# Tractability and Approximability for Subclasses of the Makespan Problem on Unrelated Parallel Machines

by

Daniel R. Page

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2014

Thesis advisor

**Ben Pak-Ching Li**

Author

**Daniel R. Page**

**Tractability and Approximability for Subclasses of the Makespan Problem on Unrelated Parallel Machines**

# Abstract

Let there be $m$ parallel machines and $n$ jobs to be scheduled non-preemptively. A job $j$ scheduled on machine $i$ takes $p_{i,j}$ time units to complete, where $1 \leq i \leq m$ and $1 \leq j \leq n$. For a given schedule, the makespan is the completion time of a machine that finishes last. The goal is to produce a schedule of all $n$ jobs with minimum makespan. This is known as the makespan problem on unrelated parallel machines (UPMs), denoted as $R||C_{max}$. In this thesis, we focus on subclasses of $R||C_{max}$. Our research consists of two components. First, a survey of theoretic results for $R||C_{max}$ with a focus on approximation algorithms is presented. Second, we present exact polynomial-time algorithms and approximation algorithms for some subclasses of $R||C_{max}$. For instance, we present $k$-approximation algorithms on par with or better than the best known for certain subclasses of $R||C_{max}$.

# Contents

# List of Figures

# List of Tables

# Acknowledgments

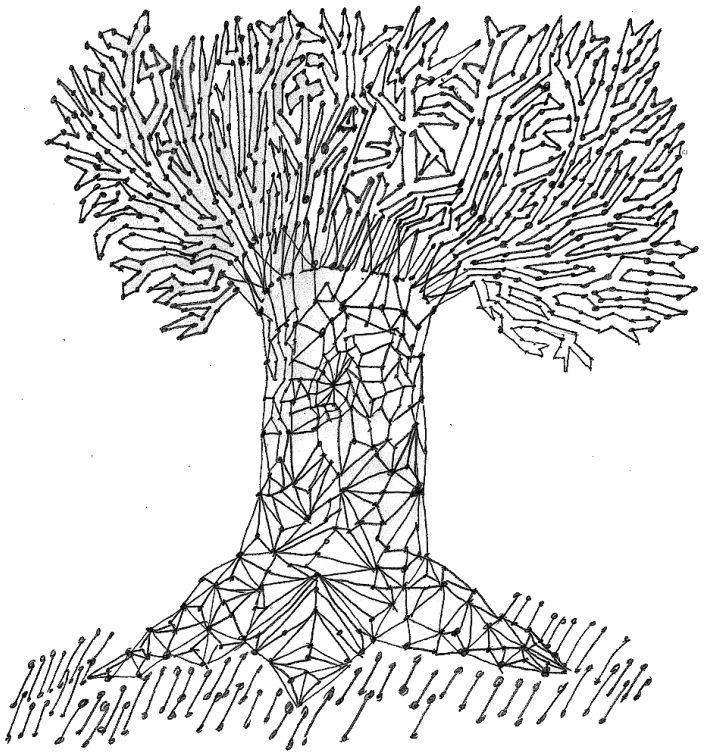
First, I would like to thank Dr. Ben Li for financially supporting my research, being a valuable supervisor to my work, and for the time spent on this research endeavour. I wanted a challenge to improve my research, and in the end I felt that was met while studying under you. In the overall development of this thesis, I want to thank my committee and my advisor for all their feedback.

Next, I must thank the Department of Computer Science as a whole for their support, and the opportunity to teach in the department. Also, with the *Special Department of Computer Science Award of Excellence* awarded by the Department of Computer Science that was made possible by the funds of the department and my advisor, I could plan towards my research and future goals. Some work-related parties I would like to acknowledge are the Department of Computer Science, and the Western Canada Aviation Museum. First, I want to thank all my co-workers at the museum for all the good times we have had. I also want to thank the faculty and fellows in the Department of Computer Science. In particular, it was a pleasure working with the teaching staff in the Department of Computer Science. I personally want to thank some of my friends in the Department, especially Dr. John van Rees and Dr. Parimala Thulasiraman to name a couple. I would like to thank every single student I have taught all these years.

Finally, I would like to mention my thanks to friends and family. There are quite a number of names I would mention but it would not fit in the margins of this page. I want to thank my family, especially my mother for understanding my goals and supporting my dreams, even if it is a bit hard to see in the short-term. Thankyou to my loved ones and closest friends, especially Charén for being there for me.

*To my Father, Michael William Page (1956–2010).*

*One of my heroes, and the most pragmatic person I've ever known.*



# Chapter 1

# Introduction

## 1.1 Motivation

Imagine that you are a production planner for a textile manufacturing plant. Your responsibility is to determine a schedule for each order within the upcoming week for manufacturing various fabrics. The manufacturing plant has three multipurpose weaving machines that independently operate in parallel. Independent of each machine, textile batches (jobs) of an order can complete at fluctuating times depending on various characteristics of input raw materials. Each job is to be scheduled on one machine, and can be assigned in any order. At initialization, a technician schedules the weaving machines to commence simultaneously. A machine continues to operate without interruption for each job until all the jobs assigned to that particular machine complete. Assuming this plant operates all hours of the day, your objective is to schedule every job of an order, such that the machines finish the order as early as possible. Based on previous experience of how the machines operate, you have

prepared estimates rounded to the next highest hour in Table 1.1.

| **Machine \ Jobs** | $\textbf{Cotton}_1$ | $\textbf{Cotton}_2$ | $\textbf{Silk}_1$ | $\textbf{Silk}_2$ | $\textbf{Wool}_1$ | $\textbf{Wool}_2$ |
|---|---|---|---|---|---|---|
| Machine 1 | 5 | 6 | 5 | 6 | 5 | 6 |
| Machine 2 | 5 | 7 | 8 | 7 | 8 | 8 |
| Machine 3 | 5 | 6 | 5 | 7 | 8 | 6 |

Table 1.1: Estimated times for a potential assignment of jobs prepared by the production planner rounded to the next highest hour. Jobs are uniquely identified by their material, and a subscript.

Your goal this week is to schedule all six jobs of this order on the weaving machines, so that the processing time of the last machine to finish is minimized. Suppose your supervisor gave you a deadline of twelve hours to have all the batches manufactured, because the machines need maintenance for the remainder of the week. Based on your estimates, is it possible to produce a schedule of all the jobs that completes within the deadline? Also, what is the minimum completion time of a machine that finishes last for this instance? A schedule that completes in twelve hours exists (see page 7), so the deadline given by your supervisor can be met.

Our goal is to find a schedule where the completion time of a machine that finishes last is minimized. This is an example of a problem instance of the makespan problem on unrelated parallel machines (UPMs), which we define in Section 1.3. This problem is called the makespan problem on UPMs, because the goal is to schedule all given independent jobs on unrelated machines while minimizing the completion time of a machine that finishes last—the *makespan*. This scheduling problem arises in numerous areas including multiprocessor computer scheduling, and is a core problem of study in combinatorial optimization. The makespan problem on UPMs is an example of an *optimization problem*.

## 1.2 Optimization Problems

The makespan problem on UPMs is an optimization problem. An optimization problem [28, Section 15.7] consists of three parts:

1. A non-empty set $I$ of *problem instances*;
2. An *objective function* $c(x, y)$ that yields a rational value called its *objective value*, where $y$ is a feasible solution to a given instance $x \in I$; and
3. A *goal* of either minimization (min) or maximization (max) for the objective function.

Each instance $x \in I$ is associated with an ordered pair $(S_x, OPT(x))$. $S_x$ is a set of feasible solutions of $x$, and $OPT(x)$ is the *optimal objective value* for a feasible solution of $x$, where the optimal objective value depends on the goal of the problem (i.e., min, or max). Given an instance $x$ of a given optimization problem, the goal is to find a feasible solution $y'$, where $c(x, y') = OPT(x)$, called an *optimal solution*. For instance, the makespan problem is a minimization problem where the goal is to find a feasible schedule whose makespan (objective value) is of minimum size.

In combinatorial optimization and theoretical computer science, many problems studied are optimization problems. The research in these areas often fall into one of three major categories:

1. Given an optimization problem, what is its computational complexity? For example, can any instance of the optimization problem be solved in polynomial time with respect to input size (i.e., polynomial-time solvable), or is the optimization problem NP-hard? Some useful computational complexity theory

references with sections on optimization problems are Cormen *et al.* [5], Korte and Vygen [28], and Papadimitriou and Steiglitz [37].

2. Is the optimization problem polynomial-time solvable? That is, is there an algorithm that solves any instance of the optimization problem in polynomial-time with respect to input size? Typically, researchers show an optimization problem is polynomial-time solvable by designing an algorithm, showing its worst-case time complexity is polynomial with respect to input size, and then proving the correctness of the algorithm. If an optimization problem has a polynomial-time algorithm, then there is no need to consider case 3.

3. Given an NP-hard optimization problem, can we design approximation algorithms for the optimization problem that yield feasible solutions in polynomial time that are guaranteed to be within a given factor of the optimal objective value? We discuss approximation algorithms and hardness of approximation in Section 1.5. As a consequence of hardness of approximation and the difficulty in designing better approximation algorithms for certain optimization problems, some researchers focus on subclasses of optimization problems. A *subclass* $(I', c', goal')$ of an optimization problem $(I, c, goal)$ is a set of sub-instances $I' \subseteq I$, with objective function $c' = c$, and $goal' = goal$. In this thesis, we consider subclasses of the makespan problem on UPMs.

## 1.3 The Makespan Problem on UPMs

We define a general scheduling problem called the *makespan problem on unrelated parallel machines* (UPMs), which is the major focus of this thesis. Let $p_{i,j} \in \mathbb{Z}^+$ be the processing time for job $j$ on machine $i$. Call $m \times n$ matrix $P = (p_{i,j})$ a *processing requirement matrix*. Given a $m \times n$ processing requirement matrix $P$, $n$ jobs, and $m$ machines, the goal is to find a non-preemptive schedule of the jobs that minimizes the completion time of a machine that finishes last. In non-preemptive job scheduling, if a job is scheduled on a machine, then the job must complete without being interrupted. The sum of job lengths of a machine that completes last is called the *makespan* of the schedule [46, Section 2.3]. As a note, for a machine, its *completion time* or *load* is the sum of job lengths to be processed by the machine. The makespan problem on unrelated parallel machines is NP-hard, because the makespan problem on identical parallel machines when $m = 2$ is NP-hard [14].

