w_1,\ w_m + w_{m+1},\ \left\lceil \frac{1}{m} \sum_{w_i \in W} w_i \right\rceil \}.$$

Preliminary testing shows that this bound is very effective for instances where $\frac{n}{m}$ is large, instances where there are plenty of jobs of small sizes, and on very small instances with only a few very large jobs. An intuition behind each of the three components of this bound goes as follows. Since the first job must be inserted somewhere, it is clear that

$C_{max} \geq w_1$ (note that the first job is the largest). For the second component, we know that each of the first m jobs will have to be assigned somewhere. If any two of them are assigned to the same processor, then we know that processor will have a makespan of at least $2w_m \geq w_m + w_{m+1}$. If none of them are assigned to the same processor, then j_{m+1} will be assigned to a processor which has one of the largest m jobs assigned to it; that processor will have a makespan of at least $w_m + w_{m+1}$. The last component of L_{TV} is an application of the pigeon-hole principle. This component describes the average load per processor when scheduling all jobs, rounded up. As such, it corresponds to the exact solution of the relaxed problem, where jobs can be subdivided arbitrarily.

Fekete and Schepers [FS01]

A lower bound for $P||C_{max}$ can be derived from lower bounds for BPP. Fekete and Schepers derive a lower bound for BPP based on dual feasible functions. For a given BPP instance with item weights W and bin size (makespan) U they begin by normalising the size of all jobs by dividing them by U (i.e. $w_i' = w_i/U$). For any $h \geq 2$ ($h \in \mathbb{N}$) and $\epsilon \in [0, \frac{1}{2}]$ they then define two functions.

$$u^{(h)}(x) = \begin{cases} x, & \text{if } x(h+1) \in \mathbb{N}, \\ \lfloor (h+1)x \rfloor \frac{1}{h}, & \text{otherwise}, \end{cases}$$

$$U^{(\epsilon)}(x) = \begin{cases} 1, & \text{if } x \geq 1-\epsilon, \\ 0, & \text{if } x \leq \epsilon, \\ x, & \text{otherwise.} \end{cases}$$

With these functions, they define a parameterised lower bound function with parameter $q \in \mathbb{N}$

$$L_{FS}^{(q)}(J) = \max_{\substack{2 \leq h \leq q, \\ \epsilon \in [0, \frac{1}{2}]}} \left\lceil \sum_{i=1}^{n} u^{(h)}(U^{(\epsilon)}(w_i')) \right\rceil .$$

This bound can be calculated in $O(q \cdot n)$, by iterating over possible values for h, and for each of them, doing a linear sweep over the values of ϵ. This provides a lower bound on the number of bins of size U required to store the given items. Alternatively, this can be interpreted as a lower bound on the minimum number of processors required to process the given jobs with a makespan $\leq U$. In order to obtain a lower bound for $P||C_{max}$ we increase U until the given lower bound is $\leq m$ [HGJ06].

Anecdotal evidence suggests that this bound is very accurate, but only on small instances, and thus is not of much use in the general case of $P||C_{max}$ solving. However, it can be invaluable in the context of a lifting procedure.

Lower Bound Lifting Procedure [HGJ06, HJ08]

Here we present two methods with which lower bounds can be tightened.

The first one, presented by Haouari et al. [HJ08] solves an SSP instance to tighten a given lower bound. Given an instance (W, m) of $P||C_{max}$, and a lower bound L, we can improve L by solving the following SSP based problem.

$$L \leftarrow \min L' \quad (L' \geq L),$$
$$\exists J' \subseteq J : \sum_{j_i \in J'} w_i = L'.$$

Intuitively, if we have a lower bound L for our given instance, but our jobs cannot be combined to reach exactly L this bound is not tight. This procedure is more effective the smaller a given instance is. Thus, while not useful in most cases, it can be very useful as part of the next lifting procedure.

The second lifting procedure is also presented by Haouari et al. [HGJ06]. Haouari and Gharbi [HG04] prove that in any $P||C_{max}$ instance, there are α processors that process at least

$$\lambda_\alpha(n, m) = \alpha\lfloor n/m\rfloor + \min(\alpha, n - \lfloor n/m\rfloor m)$$

jobs. We can then define a sub-instance I' on the $\beta \leq n$ largest jobs. For this instance, and for a suitable α, we can then create a further sub-instance which we refer to as I^α_β by taking the smallest $\lambda_\alpha(\beta, m)$ jobs and α processors. Lower bounds on I^α_β are also valid bounds on our original $P||C_{max}$ instance. With that, for a lower bound function L, we can improve it by calculating

$$\overline{L}(I) = \max_{1\leq\alpha\leq m} \{ \max_{\alpha<\beta\leq n} L(I^\alpha_\beta)\}.$$

They then show that the only values of β that have to be considered are the values such that $\beta = \gamma m + \alpha$ for $\gamma = 1, \dots, \lfloor \frac{n-\alpha}{m} \rfloor$. With that, the number of generated sub-instances is in $O(n)$. By using our aforementioned bounds, as well as the first lifting procedure, we can calculate good lower bounds for the generated sub-instances.

3. Complementary Computation

Since $P||C_{max}$ is an optimisation problem, and SAT is a decision problem, there is a lot of non-SAT solving computation necessary to efficiently solve $P||C_{max}$. These computations range from bounding techniques, to calculating partial solutions, and iterative improvement schemes for improving solutions found via SAT calls. In this section, we outline our framework for solving $P||C_{max}$ via SAT solving. For each type of complementary computation, we present different techniques we use to speed up $P||C_{max}$ solving.

3.1 Overview of SAT-Based Framework

Throughout the majority of this thesis, we discuss how we can use SAT solving for solving $P||C_{max}$. Our framework for doing so goes as follows. This framework is presented in Algorithm 3.1.

Bounding

Initially, we are provided with a $P||C_{max}$ instance (W, m). We calculate a lower and an upper bound, L, U, on the optimal makespan of the given instance.

Precomputation

For a given $U \leftarrow U - 1$ we generate the $P||C_{max}$ decision instance (W, m, U). Given the $P||C_{max}$ decision instance and L, U, there are certain assignments we can rule out / enforce and guarantee that at least one solution is maintained (if one exists). The rules we use to do this are called *simplification rules.* Oftentimes, at this point we can already detect if this instance is not solvable, by using trivial satisfiability conditions, such as measuring the remaining space, or seeing if there is a job for which all processors are forbidden.

Encoding

We then encode this decision instance into propositional logic using the techniques presented in Sections 4 and 5.

Postprocessing

Since we are only solving the decision instance, not the optimisation instance, the SAT solver can often provide solutions that are close to optimal, but not optimal. Thus, after each successful SAT call, we take the assignment provided by the solver and use an iterative improvement scheme to improve it as much as possible.

Iteration

Provided that this improvement scheme finds a solution with makespan U' we update $U \leftarrow U'$ and go back to the precomputation step.

Algorithm 3.1: Solve Instance

```
   Data: $P||C_{max}$ instance $I = (W, m)$
1  $U, L \leftarrow$ BoundInstance($I$);
2  while $U > L$ do
3      $U \leftarrow U - 1$;
4      $I' \leftarrow (W, m, U)$;
5      ForbiddenAssignments $\leftarrow$ Simplify($I'$);
6      if Unsolvable($I'$, ForbiddenAssignments) then
7          return $U + 1$;
8      Solution $\leftarrow$ EncodeAndSolve($I'$, ForbiddenAssignments);
9      if Solution = "UNSATISFIABLE" then
10         return $U + 1$;
11     Solution $\leftarrow$ ImproveSolution(Solution);
12     $U \leftarrow$ Solution.makespan;
13 return $U$;
```

3.2 Bounding and Precomputation

In this section, we discuss precomputation procedures that can simplify a given $P||C_{max}$ instance. We first discuss upper and lower bounds that we use to bound the problem, and then we present some rules that we use to decrease the size of the related $P||C_{max}$ decision problem before encoding it into SAT.

3.2.1 Lower Bounds

Here, we improve the performance of the lifting procedure presented in Section 2.9.2. In the context of our SAT solving based framework, a tight lower bound is crucial, since it allows us to avoid having to prove unsatisfiability of our SAT formula, which can be prohibitively expensive. It is thus usually worth it to invest more time into calculating a better lower bound, since it is, oftentimes, cheaper to calculate a better bound than it is to prove the unsatisfiability of a given instance.

An Improved Lifting Procedure for $P||C_{max}$

Haouari et al. [HGJ06] show that for a given $P||C_{max}$ instance, there are $O(n)$ easily obtainable sub-instances, for which a lower bound can provide a close-to-optimal lower bound on the original instance. We take this a step further by exactly solving these sub-instances. This gives us tighter bounds on our original problem if done correctly. We do this as follows.

Given an instance (W, m) of the $P||C_{max}$ problem, we can generate upper and lower bounds U, L. Using the procedure presented by Haouari et al. we can generate sub-instances, bound them from above and below, and remember the best lower bound L on our original instance. After all sub-instances have been generated and bound, we can attempt to solve any instance that might improve upon L. For any sub-instance I' with upper and lower bounds U', L' such that $U' > L$ we can attempt to solve it exactly within a given time frame. We choose our time frame in a way, such as to allow each instance to be attempted within the time allocated to this bounding technique.

Preliminary tests show that in practical scenarios, the number of such instances tends to be very small (i.e. between 0 and five). These instances also tend to be very small, meaning they can be solved exactly almost all the time. As we see in Section 8.3.1 this technique warrants its extra computational complexity.

3.2.2 Upper Bounds

Upper bounds decrease the number of SAT calls necessary to solve a given instance. For optimal runtime, it is also relatively important to have a series of upper bounding procedures of varying cost and accuracy. It is often the case that a cheap upper bound is able to exactly solve a $P||C_{max}$ instance, which allows us to restrict the usage of more expensive upper bounding techniques.

Again, our upper bounds are usually cheaper than SAT calls, so it is often worth the extra computational cost. In this section, we present two novel upper bounding techniques for the $P||C_{max}$ problem.

LPT++

In Section 5 we introduce some pruning rules that can be used to simplify the $P||C_{max}$ decision problem. These can also be used to improve the results of LPT. Thus, we define the *LPT++* heuristic for the $P||C_{max}$ decision problem as follows. We combine the LPT heuristic with the Fill-Up rule outlined in Section 5.1.3 (it is not necessary to be familiarised with the pruning rules at this point). Since the pre-computation required to apply the Fill-Up rule can be quite time-consuming, we use the simplified Fill-Up rule (Pruning-Rule 8). In essence, this rule states that given a partial assignment where processor p_x has remaining space w_i and j_i is unassigned, we can simply assign j_i to p_x and maintain satisfiability.

In order to obtain an upper bound for a given $P||C_{max}$ optimisation instance, we first obtain a lower bound L and an upper bound U and then execute LPT++ on the respective $P||C_{max}$ decision problems with makespans within $[L, U)$. The lowest value for which LPT++ returns true is then a valid upper bound for the original $P||C_{max}$ instance. For the sake of convenience, we refer to this new $P||C_{max}$ heuristic as LPT++.

LPT#

Our LPT# heuristic is loosely inspired by the *Multi-Subset* heuristic by Dell'Amico and Martello [DM95]. In the LPT# heuristic, a $P||C_{max}$ decision problem is given with a makespan U. For each processor, we solve a SSP optimisation problem with the remaining jobs. To each processor, we sequentially assign the subset of jobs that maximises the processor's workload (while maintaining that its workload is at most U). This can be done in pseudo-polynomial time using dynamic programming [Pis99]. In order to bound a given $P||C_{max}$ instance we then, similar to the LPT++ heuristic, execute the LPT# heuristic on the respective $P||C_{max}$ decision problems with makespans in $[L, U)$ and return the lowest value for which the LPT# heuristic returns true. Similar to before, we also refer to this new $P||C_{max}$ heuristic as LPT# for the sake of convenience.

3.2.3 Problem Simplification

Beyond bounding a given $P||C_{max}$ instance, before invoking the SAT solver a $P||C_{max}$ decision problem instance can also be simplified to reduce the workload required by the SAT solver. This can range from simple symmetry breaking rules, to finding valid partial assignments. In this section, we present some of the rules that can be used to simplify the problem before invoking the SAT solver. These rules are not always necessary, but are very effective at decreasing encoding size, and can also lead to exceptional performance gains for suboptimal encodings.

Applying Pruning Rules in Advance

We can apply the pruning rules defined in Section 5 before invoking the SAT solver (it is not necessary to be familiarised with the pruning rules at this point). For each unassigned job, we can use the defined pruning rules to limit the processors on which it can be assigned. For the sake of simplicity and performance, we use Pruning Rules 2, 5, and 8. We apply all of these rules sequentially on the jobs $j_1, \ldots, j_n$.

These rules are applied in the following manner. To apply Rule 2 we enforce that the i'th unassigned job can only be assigned to the first i processors of each size. To apply Rule 5 for any two jobs j_h, j_i $(h < i)$ of the same size, we assert that we cannot assign j_i to the first processor which can fit j_h but not j_h and j_i. To apply Rule 8 if any processor p_x, has w_i space remaining, and j_i is unassigned, we assign j_i to p_x.

These rules can be applied exhaustively without affecting the satisfiability of our SAT encodings (as we will later discuss). While they are a necessary part of the encoding, since they help drastically reduce formula size and computation, throughout the rest of this thesis we will not consider them in order to avoid unnecessary complexity. Any encoding we present functions analogously, whether these rules have been applied or not.

3.3 Post-Processing

In this section, we discuss how the MSS heuristic can be adapted to be used to improve assignments obtained from SAT calls. Since SAT calls do not necessarily provide globally (or locally) optimal assignments to the underlying problem, it is very often the case that an assignment given by a SAT solver can easily be improved by an iterative improvement approach.

SSSS

The issue with the MSS heuristic, is that after a local minimum is reached, the entire process is started at a different random assignment. This can be detrimental in cases where we are close to a globally optimal solution, but stuck in a neighbouring locally optimal solution. Thus, we define the *Single-Start-Subset-Sum* heuristic as follows. The MSS heuristic is executed, but instead of restarting when a local optimum is reached, we instead perturbate the current best solution. The perturbation step is defined using a perturbation factor $\alpha \ll n$. To perturbate our assignment, a processor is first selected at random and then α of the jobs assigned to that processor are assigned at random to any of the other processors. The perturbation factor can be gradually increased after every perturbation to generally explore the solution space around the current best solution.

The SSSS heuristic can also be used as an upper bound heuristic, but requires a good hot-start, since it would otherwise be searching in suboptimal neighbourhoods.

4. Base SAT Encoding

In this section, we introduce different SAT encodings for the $P||C_{max}$ decision problem. To do this, we present three encodings from the literature for *pseudo-Boolean constraints (PBC)*. Due to the similarity between these problems, the adaptation of successful encodings for PBCs deepens our understanding of the $P||C_{max}$ decision problem.

4.1 Representing Jobs and Processors

In order to represent the assignment of job i to processor x, we introduce a Boolean *assignment variable* $a_{i,x}$. The assignment of $a_{i,x} = \top$ represents the assignment $a_i = x$ in the respective $P||C_{max}$ instance. Since the execution order is irrelevant in $P||C_{max}$ we do not need to represent it.

We use the naive exactly-one encoding ($O(m^2)$ clauses for mutual exclusion, and one clause for conjunction, see [KK07]) to ensure that each job is assigned to exactly one processor.[1] This gives us a total of $O(n \cdot m)$ variables and $O(n \cdot m^2)$ clauses to represent the assignments of jobs to processors.

To encode an instance of the $P||C_{max}$ decision problem (W, m, U), what is left is to then ensure that the sum of the weights assigned to each processor are $\leq U$. We do this by defining a PBC

$$a_{1,x} \cdot w_1 + a_{2,x} \cdot w_2 + \ldots + a_{n,x} \cdot w_n \leq U,$$

for each processor x and then encoding each PBC into SAT individually. In the following sections, we discuss the different ways of encoding PBCs into SAT, and their effect on the resulting $P||C_{max}$ encoding.