An instance of the makespan problem on UPMs is given by a 3-tuple $(P, m, n)$ where $P$ is the processing requirement matrix, $m$ is the number of machines, and $n$ is the number of jobs of the instance. Given an instance $(P, m, n)$ of the makespan problem on UPMs, a *feasible solution* is a schedule that contains all $n$ jobs, where each job is scheduled on exactly one machine. We call such solutions *feasible schedules*. Also, we assume that feasible schedules do not contain idle time between any two consecutive jobs scheduled on a machine. The assignment of jobs to machines for an instance forms a schedule. This assignment of jobs can be represented as an assignment matrix $X = (x_{i,j})$, where $x_{i,j} = 1$ if job $j$ is assigned to machine $i$, and $x_{i,j} = 0$ otherwise. We interchangeably represent solutions as either a schedule, or

an assignment matrix that yields a schedule. The *objective value* of a solution is the makespan of a schedule. More precisely, given an assignment matrix $X$, the makespan of a schedule is $c((P, m, n), X) = max_{1 \leq i \leq m}\{\sum_{j=1}^{n} p_{i,j}x_{i,j}\}$. The completion time of a job that finishes last in a schedule is the same as the completion time of a machine that finishes last. So we denote the makespan of a schedule as $C_{max}$, the completion time of the job that finishes last in a schedule. An optimal solution or *optimal schedule* is a feasible schedule for which the makespan is smallest. The minimum makespan for a given instance of the makespan problem on UPMs is $OPT((P, m, n))$. For simplicity, we denote the optimal makespan of an instance instead as $OPT(P)$.

Let us consider the example we presented in Section 1.1 more formally now. The input instance is a $3 \times 6$ processing requirement matrix $P$ with $m = 3$ machines, and $n = 6$ jobs. This matrix represents the durations of possible processing times for six jobs with three machines prepared by the production planner. The goal is to produce an optimal schedule, a feasible schedule where the makespan is minimized. The production planner's table given as Table 1.1 as a processing requirement matrix is

$$P = \begin{pmatrix} 5 & 6 & 5 & 6 & 5 & 6 \\ 5 & 7 & 8 & 7 & 8 & 8 \\ 5 & 6 & 5 & 7 & 8 & 6 \end{pmatrix}.$$

To help understand the concepts introduced, let us consider a feasible schedule (that may not be optimal). Suppose we assign jobs 1, 2 and 3 to machine 1; job 6 to machine 2; and jobs 4 and 5 to machine 3. The schedule that is the result of our assignments is a feasible schedule. Figure 1.1 shows this schedule. Notice that the

makespan of this schedule is sixteen.

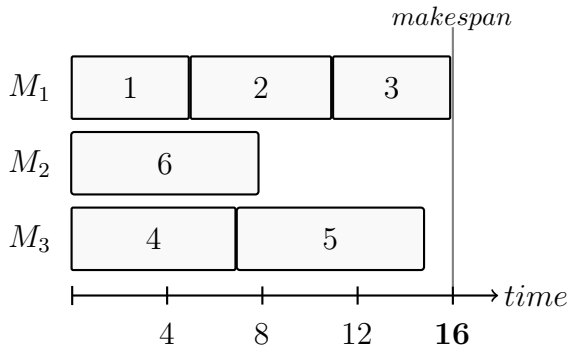



Figure 1.1: A feasible schedule for an example instance of the makespan problem on UPMs. The makespan of this schedule is sixteen.

In Section 1.1, we remarked that the production planner could meet the deadline proposed by the supervisor, and that a schedule with makespan of twelve exists. Let $X^*$ be an optimal assignment that represents an optimal schedule. Then, an optimal assignment $X^*$ is

$$X^* = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

As a result, $c((P, m, n), X^*) = OPT(P) = 12$. Figure 1.2 is the schedule produced by this assignment. Though our instance has only one schedule with optimal makespan, most instances have many different optimal schedules.

We focus on subclasses of the makespan problem on UPMs to develop theoretic results and algorithms in this thesis. By investigating subclasses of the makespan problem on UPMs, we perform two fundamental tasks. First, we search for new properties that cover many instances of the general problem. Second, we develop efficient algorithms or find improved approximation algorithms for subclasses which

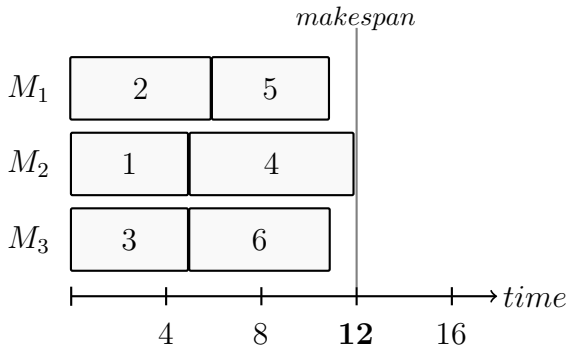



Figure 1.2: An optimal schedule for our example instance of the makespan problem on UPMs. The makespan of this schedule is twelve.

are not possible for the general makespan problem on UPMs.

## 1.4 Notation for Scheduling Problems

In this thesis, we adopt notation for describing many theoretic scheduling problems called the *Graham notation*, introduced by Graham *et al.* [17]. The notation uses three fields, $\alpha|\beta|\gamma$, where $\alpha$ specifies a *machine environment*, $\beta$ describes *job characteristics*, and $\gamma$ denotes an *optimality criterion*. We now describe in more detail each field based only on the scheduling problems we consider in this thesis.

The machine environment $\alpha =$"$\alpha_1\alpha_2$" describes the type of machines $\alpha_1$ for scheduling jobs, and a fixed number of machines $\alpha_2 \in \mathbb{Z}^+$. In Table 1.2, we define each machine type. If $\alpha_1$ is not provided, then it is assumed that $\alpha_2 = 1$. If $\alpha_2$ is not provided, we assume the number of machines $m$ is *variable* and though it is not written, $\alpha_2 = m$. If $\alpha_2$ is given as positive integer, it is assumed that the number of machines is *fixed*. Note that each machine environment presented is a special case of UPMs.

| $\alpha_1$ | **Machine Type** | **Definition** |
|---|---|---|
| | Single Machine | $1 \times n$ processing requirement matrix, where $p_{1,j} = p_j \in \mathbb{Z}^+$ and $1 \leq j \leq n$. |
| $P$ | Identical Parallel Machines | $\alpha_2 \times n$ processing requirement matrix, where $p_{i,j} = p_j \in \mathbb{Z}^+$, $1 \leq j \leq n$, and $1 \leq i \leq \alpha_2$. |
| $Q$ | Uniformly Related Parallel Machines | For $1 \leq i \leq \alpha_2$, let each machine $i$ have a speed $s_i \in \mathbb{Z}^+$. Then there is a $\alpha_2 \times n$ processing requirement matrix, where $p_{i,j} = s_i p_j \in \mathbb{Z}^+$, and $1 \leq j \leq n$. Each machine $i$ has a speedup factor of $s_i$. When each $s_i = 1$, the machines become identical parallel machines. |
| $R$ | Unrelated Parallel Machines | $\alpha_2 \times n$ processing requirement matrix, where $p_{i,j} \in \mathbb{Z}^+$, $1 \leq i \leq \alpha_2$, and $1 \leq j \leq n$. |

Table 1.2: Some machine environments under Graham notation along with their definitions.

The job characteristics further specify the scheduling problem, and what entries may be permitted in a processing requirement matrix. We organize job characteristics by four major types in Table 1.4. These four types are preemption ($B_1$), partial order ($B_2$), release dates ($B_3$), and processing times ($B_4$). Let $\beta \subseteq \{\beta_1, \beta_2, \beta_3, \beta_4\}$, where $\beta_1 \in B_1$, $\beta_2 \in B_2$, $\beta_3 \in B_3$, and $\beta_4 \in B_4$ are described in Table 1.4. Job characteristics are written as "$\beta_1, \beta_2, \beta_3, \beta_4$", and describe properties jobs may have beyond the machine environment. For $B_4$ under restricted processing times, the symbol $\infty$ denotes that a job cannot be assigned to a particular machine. In this context, $\infty$ represents an arbitrarily large positive integer, and when $p_{i,j} = \infty$, job $j$ takes too long to complete on machine $i$. Let $A$ be a set of positive integers. If $\infty \in A$, and $p_{i,j} = \infty$, then job $j$ cannot be scheduled to machine $i$. We call instances with $\infty \in A$, *restricted assignment* instances. If no job characteristics are provided, then the initial machine description is used without modification, jobs are non-preemptive,

with no precedence constraints, and job processing times are not restricted.

At last, we consider the third field, the optimality criterion $\gamma$. The optimality criterion is the property we seek to minimize in the scheduling problem. For example, we have discussed one such property, the makespan $C_{max}$ of a schedule. In Table 1.3, we give a few examples of optimality criteria explored in the literature.

| $\gamma$ | **Optimality Criterion** | **Associated Problem** |
|---|---|---|
| $C_{max}$ | Makespan/Maximum Completion Time | Find a schedule that minimizes the completion time of a machine that finishes last. |
| $L_{max}$ | Maximum Lateness | For $1 \leq j \leq n$, each job $j$ is assigned a deadline $d_j$ for an ideal completion time of a job. Lateness of a job is computed as $L_j = C_j - d_j$, where $C_j$ is the completion time of job $j$ in a schedule. Produce a schedule that minimizes the maximum lateness of a job. |
| $\sum_j w_j C_j$ | Weighted Completion Times | Each job $j$ has an assigned weight $w_j$, where $1 \leq j \leq n$. Find a schedule that minimizes the sum of weighted completion times. |

Table 1.3: Some examples of optimality criterion under Graham notation for scheduling problems.

| **Characteristic Type** | **Job Characteristic** | **Definition** |
|---|---|---|
| $B_1$ - Preemption | *pmtn* (Preemptive Jobs) | Jobs being processed can be interrupted and resumed later. |
| $B_2$ - Partial Order | *prec* (Precedence Constraints) | Let there be a directed acyclic graph $G = (V, E)$, where $V = \{1, \ldots, n\}$. Given a directed path from vertex $u$ to vertex $v$, define $u \prec v$. If $u \prec v$, then job $v$ cannot be processed until job $u$ completes. |
| | *tree* (Partial Order as Tree) | Special case of *prec*, where $G$ is a tree. |
| | *chains* (Partial Order as Chains) | Special case of *prec* when $G$ is comprised of chains: paths consisting of distinct vertices. |
| | $forest$ (Partial Order as Forest) | Special case of *prec* when $G$ is a forest: each connected component is a tree. |
| $B_3$ - Release Dates | $r_j$ (Release Dates for Jobs) | For $1 \leq j \leq n$, each job $j$ is assigned a release date $r_j$, where $r_j \geq 0$. Job $j$ cannot be schedule to a machine until $r_j$ time units have elapsed. |
| $B_4$ - Processing Times | $p_{i,j} = 1$ (Unit Times) | Given a processing requirement matrix, each entry $p_{i,j} \in \{1\}$. |
| | $l \leq p_{i,j} \leq u$ (Bounded Times) | Entries $p_{i,j}$ of a processing requirement matrix are bounded by some constant lower and upper bounds $l$ and $u$, respectively. |
| | $p_{i,j} \in A$ (Restricted Times) | Each entry $p_{i,j}$ of a processing requirement matrix is a member of a set $A$ of positive integers. |

Table 1.4: Examples of commonly-used job characteristics in Graham notation along with their definitions.

Here are some examples of the Graham notation being used to represent scheduling problems:

- $P2|prec|C_{max}$ denotes the makespan problem on two identical parallel machines with precedence constraints enforced for each job,
- $1||\sum_j w_j C_j$ represents minimizing the weighted completion times on a single machine,
- $Q|pmtn, tree|C_{max}$ denotes the makespan problem on uniformly related parallel machines with a partial order in the form of a tree, where jobs are scheduled preemptively.

The makespan problem on UPMs is represented by $R||C_{max}$. Such notation can be used also to describe subclasses of the makespan problem on UPMs. For instance, $R|p_{i,j} \in \{1, 2\}|C_{max}$ denotes the makespan problem on UPMs when processing times $p_{i,j} \in \{1, 2\}$.

## 1.5 Approximation Algorithms

Many optimization problems such as the makespan problem for UPMs are NP-hard problems. NP-hard optimization problems are not thought to be polynomial-time solvable, unless P = NP [28, Chapter 15.7]. Researchers are primarily interested in using approximation algorithms to find approximate solutions for these types of optimization problems.

An *approximation algorithm* for an optimization problem is a step-by-step finite procedure that returns in polynomial time, for all instances to a given problem, an

approximate solution. See Chapter 1 of Williamson and Shmoys [46] for a good introduction. A *k-approximation algorithm* guarantees a feasible solution whose objective value is within a factor of $k$ of the optimal objective value, and terminates in polynomial time. For example, if a 3-approximation algorithm is given for a minimization problem, then this algorithm will return a solution $A$ for an instance $x$ with objective value $c(x, A)$, where $OPT(x) \leq c(x, A) \leq 3 \cdot OPT(x)$. We call $k$ the *approximation factor*, *performance guarantee*, or *approximation ratio*. For a maximization problem, the approximation factor $k$ by convention is less than one. Furthermore, a $k$-approximation algorithm that solves a minimization problem has an approximation factor greater than one. Since the justification for applying $k$-approximation algorithms is to terminate in polynomial time, algorithms such as these give an efficient means of obtaining an approximate solution to many NP-hard problems. Another type of approximation algorithm is a $k$-absolute approximation algorithm. A *k-absolute approximation algorithm* terminates in polynomial time, and for any feasible solution of a minimization problem, $OPT(x) \leq c(x, A) \leq OPT(x) + k$.

To help us understand the process of developing an approximation algorithm, we use an example in the context of scheduling. Consider the scheduling problem $P||C_{max}$, the makespan problem on identical parallel machines. Recall from Table 1.2 that this is a special case of the makespan problem on UPMs when the rows of a processing requirement matrix $P$ are the same. To present a $k$-approximation algorithm for $P||C_{max}$, we must:

a) Develop an algorithm for $P||C_{max}$ that returns a feasible schedule $X$ for any processing requirement matrix $P$, and terminates in polynomial time.

b) For any feasible schedule $X$ produced by the algorithm, prove that $C_{max} \leq k \cdot OPT(P)$, where $k$ is the approximation factor.

c) Present a tight example. A *tight example* is an infinite set of instances where the $k$-approximation algorithm produces a schedule with makespan $C_{max} = k \cdot OPT(P)$. The existence of a tight example for a $k$-approximation algorithm implies that its approximation factor is no smaller than $k$.

When we present approximation algorithms, we present both results a) and b) together as a theorem, then follow with c). Next, we present a $(2-1/m)$-approximation algorithm by Graham [15]. It is worth noting that Graham [16] also gave a 4/3-approximation approximation for $P||C_{max}$, which we discuss in Section 4.2.1. The proof presented for this result is based on one given by Marchal [34].

**Theorem 1.5.1** (Graham [15])**.** *There is a* $(2-(1/m))$*-approximation algorithm for* $P||C_{max}$*.*

*Proof.* Suppose we are given an arbitrary processing requirement matrix $P$. For $j = 1, 2, \ldots, n$, assign job $j$ to a machine with least load. Upon assigning all $n$ jobs, return the schedule then terminate. This algorithm terminates in polynomial time. For any instance, we can clearly see that a feasible schedule is returned, as it is a schedule that contains $n$ jobs.

Next, consider the schedule produced by the algorithm. For simplicity, denote $p_j = p_{i,j}$, as the rows of the processing requirement matrix $P$ are identical. Let machine $L$ be a machine that finishes last, and job $l$ be the last task to complete on this machine. Notice that the completion time of job $l$ is the makespan $C_{max}$. Job $l$

was scheduled to machine $L$ by the algorithm, because machine $L$ had the least load at the time job $l$ was scheduled. Therefore,

$$\sum_{j=1}^{n}(p_j) - p_l \geq m(C_{max} - p_l),$$

or,

$$C_l - p_l \leq \frac{1}{m}\Big(\sum_{j=1}^{n}(p_j) - p_l\Big) \leq OPT(P) - \frac{p_l}{m}, \tag{1.1}$$

as

$$\frac{\sum_{j=1}^{n} p_j}{m} \leq OPT(P),$$

because the average processing time of a machine is a lower bound on the optimal makespan for identical parallel machines. In an optimal schedule, at most $m \cdot OPT(P)$ total processing times is done by all the machines.

Consider the makespan $C_{max}$ of a schedule produced by the algorithm. Machine $L$ finished last, so $C_{max} = C_l + p_l = (C_{max} - p_l) + p_l$. Then, by inequality (1.1),

$$C_{max} = (C_{max} - p_l) + p_l \leq OPT(P) - \frac{p_l}{m} + p_l = OPT(P) + p_l\Big(1 - \frac{1}{m}\Big).$$

Since $p_l \leq OPT(P)$,

$$OPT(P) + p_l\Big(1 - \frac{1}{m}\Big) \leq \Big(2 - \frac{1}{m}\Big) \cdot OPT(P).$$

Therefore, we have presented a $(2-(1/m))$-approximation algorithm for $P||C_{max}$.

□

We now present a tight example by Graham [15] for the $(2-(1/m))$-approximation algorithm. Let there be $n = 2m - 1$ jobs, where $p_1 = p_2 = \cdots = p_{m-1} = (m-1)$, $p_m = \cdots = p_{2m-2} = 1$, and $p_{2m-1} = m$. If applied to the $(2 - (1/m))$-approximation

algorithm, a schedule with makespan of $2m - 1$ is produced. An optimal schedule has makespan of $m$. An optimal schedule pairs jobs of length one with jobs of length $(m-1)$, then places the single job of length $m$ on a machine of its own. The approximation algorithm we presented is called the *list scheduling algorithm*. We discuss in Section 4.2.1 how this simple greedy algorithm can be used or augmented for $P||C_{max}$.

Some approximation algorithms belong to a particular family of approximation algorithms called a polynomial-time approximation scheme (PTAS). Let $\epsilon > 0$ be rational. For minimization problems, a *PTAS* is a family of algorithms $\{A_\epsilon\}$, where an algorithm $A_\epsilon$ exists for each $\epsilon > 0$, such that $A_\epsilon$ is a $(1+\epsilon)$-approximation algorithm. Similarly, for maximization problems, for each $\epsilon > 0$, algorithm $A_\epsilon$ is a $(1-\epsilon)$-approximation algorithm. Thus, if an optimization problem has a PTAS, we can solve it with great flexibility of approximability by varying $\epsilon$. Note that a PTAS can be exponential in $1/\epsilon$. The drawback of a typical PTAS is that, as we approximate closer to an optimal solution with small $\epsilon$, the worst-case running time may rise exponentially. To resolve this issue, researchers seek efficient PTASs called fully polynomial-time approximation schemes (FPTASs). A *FPTAS* is a special type of PTAS where the worst-case time complexity of every algorithm is polynomial with respect to $1/\epsilon$ and the input size.

Let us introduce *hardness of approximation*. As researchers are motivated to find $k$-approximation algorithms that terminate faster and have better approximation factors, a question arises in computational complexity theory. Given a NP-hard optimization problem, how small can approximation factor $k$ be for a $k$-approximation algorithm, assuming P $\neq$ NP? For some NP-hard minimization problems, one can

show if there were a $k$-approximation algorithm for some $k$, then $\mathsf{P} = \mathsf{NP}$. Such a result is an example of a hardness of approximation result, and gives a lower bound on the approximation factor for which there exists a $k$-approximation algorithm, assuming $\mathsf{P} \neq \mathsf{NP}$. For a good introduction to techniques for proving hardness of approximation results, refer to Chapter 16 of Williamson and Shmoys [46].

## 1.6 Problem Statement

In this thesis, we cover two major parts. First, we present a survey of theoretic results for the makespan problem on UPMs ($R||C_{max}$) with a focus on subclasses and related scheduling problems. Second, we investigate the computational complexity of certain subclasses of $R||C_{max}$. In addition to this, we investigate the boundaries of polynomial-time-solvable subclasses and their relationships with other $\mathsf{NP}$-hard subclasses of $R||C_{max}$. For instance, to the best of my knowledge, there have been no studies of subclasses of $R||C_{max}$ when processing times are restricted to $p_{i,j} \in \{p, q, \infty\}$, where $p \leq q$. Note that if $p_{i,j} = \infty$, then job $j$ cannot be scheduled on machine $i$. A particular case we consider is when processing times $p_{i,j} \in \{1, 2, \infty\}$. Also, Lenstra *et al.* [33] showed that $R||C_{max}$ is polynomial-time solvable when processing times $p_{i,j} \in \{1, 2\}$. We determine the tractability of the subclass when processing times $p_{i,j} \in \{1, 2, 4\}$, and if $\mathsf{NP}$-hard, design an approximation algorithm with approximation ratio at most two. These are three problems considered in this thesis. In Section 1.7, we give our major results from our investigation.

## 1.7 Organization and Overview of Results

In this thesis, we present a variety of constructions, examples, and proofs. We indicate the end of a construction with ✠, the end of a example with ■, and the end of a proof with □. With our problem described, we provide an overview of the remainder of this thesis. In our overview of each chapter, we summarize our results.

In Chapters 2 to 4, a literature review is provided. Our literature review focuses on approximation algorithms for subclasses of $R||C_{max}$, and related scheduling problems. First, we explore the history of approximation algorithms for $R||C_{max}$ to this date with a brief summary. To accompany this, we present the hardness of approximation result for $R||C_{max}$ by Lenstra *et al.* [33]. In Chapter 3, we present a 2-approximation algorithm for $R||C_{max}$ by Lenstra *et al.* [33], the first of its kind. Results given in their 1990 paper are fundamental to a present-day understanding of the makespan problem on UPMs and approximation algorithms. With our focus on subclasses of $R||C_{max}$, Chapter 4 is devoted to a discussion about subclasses of $R||C_{max}$ and related scheduling problems. We separate results known in the literature into two major categories; tractable problems, and NP-hard problems.