4.2 Adder Encoding

In the *adder encoding* [ES06] we use a SAT encoding of the full adder and an encoding of $\leq U$ to encode a $P||C_{max}$ decision instance (W, m, U) into SAT. We do this, by representing the weights of the jobs in binary. For job i, we define $\lceil \log(w_i) \rceil$ variables, which we refer to as a vector bw_i (binary weight), and add the required unit clauses to assert that bw_i corresponds to the binary representation of w_i. With that, we then obtain the *effective*

[1] It is important to note that in practice, feasible instances have reasonably small values for m that do not warrant a better at-most-one encoding.

weight of job i on processor x by encoding $ew_{i,x} \leftrightarrow bw_i \wedge a_{i,x}$.[2] Here $ew_{i,x} = bw_i$ if $a_{i,x} = \top$ and $ew_{i,x} = (\bot, \ldots)$ otherwise. Now that we have the effective weights of each job on each processor, we can calculate their sum using an encoding of a (bounded)[3] full-adder to get the makespan of each processor. The weights are added logarithmically (i.e. $((ew_{1,x} + ew_{2,x}) + (ew_{3,x} + ew_{4,x})) + (\ldots) \ldots)$ in order to have shorter propagation chains. Given the binary representation of the makespan of each processor, we can assert that it is $\leq U$. For a more detailed explanation of the adder encoding, we defer to the literature [ES06].

This encoding has many problems. Beyond those discussed in the literature, such as this encoding not maintaining GAC or inconsistency detection, it is also very difficult to encode the pruning rules from the literature into it. Let us take Pruning Rule 2 as an example. In order to apply Pruning Rule 2 to a job i and two processors, x, y, we need to know (i.e. have the binary representation of) the makespans of p_x and p_y at the point where we decide to assign j_i to one of them. In other words, we need the partial sum of the effective weights on each processor. If we do not add up the effective weights on each processor sequentially, then it is difficult to obtain the partial sum of the effective weights. Assuming we can obtain the partial sum of the effective weights, we would have to do a pairwise equality check of the corresponding values of the sums. For a job j_i and any two processors p_x, p_y, letting $ps_{i,x}$ ($ps_{i,y}$) be the makespan of p_x (p_y) at decision level i, we would need to encode $e_{i,x,y} \leftrightarrow (ps_{i,x} = ps_{i,y})$. Keep in mind that $ps_{i,x}$ ($ps_{i,y}$) is a potentially quite long vector of variables. The variable $e_{i,x,y}$ can then be used to imply that job i cannot be applied to processor y (assuming $x < y$).

In this case, we have had to sacrifice quite a lot in order to encode this pruning rule using the adder encoding. It forces us to add a few very long clauses, and also greatly increases the length of propagation chains during solving, both due to these clauses, and due to the fact that we create long chains when adding the effective weights sequentially (as opposed to logarithmically). This burdens the solver and outweighs the benefits of the pruning rule. It is also commonly believed to be beneficial to maintain GAC or inconsistency detection.

4.3 Binary-Merge Encoding

In order to alleviate the problems of having very long propagation chains, and a lack of inconsistency detection, we look into the Binary-Merge Encoding [MPS14]. The Binary-Merge encoding detects inconsistencies via unit propagation, but does not maintain GAC in practice. This encoding uses the concept of *cardinality networks* [ANORC09]. These are two operations, $Sort$ and $Merge$ that respectively can be used to sort a list of SAT variables, or merge two sorted lists of SAT variables. It is also useful for us to limit the number of true variables. Thus, for a given list L of SAT variables, such that no more than α of them can be true, $SORT(L, \alpha)$ returns a list L' of α variables, ensuring that no more than α variables in L are true, that a prefix of L' is true and the rest of the variables are false, and that L and L' contain the same number of true variables. Intuitively, for two lists of SAT variables, A, B, their merging $Merge(A, B, \alpha)$ returns a sorted list containing as many true variables as A and B combined (but at most α).

Given these two functions, the Binary-Merge Encoding for a PBC goes as follows. First, we normalise the PBC such that it is of the form

$$1 \cdot \beta + a_{1,i} \cdot w_1 + a_{2,i} \cdot w_2, \ldots, a_{n,i} + w_n < \gamma \cdot 2^{\delta},$$

where β, γ, and δ are known constants and $\delta \geq \max_i \lceil \log w_i \rceil$. Here $1, a_{1,i}, a_{2,i}, \ldots$ act as decider variables, taking on values in $\{0, 1\}$. We then insert the decider variables into bins.

[2]Here the $\wedge$ represents an element-wise and of the elements in bw_i and $a_{i,x}$.

[3]A full-adder encoding that is unsatisfiable if the bitlength of the result is too large

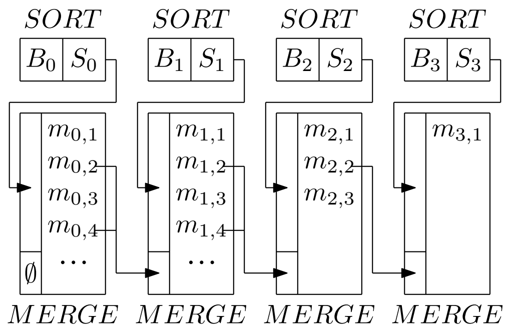


Figure 4.1: The binary merge encoding for a PBC with a limit of 2^3.

In bin B_ϵ we place every decider variable whose respective weight's binary representation has a 1 in the ϵ'th place. We then set $S_\epsilon := SORT(B_\epsilon, \alpha)$, with α being the maximum number of jobs with decider variables in B_ϵ that we can fit in a processor. Each true variable in S_ϵ represents an added workload of size 2^ϵ to this processor. Meaning, if we assign a job of size 7 to a processor, it will cause one variable in each of S_0, S_1 and S_2 to be set to true. Hence, we need to "add up" the sorted sets. Each variable in S_ϵ is weighted twice as heavily as the variables in $S_{\epsilon-1}$. Thus, we begin by setting $M_{-1} := \emptyset$ and proceed as follows. For $\epsilon \geq 0$, and for $S_\epsilon = \{s_{\epsilon,1}, s_{\epsilon,2}, \ldots\}$ we define M_ϵ, by merging S_ϵ with the set of variables from $M_{\epsilon-1}$ with even index (i.e. $m_{\epsilon-1,2}, m_{\epsilon-1,4}, m_{\epsilon-1,6} \ldots$). With that, all that remains is asserting $m_{\delta,\gamma} = \bot$. This process is represented in Figure 4.1.

This encoding has been shown to provide good performance on PBC benchmarks [PS15], and performs surprisingly well when used in this manner as well. While this encoding itself does not maintain GAC (in practice), it does maintain inconsistency detection.

However, it is also not without its problems. The encodings proposed by Achá et al. [ANORC09] for the functions *SORT* and *MERGE* could be slightly improved upon. For $A = SORT(B, \alpha)$ Achá encodes the statement *A is sorted and has at least as many true variables as B*. Similarly, for $A = MERGE(B, C, \alpha)$, Achá encodes the statement *A is sorted and has at least as many true variables as B and C combined.* This is due to the fact that they use a Tseitin encoding [Tse83] with the Plaisted-Greenbaum optimisation [PG86]. In order to check the satisfiability of a set of PBCs this is acceptable, but for a CDCL based solver, this can sometimes be counter-intuitively worse, especially if the output of the circuit is used for further computation. Achá et al. focus heavily on the size of the encoding of the cardinality networks. To encode the *MERGE* function, Achá et al. use $O(n \log n)$ clauses, and $O(n \log n)$ auxiliary variables. However, when setting an input variable to $\top$, $O(\log n)$ variables must be propagated before the value of the corresponding output variable is learned.

In our case, however, $\alpha \in O(n/m)$ is usually quite small in comparison to n. It is therefore quite beneficial to have a potentially larger encoding for the cardinality networks, if it speeds up propagation for small networks. We instead opt for using $O(1)$ auxiliary variables, $O(n + \alpha^2)$ clauses, and a propagation chain length in $O(1)$.

4.3.1 A New Binary-Merge Encoding

In order to achieve this, we provide a new Binary-Merge encoding for PBCs, and a new encoding for cardinality networks in order to be better suited for $P||C_{max}$. From this point on, we redefine the *SORT* and *MERGE* functions.

We begin by redefining *MERGE*. For two lists of sorted variables, $MERGE(A, B, \alpha)$ is defined as follows. First, we assume w.l.o.g. that $|A| = |B| = \alpha$ (since we can pad shorter lists with $\bot$, and assert the suffix of lists that are too long is false). $MERGE$ returns a set of new variables $C := \{c_1, \ldots, c_\alpha\}$ and makes five types of assertions. For $A = \{a_1, \ldots, a_\alpha\}$ and $B = \{b_1, \ldots, b_\alpha\}$ we assert

$$
\begin{aligned}
a_\beta &\implies c_\beta & (\beta <= \alpha) \quad (4.1)\\
b_\beta &\implies c_\beta & (\beta <= \alpha) \quad (4.2)\\
a_\beta \wedge b_\gamma &\implies c_{\beta+\gamma} & (\beta + \gamma \leq \alpha + 1), \quad (4.3)\\
\overline{a_\beta} \wedge \overline{b_\gamma} &\implies \overline{c_{\beta+\gamma}} & (\beta + \gamma \leq \alpha + 1) \quad (4.4)\\
c_{\alpha+1} &= \bot & (4.5)
\end{aligned}
$$

Consequently, C is sorted, and contains exactly as many true variables as A and B combined. To define $SORT$, we use $MERGE$ to recursively merge sublists of a given list in the same manner as Merge-Sort, similar to what is done by Achá et al. [ANORC09].

We move on to discussing how to encode a PBC. For the best performance, we normalise the formula and proceed as before. For the purpose of testing, we choose this option. With this option, however, it is difficult to extract the exact value of the sum; something which could come in handy in a different context. Hence, for demonstratory purposes, we introduce a slightly altered variant that allows for further computation.

This option requires leaving the PBC de-normalised, and proceeding as follows. We calculate the sets $S_1, \ldots$ and $M_1, \ldots$ as before. Then, in order to obtain the assigned makespan for a given processor, we calculate the parity of the variables in each list $M_1, \ldots$, into variables $p_1, \ldots$. We do this for a given M_β by asserting the following.

$$
\begin{aligned}
m_{\beta,\gamma} \wedge \overline{m_{\beta,\gamma+1}} &\implies p_\beta & \text{for odd } \gamma < |M_\beta|\\
m_{\beta,\gamma} \wedge \overline{m_{\beta,\gamma+1}} &\implies \overline{p_\beta} & \text{for even } \gamma < |M_\beta|\\
m_{\beta,|M|} &\implies p_\beta & \text{for odd } |M_\beta|\\
m_{\beta,|M|} &\implies \overline{p_\beta} & \text{for even } |M_\beta|
\end{aligned}
$$

The variables $p_1, \ldots$ represent the binary representation of the makespan of the given processor.[4] We can simply assert that for $p = (p_1, \ldots)$, $p \leq U$, similar to what we did in Section 4.2. Doing this for each processor gives us a valid SAT encoding of $P||C_{max}$. In doing this, we lose inconsistency detection, since p can only be determined once all input variables have been decided. On the other hand, p can now be used for further computation.

However, one problem remains. Since this encoding does not give us the partial sum of the makespan of each processor, we are unable to encode the pruning rules into it effectively.

4.4 BDD Encoding

This leads us to our final SAT encoding for $P||C_{max}$. In the *BDD encoding* for PBCs, the weighted $\leq$ function is represented as a BDD and the calculated BDD is encoded into SAT.

4.4.1 The BDD Encoding for PBCs

The BDD encoding for PBCs goes as follows. Given a PBC $a_1 \cdot w_1 + a_2 \cdot w_2, \ldots, a_n + w_n \leq U$, we construct a BDD for said PBC with decision variables $a_1, \ldots, a_n$. An example of such a BDD is given in Figure 4.2.

[4]This allows this encoding to be used as a general n-way adder with far better performance than the adder-encoding.

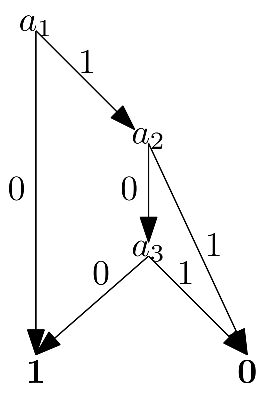


Figure 4.2: The BDD of the PBC $a_1 \cdot 7 + a_2 \cdot 4 + a_3 \cdot 3 \leq 8$

Given a BDD of any monotonic function (such as a PBC), we can encode it into a SAT formula in the following manner, described by Abío et al. [ANORC11, ES06]. For every node in the BDD, we define an auxiliary variable. Then for each node with auxiliary variable α on level i, we insert the clauses

$$\begin{aligned} \alpha &\implies \alpha_{\text{left}}, \\ \alpha \wedge a_i &\implies \alpha_{\text{right}}, \end{aligned}$$

where $\alpha_{\text{left}}, \alpha_{\text{right}}$ are the auxiliary variables of the left and right children respectively. We then also enforce that the auxiliary variable of the "False" node $\alpha_{\text{False}} = \bot$ and the auxiliary variable of the "True" node $\alpha_{\text{True}} = \top$. In order to maintain GAC, it is also necessary to enforce that the auxiliary variable of the root node $\alpha_{\text{root}} = \top$. A more detailed description along with proofs for these statements is presented by Abío et al. [ANORC11].

This encoding maintains GAC, and has been experimentally shown to have superior performance to the other presented encodings for PBCs [PS15]. However, it is not without cost. The first and largest cost is the size of the encoding. Abío et al. show that there are infinitely many PBCs that require an exponentially large BDD regardless of variable ordering. They also, however, present a workaround by splitting any PBC into multiple PBCs that together no longer need an exponentially large encoding. This greatly decreases performance, which is why we do not consider it here.

The other daunting problem is that it does not allow us to incorporate our pruning rules into the encoding, for two major reasons. The first, is that when using this for $P||C_{max}$ it does not directly provide us with the partial sum of the makespan of each processor. This problem, we can work around, but the other problem we cannot. In order to clarify the other problem, we give a meaning to the added auxiliary variables. Assuming we are traversing the BDD top-down, the assignment of the auxiliary variable $\alpha = \top$ states *from this point, the (partial) assignment of the variables below me does not force us to reach the "False" node.*

Let us take the example from Figure 4.2. Assigning $a_3 = 1$ automatically means that the auxiliary variable belonging to its only node must be false, causing the auxiliary variable of the node above it to also be false. This automatically tells us (via unit propagation) that we must set $a_1 = 0$. This is represented in Figure 4.3.

This meaning behind the auxiliary variables is what causes this encoding to maintain GAC, and is the major reason behind its great performance. The cost, however, is that we cannot tell exactly which node in the BDD a partial assignment brings us to, let alone what the current makespan is.

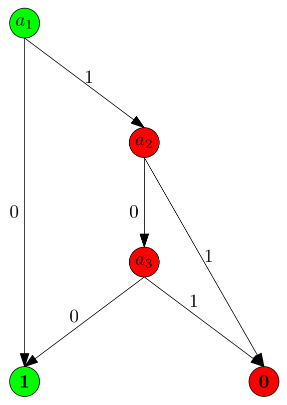


Figure 4.3: A representation of the assignment of the auxiliary variables that can be obtained via unit propagation, given the partial assignment $\{a_3 = \top\}$. Green represents the assignment $\top$ and red represents the assignment $\bot$.

4.4.2 A Worse BDD Encoding for PBCs

To fix this, we introduce a worse (i.e. less performant) BDD encoding for PBCs. This encoding proceeds in much the same way as that introduced in the previous section. First we introduce an auxiliary variable for each node of the BDD, we set $\alpha_{\text{root}} = \alpha_{\text{True}} = \top$ and $\alpha_{\text{False}} = \bot$. Then for each node with auxiliary variable α on level i, we insert the clauses

$$\begin{aligned} \alpha \wedge \overline{a_i} &\implies \alpha_{\text{left}}, \\ \alpha \wedge a_i &\implies \alpha_{\text{right}}, \end{aligned}$$

where $\alpha_{\text{left}}, \alpha_{\text{right}}$ are the auxiliary variables of the left and right children respectively. The difference between this encoding and the prior one is in that we now also consider the value of a_i when going left. Now, if an auxiliary variable α must be assigned $\alpha = \top$, then it can be interpreted as *the assignment of the decision variables above forces this node to be reached.* Consequently, this encoding no longer maintains GAC, and its performance suffers. However, now if we are given a partial assignment that lands us at a given node, its auxiliary variable must be true, and all other auxiliary variables on the same level can be, but are not forced to be, true.