In Chapters 5 and 6, we develop new results for particular subclasses of $R||C_{max}$. We concentrate our study on new or pragmatic subclasses that are similar to known NP-hard subclasses of $R||C_{max}$. We are especially interested in three-valued subclasses, which are not as well understood in the literature. Upon gathering our results, we organize our outcomes in two chapters by subclass, polynomial-time-solvable subclasses and NP-hard subclasses of $R||C_{max}$, respectively. We summarize the contributions we present in Table 1.5 by section number.

Finally, in Chapter 7 we summarize our investigation, and give our conclusions.

| **Section** | **Subclass of $R\|\|C_{max}$** | **Contribution** |
|---|---|---|
| 5.1 | $R\|p_{i,j} \in \{\omega, \infty\}\|C_{max}$ with initial loads | Polynomial-time algorithm (Theorem 5.1.2) |
| 6.1 | $R\|p_{i,j} \in \{1, 2, 4\}\|C_{max}$ | NP-hard (Theorem 6.1.1) |
| | $R\|p_{i,j} \in \{1, p, q\}\|C_{max}$ | NP-hard (Theorem 6.1.5) |
| | $R\|p_{i,j} \in \{p, q, r\}\|C_{max}$ | NP-hard (Corollary 6.1.6) |
| | $R\|p_{i,j} \in \{1, 2, 4\}\|C_{max}$ | 2-approximation algorithm* (Theorem 6.1.7) |
| 6.2 | $R\|p_{i,j} \in \{1, 2, \infty\}\|C_{max}$ | NP-hard (Theorem 6.2.1) |
| | $R\|p_{i,j} \in \{p, q, \infty\}\|C_{max}$ | NP-hard (Theorem 6.2.3) |
| | $R\|p_{i,j} \in \{p, q, \infty\}\|C_{max}$ | $(q/p)$-approximation algorithm* (Theorem 6.2.4) |
| | $R\|p_{i,j} \in \{1, 2, \infty\}\|C_{max}$ | 2-approximation algorithm* (Theorem 6.2.6) |
| | $R\|p_{i,j} \in \{p, q\}\|C_{max}$ | Linear-time $(q/p)$-approximation algorithm* (Theorem 6.2.7) |
| | $R\|p_{i,j} \in A(p, q) \cup \{\infty\}\|C_{max}$ | $(q/p)$-approximation algorithm* (Theorem 6.2.9) |
| | $R\|p \leq p_{i,j} \leq q\|C_{max}$ | Linear-time $(q/p)$-approximation algorithm* (Corollary 6.2.10) |

Table 1.5: Summary of our major results presented in Chapters 5 and 6 of this thesis. Results with * indicate an approximation algorithm with approximation factor that has the potential to be on par with or better than best-known approximation factor to date.

# Chapter 2

# Results for $R||C_{max}$

In this chapter, we give a brief survey of results for $R||C_{max}$. First, we summarize some of the major developments in approximation algorithms for $R||C_{max}$. Then, we present hardness of approximation results for $R||C_{max}$ that were given by Lenstra *et al.* [33].

## 2.1 Approximation Algorithms

We now provide a synopsis of the known results on approximation algorithms for $R||C_{max}$. Table 2.1 outlines approximation algorithms for $R||C_{max}$. Though subclasses such as $P||C_{max}$ have been studied since 1966 [15], the first approximation algorithm for $R||C_{max}$ was found in 1977 by Ibarra and Kim [20]. Ibarra and Kim gave an $m$-approximation that is similar to the list scheduling algorithm. Instead of assigning each job to a machine with least load like the list scheduling algorithm, their $m$-approximation algorithm assigns each job to a machine that completes the job ear-

| **Year** | **Researchers** | **k-Approximation Algorithm** |
|---|---|---|
| 1977 | Ibarra and Kim [20] | $m$-approximation algorithm. |
| 1981 | Davis and Jaffe [7] | $2\sqrt{m}$-approximation algorithm |
| 1990 | Lenstra *et al.* [33] | 2-approximation algorithm. |
| 2005 | Shchepin and Vakhania [42] | $(2-(1/m))$-approximation algorithm. |
| 2007 | Gairing *et al.* [13] | 2-approximation algorithm, improved time-complexity compared to [33; 42]. |

Table 2.1: Summary of $k$-approximation algorithms for $R||C_{max}$.

liest. In 1981, Davis and Jaffe [7] developed a $2\sqrt{m}$-approximation algorithm using two additional techniques when the assignments are made by the $m$-approximation algorithm. These two techniques are precomputation to develop estimates for potential job assignments, and activating or deactivating machines using a threshold. Before 1990, no constant-factor approximation algorithm existed for $R||C_{max}$.

In 1985, Potts [38] presented an algorithm for $R||C_{max}$ with an approximation factor of two. The algorithm by Potts has worst-case time complexity of $O(m^{m-1})$, so it is not a 2-approximation algorithm. Though its running time is exponential, this algorithm uses polynomially-bounded space. Lenstra *et al.* [33] refined Potts' result, and obtained a polynomial-time algorithm for $R||C_{max}$ with the same approximation factor. This algorithm employs a variation of fractional rounding and linear programming. In Chapter 3, we present the 2-approximation algorithm by Lenstra *et al.* [33] for $R||C_{max}$. Also, it is worth noting that Lenstra *et al.* presented a PTAS for $Rm||C_{max}$, but it is not a FPTAS. A FPTAS for $Rm||C_{max}$ was given later by Jansen and Porkolab [22].

In 2005, building upon results developed by Potts [38] and Lenstra *et al.* [33], Shchepin and Vakhania [42] designed a $(2-(1/m))$-approximation algorithm for the

makespan problem on UPMs. The algorithm developed by Shchepin and Vakhania [42] uses a linear programming formulation based on the linear program by Lenstra *et al.* [33] that includes more constraints. Upon finding a feasible solution to the program, the algorithm rounds the values of a given solution in a manner given by the authors. This algorithm provides an improved approximation factor, but for arbitrarily large number of machines $m$, this result still only provides an approximation ratio of two. The authors suggested that this result is the best approximation result that can be obtained through the rounding technique.

Most recently, Gairing *et al.* [13] in 2007 showed the linear programming aspects of the 2-approximation algorithm by Lenstra *et al.* [33] can be replaced by solving a generalized network-flow problem. The authors showed that this generalized network-flow problem can be adapted and solved by combinatorial techniques in polynomial time, and yield a more efficient 2-approximation algorithm for $R||C_{max}$. At this point, for any $m$, there is no better constant approximation factor than two for $R||C_{max}$.

## 2.2 Hardness of Approximation

The most recent results for the hardness of approximation of $R||C_{max}$ were given in 1990 by Lenstra *et al.* [33]. Lenstra *et al.* showed that there does not exist a $k$-approximation algorithm with an approximation factor of $k < 3/2$ for $R||C_{max}$, unless $\mathsf{P} = \mathsf{NP}$. We present their second corollary, though the proof resides in their Theorem 5. Theorem 5 [33] states that given an instance of $R||C_{max}$, to determine if there is a schedule with makespan at most two is $\mathsf{NP}$-complete. In order to show this result, we use the 3-dimensional matching (3DM) problem, which is known to be

NP-complete [25].

**Problem 2.2.1** (3-dimensional matching ($3DM$)). *Given three disjoint $n'$-sets*

$$A = \{a_1, a_2, \ldots, a_{n'}\}, B = \{b_1, b_2, \ldots, b_{n'}\}, C = \{c_1, c_2, \ldots, c_{n'}\},$$

*and a family of 3-sets $F = \{T_1, \ldots, T_{m'}\}$, where $|T_\gamma \cap A| = |T_\gamma \cap B| = |T_\gamma \cap C| = 1$, for $1 \leq \gamma \leq m'$, is there a 3DM? That is, is there a subfamily $F' \subseteq F$, such that $|F'| = n'$, and $\bigcup_{T_\gamma \in F'} T_\gamma = A \cup B \cup C$?*

This decision problem is a special case of the $q$DM problem (Problem 4.2.1) when $q = 3$. We define an instance of the 3DM problem as $I = (m', n', A, B, C, F)$.

**Theorem 2.2.2** (Lenstra *et al.* [33], Corollary 2). *For any $\alpha < 3/2$, there does not exist an $\alpha$-approximation algorithm for $R||C_{max}$, unless $\mathsf{P} = \mathsf{NP}$.*

*Proof.* Let us assume there exists a $\alpha$–approximation algorithm, for some $\alpha < 3/2$. Consider the following decision problem.

**Problem 2.2.3** (MAKESPANUPM2). *Given a $m \times n$ processing requirement matrix $P$ for UPMs, does there exist a schedule with makespan at most two?*

Under the assumption that we have an approximation algorithm with approximation ratio strictly less than 3/2, we develop a polynomial-time procedure that solves any instance of the 3DM problem. To begin, we show how to create an instance $I' \in$ MAKESPANUPM2 from any instance $I \in 3DM$ in polynomial time.

**Construction 2.2.4.** Given an instance $I = (m', n', A, B, C, F)$ of the 3DM problem, we wish to create an instance $I' = (P, m, n) \in$ MAKESPANUPM2. Given an element

$a \in A$, define $k(a)$ where

$$k(a) = |\{a \in T_\gamma \mid T_\gamma \in F, 1 \leq \gamma \leq m'\}|.$$

If $m' < n'$, or there exists $a \in A$ such that $k(a) = 0$, create a trivial "no" instance. A 3DM will not exist when there are not enough 3-sets to contain the members of the three $n'$-sets, and each element $a \in A$ needs to appear in at least one of the 3-sets in $F$. After this point, we assume $m' \geq n'$, and $k(a) > 0$ for all $a \in A$. Let there be $m = m'$ machines, and $n = 2n' + \sum_{a \in A} (k(a) - 1)$ jobs. For each 3-set of the family $F$, introduce a machine. For the 3-set $T_\gamma \in F$, we denote the corresponding machine by $\gamma$. A machine $\gamma$ is said to be of *machine type* $a$ if $a \in T_\gamma$, where $a \in A$.

There will be two types of jobs, "element jobs" and "dummy jobs". Each *element job* corresponds to an element contained in $B \cup C$. If an element is in a 3-set $T_\gamma$, then its corresponding element job can be scheduled on machine $\gamma$ to take one time unit, and two time units otherwise. Each *dummy job* is of a particular machine type. A dummy job of machine type $a$ can be scheduled on a machine of type $a$ for two time units. For all other machines, such a job can be scheduled to take $\infty$ time units. For each $T_\gamma \in F$, recall that $|T_\gamma \cap A| = 1$. If $k(a) > 1$, then there are at least two 3-sets in $F$ that contain element $a$. The dummy jobs ensure that if a feasible schedule of length two exists, only one machine of type $a$ can be assigned two element jobs, while each remaining $(k(a) - 1)$ machines must be assigned one dummy job. There are $2n'$ element jobs, and for each $a \in A$, there are $(k(a) - 1)$ dummy jobs of machine type $a$. As a result, an instance $I' \in$MAKESPANUPM2 is constructed in polynomial time.

✠

With a scheduling instance $I' \in$ MAKESPANUPM2 created by Construction 2.2.4,

we show that there is a 3DM for $I \in 3DM$ if and only if there is a schedule with makespan at most two. We use this result to develop a polynomial-time algorithm that solves the 3DM problem assuming that there is an $\alpha$-approximation algorithm, where $\alpha < 3/2$.

**Lemma 2.2.5** (Lenstra *et al.* [33], Theorem 5)**.** *There is a matching (3DM) for instance $I \in 3DM$ if and only if the scheduling instance $I' \in$ MAKESPANUPM2 produced by Construction 2.2.4 using $I$ has a schedule with makespan at most two.*

*Proof.* Suppose there is a matching $F'$. For each 3-set $T_\gamma \in F'$, schedule element jobs that corresponds to elements in $B \cup C$ on machine $\gamma$ in one time unit. Each of these machines have load 2 upon scheduling the jobs of $B \cup C$. Next, consider each $a \in A$. There are exactly $(k(a) - 1)$ machines of type $a$ available with no jobs assigned yet. Schedule $(k(a) - 1)$ dummy jobs on each remaining machine of type $a$ for two time units. This assignment produces a schedule with makespan of two.

Next, suppose there is a schedule with makespan of two. In order to respect the makespan, each dummy job of machine type $a$ must be scheduled on a machine of type $a$ for two time units. Since there are exactly $(k(a) - 1)$ dummy jobs of machine type $a$ each, there is exactly one machine of machine type $a$ available to schedule element jobs. There are $2n'$ element jobs that need to be scheduled on $n'$ machines. To respect the makespan of two, element jobs must all be scheduled to take one time unit on a machine, such that only two element jobs are scheduled on each remaining machine. If element jobs that correspond to $b \in B$ and $c \in C$ are scheduled for one time unit on a machine of type $a$, then this assignment corresponds to a 3-set $\{a, b, c\} \in F$. As there are $n'$ machines that schedule element jobs, then the corresponding set of 3-sets

is a subfamily $F' \subseteq F$, and $|F'| = n'$. If the makespan cannot be respected, then a matching $F'$ cannot exist. Thus, $F'$ is a matching.

Therefore, there is a matching if and only if there exist a schedule with makespan at most two. □

If there is an $\alpha$-approximation algorithm with approximation ratio of $\alpha < 3/2$, then we can use it to solve the 3DM problem in polynomial time, as follows. Given an arbitrary instance $I \in 3DM$, construct an instance $I'$ of the makespan problem on UPMs using Construction 2.2.4. Since $\alpha < 3/2$, $2\alpha < 3$. The makespan is of integral value, so the approximation algorithm will guarantee a schedule with makespan at most two only if a 3DM exists. By Lemma 2.2.5, if the schedule produced by the approximation algorithm has makespan less than or equal to two, say "yes" as there is a 3DM. If the schedule returned by the approximation algorithm has makespan greater than or equal to three, then say "no". This procedure will complete in polynomial time.

Therefore, there is no $\alpha$-approximation algorithm with approximation $\alpha < 3/2$ for $R||C_{max}$, unless $\mathsf{P} = \mathsf{NP}$. □

To help us understand the decision procedure found in Theorem 2.2.2 that employs Construction 2.2.4, let us apply the procedure to an example 3-dimensional matching ($3DM$) instance.

**Example 2.2.6.** Let

$$A = \{1, 2, 3, 4\}, B = \{5, 6, 7, 8\}, C = \{9, 10, 11, 12\},$$

and a family of 3-sets,

$$F = \{\{1, 5, 9\}, \{2, 5, 9\}, \{2, 7, 10\}, \{2, 6, 10\}, \{3, 6, 11\}, \{4, 8, 12\}\}.$$

In this example, we see that $m' = 6$, and $n' = 4$.

Next, apply Construction 2.2.4 to the $3DM$ instance. There will be $m = m' = 6$ machines, and $n = 2n' + \sum_{a \in A} (k(a) - 1)$ jobs. Of the $n$ jobs, there will be $2n' = 8$ element jobs, and $\sum_{a \in A} (k(a) - 1)$ dummy jobs over at most $|A| = n' = 4$ machine types. Recall that $k(a)$ equals the number of 3-sets in $F$ that contain element $a \in A$. In Table 2.2, we show these calculations.

| $a \in A$ | $k(a)$ | $k(a) - 1$ |
|---|---|---|
| 1 | 1 | 0 |
| 2 | 3 | 2 |
| 3 | 1 | 0 |
| 4 | 1 | 0 |

Table 2.2: For each element $a \in A$ in an example instance of the 3DM problem, the calculation of $k(a)$ to determine the number of dummy jobs of a particular machine type.

So in the example, we have eight element jobs, and two dummy jobs of machine type 2. The dummy jobs of machine type 2 will be scheduled to take two time units. These dummy jobs will be scheduled on machines that correspond to 3-sets that contain the element $2 \in A$, and $\infty$ time units otherwise. As a result, we obtain the

processing requirement matrix

$$P = \begin{pmatrix} \textcircled{1} & 2 & 2 & 2 & \textcircled{1} & 2 & 2 & 2 & \infty & \infty \\ 1 & 2 & 2 & 2 & 1 & 2 & 2 & 2 & \textcircled{2} & 2 \\ 2 & 2 & \textcircled{1} & 2 & 2 & \textcircled{1} & 2 & 2 & 2 & 2 \\ 2 & 1 & 2 & 2 & 2 & 1 & 2 & 2 & 2 & \textcircled{2} \\ 2 & \textcircled{1} & 2 & 2 & 2 & 2 & \textcircled{1} & 2 & \infty & \infty \\ 2 & 2 & 2 & \textcircled{1} & 2 & 2 & 2 & \textcircled{1} & \infty & \infty \end{pmatrix}.$$

Circled in the entries of $P$ above is an optimal solution to this scheduling instance. Notice that each machine that schedules element jobs has exactly two element jobs on a machine. Dummy jobs are scheduled on each remaining machine. This schedule has makespan of two, which corresponds to a matching that consists of a subfamily of 3-sets of $F$ that match the machines that are assigned the element jobs in the schedule. ■

If there was an $\alpha$-approximation algorithm with $\alpha < 3/2$, then the objective value is guaranteed to be strictly less than three if there is a matching. With such an approximation algorithm, one could solve the the 3DM problem in polynomial time.

# Chapter 3

# A 2-Approximation Algorithm for $R||C_{max}$

In 1990, Lenstra *et al.* [33] developed the first 2-approximation algorithm for the makespan problem on unrelated parallel machines. Before their result, the best approximation factor for any approximation algorithm for $R||C_{max}$ was $2\sqrt{m}$. Having an approximation algorithm that does not have a constant approximation factor is undesirable, because the quality of a solution depends on the input instance. To this date, for any number of machines $m$, the best known approximation factor is 2. Since many of the known 2-approximation algorithms for $R||C_{max}$ are derived from the 2-approximation algorithm by Lenstra *et al.* [33], this chapter is dedicated to presenting their 2-approximation algorithm. To understand this algorithm, some background in linear programming is required. We recommend reading Appendix A if the reader is not familiar with basic feasible solutions (BFSs), integer programs (IPs), integrality gap, linear programs (LPs), relaxed LPs, or the rounding technique.

## 3.1 Background and Overview of the Algorithm

A common technique in approximation algorithms is the routnding technique. In the *rounding technique*, the idea is to formulate the problem of interest as an IP, then relax the integral constraints and solve its relaxed LP. Once the relaxed LP is solved, round each component in the solution of the relaxed LP to form a feasible solution to the original IP. Then, the approximation factor is at least the integrality gap of the IP. For an example of this process, refer to our example with the minimum vertex cover problem in Section A.4. We use the rounding technique. Let us begin by presenting an IP formulation $IP_0(P,t)$ for $R||C_{max}$. Given a processing requirement matrix $P$, let *decision variable* $x_{i,j} = 1$ if job $j$ is assigned to machine $i$ to take $p_{i,j} > 0$ time units, and $x_{i,j} = 0$ otherwise. Assume $t \in \mathbb{Z}^+$ is fixed, then define $IP_0(P,t)$ as

$$\text{minimize} \quad 0 \qquad (IP_0(P,t))$$

$$\text{subject to} \quad \sum_{i=1}^{m} x_{i,j} = 1 \ , \ \text{for } j = 1,2,\ldots,n \qquad (3.1)$$

$$\sum_{j=1}^{n} p_{i,j}x_{i,j} \leq t \ , \ \text{for } i = 1,2,\ldots,m \qquad (3.2)$$

$$x_{i,j} \in \{0,1\}, \ \text{for } i = 1,2,\ldots,m, \ j = 1,2,\ldots,n. \qquad (3.3)$$

For $IP_0(P,t)$, the makespan is represented as $t$. For this IP, the smallest value $t$ when $IP_0(P,t)$ is feasible is the optimal makespan. Notice the objective function of $IP_0(P,t)$ is 0—a feasible schedule with makespan at most $t$ follows from the constraints of this IP. Constraints (3.1) enforce that each job is scheduled on exactly one machine. Also, constraints (3.2) maintain that each machine takes at most $t$ time units to complete. Since each decision variable $x_{i,j} \in \{0,1\}$ in $IP_0(P,t)$, $IP_0(P,t)$

models $R||C_{max}$ when $t$ is the optimal makespan. We use the subscript 0 in $IP_0(P,t)$ to indicate that this is our first attempt at devising an IP for $R||C_{max}$. To develop an approximation algorithm with a constant approximation factor, relax $IP_0(P,t)$ so that each $x_{i,j} \in [0,1]$. We obtain $LP_0(P,t)$, where

$$\text{minimize} \quad 0 \qquad (LP_0(P,t))$$

$$\text{subject to} \sum_{i=1}^{m} x_{i,j} = 1 \ , \ \text{for } j = 1,2,\ldots,n \qquad (3.1)$$

$$\sum_{j=1}^{n} p_{i,j}x_{i,j} \leq t \ , \ \text{for } i = 1,2,\ldots,m \qquad (3.2)$$

$$x_{i,j} \geq 0, \ \text{for } i = 1,2,\ldots,m, \ j = 1,2,\ldots,n. \qquad (3.4)$$

Since $LP_0(P,t)$ permits rational values for each decision variable $x_{i,j}$, the processing time needed for each job can be distributed across more than one machine while satisfying all the constraints. Unfortunately, the integrality gap of $IP_0(P,t)$ is unbounded. For example, consider a $m \times 1$ processing requirement matrix $P$, where each $p_{i,1} = m$. The minimum value of $t$ for which $LP_0(P,t)$ is feasible is 1 as one could assign each $x_{i,1} = 1/m$. Unlike $LP_0(P,t)$, the minimum $t$ when $IP_0(P,t)$ is feasible is $m$; one job assigned to one machine. So in this example, the worst-case ratio between $IP_0(P,t)$ and $LP_0(P,t)$ is $m$. Thus, we cannot develop a rounding strategy as easily as we did for the 2-approximation algorithm for the minimum vertex cover problem in Section A.4, and the integrality gap is not as good an indicator as before.