4.4.3 Encoding the Pruning Rules

The last issue at hand is the issue of the partial sum of the makespan of each processor. As previously mentioned, our new BDD encoding tells us the path taken through the BDD by a given assignment, but it is not that simple to extract the makespan at each decision level.

In this section, we give an intuitive understanding of the nodes in our BDD, which we later use to develop new and improved pruning rules that we can easily incorporate into the BDD encoding.

To recall, a BDD is constructed in two distinct steps. In the first step, a complete decision diagram is constructed to represent the underlying formula. In the case of a PBC, each node in this decision diagram represents the sum of a subset of the weights above it. In the context of $P||C_{max}$ this means that a node which we call $n_{i,u}$ is present on the i'th level of the decision diagram if and only if a subset of $\{w_1, \ldots, w_{i-1}\}$ can be summed up to result in u.

In the second step of building the BDD, equivalent nodes are progressively merged (and paths shortcut). Two nodes are merged if and only if their underlying subgraphs are equivalent. Let us take two nodes, $n_{i,u}$ and $n_{i,u'}$, in the decision diagram with equivalent subgraphs. In the context of traversing the decision diagram, whether we are currently at $n_{i,u}$ or $n_{i,u'}$ does not matter, since any decision we make will have the same outcome in either case. What this means is that in the context of the $P||C_{max}$ decision problem, whether a processor is currently assigned a workload u or u' is irrelevant.

To build upon this example, let us consider Pruning Rule 2. Again, we take two nodes, $n_{i,u}$ and $n_{i,u'}$, in the decision diagram with equivalent subgraphs, with the exception that $n_{i,u}$ is a node in the decision diagram belonging to p_x , and $n_{i,u'}$ is a node in the decision diagram belonging to p_y. Pruning Rule 2 states that if we reach $n_{i,u}$ and $n_{i,u'}$, and $u = u'$, then there is no need to consider assigning job j_i to p_y (assuming $x < y$). However, since their respective subgraphs are equivalent, assigning j_i to p_x will lead to a collision if and only if assigning it to p_y does as well. Thus, there is no need to consider assigning job j_i to p_y even if $u \neq u'$.

This makes it clear, that when considering the $P||C_{max}$ decision problem, there are stronger pruning rules to be found, that can help us improve the performance of our SAT encodings. These new rules are formalised and applied to our BDD encoding in the next section.

5. Novel Pruning Rules and the BDD Encoding

In order to decrease the search space of our SAT solver, we incorporate pruning rules for BnB algorithms into our encoding. Inspired by the unique structural properties of the BDD encoding, we develop new and improved pruning rules for more aggressive search space optimisation.

5.1 Novel Pruning Rules for the $P||C_{max}$ decision problem

As we have discussed, there are more symmetry breaking rules to be found when considering the $P||C_{max}$ decision problem. In this section, we assume the development of a BnB algorithm for the $P||C_{max}$ decision problem, and develop novel pruning rules for such a scenario. Keep in mind that we intend to incorporate these rules into the BDD encoding.

5.1.1 Precedence Relations

Pruning Rule 1 is very effective at enforcing an ordering onto the jobs. Effectively, Pruning Rule 1 states that if two jobs have the same weight then the first one starts at an earlier time than the second one. Since this rule requires making a case distinction over C_x, it can potentially cause a blow-up in formula size, since we would have to add a new clause for every possible value for C_x. Instead, we introduce the following rule.

Pruning Rule 5. *For any two jobs j_h, j_i $(h < i)$, s.t. $w_h = w_i$, we enforce that $a_h \leq a_i$.*

The correctness of this condition is quite trivial to prove. For any assignment where this is not the case, we can simply swap a_h and a_i (iteratively for every h, i that break this condition) to obtain an assignment with the same resulting makespan where this condition does hold.

There is also a stronger pruning rule to be obtained here. Let us assume we have three jobs, j_g, j_h, j_i, s.t. $w_g = w_h + w_i$. If j_h and j_i are assigned to the same processor, then swapping the assignments of j_h and j_i with the assignment of j_g leads to an assignment with the same makespan. We can also clearly generalise this for an arbitrary number of jobs. This leads us to the following generalisation of Pruning Rule 5.

Pruning Rule 6. *For any set of jobs $\{j'_1, \ldots, j'_\alpha\}$ with assignments $\{a'_1, \ldots, a'_\alpha\}$ and weights $\{w'_1, \ldots, w'_\alpha\}$, and a job j_g s.t. $w_g = \sum_{\beta \in [1,\alpha]} w'_\beta$, if for all $h, i \in [1, \alpha]$, $a'_h = a'_i$, then we can enforce $a_g \leq a'_1$.*

While this rule is theoretically very strong, it can pose some issues when encoding it into SAT. Firstly, the larger α gets, the more such relations we would have, which could lead to a huge blow-up in formula size. Furthermore, the larger α gets, the less useful information we obtain from each condition. Thus, for practical purposes. we only use Pruning Rule 5.

5.1.2 Equivalencies in Partial Assignments

Our second pruning rule is strongly inspired by Pruning Rule 2, and involves deciding when two possible assignments for a job are equivalent. Given a partial assignment of the jobs, where we are now deciding where to place job j_i, if two processors have the same current workload, then assigning j_i to the former processor is equivalent to assigning it to the latter. When restricting ourselves to the $P||C_{max}$ decision problem, we can find a stronger equivalency condition.

Given an instance (W, m, U) of the $P||C_{max}$ decision problem, consider an element j_i, a number $u \leq U$ and the set $J_i = \{j_i, \ldots, j_n\}$. We can calculate the function

$$\phi(j_i, u) = \{J' \subseteq J_i \mid u + \sum_{j_\alpha \in J'} w_\alpha \leq U\}.$$

An intuition behind ϕ goes as follows. Assume we have already assigned u work to a processor, and now want to decide on assigning j_i to this processor as well. ϕ lists all possible combinations of jobs in J_i that we can now assign to this processor and still have an assigned workload (on this processor) that is at most U. With that we obtain the following pruning rule.

Pruning Rule 7. *Given an instance (W, m, U) of the decision problem of $P||C_{max}$ and a partial assignment A, if there are two processors with assigned workload C^A_x and C^A_y, s.t.*

$$\phi(j_i, C^A_x) = \phi(j_i, C^A_y),$$

for the largest unassigned job j_i, then we only need to consider assigning j_i to one of the two processors (which one is irrelevant).

Proof. Let us assume, w.l.o.g., that assigning j_i to p_x leads to a valid solution of the $P||C_{max}$ decision problem. In that valid solution, consider all the jobs, J', that have been assigned to p_x after C^A_x. Also consider all the jobs, J'', that have been assigned to p_y after C^A_y. Since $\phi(j_i, C^A_x) = \phi(j_i, C^A_y)$ and $J', J'' \subseteq J_i$ we know that $J', J'' \in \phi(j_i, C^A_x)$ and $J', J'' \in \phi(j_i, C^A_y)$. Thus, we can swap the assignment of the elements in J' with the assignment of the elements in J'', obtaining another solution to the $P||C_{max}$ decision problem. □

In the proof of Pruning Rule 7 we swap two parts of a satisfying assignment to obtain another satisfying assignment. It is worthy to note that doing this can provide worse solutions, but guarantees that they have makespan $\leq U$.

5.1.3 The Fill-Up Rule

Another pruning rule involves the maximum element in $\phi(j_i, C_x)$. For the sake of understandability, however, we first present a simpler rule. Given are an instance of the $P||C_{max}$ decision problem (W, m, U) and a partial assignment of the variables. Assume we have a processor that has a currently assigned workload C_x, and a job j_i such that $w_i + C_x = U$. We know, that if the partial assignment we have is correct, then any valid continuation of this assignment cannot assign a larger workload to p_x than that obtained by assigning j_i to p_x. Thus, given any valid continuation of our partial assignment, we can simply swap the elements assigned to p_x (that continue our partial assignment) with j_i. This is represented in Figure 5.1.

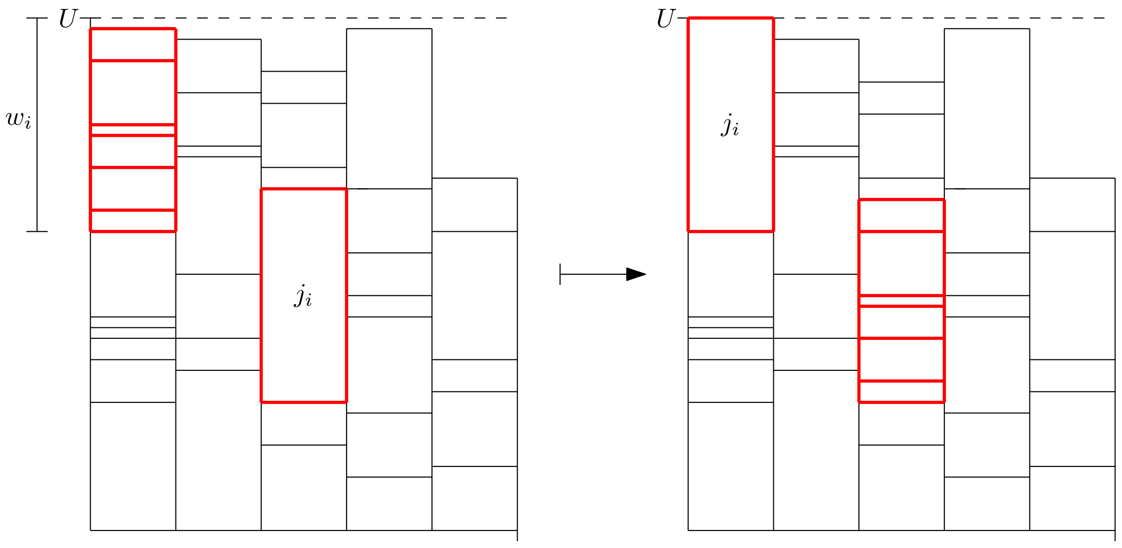


Figure 5.1: A simple example of the effect of the Fill-Up Rule.

With that remark, we obtain the following pruning rule.

Pruning Rule 8. *Given an instance (W, m, U) of the decision problem of $P||C_{max}$ and a partial assignment A, if there exists a processor with assigned workload C_x^A and a job j_i with $w_i + C_x^A = U$, then we can directly assign j_i to p_x.*

With that in mind, we can easily generalise this rule for cases where $w_i + C_x^A$ is not necessarily equal to U, but is as close as we can get to U. The proof of this is the exact same as that of Rule 8, and the statement of this rule goes as follows.

Pruning Rule 9 (The Fill-Up Rule (FUR))**.** *Given an instance (W, m, U) of the decision problem of $P||C_{max}$ and a partial assignment A, let p_x be a processor with assigned workload C_x^A and j_i be the largest unassigned job that can still be assigned to p_x (i.e. $w_i + C_x^A \leq U$). If*

$$w_i = \max\left\{\sum_{j_\alpha \in J'} w_\alpha \mid J' \in \phi(j_i, C_x^A)\right\}$$

then we can directly assign j_i to p_x.

Proof. Let us consider a completion S of A where j_i is assigned to p_y, with $y \neq x$. Let us consider all of the jobs $j'_1, \ldots$ with weights $w'_1, \ldots$ that are assigned to p_x and are not already assigned in A. We know that $\sum_\alpha w'_\alpha \leq w_i$. With that, we know that by swapping j_i with $j'_1, \ldots$ we obtain a solution S' where $C_x^S \leq C_x^{S'} \leq U$ and $C_y^{S'} \leq C_y^S \leq U$. Thus, with partial assignment A we can directly assign j_i to p_x and maintain satisfiability. □

We can efficiently check if the FUR can be applied by checking if $\phi(j_i, C_x) = \phi(j_i, U - w_i)$.

5.2 Integrating Pruning Rules into the BDD Encoding

In this section, we integrate our new Pruning Rules into the BDD encoding.

Precedence Relations

Incorporating Pruning Rule 6 into any of our SAT encodings is relatively simple. Given a set of jobs $\{j'_1, \dots\}$ with weights $\{w'_1, \dots\}$ and assignments $\{a'_1, \dots\}$ and a job j_i s.t. $w_i = \sum_\alpha w'_\alpha$; for each processor p_x, and each $y > x$ $(x < m)$ we add the clause

$$a'_{1,x} \wedge a'_{2,x} \wedge \dots \implies \overline{a_{i,y}}.$$

In words, if all of $j'_1, \dots$ are assigned to the same processor, then j_i cannot be assigned to a later processor.[1]

The Fill-Up Rule

Incorporating Pruning Rule 7 relies on the similarity between the ϕ set, and the nodes of the BDD. To recall, each node in the BDD represents a set of values, which we refer to as the *MERGED* set at the end of Section 2.5.

Theorem 5.1. *Let (W, m, U) be an instance of the $P||C_{max}$ decision problem, we construct the respective ϕ set and the respective BDD for the underlying PBC. For any node n on level i in the BDD, any $u \in MERGED(n)$, and for all $u' \in \{\sum_{\alpha \in J'} w_\alpha \mid J' \subseteq \{1, \dots, i-1\}\}$,*

$$\phi(j_i, u) = \phi(j_i, u') \iff u' \in MERGED(n).$$

Proof. The first direction is clear due to the definition of the ϕ set. If $\phi(j_i, u) = \phi(j_i, u')$, then the subgraphs of the nodes $n_{i,u}$ and $n_{i,u'}$ must be identical, and they therefore must be merged during the creation of the BDD.

To prove the opposite direction we see that if $u, u' \in MERGED(n)$, then the underlying subgraphs of $n_{i,u}$ and $n_{i,u'}$ must be identical. This alone implies that

$$\{J' \subseteq J_i \mid u + \sum_{j_\alpha \in J'} w_\alpha \leq U\} = \{J' \subseteq J_i \mid u' + \sum_{j_\alpha \in J'} w_\alpha \leq U\},$$

meaning that $\phi(j_i, u) = \phi(j_i, u')$. □

In order to encode the FUR, all we have to do is find all nodes in the BDD where the FUR can be applied using Theorem 5.1, and encode the statement *if j_i has still not been assigned, and this node is reached, then it cannot be assigned to any of the later processors.* This helps us uniquely assign j_i to the first available position where the FUR can be applied.

However, Theorem 5.1 only applies to level i and job i. If we consider job i and level h with $h \neq i$, then we have to manually check that $MERGED(n) \subseteq \{u \mid \phi(j_i, U - w_i) = \phi(j_i, u)\}$. Fortunately, this is not too difficult since $\{u \mid \phi(j_i, U - w_i) = \phi(j_i, u)\}$ is a range [2], and we can easily remember the min and max of each $MERGED(n)$. For $h < i$, this encoding can also cause some trouble; if there are multiple jobs of the same size then this encoding would forbid both of them from being assigned in the future, so for the sake of simplicity we only encode this rule for $h \geq i$.

[1]Refer back to Example 2.3.1 to see why we choose this encoding for the precedence relations.
[2]A formal proof of this can be found in [CKE08], we show this indirectly in Section 6.1.

Thus, in order to encode the FUR, all we have to do is for each job i find the node n on each level $h \geq i$ such that $\forall u \in MERGED(n) : \phi(j_i, u) = \phi(j_i, U - w_i)$. For each processor x and each n, let α be the respective auxiliary variable of n and add the clauses

$$\alpha \implies \overline{a_{i,y}},$$

for all $x < y$.