We want to eventually obtain a feasible solution for $IP_0(P,t)$. As with the rounding technique, a rounding strategy is needed. An ideal rounding strategy would be to round all positive decision variables to integral assignments so that $x_{i,j} = 0$ means that the job is not assigned to a machine. For some values of $k, j$, constraints (3.1)

enforce when a decision variable $x_{k,j} = 1$, each $x_{i,j} = 0$, where $k \neq i$. What about when $x_{i,j} \in (0, 1)$? It is not clear how these decision variables should be rounded. We need to assign jobs carefully so the constraints of $IP_0(P, t)$ are satisfied. First, consider constraints (3.2). For some processing requirement matrix $P$ and fixed $t$, an immediate issue arises with the formulation of $LP_0(P, t)$. For $IP_0(P, t)$, if $p_{i,j} > t$, it is guaranteed that $x_{i,j} = 0$ as $x_{i,j} \in \{0, 1\}$. Unlike this, a feasible solution of $LP_0(P, t)$ can assign some $x_{i,j} > 0$, even when $p_{i,j} > t$. For example, consider processing requirement matrix

$$P = \begin{pmatrix} 8 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 \end{pmatrix}.$$

Fix $t = 3$, the optimal makespan for this instance of $R||C_{max}$. A feasible solution to $LP_0(P, 3)$ is

$$(x_{1,1} = 2/8, x_{1,2} = 0, x_{1,3} = 1, x_{1,4} = 0, x_{2,1} = 6/8, x_{2,2} = 1, x_{2,3} = 0, x_{2,4} = 1).$$

But, $p_{1,1} = 8 > 3$. If we were to perform some kind of rounding strategy, such decision variables could potentially lead to an infeasible solution to $IP_0(P, 3)$. For instance, assigning job 1 to machine 1 would violate one of constraints (3.2). We need to include more constraints to ensure no decision variable in $LP_0(P, t)$ is non-zero when $p_{i,j} > t$. Introduce *artificial constraints* that impose that each decision variable $x_{i,j} = 0$, when $p_{i,j} > t$. We give the new LP $LP_1(P, t)$ in standard form. For

some processing requirement matrix $P$ and fixed $t \in \mathbb{Z}^+$, define $LP_1(P,t)$, where

$$\text{minimize} \quad 0 \qquad (LP_1(P,t))$$

$$\text{subject to} \sum_{i=1}^{m} x_{i,j} = 1 \ , \ \text{for } j = 1,2,\ldots,n \tag{3.1}$$

$$\sum_{j=1}^{n} p_{i,j}x_{i,j} + s_i = t \ , \ \text{for } i = 1,2,\ldots,m \tag{3.2}$$

$$x_{i,j} = 0 \ , \ \text{for } i,j, \ \text{where } p_{i,j} > t \tag{3.5}$$

$$x_{i,j} \geq 0, \ \text{for } i = 1,2,\ldots,m, \ j = 1,2,\ldots,n \tag{3.4}$$

$$s_i \geq 0, \ \text{for } i = 1,2,\ldots,m. \tag{3.6}$$

By including artificial constraints (3.5), we have introduced an alternate version of the same LP that Lenstra *et al.* [33] used for their 2-approximation algorithm. The 2-approximation algorithm by Lenstra *et al.* [33] gives a rounding strategy used to satisfy constraints (3.1), while maintaining an approximation factor of 2.

Now we give a brief description of the 2-approximation algorithm to outline the remainder of this chapter. We present the 2-approximation algorithm by Lenstra *et al.* [33] as pseudocode in Algorithm 3.1.1. First, the algorithm finds a value $t = d_{best}$, such that $d_{best} \leq OPT(P)$ and $LP_1(P, t = d_{best})$ is feasible. Section 3.2 describes how to calculate $d_{best}$ by constructing a greedy schedule with makespan we denote as $\kappa$, and perform binary search on $LP_1(P,t)$ over $[\lfloor \kappa/m \rfloor, \kappa]$ in polynomial time. Next, the algorithm finds a BFS of $LP_1(P, d_{best})$. As elaborated in Section 3.3, a BFS of $LP_1(P,t)$ guarantees properties we can use to develop a rounding strategy for any jobs when decision variables are not integral. Then, in the same section we present graph theoretic properties between pseudoforests and a BFS of $LP_1(P,t)$. Upon obtaining a BFS of $LP_1(P, d_{best})$, two sets of jobs are assigned: integrally set and fractionally set

jobs. In Section 3.4, we discuss the definitions of each job type and present some of their properties. Consequently, we show how the algorithm "rounds" the fractionally set jobs to obtain a feasible schedule. This feasible schedule has makespan at most $2 \cdot OPT(P)$. In Section 3.5, we present the complete algorithm, and prove that it is a 2-approximation algorithm.

**Algorithm 3.1.1** 2-approximation algorithm for $R||C_{max}$

**procedure** 2APPROXMAKESPANUPMS($P, m, n$)
  Let $X$ be a $m \times n$ matrix with zero entries; ▷ $m \times n$ assignment matrix
  $F := \varnothing$; ▷ Will contain a matching
  Construct schedule $S$ using the greedy algorithm in Section 3.2;
  Let $\kappa$ be the makespan of $S$;
  Obtain lower bound $d_{best}$ by applying the binary search procedure given in Section 3.2;
  Let $(\tilde{\mathbf{x}}, s_1, \ldots, s_m)$ be a basic feasible solution of $LP_1(P, t = d_{best})$;
  Apply Construction 3.3.2 with decision variables $\tilde{\mathbf{x}}$ to get $G(\tilde{\mathbf{x}}) = (M \cup J, E)$;
  **for each** $j \in J$ **do** ▷ Construct partial schedule of integrally set jobs
    **if** $deg_{G(\tilde{\mathbf{x}})}(j) == 1$ **then** ▷ Is $\tilde{x}_{i,j} = 1$?
      $e := \{i, j\} \in E$; ▷ Exactly one edge to some $i \in M$
      $X_{i,j} := 1$; ▷ Integral job; assign job $j$ to machine $i$
      $E := E \setminus \{i, j\}$
      $J := J \setminus \{j\}$;
    **end if**
  **end for**
  $J' := J$;
  $E' := E$;
  Let $G'(\tilde{\mathbf{x}}) = (M \cup J', E')$;
  Let $F$ be a maximum matching in $G'(\tilde{\mathbf{x}})$;
  **for each** $\{i, j\} \in F$, where $i \in M$ and $j \in J$ **do** ▷ Complete partial schedule
    $X_{i,j} := 1$; ▷ Round a fractionally set job; assign job $j$ to machine $i$
  **end for**
  **return** $X$; ▷ Return schedule
**end procedure**

## 3.2 Finding a Lower Bound on the Optimal Makespan

Algorithm 3.1.1 begins by constructing a greedy schedule as follows. Assign each job to a machine that takes the least time to process the job. That is, for $j = 1, \ldots, n$, assign each job $j$ to machine $\text{argmin}_{1 \leq i \leq m}\{p_{i,j}\}$. For any instance of $R||C_{max}$, such a schedule can be produced in polynomial time by this procedure; we denote its makespan as $\kappa$.

Now, we will prove that $\lfloor \kappa/m \rfloor$ is a lower bound on the optimal makespan $OPT(P)$.

**Lemma 3.2.1** (Lenstra *et al.* [33], Lemma 1)**.** *Suppose we obtain a greedy schedule with makespan $\kappa$ using $P$ as described in Section 3.2. Then, $\lfloor \kappa/m \rfloor \leq OPT(P)$.*

*Proof.* Observe $\lfloor \kappa/m \rfloor \leq OPT(P)$ is true if and only if $\kappa \leq m \cdot OPT(P)$. Notice that the total processing time of an optimal schedule

$$\sum_{i=1}^{m}\sum_{j=1}^{n} p_{i,j}x^*_{i,j} \leq m \cdot max_{1 \leq i \leq m}\sum_{j=1}^{n} p_{i,j}x^*_{i,j} = m \cdot OPT(P).$$

Consider the makespan of the greedy schedule $\kappa = max_{1 \leq i \leq m}\sum_{j=1}^{n} p_{i,j}x_{i,j}$. Each job in the greedy schedule is assigned to a machine for which the job takes the least amount of time. Since the total processing time of the greedy schedule is at least the makespan of the greedy schedule,

$$\kappa \leq \sum_{i=1}^{m}\sum_{j=1}^{n} p_{i,j}x_{i,j} \leq \sum_{i=1}^{m}\sum_{j=1}^{n} p_{i,j}x^*_{i,j} \leq m \cdot OPT(P).$$

Therefore, $\kappa \leq m \cdot OPT(P)$, and $\lfloor \kappa/m \rfloor \leq OPT(P)$. □

It is worth noting that since this greedy schedule can be constructed in polynomial time and $\kappa \leq m \cdot OPT(P)$, we have described a $m$-approximation algorithm for

$R||C_{max}$. A tight example for this $m$-approximation algorithm is a $m \times m$ processing requirement matrix, where each entry is $m$. An optimal schedule will have makespan of $m$, and the greedy schedule produced can have makespan at most $m^2$.

In the next step, Algorithm 3.1.1 employs *binary search* over a range of values to determine the smallest value $d_{best}$ for which $LP_1(P, t = d_{best})$ is feasible. Using the algorithm binary search to find such lower bounds for LPs is a common technique in linear programming; see page 171 of Papadimitriou and Steiglitz [37]. The goal for the 2-approximation algorithm at this stage is to find a feasible LP with the smallest value $d_{best}$ between $\lfloor \kappa/m \rfloor$ and $\kappa$, inclusively. Since $\lfloor \kappa/m \rfloor$ is a lower bound on $OPT(P)$ by Lemma 3.2.1, when the smallest value for $d_{best}$ is found where $LP_1(P, t = d_{best})$ is feasible, $d_{best} \leq OPT(P)$.