Equivalencies in Partial Assignments

Encoding Pruning Rule 7 greatly increases formula size, and yet still does result in performance benefits. In order to encode Pruning Rule 7, for every node n in the BDD, and every pair of processors p_x, p_y, $(x < y)$, let α_x and α_y be the auxiliary variables of the node n belonging to p_x and p_y respectively. We add the clause

$$\alpha_x \wedge \alpha_y \implies \overline{a_{i,y}}.$$

5.2.1 Collisions Between Pruning Rules

While we know that on their own our Pruning Rules maintain at least one valid solution, it is unclear whether the usage of all of them also maintains at least one valid solution. Here we show informally that using Pruning Rules 5, 7, and 9 together maintains at least one valid solution.

Given an instance of the $P||C_{max}$ decision problem, and an arbitrary series of partial assignments $A_0 \subset \ldots \subset A_n$ that does not violate Pruning Rule 9; $A_i = A_{i-1} \cup \{a_i = x\}$ for some x.

In order to obtain a solution that does not violate Pruning Rules 7 and 9, we iterate through the jobs from largest to smallest, defining partial assignments $A'_0, \ldots, A'_n$. For every job j_i, that has been assigned to processor p_x in A_i, we find the smallest y such that $\phi(j_i, C_y^{A'_{i-1}}) = \phi(j_i, C_x^{A'_{i-1}})$ and assign it there. At the end, it is clear that we obtain a valid solution that fulfils Rule 7. Since we only swap completions between processors within the same equivalence range, it does not affect any FUR applications, meaning that Pruning Rule 9 is also satisfied by our current solution. To obtain a solution that fulfils all three pruning rules, we simply reorder the jobs of the same size, fulfilling also Pruning Rule 5.

6. Efficient Computation

As it stands, our results are of a more theoretical nature. Unfortunately, since we need to calculate the ϕ function, it is hard to apply these rules in a variety of contexts. In this section, we focus on the efficient computation of equalities in the ϕ function. As a side effect of this, we develop a novel state-of-the-art approach for constructing BDDs for PBCs. This algorithm behaves especially well for constructing BDDs where there are many levels with roughly U nodes, such as those corresponding to the PBCs that underlay $P||C_{max}$ instances.

6.1 The Range Equivalency Table

In Section 5.1.2, we introduce a pruning rule with which we can draw equivalencies between partial assignments using a function which we refer to as ϕ. Since calculating ϕ explicitly would be prohibitively expensive, we introduce the auxiliary *range equivalency table (RET)*, which allows us to implicitly determine when $\phi(j_i, C_x) = \phi(j_i, C_y)$.

For the construction, we highlight two interesting properties of ϕ. Intuitively, one can see that $\phi(j_i, u) \supseteq \phi(j_i, u+1)$ for all $u < U$. This is because the sets of admissible jobs will never decrease when increasing the available processing time. The second important property is the fact that for $i < n$

$$\phi(j_i, u) = \phi(j_{i+1}, u) \cup \{j_i \cup J' \mid J' \in \phi(j_{i+1}, u + w_i)\}. \tag{6.1}$$

In words, the valid ways to assign jobs that may or may not include j_i is equal to the union of valid ways of assigning jobs that do not include j_i, and the valid ways of assigning jobs that do.

The RET is an $n \times (U+1)$ table with entries in $\mathbb{N}$. The first dimension of the RET is indexed from 1 to n, since each row of the RET represents the equivalence ranges for a single job. The second dimension of the RET is indexed from 0 to U —one entry for each possible assigned workload. For a job j_i, an *equivalence range* is a range of workloads $u, \ldots, u'$ such that $RET[i][u] = \ldots = RET[i][u']$. For such a range, we assert that $\phi(j_i, u) = \ldots = \phi(j_i, u')$.

We construct the RET from the smallest (j_n) to the largest job (j_1). For j_n, $\phi(j_n, u) = \{\emptyset, \{j_n\}\}$ if $u + w_n \leq U$, and $\phi(j_n, u) = \{\emptyset\}$ otherwise. We thus initialise two equivalence ranges: $RET[n][U - w_n + 1] = \ldots = RET[n][U] = 1$ and $RET[n][0] = \ldots = RET[n][U -$

$w_n] = 2$. For job j_i $(i < n)$, we denote the two entries of j_{i+1} that are relevant for applying property (6.1) as $\mathit{left}(i,u) := \mathit{RET}[i+1][u]$ and $\mathit{right}(i,u) := \mathit{RET}[i+1][u+w_i]$ (with $\mathit{right}(i,u) := 0$ if $u + w_i > U$). We start by setting $\mathit{RET}[i][U] := 1$ and then proceed sequentially for $u = U-1, U-2, \dots, 0$:

$$\mathit{RET}[i][u] := \begin{cases} \mathit{RET}[i][u+1], & \text{if } \mathit{left}(i,u) = \mathit{left}(i,u+1) \\ & \wedge\ \mathit{right}(i,u) = \mathit{right}(i,u+1), \\ \mathit{RET}[i][u+1]+1, & \text{otherwise.} \end{cases}$$

In words, we initialise a new equivalency range for j_i if *left* or *right* changes from $u+1$ to u, and we extend the prior range otherwise.

This can be quite daunting to begin with, but can be made very clear with an example.

Example 6.1.1. Given a set of 5 jobs with sizes $\{11, 7, 5, 3, 2\}$, we construct the *RET* for makespan $U = 13$. The first row is defined explicitly as follows.

2	2	2	2	2	2	2	2	2	2	2	2	1	1

Table 6.1: The bottom row of an exemplary *RET* table.

The second row, (for job number 4 with size 3) changes at index 11 where the bottom row changes, but also at index 10 when $right(4, 10) \neq 0$ for the first time. It then changes for the last time at index 8 where $right(4, 8) = 2$ for the first time. This is represented in Table 6.2.

4	4	4	4	4	4	4	4	4	3	3	2	1	1
2	2	2	2	2	2	2	2	2	2	2	2	1	1

Table 6.2: The bottom two rows of an exemplary *RET* table.

The complete table is then represented in Table 6.3.

11	10	9	8	7	6	5	4	4	3	3	2	1	1
9	9	8	8	7	6	5	4	4	3	3	2	1	1
7	7	7	7	6	6	5	4	4	3	3	2	1	1
4	4	4	4	4	4	4	4	4	3	3	2	1	1
2	2	2	2	2	2	2	2	2	2	2	2	1	1

Table 6.3: An exemplary *RET* table.

The purpose of the RET, is to serve as an abstraction of ϕ, such that we do not need to explicitly calculate it. We introduce the following theorem.

Theorem 6.1. *Given an instance (W, m, U) of the $P||C_{max}$ decision problem, for any $i \in \{1, \dots, n\}$ and $u, u' < U$*

$$\mathit{RET}[i][u] = \mathit{RET}[i][u'] \iff \phi(j_i, u) = \phi(j_i, u').$$

Proof. By induction. For $i = n$ this property is clear.

For $i < n$, we assume that the statement holds true for $i+1$. Given that $RET[i][u] = RET[i][u']$, we know that $RET[i+1][u] = RET[i+1][u']$ and $RET[i+1][u+w_i] = RET[i+1][u'+w_i]$. With our inductive assumption, this implies that

$$\begin{aligned}\phi(j_i, u) &= \phi(j_{i+1}, u) \cup \{j_i \cup x \mid x \in \phi(j_{i+1}, u + w_i)\} \\ &= \phi(j_{i+1}, u') \cup \{j_i \cup x \mid x \in \phi(j_{i+1}, u' + w_i)\} \\ &= \phi(j_i, u').\end{aligned}$$

Given that $RET[i][u] \neq RET[i][u']$, we know either $left(i,u) \neq left(i,u')$ or $right(i,u) \neq right(i,u')$. We assume w.l.o.g. that $u < u'$. If $left(i,u) \neq left(i,u')$, then by our inductive assumption, $\phi(j_{i+1}, u) \supsetneq \phi(j_{i+1}, u')$. We thus know that $\phi(j_i, u) \supsetneq \phi(j_i, u')$ since all other elements in either set must contain j_i. The case where $right(i,u) \neq right(i,u')$ proceeds analogously. Induction thus proves both directions in the above theorem. □

6.2 Compressed RET

The RET is a useful for efficiently pruning certain branches of the search tree. However, since it takes $O(U \times n)$ space, the RET can have a significant memory footprint for large instances. Thus, in this section we compress the RET to require only $O(U)$ memory, at the cost of having a worst case lookup time of $O(U)$.

In order achieve this, there is one crucial property of note.

Lemma 6.2. *For $i < n$, $RET[i+1][u] \neq RET[i+1][u+1] \implies RET[i][u] \neq RET[i][u+1]$.*

This property comes from the fact, that $RET[i][u] \neq RET[i][u+1]$ if $left(i,u) \neq left(i,u+1)$. We can therefore create an array, $CRET$, of size U (indexed from $[0, U-1]$) that saves the locations of the changes between ranges, and the index of the smallest job at which this change occurs. To be specific, $CRET[u]$ stores the largest index i such that $RET[i][u] \neq RET[i][u+1]$.

We build the $CRET$ as follows. We initialise all entries of the $CRET$ with 0 and set $CRET[U - w_n] = n$. Then, for each job j_i from j_{n-1} to j_1, and each entry with $CRET[u] = 0$, we set $CRET[u] = i$ if $CRET[u + w_i] > i$ or $u + w_i = U$.

Example 6.2.1. In this example, we build the $CRET$ corresponding to the RET presented in Example 6.1.1.

1	2	1	3	2	3	3	0	4	0	4	5	0

Table 6.4: The $CRET$ corresponding to the RET from Example 6.1.1.

In order to check if $RET[i][u] = RET[i][u']$ $(u < u')$ we have to check if $\forall \alpha \in [u, u') : CRET[\alpha] < i$. Checking this has a theoretical worst case runtime of $O(U)$, but due to the small size of most ranges, performance does not take a noticeable hit in most practical scenarios. We verify this experimentally in Section 8.3.3.

6.3 Iterative CRET Creation

In the context of optimisation, we would need to solve multiple $P||C_{max}$ decision instances with varying U's. Since the construction of the CRET comes with an inherent cost, we do not want to recompute the CRET every time U is varied. In this section, we present a way with which the CRET can be reused for varying values of U.

Theorem 6.3. *Let $CRET^U$ be the CRET for the $P||C_{max}$ decision problem (W, m, U). Let $CRET^{U-1}$ be the CRET for the $P||C_{max}$ decision instance $(W, m, U-1)$, then*

$$CRET^{U-1}[u] = CRET^U[u+1] \quad (u \in [0, U-2]).$$

Proof. In order to prove this, we will simply consider the ϕ functions of the underlying PBCs. Let ϕ^U and ϕ^{U-1} be the underlying functions represented by $CRET^U$ and $CRET^{U-1}$ respectively. For any $u \in [0, U-1]$ and any j_i we see that

$$\begin{aligned}\phi^{U-1}(j_i, u) &= \{J' \subseteq J_i \mid u + \sum_{j_\alpha \in J'} w_\alpha \leq U - 1\} \\ &= \{J' \subseteq J_i \mid u + 1 + \sum_{j_\alpha \in J'} w_\alpha \leq U\} \\ &= \phi^U(j_i, u+1).\end{aligned}$$

Thus, since $\phi^{U-1}(j_i, u) = \phi^U(j_i, u+1)$, $CRET^{U-1}[u] = CRET^U[u+1]$. □

In words, given the CRET for a $P||C_{max}$ decision instance with upper bound U, all we need to do in order to obtain the CRET for the respective $P||C_{max}$ decision instance with upper bound $U-1$ is shift all entries in the CRET to the left by one slot.

6.4 Faster BDD Creation for PBCs

The creation of BDDs is a computationally expensive but important task in many areas of discrete optimisation [CCB22]. While there are a vast array of approaches of varying performance to build arbitrary BDDs [CCB22], domain knowledge generally helps improve the performance of BDD creation. Here we use the duality between the state transition graph (from dynamic programming literature [Ber05]) used to construct the RET and the nodes of the BDD [CCB22] in order to efficiently create BDDs for PBCs.

There have been a few constructions of BDDs for PBCs. Eén et al. [ES06] remember the sum value reached at the construction of each node to effectively prevent some (not all) equivalent nodes from being generated before being merged. Behle [Beh07] remembers a range for each node, effectively preventing the unnecessary creation of identical subtrees. Christian et al. [CKE08] uses the same algorithm as Behle, but also uses the ϕ set to formalise these ranges. That approach is top-down, and has a runtime of $O(n \cdot w \cdot \log w)$, where n is the number of variables in the PBC and w is the width of the widest level in the BDD. In our case, since we are generating BDDs for PBCs for $P||C_{max}$ we can also use the fact that n is relatively large, and only a small portion $(1/m)$ of the variables can be set to true for each PBC. This means that in our case usually $w \approx U$, or at least in the same order of magnitude. Using the *RET* to help generate our BDD bottom-up, we can then bring the runtime of BDD construction down to $O(n \cdot U)$.

Doing this is quite simple. We know that each range in the *RET* represents at most one node in the BDD, and each node in the BDD is represented by exactly one range in the *RET* (Theorem 5.1). Thus, we can construct our BDD as follows.

We begin by marking all the entries $[i][u]$ in our *RET* such that

$$\exists J' \subseteq \{1, \ldots, i-1\} : u = \sum_{j \in J'} w_j, \quad (J' = \emptyset \text{ for } i = 1).$$

This can be done in $O(n \cdot U)$ using dynamic programming. Then, from the bottom of the *RET* up, we iterate through the ranges, creating a node for each range where there is at

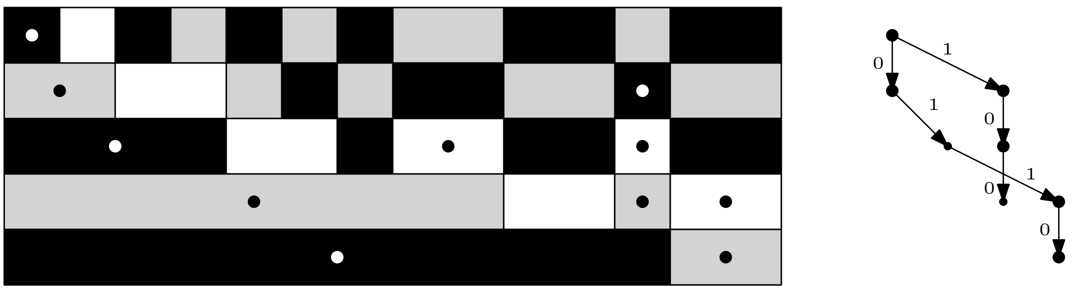


Figure 6.1: The RET, with ranges represented as singular colours, and markings represented by dots. On the right is the BDD constructed from the RET and its markings. Ignored 0 edges lead to $True$, and ignored 1 edges lead to $False$.

least one entry marked and storing pointers to its left and right children. We save the index of the respective node at each marking for efficient look-up on higher levels. For any range where *left* and *right* point to the same range, we store the index of the node belonging to that range instead of creating a new node. This also takes a total runtime of $O(n \cdot U)$.

Example 6.4.1. Building on Example 6.1.1, Figure 6.1 represents how our algorithm constructs a BDD for the PBC instance with weights $\{11, 7, 5, 3, 2\}$ and limit 13.

7. Incorporating Our Pruning Rules into Existing Exact Solvers for $P||C_{max}$

To demonstrate the effectiveness of our pruning rules in different contexts, in this section, we incorporate them into two BnB algorithms, as well as an ILP translation. The goal here, is to demonstrate how adding these extra conditions on solution structure can reduce search space size, resulting in faster completion. Since our rules apply to the $P||C_{max}$ decision problem, it will require a bit of effort to incorporate them into branching schemes for the $P||C_{max}$ optimisation problem.

7.1 Dell'Amico Martello

We now present an adaptation of the algorithm by Dell'Amico and Martello [DM95] to demonstrate how our new pruning rules can be integrated into BnB algorithms for the $P||C_{max}$ optimisation problem (see also [Law17]). Before we begin, we introduce the following theorem that helps us better integrate the FUR into our scheme.