The algorithm performs a binary search procedure using $U = \kappa$ and $L = \lfloor \kappa/m \rfloor$ as initial upper and lower bounds, respectively. Set $d = \lfloor (L + U)/2 \rfloor$ and test the feasibility of $LP_1(P, t)$, when $t = d$. The feasibility of $LP_1(P, t = d)$ can be determined by solving $LP_1(P, t = d)$, which can be done in polynomial time [24; 26]. Note that when the the objective function of a LP is set to zero, determining the feasibility of an LP is the same as determining the feasibility of a system of linear inequalities (LSI), which is just as difficult computationally [37]. If $LP_1(P, d)$ is feasible, set $d_{best} = d$, and let $U = d$. Otherwise, let $L = d+1$. Repeat this process until $U = L$. Let $p_{max}$ be the longest processing time in processing requirement matrix $P$. The smallest value $d_{best}$ for which $LP_1(P, d_{best})$ is feasible can be found in $O(\log_2(p_{max}n))$ feasibility checks. If the LP is found to be feasible at any step of this process, the algorithm can keep the best solution obtained so far as this method will find a BFS $(\tilde{\mathbf{x}}, s_1, \ldots, s_m)$ when

feasible. A BFS of $LP_1(P, t = d_{best})$ is needed following the binary search procedure.

## 3.3 Basic Feasible Solutions and Pseudoforests

Following the binary search procedure that finds a lower bound $d_{best}$ for which $LP_1(P, t = d_{best})$ is feasible, the 2-approximation algorithm finds a BFS $(\tilde{\mathbf{x}}, s_1, \ldots, s_m)$. Finding a BFS of $LP_1(P, t = d_{best})$ can be done in polynomial time as this LP is bounded, and we can solve $LP_1(P, t = d_{best})$ in polynomial time [24; 26]. An optimal solution is one of the BFSs of $LP_1(P, t = d_{best})$ as a consequence of Theorem A.3.1. We show a particular property of a BFS for $LP_1(P, d_{best})$. Later, we exploit this property when finding a feasible schedule.

**Lemma 3.3.1** (Lenstra *et al.* [33], Theorem 1)**.** *Given the decision variables $\tilde{\mathbf{x}}$ in a BFS of $LP(P, d_{best})$, there are at most $(m + n)$ non-zero decision variables.*

*Proof.* There are $m + n + z$ constraints in $LP(P, d_{best})$, where $n$ constraints are from constraints (3.1), $m$ constraints are from constraints (3.2), and $z$ of the constraints are the artificial constraints (constraints (3.5)). By the definition of a BFS, there are at most $m + n + z$ non-zero variables. From the $z$ artificial constraints, each decision variable $\tilde{x}_{i,j}$ is set to zero when $p_{i,j} > t$. Since these $z$ constraints set $z$ decision variables to zero, there remain at most $(m+n)$ non-zero decision variables. Therefore, there are at most $(m + n)$ non-zero decision variables in a BFS of $LP(P, d_{best})$. □

Next, Algorithm 3.1.1 uses the decision variables $\tilde{\mathbf{x}}$ of a BFS in $LP_1(P, d_{best})$, and constructs a bipartite graph $G(\tilde{\mathbf{x}}) = (M \cup J, E)$ in polynomial time by applying Construction 3.3.2.

**Construction 3.3.2.** Fix $t \in \mathbb{Z}^+$. Consider the decision variables $\mathbf{x}$ of a feasible solution to $LP_1(P,t)$. Let there be machine vertices $M = \{1, 2, \ldots, m\}$ for the machines, and job vertices $J = \{1, 2, \ldots, n\}$ for the jobs. Construct a bipartite graph $G(\mathbf{x}) = (M \cup J, E)$, where $E = \{\{i,j\} \mid i \in M,\ j \in J,\ x_{i,j} > 0\}$. Delete all machine vertices with degree zero. ❋

A *pseudotree* is an undirected connected graph with at most one cycle. As a consequence, a pseudotree is a tree or a tree with an extra edge that forms a cycle. A *pseudoforest* is an undirected graph for which each connected component is a pseudotree. We wish to show that the resulting bipartite graph $G$ is a pseudoforest when decision variables $\tilde{\mathbf{x}}$ are used from any BFS in $LP_1(P, t = d_{best})$.

**Lemma 3.3.3** (Lenstra *et al.* [33], Theorem 1)**.** *The bipartite graph $G(\tilde{\mathbf{x}})$ is a pseudoforest.*

*Proof.* For some processing requirement matrix $P$ and fixed $t \in \mathbb{Z}^+$, consider the decision variables $\tilde{\mathbf{x}}$ of a BFS in $LP_1(P,t)$. By definition, a pseudoforest is comprised of pseudotrees. In order to show $G(\tilde{\mathbf{x}})$ is a pseudoforest, we consider any connected component and show the number of edges cannot exceed the number of vertices. Thus, each connected component would be a pseudotree. Let $G_C(\tilde{\mathbf{x}}) = (M_C \cup J_C, E_C)$ be a connected component of $G(\tilde{\mathbf{x}})$, where $M_C \subseteq M$, $J_C \subseteq J$, and $E_C \subseteq E$.

For $LP_1(P,t)$, consider its constraint matrix $A$. In the constraint matrix $A$, there are $n$ rows that correspond to each job, $m$ rows of $A$ that correspond to each machine, and a row for each processing time $p_{i,j}$ when $p_{i,j} > t$. Let $z$ denote the number of rows that correspond to the last set of rows we specified, the artificial constraints (i.e., constraints (3.5)). Restrict the rows of $A$ to those corresponding to machines $M_C$ and

jobs $J_C$, and columns indexed by each decision variable $\tilde{x}_{i,j}$, where edge $\{i,j\} \in E_C$. In addition, remove any rows that correspond to artificial constraints with $x_{i,j} = 0$, where $\{i,j\} \notin E_C$. Say there are $z'$ rows remaining from the $z$ artificial constraints. Call this submatrix $A'$. Notice that $A'$ has $|M_C \cup J_C| + z'$ rows and $|E_C|$ columns.

Next, restrict the decision variables $\tilde{\mathbf{x}}$ to those that index the columns of $A'$. Observe that each column of $A'$ can be indexed by a non-zero decision variable in $\tilde{\mathbf{x}}$. The columns of $A$ indexed by non-zero decision variables are linearly independent. Since the columns of $A'$ are a subset of the columns that are indexed by non-zero decision variables in $A$, the columns of $A'$ are also linearly independent. So by the definition of a BFS, there are at most $|M_C \cup J_C| + z'$ non-zero decision variables. Then, consider the $z'$ rows of $A'$ that correspond to artificial constraints. An artificial constraint appears in $A'$ if $p_{i,j} > t$, and $\{i,j\} \in E'$. But, if $p_{i,j} > t$, then $\tilde{x}_{i,j} = 0$. The $z'$ artificial constraints will set $z'$ decision variables to zero. Thus, $|E_C| \leq |M_C \cup J_C|$.

Since $E_C = \{\{i,j\} \in E \mid i \in M_C,\ j \in J_C,\ \tilde{x}_{i,j} > 0\}$ and $|E_C| \leq |M_C \cup J_C|$, connected component $G_C(\tilde{\mathbf{x}})$ is a pseudotree. Therefore, for the decision variables $\tilde{\mathbf{x}}$ of any BFS in $LP_1(P,t)$, $G(\tilde{\mathbf{x}})$ is a pseudoforest. □

## 3.4 Jobs and Matchings

In this section, we discuss how the Algorithm 3.1.1 creates a feasible schedule. Using a BFS in $LP_1(P, t = d_{best})$ and bipartite graph $G(\tilde{\mathbf{x}})$, matching techniques are used as a rounding strategy to obtain a feasible schedule.

For the bipartite graph $G(\tilde{\mathbf{x}})$, consider each job vertex $j \in J$. If $deg_{G(\tilde{\mathbf{x}})}(j) = 1$ and $\{i,j\} \in E$, assign job $j$ to machine $i$. Each of these jobs will correspond to some

$\tilde{x}_{i,j} = 1$ from the decision variables $\tilde{\mathbf{x}}$, because each job $j$ is scheduled on exactly one machine due to constraints (3.1) in $LP_1(P, t = d_{best})$. Upon scheduling each job, for $G(\tilde{\mathbf{x}})$, remove vertex $j$, and edge $\{i, j\}$. After all the vertices of $J$ are considered, call the resulting bipartite graph $G'(\tilde{\mathbf{x}}) = (M \cup J', E')$.

We now show at most $m$ jobs still need to be scheduled following the assignment of the integral jobs. Consider the decision variables $\tilde{\mathbf{x}}$ of a BFS in $LP_1(P, t = d_{best})$. Say that a job $j$ is *fractionally set* if there is a machine $i$, such that $0 < \tilde{x}_{i,j} < 1$. To contrast, for all machines $i$, a job $j$ is *integrally set* if $\tilde{x}_{i,j} \in \{0, 1\}$.

**Lemma 3.4.1** (Lenstra *et al.* [33], Theorem 1)**.** *For any BFS in $LP_1(P, t = d_{best})$, let $\tilde{\mathbf{x}}$ be its decision variables. There are at most $m$ fractionally set jobs.*

*Proof.* Suppose we are given decision variables $\tilde{\mathbf{x}}$ in a BFS of $LP_1(P, d_{best})$. Let $\alpha$ and $\psi$ be the number of integrally set and fractionally set jobs, respectively. Since these two types of jobs are disjoint and there are $n$ jobs, $\alpha + \psi = n$. By definition, each fractionally set job is assigned to at least two machines. For each fractionally assigned job, there are at least two non-zero entries contributed to $\tilde{\mathbf{x}}$. By Lemma 3.3.1, $\alpha + 2\psi \leq (m + n)$. Recall that $\alpha + \psi = n$, so

$$(n + \psi) - n \leq (m + n) - n,$$

and

$$\psi \leq m.$$

Therefore, there are at most $m$ fractionally set jobs. □

At this point in Algorithm 3.1.1, there are at most $m$ jobs left to be assigned to $m$ machines. At least $(n - m)$ jobs were assigned, because there are at least $(n - m)$

integrally set jobs. Next, the algorithm builds a matching $F$ of $G'(\tilde{\mathbf{x}}) = (M \cup J', E')$ that saturates all every $j \in J'$. In Lemma 3.4.2, we show such a matching exists, and that it can be found in polynomial time.