Theorem 7.1. *Given is a $P||C_{max}$ decision instance (W, m, U) and a partial assignment A, such that the FUR can be applied to job j_i and processor p_x, giving us the partial assignment A'. If an optimal completion of A' has makespan U' such that $C_x^{A'} \neq U'$, then this is also an optimal completion of A.*

Proof. Assume that A has a completion S with makespan $\overline{U} < U'$. In this completion, j_i must have been assigned to processor p_y with $x \neq y$ (otherwise it would also be a completion of A'). Now consider the set of jobs $J' = \{j'_1, \ldots\}$ with weights $\{w'_1, \ldots\}$ that are assigned to p_x in S and are not already assigned in A. We know due to the FUR that $\sum_{j'_\alpha \in J'} w'_\alpha \leq w_i$. Thus, if we swap those jobs with j_i, we obtain a solution S' with $C_y^{S'} \leq C_y^S$, $C_x^{S'} \geq C_x^S$, and all other processors staying the same. Thus, we have two cases.

If $C_x^{S'} \leq \overline{U}$, then this is also a valid (and thus optimal) completion for A' with makespan $\overline{U}$.

If $C_x^{S'} > \overline{U}$, then clearly this is an optimal completion for A' and thus $C_x^{S'} = U'$. $\square$

In essence, what this theorem suggests, is that when applying the FUR, this assignment will either directly cause an increase in makespan, or will not affect it at all. In the context of BnB algorithms for the $P||C_{max}$ optimisation problem, we can exploit this to further reduce the search space.

We now integrate our rules into the branching scheme of Dell'Amico and Martello. We begin with initialising upper and lower bounds U, L for the given instance. Next, we set $U \leftarrow U - 1$ and consider the decision problem $I = (W, m, U)$.

We then call Algorithm 7.1 with $A = \emptyset$. This recursive algorithm returns TRUE if and only if A can be completed to provide a better solution than the current best. It proceeds as follows: First, we perform some infeasibility checks given A, where we see if the remaining space on the processors where at least one job can be inserted is large enough for the remaining jobs. We then apply Rule 3 if only three unassigned jobs remain (l. 2–5).[1] Otherwise, we check if the FUR can be applied, and if so we apply it and recurse (l. 6–10). If that recursion is unsuccessful, then the respective $P||C_{max}$ solution cannot be improved upon using the current partial assignment, and we return with the current best solution (l. 14). If the recursion is successful, then we check if Theorem 7.1 can be applied, and if so we apply it (l. 11–13). Otherwise, we assume U and the RET have been updated accordingly. We undo the assignment made by the FUR and need to recurse again in case further improvements are possible given the new bound (l. 12). Lastly, if all else fails, we branch over all processors (l. 15–20) while using Rules 1, 4, and 7. Note that the modifications made by a recursion may change which pruning rules apply to later processors, which is why we perform pruning individually and just-in-time.

We also need to consider any collisions between the pruning rules. By analogous remarks to those made in Section 5.2.1 we know that if a solution to the $P||C_{max}$ decision instance exists, then there exists one that fulfils all of our pruning rules. However, when applying Rules 7 and 9 there can be multiple assignments to choose from, where, potentially, only one of which leads to a solution that can be obtained using all three rules. Rules 7 and 9 do not affect each other (up to isomorphisms), so we only need to consider their effects on Rule 1. When applying the FUR, there can be multiple jobs of the same size to choose from. Thus, in order to restrict as few assignments as possible in the future, we select the job with the largest index, and then ensure that the order defined by Rule 1 is maintained by swapping the assigned jobs of the same size until they are ordered.[2] When applying Rule 7 there can be multiple processors to choose from. In order to restrict as few assignments as possible in the future, we can simply assign the selected job to the least loaded of the processors. Fortunately, this is already done by our branching order.

7.2 Haouari and Jemmali

Now we adapt a more modern approach by Haouari and Jemmali [HJ08], to demonstrate how our pruning rules can affect the performance of this algorithm as well. In this algorithm, processors are loaded sequentially instead of simultaneously. In other words, when branching, instead of deciding which processor the current job will be assigned to, we decide which job will be assigned to the current processor. The adapted algorithm proceeds as follows. Much like before, we begin with initialising upper and lower bounds U, L for the given instance. Next, we set $U \leftarrow U - 1$ and consider the decision problem $I = (W, m, U)$.

We then call Algorithm 7.2 with $A = \emptyset$ and $x = 1$. This recursive algorithm returns TRUE if and only if it found an improved solution. It proceeds as follows: First, we perform some infeasibility checks given A, as before (l. 1). If this processor is now full, then we recurse on the next processor (l. 2-3). If only two empty processors remain, then we solve them

[1] In this algorithm, we assume w.l.o.g. that $n \geq 3$. The problem is trivial if $n < 3$.

[2] In practice, in order to save computations we can simply ignore Rule 1 for jobs assigned with the FUR

Algorithm 7.1: DMFUR

```
    Global Data: $W$, $m$, $RET$, $L$, $U$, $\textsc{BestSolution} = \bot$
    Data: Partial Assignment $A$
 1  if $Infeasible(A)$ then return FALSE;
 2  if $|A| = n - 3$ then
 3      $\textsc{BestSolution}, U \leftarrow$ Apply Rule 4;
 4      $RET \leftarrow RET :: new(W, m, U)$;
 5      return TRUE;
 6  if $\exists\, i, x$ : as in the FUR $\wedge j_i$ unassigned then
 7      $i \leftarrow$ the largest such $i$;
 8      $A' \leftarrow A \cup \{a_i = x\}$;                          // Apply FUR
 9      Reorder jobs of size $w_i$;
10      if $DMFUR(A')$ then
11          if $C_x^{A'} = U$ then
12              $DMFUR(A)$;                                      // retry with new $U$
13          return TRUE;
14      else return FALSE;                                      // infeasible!
15  $j_i \leftarrow$ largest unassigned job;  $improved \leftarrow$ FALSE;
16  for each processor $p_x$ by free space descendingly do
17      if $a_i = x$ can be pruned via Rules 4 or 7 then continue;
18      $A' \leftarrow A \cup \{a_i = x\}$;
19      $improved \leftarrow improved \vee DMFUR(A')$;
20  return $improved$
```

to optimality by solving the underlying SSP using dynamic programming (l. 4–7).[3] We then check if the FUR can be applied, and if so, apply it like in the previous algorithm (l. 8–18). However, if we see that the FUR can be applied to a job that has previously been rejected from being assigned to this processor, we know that assigning this job to this processor cannot lead to an improvement and thus (due to the FUR) we can return FALSE (l. 9–10). We then attempt to assign the first job of each size in descending order of size, and recursing (l. 19–25). For each job we choose not to insert, we remember the current makespan of the processor; the index at which this job was rejected on this processor (l. 11–26). If a job has previously been rejected at a given makespan or a makespan that is equivalent via the RET on a previous processor, then there is no need to assign it at the same makespan again on the current processor (proof analogous to that of Pruning Rule 7) (l. 20–22). After all jobs have been rejected, we forget all rejections made at this level and backtrack (l. 27–28).

[3]In this algorithm, we assume w.l.o.g. that $m \geq 2$. The problem is trivial if $m < 2$.

Algorithm 7.2: HJFUR

```
Global Data: $W$, $m$, $RET$, $L$, $U$, BestSolution $=\perp$
Data: Partial Assignment $A$, Current Processor $x$
1  if $Infeasible(A)$ then return FALSE;
2  if no jobs fit on $p_x$ then
3      return $HJFUR(A, x+1)$;
4  if $x = m-1$ then
5      BestSolution, $U \leftarrow$ Solve SSP;
6      $RET \leftarrow RET::new(W, m, U)$;
7      return TRUE;
8  if $\exists$ unassigned $j_i$ as in the FUR then
9      if $j_i$ has previously been rejected from $p_x$ then
10         return FALSE;
11     $i \leftarrow$ the largest such $i$;
12     $A' \leftarrow A \cup \{a_i = x\}$ ;                          // Apply FUR
13     Reorder jobs of size $w_i$;
14     if $HJFUR(A', m)$ then
15         if $C_x^{A'} = U$ then
16             $HJFUR(A, m)$;                                  // retry with new $U$
17         return TRUE;
18     else return FALSE;                                      // infeasible!
19 for each unassigned, non-rejected job $j_i$ by size in descending order do
20     if $a_i = x$ can be pruned via Rule 5 or $j_i$ has been previously rejected at
       makespan $C_y^{A'}$ with $RET[i][C_y^{A'}] = RET[i][C_x^A]$ then
21         Remember that job $j_i$ was rejected;
22         continue;
23     $A' \leftarrow A \cup \{a_i = x\}$;
24     if $HJFUR(A', m)$ then
25         $improved \leftarrow$ TRUE;
26     Remember that job $j_i$ was rejected at makespan $C_x^A$;
27 Forget all rejections that happened this recursion;
28 return $improved$
```

7.3 Mrad and Souayah

Mrad and Souayah [MS18] use an ILP translation of $P||C_{max}$ and solve it using the generic ILP-solver CPLEX [Cpl09]. In this section, we present their ILP encoding, provide minor improvements such that it is better suited for the more modern Gurobi [Gur23] optimiser, and discuss how our pruning rules can be incorporated into the encoding.

The pseudo-polynomial $P||C_{max}$ encoding presented by Mrad and Souayah defines a directed multi-graph with $U+1$ nodes labelled $u_0, \dots u_U$. For each job j_i we define an edge $e_{\alpha,i} = u_\alpha u_{\alpha+w_i}$ if there is a subset of the jobs $j_1, \dots, j_{i-1}$ whose weights sum up to α (and $\alpha + w_i \leq U$). Mrad and Souayah then also define an edge $f_\alpha = u_\alpha u_U$ for each node. Solving a $P||C_{max}$ decision problem is then equivalent to finding m edge distinct $u_0 - u_U$ paths, with the extra condition that you use exactly one of the edges belonging to each job and the paths do not need to be distinct in the f edges. Solving a $P||C_{max}$ optimisation problem is then equivalent to minimising the index of the last node visited without the f edges.

This is done in the following manner. We define an integer variable *makespan*, a pseudo-Boolean variable $e_{\alpha,i}$ for each respective edge, and an integer variable f_α for each f_α edge. We then define the following constraints

$$\text{Minimise } \mathit{makespan} \tag{7.1}$$

$$\mathit{makespan} \geq (\alpha + w_i)e_{\alpha,i} \qquad (\forall e_{\alpha,i}) \tag{7.2}$$

$$\sum_{i=1,\ldots,n} e_{0,i} = m \tag{7.3}$$

$$\sum_{i=1,\ldots,n} e_{\alpha-w_i,i} - \sum_{i=1,\ldots,n} e_{\alpha,i} - f_\alpha = 0 \qquad (\forall \alpha \in [1, U-1]) \tag{7.4}$$

$$\sum_{\alpha=0,\ldots,U} e_{\alpha,i} = 1 \qquad (\forall j_i) \tag{7.5}$$

whereby undefined variables are replaced by 0.

An intuition behind each of the formulas goes as follows. Constraints 7.1 and 7.2 ensure that the correct minimum makespan is calculated, by minimising the index of the last visited node without the f edges. Constraint 7.3 ensures that there are m paths going out of u_0. Constraint 7.4 ensures that the number of paths that visit any node (besides u_0 and u_U) is the same as the number of paths leaving it. Constraint 7.5 ensures that only one edge belonging to each job is ever used.

In order to improve this encoding and increase the performance of the Gurobi optimiser, we replace Constraint 7.4 with

$$\sum_{i=1,\ldots,n} e_{\alpha-w_i,i} - \sum_{i=1,\ldots,n} e_{\alpha,i} >= 0 \qquad (\forall \alpha \in [1, U-1]),$$

and eliminate all f_α variables. We also add a constraint to provide the solver with the lower bound

$$\mathit{makespan} \geq L,$$

which allows the solver to break early on a significant number of instances.

Unfortunately, due to the structure of the encoding, and there being no way to read which job is assigned to which processor, there is no direct way to incorporate Pruning Rules 7 and 9. However, we can still incorporate Rule 1 to see how the addition of pruning rules affects this encoding.

In order to incorporate Rule 1, we assert that for any two jobs of the same size, the one with a lower index will be inserted at an earlier time-point. Thus, for any two jobs j_h, j_i of the same size, with $h < i$ we add the constraint

$$\sum_{\alpha=0,\ldots,U} \alpha \cdot e_{\alpha,h} - \sum_{\alpha=0,\ldots,U} \alpha \cdot e_{\alpha,i} \leq 0.$$

With that, we force an ordering onto jobs of the same size.

8. Experimental Evaluation

In this section, we test our different schemes in search of trends in our data.

8.1 Setup

For testing, we use an ARM Neoverse-N1 with 80 cores at 3 GHz and 256 GB of RAM, running Ubuntu for ARM 22.04. We run the tests in parallel over all instance (sequentially over each instance). We implement our algorithms in RUST 1.75.0. For SAT solving we use Kissat 3.1.1 [BFFH20, Bie23] and for ILP-solving we use Gurobi 11.0.0 [Gur23].

8.2 Benchmarks

For testing, we use three benchmarks that each vary in structure and difficulty.

Lawrinenko

The Lawrinenko (**Law**) instances (defined here: [Law17], but summarised and extended here [MS18]) are difficult due to combinatorial properties. This benchmark was inspired by Haouari and Jemmali [HJ08], that discover the first set of difficult instances; instances where $n/m = 2.5$. Thus, this benchmark focuses around instances with n/m being roughly 2.5. We use $n/m \in \{2, 2.25, 2.5, 2.75, 3\}$. For each n/m we have:

- $n \in \{20, 40, 60, 80, 100, 120, 140, 160, 180, 200\}$ if $n/m \in \{2, 2.5\}$,
- $n \in \{36, 54, 72, 90, 108, 126, 144, 162, 180, 198\}$ if $n/m \in \{2.25, 3\}$,
- $n \in \{22, 44, 66, 88, 110, 132, 154, 176, 198, 220\}$ if $n/m \in \{2.75\}$.

For each (n, m) pair, we have 7 classes used to generate job sizes.

1. discrete uniform distribution in $[1, 100]$
2. discrete uniform distribution in $[20, 100]$
3. discrete uniform distribution in $[50, 100]$
4. normal distribution with $\mu = 100$ and $\sigma = 20$
5. normal distribution with $\mu = 100$ and $\sigma = 50$
6. discrete uniform distribution in $[n, 4n]$
7. normal distribution with $\mu = 4n$ and $\sigma = n$

For each (n, m) pair and each class, we generate 10 instances, for a total of 3500 instances.

Franca Frangioni

The Franca Frangioni (**FF**) set [FGLM94, ANS04][1] is difficult despite having large n/m values. This set is difficult due to the distribution of job sizes, and due to the size of the makespan of generated instances, since very large makespans make pseudo-polynomial speed-up techniques ineffective. A lot of our algorithms were also very memory-bound on this benchmark set. This benchmark consists of two families of instances, the *uniform* and the *non-uniform*. In uniform instances, job sizes are generated using a uniform distribution. In the non-uniform instances, for a range $[a, b]$, $x\%$ of the jobs are generated using a uniform distribution from $[(b-a)y, b]$ and the other jobs are generated using a uniform distribution from $[a, (b-a)z]$. They choose $x = 98, y = 0.9$, and $z = 0.02$. In order to generate instances, they use $m \in \{5, 10, 25\}$ and $n \in \{50, 100, 500, 1000\}$. For $m = 5$, they also test $n = 10$. For each (n, m) pair, they generate 10 instances of each family, for a total of 780 instances.

Berndt Deppert

The Berndt Deppert (**BD**) set [BDJR22] consists of comparably easy instances, with high n/m values, reasonably sized makespans and job sizes generated from a uniform distribution. This benchmark set consists of 5 families. These five families are presented in Table 8.1. We generate 30 instances for each configuration, of which there are 102, for a total of 3060 instances.

	m	n	Job Sizes
E1	$3,4,5$	$2m, 3m, 5m$	$[1,20],[20,50]$
E2	$2,3$	$10,30,50,100$	$[100,800]$
	$4,6,8,10$	$30,50,100$	$[100,800]$
E3	$3,5,8,10$	$3m+1, 3m+2$	$[1,100],[100,200]$
	$3,5,8,10$	$4m+1, 4m+2$	$[1,100],[100,200]$
	$3,5,8,10$	$5m+1, 5m+2$	$[1,100],[100,200]$
E4	2	10	$[1,20],[20,50],[1,100],[50,100],[100,200],[100,800]$
	3	9	$[1,20],[20,50],[1,100],[50,100],[100,200],[100,800]$
BIG	$25,50,75,100$	$4m$	$[1,1000]$

Table 8.1: A table presenting all configurations of the **BD** benchmark set. “Job Sizes” denotes the range from which jobs sizes are sampled.

8.3 Experiments

In this section, we perform various experiments on the techniques presented in this thesis. We split up the experiments into various categories, depending on what they are meant to test. First, we test the behaviour of the bounding heuristics presented in Chapter 3. Next, we run a set of experiments to show the effectiveness of pruning rules presented in Chapter 5 in different scenarios. Building on that, we test our different SAT encodings against each other. A set of experiments is conducted to see the effect of the techniques presented in Chapter 6. Lastly, we use all techniques discussed in this thesis and present the different run-times.

8.3.1 Evaluation of Bounding Techniques

In this section, we evaluate the accuracy of our bounding techniques on our instances. We evaluate the following bounds. For our lower bounds, we have

[1]`https://site.unibo.it/operations-research/en/research/library-of-codes-and-instances-1`

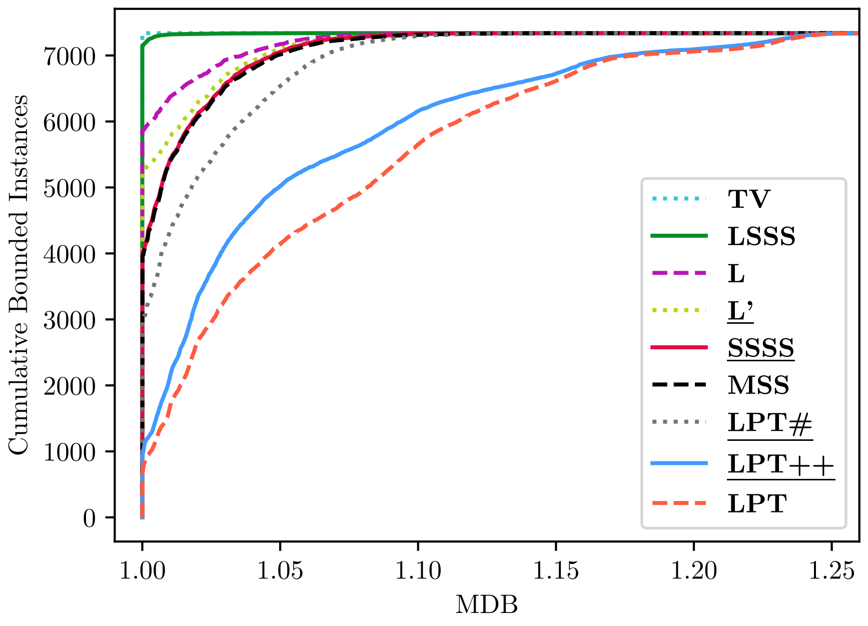


Figure 8.1: The number of cumulative bounded instances relative to the multiplicative distance from a baseline. I.e. the line goes through (x, y) if we can bound y instances with MDB at most x (the baseline here is **TV**). Lower is better for lower bounds, and higher is better for upper bounds. Our algorithms are underlined; bounding algorithms from the literature are not.

- **TV**: the trivial lower bound
- **LSSS**: the lifted **TV**, using the SSS based lifting procedure presented by Haouari et al. [HJ08]
- **L**: The lifting procedure presented by Haouari et al. [HGJ06]
- **L'**: the lifting procedure presented in Section 3.2.1, where during the application of **L** we attempt to solve select instances exactly using **DMFUR** (see next section)

For our upper bounds, we have (with the same names as in Sections 2.9.1 and 3.2.2)

- **LPT**
- **LPT++**
- **LPT#**
- **MSS** with **LPT** providing a warm start
- **SSSS** with the best of **LPT**, **LPT++**, and **LPT#** providing a warm start

We run our bounds on all three benchmark sets, with a timeout of 10 seconds for each instance. In order to present the data, we use **TV** as a baseline and measure the multiplicative distance between **TV** and each bound. We refer to this value as the *multiplicative distance from baseline (MDB)*. The number of cumulative bounded instances relative to MDB is represented in Figure 8.1.

Out of the 7277 instances we are able to solve throughout this thesis, **L** is able to tightly bound 5308 of them while **L'** was able to tightly bound 6260 of them. Meanwhile, from above, the **MSS** heuristic with **LPT** as a warm start is able to tightly bound 4430 instances.

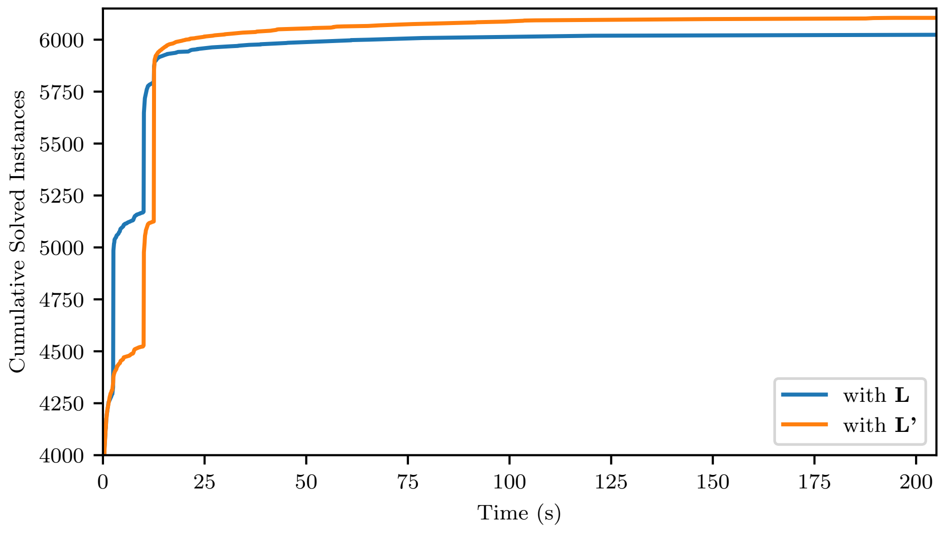


Figure 8.2: The result of **DMFUR** when using **L** versus when using **L'**. Note that the scale starts at 4000.

The **SSSS** heuristic with the best of **LPT**, **LPT++**, and **LPT#** as a warm start, was able to tightly bound 6268 instances.

It is clear that the **L'** heuristic is superior to the **L** heuristic from the literature in terms of accuracy. However, one can reasonably question whether it is beneficial in a practical context. The reason it is practical lies in the exponential nature of exact solving algorithms for $P||C_{max}$. When using **L'** we generate many small sub-instances which we also try to solve in order to bound the original instance. Oftentimes, this bound is tight, meaning that we no longer need to prove the optimality of a given solution for our original instance. This is practically useful since, due to the exponential nature of exact solving algorithms for $P||C_{max}$, exactly solving many small instances is far cheaper than proving the optimality of a solution for one large instance.

To test this, we run **DMFUR** (see next section) on all 7340 instances with a 200-second timeout (including preprocessing time). We do this once with the **L** heuristic, and once with the **L'** heuristic. For lower bounds we use **TV** and **LSSS**, and for upper bounds we use **LPT**, **LPT++**, **LPT#**, and **SSSS**. We give each bound a 10-second time limit, which is part of the allocated 200 seconds. The results are presented in Figure 8.2.

What we see, is that indeed the **L** heuristic is faster than **L'**, causing plenty of instances to take an initial 10-second performance hit. However, since **L'** tightly bounds more instances, we see that the **SSSS** heuristic (which is executed after the **L** or **L'** heuristics) has to run for the full 10 seconds on many more instances when using **L**, which is why the variant with **L'** quickly catches up. By the end of the 200 seconds, the variant with **L** solves 6024 instances, while the variant with **L'** solves 6104 instances (+1.3%), a significant improvement in number of instances solved. Keep in mind that the true difference is blinded by the **BD** instances, which are very easy, causing **L'** to bring no performance benefits. If we disregard the **BD** instances and focus only on the difficult instances, we see an increase in the number of solved instances from 2968 to 3048 (+2.7%). It is clear that the **L'** heuristic brings practical performance benefits, even though plenty of instances need to take a minor performance penalty initially.

8.3.2 Evaluation of Our Pruning Rules

In this section, we evaluate the effect of our pruning rules in different settings.

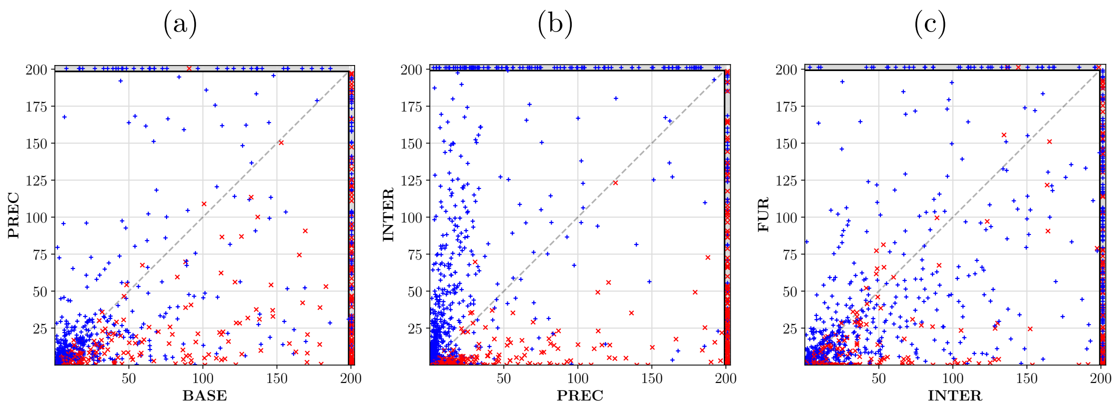


Figure 8.3: Per-instance scatter plot of (a) **BASE** vs. **PREC**, (b) **PREC** vs. **INTER**, and (c) **INTER** vs.**FUR**. A blue '+' represents a satisfiable instance, whereas a red '×' represents an unsatisfiable instance.

SAT Solving

In the first test, we apply the pruning rules to the BDD encoding. Here there are four variants.

- The **BASE** variant is the pure BDD encoding for $P||C_{max}$ as described in Section 4.4.1.
- The **PREC** variant extends the previous encoding with the precedence relations defined by Pruning Rule 5 as presented in Section 5.2.
- The **INTER** variant uses the worse BDD encoding presented in Section 4.4.2, which allows us to encode Pruning Rule 7 as presented in 5.2.
- The **FUR** variant extends the previous variant by also incorporating Rule 9.

We test these instances on 6674 instances of the $P||C_{max}$ decision problem, generated from the **Law** benchmark in the following manner. For each instance in the **Law** benchmark, we use **TV** to bound the instance from below. For a lower bound L, we take the two instances with $U \in \{L, L+1\}$. We then eliminate all instances that can be decided using the simplification rules presented in Section 3.2.3 to get realistic SAT calls.

We run each encoding on each instance with a 200-second timeout for SAT solving. The data is presented in Figure 8.3.

From this data there is a very clear outcome. Our pruning rules add extra conditions on the structure of the solution of the underlying $P||C_{max}$ decision instance. This can make it harder to find a solution if one exists, but almost always makes it easier to prove the lack thereof if one does not.

Integer Linear Programming

In Section 7.3 we discuss how Pruning Rule 1 can be incorporated into the ILP encoding presented by Mrad and Souayah [MS18]. In order to truly see the effect of the pruning rule on the performance of the ILP optimiser, we experiment on singular decision instances of the $P||C_{max}$ problem. Analogous to our experiments on our SAT encodings, we use 6674 instances of the $P||C_{max}$ decision problem generated for the **Law** instances and use a 200-second timeout. In order to test the decision instances, we limit the Gurobi solver to the ILP decision problem.

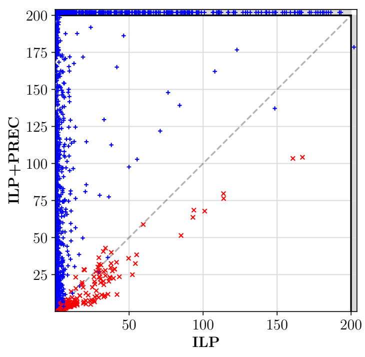


Figure 8.4: Per-instance scatter plot of **ILP** vs. **ILP+PREC**. A blue '+' represents a satisfiable instance, whereas a red '×' represents an unsatisfiable instance.

We refer to our adapted ILP encoding as **ILP**, and **ILP+PREC** refers to the variant where we also incorporate the precedence relations. The results are presented in Figure 8.4.

The results are as surprising as they are drastic. The performance on satisfiable instances takes a dramatic hit, whereas the performance on unsatisfiable instances improves significantly, but to a lesser extent. We see the same trend (but less emphasised) in the SAT encodings, but it seems that the internal heuristics of the Gurobi solver specifically do not adapt well to the shrinking search space.

For reference, **ILP** solves a total of 6624 of the 6674 instances, whereas **ILP+PREC** solves merely 5854.

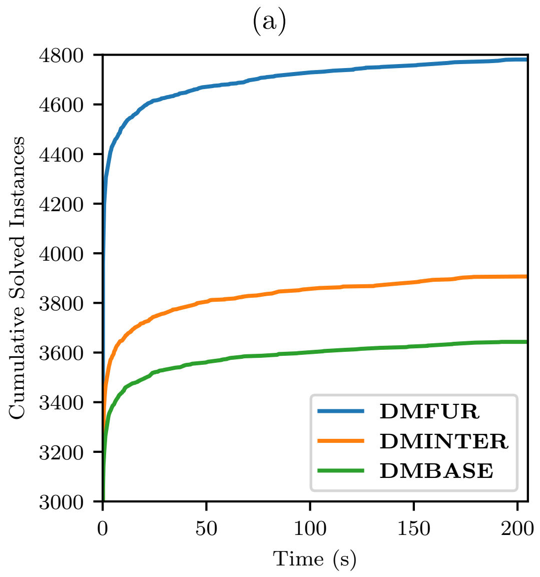
Branch-and-Bound

Next, we test the effectiveness of our rules on the BnB algorithms presented in Sections 7.1 and 7.2. Much like with the SAT instances, we define several variants of our algorithms.

- **DMBASE** is the baseline version of the Dell'Amico Martello algorithm, using Pruning Rules 1, 2, 3, and 4.
- **DMINTER** builds the CRET whenever a new bound is found and replaces Rule 2 with Rule 7.
- **DMFUR** extends **DMINTER** by also integrating Rule 9.
- **HJBASE** is the baseline version of the Haouari and Jemmali algorithm, using Pruning Rules 2 and 5.
- **HJINTER** builds the CRET whenever a new bound is found and replaces Rule 2 with Rule 7.
- **HJFUR** extends **HJINTER** by also integrating Rule 9.

In order to maximise the number of instances that require branching, we use **TV** as a lower bound and **LPT** as an upper bound. We run our algorithms on all three benchmark sets for a total of 7340 $P||C_{max}$ instances. We use a 200-second timeout.

DMBASE was able to solve 3600 instances within 200 seconds. **DMINTER** was able to solve 256 more problems (+7.1%) than **DMBASE** and **DMFUR** was able to solve 1181 more problems (+32.8%) than **DMBASE** (see Fig. 8.5a). Rule 7 thus appears to bring modest improvements, while the FUR results in substantial performance benefits. In line with this observation, the number of explored nodes per problem decreases by a geometric mean factor of 1.71 (max: 3.6×10^6, min: 1) from **DMBASE** to **DMINTER** and 6.89



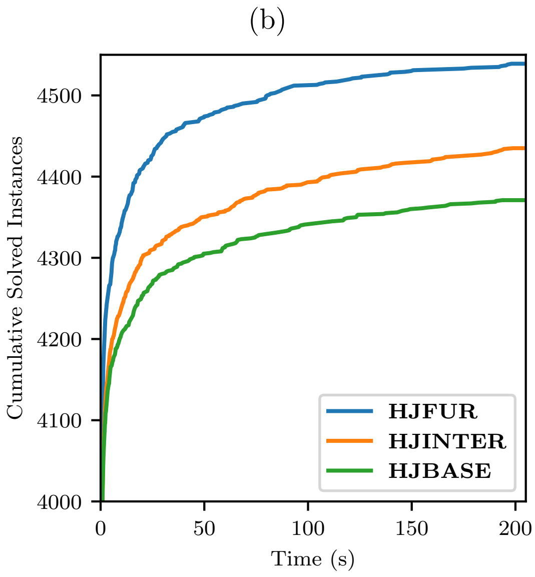


Figure 8.5: Cumulative solved instances relative to running time of (a) DM variants, (b) HJ variants. Note that the y-axis of the two graphs start and end at different values.

(max: 6.2×10^7, min: 0.011) from **DMINTER** to **DMFUR** (8.77 from **DMBASE** to **DMFUR**, max: 6.5×10^7, min: 0.017). The P2 scores of the three variants are 1496000, 1402042, and 1040472 for **DMBASE**, **DMINTER**, and **DMFUR** respectively. The optimal makespans reported by **DMFUR** range from 13 to 1012869 (median 231, mean 6285). Out of the 16872 seconds that **DMFUR** spends on the solved instances, it spends a total of 283 seconds (1.7% of the total time) building the CRET.

For HJ, the results are similar but less pronounced. **HJINTER** was able to solve 64 more problems (+1.46%) than **HJBASE** and **HJFUR** was able to solve 168 more problems (+3.84%) than **HJBASE** (see Fig. 8.5b). The FUR thus appears to bring moderate performance improvements to the HJ algorithm, especially in comparison to its effect on DM. However, this is somewhat to be expected, since the structure of the CRET is better suited to the branching order of DM than to that of HJ.

Comparison to Related Work

In his dissertation, Lawrinenko [Law17] develops a new pruning rule for the $P||C_{max}$ problem that uses a symmetry-breaking criteria on the structure of the solution for an abstract (without job sizes) $P||C_{max}$ problem. Given a partial solution, one can evaluate in $O(mn)$ (amortised) if this condition can still be satisfied and if not, backtrack. In order to test the effectiveness of this pruning rule, he incorporates it into a **DM**-like BnB scheme and tests it against the results presented by Haouari and Jemmali [HJ08]. We compare **DMFUR** against the results presented by Lawrinenko [Law17].

Before we start, we note that the following results are a very rough comparison. First, since we are mainly comparing our algorithm to **WL** presented by Lawrinenko [Law17], we also use the same upper and lower bounds that were used in that dissertation (**TV** as a lower bound and **LPT** as an upper). Haouari and Jemmali [HJ08] use tighter bounds for their computation. Furthermore, the results presented by Haouari and Jemmali [HJ08] were for a Microsoft Visual C++ implementation running on a Pentium IV 3.2 GHz with 1 GB RAM and a time limit of 1200 seconds. The results presented by Lawrinenko [Law17] were for a Java 7.2 implementation running on an Intel Core i7-2600 and 8 GB RAM while running Windows 7 Professional SP 1 (64-bit), with a 900-second time limit. We are running on a 3 GHz ARM processor running Ubuntu for ARM 22.04, and give our algorithm a 200-second timeout. We also do not provide run-time, since over the course of 16 years they would be practically meaningless. To put our results in context with theirs, we compare only the number of solved instances; this should provide some frame of reference as to the behaviour of each heuristic.

The benchmarks used are as follows. For classes 1-5 of the **Law** instances, 10 instances are generated for each constellation of the (n, m)-constellations $(10, 3)$, $(10, 5)$, $(25, 10)$, $(25, 15)$, $(50, 15)$. From the **FF** benchmark 70 instances are selected, 10 of each of the following classes.

- $n = 50$, $m = 25$, jobs sizes uniformly selected from $[1, 10^2]$
- $n = 50$, $m = 25$, jobs sizes uniformly selected from $[1, 10^3]$
- $n = 100$, $m = 25$, jobs sizes uniformly selected from $[1, 10^3]$
- $n = 50$, $m = 10$, jobs sizes uniformly selected from $[1, 10^4]$
- $n = 50$, $m = 25$, jobs sizes uniformly selected from $[1, 10^4]$
- $n = 100$, $m = 25$, jobs sizes uniformly selected from $[1, 10^4]$
- $n = 100$, $m = 25$, jobs sizes non-uniformly selected from $[1, 10^4]$

For the last set of tests, we uniformly select job sizes from $[n/2, n/5]$, and generate instances with $n/m = 2.5$. For each $n \in \{20, 30, 40, 50, 60, 80\}$ 20 instances are generated. In total, there are 350 instances, 230 with $n/m \leq 2.5$ and 120 with $n/m > 2.5$. The results are presented in Table 8.2.

n	m	Class	#Sol. **HJ**	#Sol. **WL**	#Sol. **DMFUR**
		1	10	10	9
50	15	4	7	6	0
		5	8	5	6

NU/U	n	m	Job Sizes	#Sol. **HJ**	#Sol. **WL**	#Sol. **DMFUR**
	50	25	$[1, 10^2]$	8	10	10
	50	25	$[1, 10^3]$	8	10	10
	100	25	$[1, 10^3]$	10	1	0
U	50	10	$[1, 10^4]$	7	0	0
	50	25	$[1, 10^4]$	9	10	10
	100	25	$[1, 10^4]$	0	0	0
NU	100	25	$[1, 10^4]$	7	0	0

n	#Sol. **HJ**	#Sol. **WL**	#Sol. **DMFUR**
30	15	20	20
40	16	20	20
50	12	20	20
60	15	19	16
80	13	10	15

Table 8.2: The number of instances solved in each configuration of each of the three sets, comparing the results presented by Haouari and Jemmalli [HJ08], Lawrinenko [Law17], and **DMFUR**, presented here. Configurations not present in the tables were completely solved by all three algorithms.

In total, **HJ** solves 196 (of 230) instances with $n/m \leq 2.5$, **WL** solves 219, and **DMFUR** solves 221. For instances with $n/m > 2.5$ **HJ** solves 99 (of 120), **WL** solves 72, and **DMFUR** solves 65. However, as mentioned by Lawrinenko, most of the 99 instances with

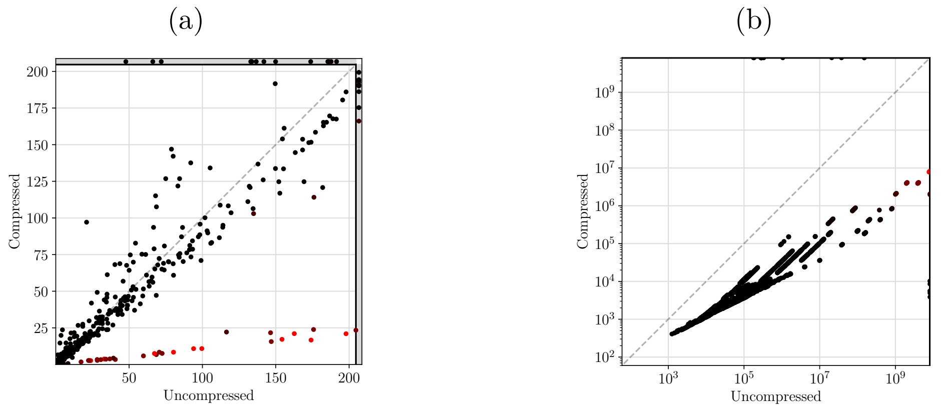


Figure 8.6: Per-instance scatter plot of (a) the performance (in seconds) of **MDFUR** with the CRET vs. the RET and the (b) data usage in bytes. The brightness of each point is proportional to the makespan of the instance.

$n/m > 2.5$ solved by **HJ** were solved at the root node and did not require branching due to their superior bounding techniques. As discussed by Haouari et al., the performance of an exact solving algorithm is more critical on instances with $n/m \approx 2.5$ since other instances can almost always be solved exactly by simple heuristics.

Our performance edges out that of **WL** on instances with low n/m, and achieves better scalability since it can be evaluated much faster at each node. It stands to reason that our pruning techniques are cheaper and yet as effective as those presented by Lawrinenko [Law17], and thus lead to superior performance.

8.3.3 Evaluation of Efficient Computation

In this section, we evaluate the improvements presented in Chapter 6. We compare the performance of the BnB algorithm presented in Section 7.1 with the RET and the CRET. We run them on all 7340 $P||C_{max}$ instances with a timeout of 200 seconds. We also measure the memory usage on the heap. The results are presented in Figure 8.6. For most instances, the performance with the CRET is roughly equivalent to that with the RET. For instances with very large makespans, look-ups over the CRET are faster due to memory locality, causing massive speed-ups. Some instances with small makespan benefit from the usage of the RET over the CRET. However, the memory consumption of the CRET makes it a far superior solution in any scenario where memory is limited. For example, when running these experiments on our server, we were forced to restrict the RET tests to only 10 threads on the **FF** benchmark set.[2]

With the RET, we can solve 4 more instances (+0.08%) than with the CRET. In total, the memory consumption over commonly solved instances for the RET is $137.7GB$, compared to only $0.22GB$ for the CRET.

In our experiments, as previously mentioned, we spend a total of 1.7% of the total computation time constructing the CRET. For our instances, testing the optimisation presented in Section 6.3 would therefore not make much sense. We do maintain, however, that there are likely certain scenarios where this optimisation would have a more observable effect.

[2]Since we implement this algorithm recursively, a lot more memory is used than that of the (C)RET, due to the extremely large stack size, and all the information that is stored in each recursion (e.g. the branch-order). All of this extra waste can be averted via an iterative more memory-conscious implementation, which for us was not necessary.

8.3.4 End-to-End Evaluation

In the last set of experiments, we use all upper and lower bounding techniques, and use a 900-second timeout. We allow 10 seconds for each bound, this time is not included in the results. We test the following techniques

- **ILP**: the adapted Mrad and Souayah encoding (no precedence relations)
- **ILPBASE**: the unedited version of the Mrad and Souayah encoding
- **DMFUR**
- **HJFUR**
- **MULTI**: two complementary SAT encodings are used, and solved concurrently[3]
- **BDDFUR**: the BDD encoding with all optimisations active
- **BINMERGE**: the adapted binmerge encoding introduced in Section 4.3.1, with integrated precedence relations (see Section 5.1.1).
- **ADDER**: the adder encoding introduced in Section 4.2, with integrated precedence relations (see Section 5.1.1).

We test our techniques on all 7340 instances from all three benchmark sets. The results are presented in Figure 8.7.

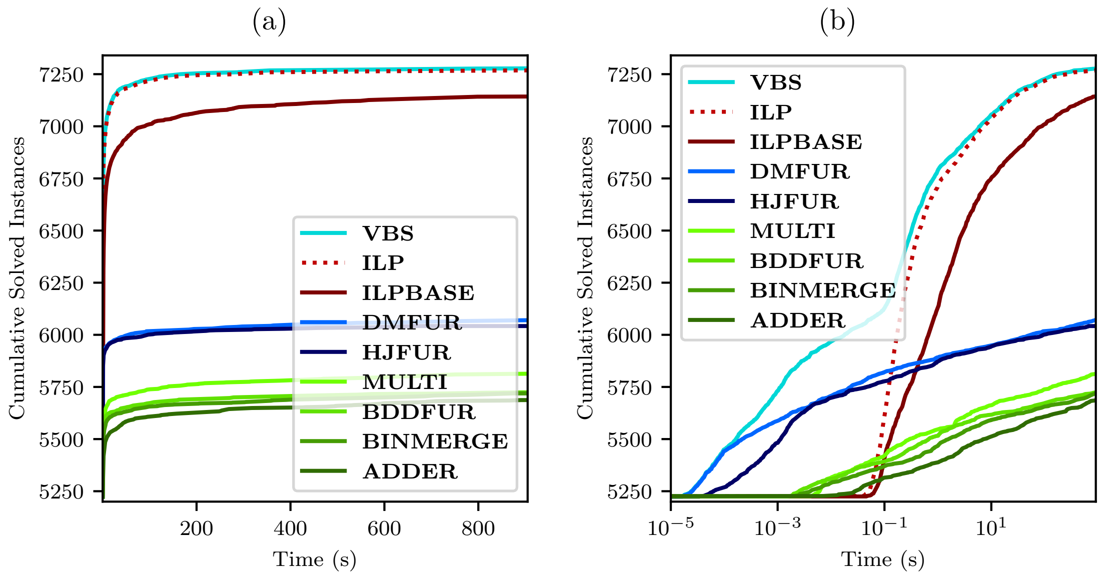


Figure 8.7: The number of solved instances over all presented solving techniques, as well as the virtual best solver. Note that the y-axis starts at 5200, and that the legend is sorted from best to worst.

Our bounds can correctly solve 5222 out of the 7340 instances within the allocated time period. In order from worst to best, our techniques solve 5686, 5718, 5723, 5814, 6041, 6070, 7142, and 7267 instances. The *virtual best solver (**VBS**)* was able to solve 7277 instances; only 10 more than **ILP**. Between our SAT encodings, we see that on the solved instances **BDDFUR** had 334 satisfiable calls and 277 unsatisfiable calls, **BINMERGE** had 331 satisfiable calls and 253 unsatisfiable calls, and **ADDER** had 295 satisfiable calls and 245 unsatisfiable calls.

[3]The best portfolio is achieved by **BDDFUR** and **BINMERGE**.

Looking at Figure 8.7b we see that our BnB algorithms start out faster. This is mostly due to the overhead required to start a new process, and the ingestion period required. Nevertheless, it only takes 0.13 seconds for **ILP** to be more effective. Over all solved instances, **ILP** requires a total of 1.26 GB (0.6 MB per branched and solved instance) of formula size. Meanwhile, the CRET requires 0.0043 GB (0.005 MB per branched and solved instance for **DMFUR**). Keep in mind that in general, for **ILP**, there were numerous instances that required much more data, that could not be solved.

9. Discussion

In this final section, we present some final notes on the $P||C_{max}$ decision problem and the techniques presented in this thesis. A lot of work has been done to speed up $P||C_{max}$ solving. However, as with almost all research, this thesis opens more avenues for future research than it closes. We discuss what we have done to further the art of $P||C_{max}$ solving, and what new avenues can be explored that directly result from our work. The large positive note is that the future of $P||C_{max}$ solving is guaranteed to be rich in advancement.

9.1 The State Space of $P||C_{max}$

Throughout this thesis, we have developed new pruning rules that exploit isomorphisms in the state space of $P||C_{max}$. In this section, we model the state space of $P||C_{max}$. The purpose of this is twofold. First, this helps us understand which instances of $P||C_{max}$ are difficult. Second, this helps us present how our pruning rules can be extended to develop more advanced and complex pruning techniques in the future.

For a $P||C_{max}$ decision instance (W, m, U), the state space can be represented as a multi-set of size m of integers in $[0, U]$, as well as a set $J' \subseteq \{1, \dots, n\}$. The current state is a multi-set of the makespans of each processor and the set of unassigned jobs. Thus, from this perspective, the size of the state space of a $P||C_{max}$ decision instance is in $O(\binom{U+m}{m} \cdot 2^n)$.[1] This is why instances with low n/m values tend to be harder in practice, since m is very large. On the other hand, instances with large U values are also difficult.

9.2 BnB Algorithms for $P||C_{max}$

As we have seen, a "[BnB] procedure can get hopelessly mired by being forced to explore and fathom symmetric reflections of various solutions during the search procedure" [SS01]. Throughout this section, we take the Dell'Amico and Martello branching scheme as an example to present the problem with BnB algorithms for $P||C_{max}$, and how this can potentially be solved in the future. While our pruning rules do help a great deal in avoiding the exploration of equivalent states, there remains a lot of work to be done for the $P||C_{max}$ problem.

[1]The binomial coefficient in this formula is the number of m sized multisets with elements in $[0, U]$. For an intuitive understanding as to where this comes from, consider all the different ways of placing U separators between m marbles, or equivalently, taking $U + m$ spaces and using them to place m marbles before filling the rest of the spaces with separators.

In the Dell'Amico and Martello branching scheme, the state space can be represented as a multi-set of size m of elements in $[0, U]$ and an integer $i \in [0, n]$, since the unassigned jobs are exactly J_i.[2] At each decision node, our pruning rules can tell us whether two branches lead us to equivalent states. This is very practical, since it allows us to only explore one of the two equivalent states. The problem, in general with BnB algorithms, is that after visiting a state and backtracking many times it is possible to visit the same state (or an equivalent state) again.

Take the two partial assignments represented in Figure 9.1 for example. These two partial assignments land us in the same state. Nevertheless, both of these partial assignments can be obtained by **DMFUR** during its search. The indices of the jobs are written on each job, such that the reader may verify this themselves. Here we assume that there remain a very large amount of small unassigned jobs.

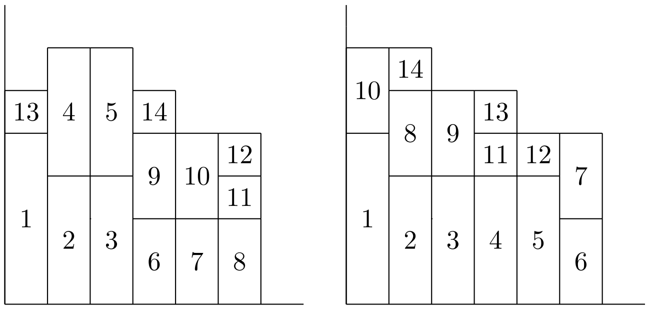


Figure 9.1: A representation of two equivalent $P||C_{max}$ partial assignments that are both explored by **DMFUR**. Each job is represented as a bar of a given size. Each processor is represented as a stack of bars that have been assigned to it. The number on each job represents the job's index.

Based on a backtracking search, the CDCL algorithm also faces this problem. However, after every conflict, CDCL attempts to learn a new clause from the conflict. It does this by recognising the part of the state that is responsible for this conflict and heuristically deriving a clause from it. Due to memory constraints, at some point it is necessary to forget some of these learned clauses, at which point this partial state might be revisited.

Similarly, for a **DMFUR**-like BnB algorithm, one could remember a series of multi-sets (along with their indices) representing states which have been deemed unviable. We can then avoid visiting states which we have previously visited. The RET can also be used here to draw equivalencies between visited states. Instead of remembering the set of makespans of each processor at a state, we can simply remember a multiset of the equivalency range each processor was in. This would allow us to detect and prune states which are equivalent to previously visited states.

From an algorithm engineering perspective, the question still remains how this can be implemented efficiently. There is a large design space for the storing of states for maximal memory efficiency and fast look-up times, as well as forgetting rarely visited states. This has unfortunately been outside the scope of this thesis, but is a very interesting area for future research.

Further Improvements to BnB Algorithms for $P||C_{max}$

Other than trying to avoid visiting the same state multiple times, there are plenty of potential improvements to BnB algorithms that can provide excellent returns. By drawing

[2]In this context the i is redundant and can be inferred from the set. However, if we choose to store the index of the equivalency range each processor is in instead of their makespans, this no longer becomes possible.

inspiration from CDCL based SAT solvers, we see that there are plenty of techniques that can be transferred to $P||C_{max}$ solving. Here we list, in no particular order, some techniques present in SAT solving that can be transferred over to $P||C_{max}$ BnB algorithms. The following techniques are described by Biere et al. [BHvMW21]

First and foremost, SAT solvers use advanced branching heuristics to determine, not only which variable to branch on, but also which value to branch on for the selected variable. In our implementation, we use very simple branching heuristics, and even these lead to massive performance improvements. It is possible that more advanced branching heuristics can lead to far greater returns.

Secondly, SAT solvers use non-chronological backtracking to only backtrack on the parts of the partial-assignment that cause collisions. This is not as simple for $P||C_{max}$, since the implications between variables are not easily separable into distinct neighbourhoods, like they sometimes can be for SAT formulas. Nevertheless, especially for branching orders such as that of Haouari and Jemmali, it is plausible that such backtracking techniques would result in massive speed-ups.

Thirdly, SAT solvers use advanced restart techniques to avoid getting stuck in inconvenient branches. The problem with BnB algorithms, is that if we enter a complex unsolvable branch, while another easily solvable branch remains open, we can get endlessly stuck in the complex branch. Thus, SAT solvers restart at certain intervals, and use complex heuristics to explore new areas of the search space. This also leads to great performance benefits and can be integrated into BnB algorithms for $P||C_{max}$.

Lastly, we are interested in the aspect of variable elimination. During precomputation, SAT solvers detect variables that can be eliminated, and eliminate them from the formula. New clauses are added to ensure that the resulting formula is equivalent to the original. Something similar can be done for the $P||C_{max}$ decision problem. For example, we could eliminate the smallest i jobs via an extensive search, and store the set of states at level $n - i$ which are unviable. This could lead to dramatic performance improvements, since we now guarantee that no state in the bottom i levels is explored twice.

9.3 The DIMM Heuristic and ILP Approaches

Beyond BnB algorithms, there are other promising areas of future research that could directly benefit from the results we present here. The DIMM [DIMM08] heuristic is based on an ILP encoding of the $P||C_{max}$ decision problem and a specialised BnB approach for solving these ILP formulas. As we have seen, the reduction of the search space via the incorporation of our pruning rules into a BnB algorithm or ILP formula can be greatly beneficial. It is therefore possible to integrate our pruning rules into either the ILP formula, or the BnB algorithm of the DIMM heuristic to obtain greater performance benefits. We would also be interested in finding an ILP encoding in which our pruning rules can be better incorporated.

The size of the design space for this has unfortunately made it outside the scope of this thesis. However, successfully doing so would most likely lead to performance improvements and a successful advancement of the state-of-the-art.

9.4 Conclusion

In this thesis, we have investigated how SAT solving can be used to solve the $P||C_{max}$ problem.

We begin by investigating how upper and lower bounding techniques from the literature can be improved upon and successfully develop a suite of heuristics which together can solve 71.14% of our instances, and tightly bound 85.29% from below, and 85.39% from above. Improvements over the literature are measurable and significant.

Next, we observe that our pruning rules are effective at limiting the search space of different types of search algorithms. We detect some similarity across the different solving schemes. The incorporation of our pruning rules decreases the size of the search space that any exact algorithm would need to explore. On the one hand, this makes it easier for a solver to prove the optimality of a given solution. On the other hand, if a solution is not optimal, it makes it harder for the solver to find a better solution. Since our lower and upper bounding techniques are both equally accurate, it is likely that the optimal technique is simply a portfolio; one algorithm that attempts to find a solution, and another that attempts to prove the optimality of the existing solution. We see that, at least in the case of SAT solving, this does indeed provide the best performance.

Furthermore, we improve upon the state-of-the-art of BnB algorithms for $P||C_{max}$. While this does still fall short of generic solvers, this is somewhat to be expected. A dedicated algorithm still requires a large amount of work, but the techniques presented here can bring both exact algorithms and generic translations one step closer to efficiently solving $P||C_{max}$.

Bibliography

[AA20] N. Abd-Alsabour. The Subset-Sum Problem as an Optimization Problem. In Suresh Chandra Satapathy, Vikrant Bhateja, J. R. Mohanty, and Siba K. Udgata, editors, *Smart Intelligent Computing and Applications*, pages 693–700, Singapore, 2020. Springer Singapore.

[ANORC09] R. Achá, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 167–180, 06 2009.

[ANORC11] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. BDDs for Pseudo-Boolean Constraints – Revisited. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 61–75, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[ANS04] F. Antonio, E. Necciari, and M. Scutellà. A Multi-Exchange Neighborhood for Minimum Makespan Machine Scheduling Problems. *J Combinat Optimizat*, 01 2004.

[BCS74] J. Bruno, E. G. Coffman, and R. Sethi. Scheduling Independent Tasks to Reduce Mean Finishing Time. *Commun. ACM*, 17(7):382–387, jul 1974.

[BCSV20] M. Bofill, J. Coll, J. Suy, and M. Villaret. SMT encodings for Resource-Constrained Project Scheduling Problems. *Computers & Industrial Engineering*, 149:106777, 2020.

[BDJR22] S. Berndt, M. Deppert, K. Jansen, and L. Rohwedder. *Load Balancing: The Long Road from Theory to Practice*, pages 104–116. 01 2022.

[Beh07] M. Behle. On Threshold BDDs and the Optimal Variable Ordering Problem. In Andreas Dress, Yinfeng Xu, and Binhai Zhu, editors, *Combinatorial Optimization and Applications*, pages 124–135, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Ber05] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume I. Athena Scientific, Belmont, MA, USA, 3rd edition, 2005.

[BFFH20] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[BHvMW21] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 2021.

[Bie23] A. Biere. Kissat. `https://github.com/arminbiere/kissat`, 2023.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BS97] R. J. Bayardo and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 203–208. AAAI Press, 1997.

[CCB22] M. P. Castro, A. A. Cire, and J. C. Beck. Decision Diagrams for Discrete Optimization: A Survey of Recent Advances. *INFORMS J. on Computing*, 34(4):2271–2295, jul 2022.

[CKE08] V. Christian, J. Kaspar, and M. Eichberger. Towards Solving a System of Pseudo Boolean Constraints with Binary Decision Diagrams. 2008.

[Cpl09] IBM ILOG Cplex. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[DG84] W. F. Dowling and J. H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[DIMM08] M. Dell'Amico, M. Iori, S. Martello, and M. Monaci. Heuristic and Exact Algorithms for the Identical Parallel Machine Scheduling Problem. *INFORMS Journal on Computing*, 20:333–344, 08 2008.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7):394–397, jul 1962.

[DM95] M. Dell'Amico and S. Martello. Optimal Scheduling of Tasks on Identical Parallel Processors. *INFORMS Journal on Computing*, 7:191–200, 05 1995.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, jul 1960.

[DSV93] W. Domschke, A. Scholl, and S. Voss. *Produktionsplanung: Ablauforganisatorische Aspekte*. Springer Verlag, 1993.

[ES06] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2(1-4):1–26, 2006.

[FGLM94] P. M. França, M. Gendreau, G. Laporte, and F. M. Müller. A Composite Heuristic for the Identical Parallel Machine Scheduling Problem with Minimum Makespan Objective. *Computers & Operations Research*, 21(2):205–210, 1994.

[FS01] S. P. Fekete and J. Schepers. New Classes of Fast Lower Bounds for Bin Packing Problems. *Math. Program.*, 91(1):11–31, 2001.

[GJ78] M.R. Garey and D.S. Johnson. Strong NP-completeness Results. *J. ACM*, 25(3):499–508, 1978.

[GLLK79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and AHG R. Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. In *Annals of discrete mathematics*, volume 5, pages 287–326. 1979.

[Gra66] R. L. Graham. Bounds for Certain Multiprocessing Anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966.

[Gra69] Ronald L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[GRT01] J. N. D. Gupta and A. J. Ruiz-Torres. A LISTFIT Heuristic for Minimizing Makespan on Identical Parallel Machines. *Production Planning & Control*, 12(1):28–36, 2001.

[Gur23] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[Heb12] E. Hebrard. Scheduling and SAT. `https://web.imt-atlantique.fr/x-info/cpaior-2012/uploads/slides%20master%20class/5-scheduling-Hebrard.pdf`, 2012.

[HG04] M. Haouari and A. Gharbi. Lower Bounds for Scheduling on Identical Parallel Machines with Heads and Tails: Models and Algorithms for Planning and Scheduling Problems (Editors: P. Baptiste, J. Carlier, A. Munier, A. S. Schulz). *Annals of Operations Research*, 129, 01 2004.

[HGJ06] M. Haouari, A. Gharbi, and M. Jemmali. Tight Bounds for the Identical Parallel Machine-Scheduling Problem. *International Transactions in Operational Research*, 13:529 – 548, 10 2006.

[HJ08] M. Haouari and M. Jemmali. Tight Bounds for the Identical Parallel Machine-Scheduling Problem: Part II. *International Transactions in Operational Research*, 15(1):19–34, 2008.

[KK07] W. Klieber and G. Kwon. Efficient CNF Encoding for Selecting 1 from N Objects. 2007.

[Law17] A. Lawrinenko. *Identical Parallel Machine Scheduling Problems: Structural Patterns, Bounding Techniques and Solution Procedures.* PhD thesis, Friedrich-Schiller-Universität Jena, Jena, 2017. Dissertation, Friedrich-Schiller-Universität Jena, 2017.

[LLRS93] E. L. Lawler, J K. Lenstra, A. H.G. Rinnooy Kan, and D. B. Shmoys. Chapter 9 Sequencing and Scheduling: Algorithms and Complexity. In *Logistics of Production and Inventory*, volume 4 of *Handbooks in Operations Research and Management Science*, pages 445–522. Elsevier, 1993.

[McN59] R. McNaughton. Scheduling with Deadlines and Loss Functions. *Manage. Sci.*, 6(1):1–12, oct 1959.

[MKHT20] F. Molnár, S. R. Kharel, X. Sharon Hu, and Z. Toroczkai. Accelerating a Continuous-Time Analog SAT Solver Using GPUs. *Computer Physics Communications*, 256:107469, 2020.

[MMZ$^+$01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001.

[Mok04] E. Mokotoff. An Exact Algorithm for the Identical Parallel Machine Scheduling Problem. *European Journal of Operational Research*, 152(3):758–769, 2004. Applications of Soft O.R. Methods.

[MPS14] N. Manthey, T. Philipp, and P. Steinke. A More Compact Translation of Pseudo-Boolean Constraints into CNF Such That Generalized Arc Consistency Is Maintained. In Carsten Lutz and Michael Thielscher, editors, *KI 2014: Advances in Artificial Intelligence*, pages 123–134, Cham, 2014. Springer International Publishing.

[MS18] M. Mrad and N. Souayah. An Arc-Flow Model for the Makespan Minimization Problem on Identical Parallel Machines. *IEEE Access*, 6:5300–5307, 2018.

[MSS03] J. P. Marques-Silva and K. A. Sakallah. *Grasp—A New Search Algorithm for Satisfiability*, pages 73–89. Springer US, Boston, MA, 2003.

[MT90] S. Martello and P. Toth. Lower Bounds and Reduction Procedures for The Bin Packing Problem. *Discrete Applied Mathematics*, 28(1):59–70, 1990.

[Pfa10] B. Pfahringer. *Conjunctive Normal Form*, pages 209–210. Springer US, Boston, MA, 2010.

[PG86] D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *J. Symb. Comput.*, 2:293–304, 1986.

[Pis99] D. Pisinger. Linear Time Algorithms for Knapsack Problems with Bounded Weights. *Journal of Algorithms*, 33(1):1–14, 1999.

[PS15] T. Philipp and P. Steinke. PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 9–16, Cham, 2015. Springer International Publishing.

[Rot66] M. H. Rothkopf. Scheduling Independent Tasks on Parallel Processors. *Management Science*, 12(5):437–447, 1966.

[RvBW06] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience, 1986.

[SS01] H. D. Sherali and J. C. Smith. Improving Discrete Model Representations via Symmetry Considerations. *Management Science*, 47(10):1396–1407, 2001.

[SS21] D. Schreiber and P. Sanders. Scalable SAT Solving in the Cloud. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 518–534, Cham, 2021. Springer International Publishing.

[TGP$^+$09] S. K. Tiwary, A. Gupta, J. R. Phillips, C. Pinello, and R. Zlatanovici. First Steps Towards SAT-Based Formal Analog Verification. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, page 1–8, New York, NY, USA, 2009. Association for Computing Machinery.

[Tse83] G. S Tseitin. On the Complexity of Derivation in Propositional Calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

[YSV$^+$18] X. Yin, B. Sedighi, M. Varga, M. Ercsey-Ravasz, Z. Toroczkai, and X. S. Hu. Efficient Analog Circuits for Boolean Satisfiability. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(1):155–167, 2018.