**Lemma 3.4.2** (Lenstra *et al.* [33], Theorem 1)**.** *The bipartite graph $G'(\tilde{\mathbf{x}}) = (M \cup J', E')$ contains a matching that saturates all vertices $j \in J'$. Such a matching can be found in polynomial time.*

*Proof.* By Lemma 3.3.3, $G(\tilde{\mathbf{x}})$ is a pseudoforest. Since $G'(\tilde{\mathbf{x}})$ is formed by removing an equal number of edges and vertices from $G(\tilde{\mathbf{x}})$, $G'(\tilde{\mathbf{x}})$ is also a pseudoforest. In the construction of $G'(\tilde{\mathbf{x}})$, job vertices of degree one and their incident edges were removed from the original graph $G(\tilde{\mathbf{x}})$. As a result, for each $j \in J'$, $deg_{G'(\tilde{\mathbf{x}})}(j) \geq 2$. Also, the only vertices of $G'(\tilde{\mathbf{x}})$ that can have degree one are in $M$.

Since $G'(\tilde{\mathbf{x}})$ is a pseudoforest, each connected component is a pseudotree. Also, the only vertices with degree of one are machine vertices of $M$. For each $i \in M$, if $deg_{G'(\tilde{\mathbf{x}})}(i) = 1$ and $\{i, j\} \in E'$, then match job vertex $j$ to machine vertex $i$. Upon doing so, delete job vertex $j$, and machine vertex $i$ at each stage. When all $i \in M$ have been considered after each step and no remaining machine vertices have degree of one, call the resulting bipartite graph $G''(\tilde{\mathbf{x}}) = (M' \cup J'', E'')$.

Since $G''(\tilde{\mathbf{x}})$ is a bipartite pseudoforest, what remains are vertex-disjoint cycles of even length. For each connected component, apply a depth-first search (DFS) procedure to find each cycle. There are at most $m$ connected components at this stage, so this procedure will find all the cycles in polynomial time. Since all the leaf vertices have been removed, each connected component is now an even-length cycle, and there be no job vertices excluded from such a cycle. Match alternating edges of

the cycle. Remove any edges that are matched at each step. As a result, a matching $F$ that saturates all the job vertices is found in polynomial time. □

To obtain a matching $F$ that saturates all remaining job vertices of $G'(\tilde{\mathbf{x}})$ apply the procedure described in the proof of Lemma 3.4.2. Note that any polynomial-time maximum matching algorithm for bipartite graphs can also be used, but the method we described is efficient and exploits properties of any BFS in $LP_1(P, t = d_{best})$. For each edge $\{i, j\} \in F$, schedule job $j$ on machine $i$. Since the matching $F$ saturates each vertex $j \in J'$, the algorithm returns a schedule that contains all $n$ jobs, and terminates in polynomial time. The algorithm then terminates with a feasible schedule.

## 3.5 Algorithm

In this section, we first present the 2-approximation algorithm by Lenstra *et al.* [33] as we have presented all the essential pieces needed to understand this algorithm. Following this, we give a tight example, and provide an example execution of the 2-approximation algorithm.

**Theorem 3.5.1** (Lenstra *et al.* [33], Theorem 2)**.** *There is a 2-approximation algorithm for $R||C_{max}$.*

*Proof.* Let there be an arbitrary $m \times n$ processing requirement matrix $P$. First we describe the polynomial-time algorithm, then show that it has an approximation factor of 2. Pseudocode for the algorithm is provided as Algorithm 3.1.1.

To begin, the algorithm constructs a greedy schedule by assigning each job to a machine on which the job takes the least processing time. Denote the makespan

of the greedy schedule as $\kappa$. Next, the algorithm uses the binary search procedure described in Section 3.2 over the range $[\lfloor \kappa/m \rfloor, \kappa]$. The binary search procedure finds in polynomial time the smallest value $t = d_{best}$ that $LP_1(P, t)$ is feasible. Since $d_{best}$ is the smallest value for $t$ that $LP_1(P, t)$ is feasible, and $\lfloor \kappa/m \rfloor \leq OPT(P)$ by Lemma 3.2.1, $d_{best} \leq OPT(P)$.

Then, the algorithm finds a BFS $(\tilde{\mathbf{x}}, s_1, \ldots, s_m)$ of $LP_1(P, t = d_{best})$, where $\tilde{\mathbf{x}}$ are the decision variables and $s_1, \ldots, s_m$ are the slack variables of $LP_1(P, t = d_{best})$. The algorithm then considers the decision variables $\tilde{\mathbf{x}}$ in the BFS. Using the decision variables, the procedure uses Construction 3.3.2 to build in polynomial time a bipartite graph $G(\tilde{\mathbf{x}}) = (M \cup J, E)$, where $E = \{\{i, j\} \mid i \in M,\ j \in J,\ \tilde{x}_{i,j} > 0\}$. The algorithm assigns all the integrally set jobs by considering each job vertex $j \in J$. For each $j \in J$, if $deg_{G(\tilde{\mathbf{x}})}(j) = 1$ and $\{i, j\} \in E$, schedule job $j$ on machine $i$. Note that this assignment corresponds to when decision variable $\tilde{x}_{i,j} = 1$. Upon scheduling each job that satisfied this condition, remove job vertex $j$ from $J$. Call the resulting set of job vertices $J'$ and set of edges $E'$, then denote the bipartite graph as $G'(\tilde{\mathbf{x}}) = (M \cup J', E')$. By Lemma 3.4.1, there remains at most $m$ fractionally set jobs to be assigned, and $|J'| \leq m$. Assign the fractionally set jobs by finding a matching $F$ in $G'(\tilde{\mathbf{x}})$ in polynomial time as described in Lemma 3.4.2. For each edge $\{i, j\} \in F$, assign job $j$ to machine $i$. Return the schedule, and then terminate.

Now, let us show that this polynomial-time algorithm has an approximation factor of 2. As previously stated, when the smallest value $d_{best}$ is found by the algorithm that $LP_1(P, t = d_{best})$ is feasible, $d_{best} \leq OPT(P)$. So, when the algorithm obtains a BFS $(\tilde{\mathbf{x}}, s_1, \ldots, s_m)$ of $LP_1(P, t = d_{best})$, the algorithm schedules an integrally set

job whenever $\tilde{x}_{i,j} = 1$. The assignment of integrally set jobs produces a partial schedule of length at most $d_{best}$, because the BFS $(\tilde{\mathbf{x}}, s_1, \ldots, s_m)$ is a feasible solution to $LP_1(P, t = d_{best})$. Only the fractionally set jobs remain after all the integrally set jobs are scheduled by the algorithm. By Lemma 3.4.1, there are at most $m$ fractionally set jobs. The matching $F$ assigns one fractionally set job to each machine. Observe that an edge $\{i, j\} \in E'$ appears in $G'(\tilde{\mathbf{x}})$ if $0 < \tilde{x}_{i,j} < 1$. This implies that $p_{i,j} < t = d_{best}$ for any fractionally set job $j$ assigned to machine $i$ as a consequence of the matching $F$. Thus, when the schedule contains all $n$ jobs, the schedule has length at most $d_{best} + d_{best} = 2 \cdot d_{best} \leq 2 \cdot OPT(P)$. Therefore, this is a 2-approximation algorithm for $R||C_{max}$. □

Consider the following tight example for the 2-approximation algorithm presented by Lenstra *et al.* [33]. Let there be $m^2 - m + 1$ jobs and $m$ identical parallel machines. Then, let job 1 take $m$ time units, and every other job take one time unit. Notice that the optimal makespan for such an instance is $m$ because one machine is assigned job 1, then the remaining $m^2 - m = m(m-1)$ jobs of length one are assigned to the remaining machines to respect the makespan of $m$. When a greedy schedule is produced, and the binary search procedure is carried out, the smallest value $d_{best}$ for which $LP_1(P, t)$ results in a feasible LP is $m$. Job 1 can distribute one time unit of its processing to each machine while maintaining a feasible solution to $LP_1(P, d_{best})$. When $d_{best} = m$, a possible BFS with decision variables $\tilde{\mathbf{x}}$ in $LP_1(P, d_{best})$ can consist of one time unit of job 1 (which is of length $m$) on each machine, and then $(m-1)$ unit time jobs assigned to each machine. When the bipartite graph $G(\tilde{\mathbf{x}})$ is constructed through rounding, notice that job 1 must be forced onto a machine that also assigns

$(m-1)$ jobs that take one time unit each when the matching is found. This results in a schedule with makespan of $2m-1$.

Now we give an example execution of the 2-approximation algorithm by Lenstra *et al.* [33].

**Example 3.5.2.** Consider the following processing requirement matrix

$$P = \begin{pmatrix} 5 & 2 & 3 & 3 & 2 \\ 1 & 3 & 2 & 1 & 3 \\ 2 & 2 & 3 & 4 & 1 \end{pmatrix}.$$

First, a greedy schedule is produced by assigning each job to a machine for which it takes the least time. Such a greedy schedule can assign job 2 to machine 1, jobs 1, 3, and 4 to machine 2, and job 5 to machine 3. As a result, we can obtain a feasible schedule with makespan of four (i.e., $\kappa = 4$). Next, the algorithm assigns an initial upper bound $U = 4$, and initial lower bound $L = \lfloor 4/3 \rfloor = 1$. So $d = \lfloor (L+U)/2 \rfloor = \lfloor (4+1)/2 \rfloor = 2$. We test the feasibility of $LP_1(P, t)$, when $t = d = 2$.

$$\begin{aligned}
&\text{minimize} \quad 0 && \text{(Testing feasibility)}\\
&\text{subject to } x_{1,1} + x_{2,1} + x_{3,1} = 1\\
&\qquad x_{1,2} + x_{2,2} + x_{3,2} = 1\\
&\qquad x_{1,3} + x_{2,3} + x_{3,3} = 1\\
&\qquad x_{1,4} + x_{2,4} + x_{3,4} = 1\\
&\qquad x_{1,5} + x_{2,5} + x_{3,5} = 1\\
&\qquad 5x_{1,1} + 2x_{1,2} + 3x_{1,3} + 3x_{1,4} + 2x_{1,5} + s_1 = 2\\
&\qquad 1x_{2,1} + 3x_{2,2} + 2x_{2,3} + 1x_{2,4} + 3x_{2,5} + s_2 = 2\\
&\qquad 2x_{3,1} + 2x_{3,2} + 3x_{3,3} + 4x_{3,4} + 1x_{3,5} + s_3 = 2\\
&\qquad x_{1,1} = 0,\ x_{1,3} = 0,\ x_{1,4} = 0\\
&\qquad x_{2,2} = 0,\ x_{2,5} = 0\\
&\qquad x_{3,3} = 0,\ x_{3,4} = 0\\
&\qquad x_{i,j} \geq 0,\ for\ j = 1,2,3,4,5,\ i = 1,2,3.
\end{aligned}$$

After considering the artificial constraints, if constraints "$0x_{1,3} + x_{2,3} + 0x_{3,3} = 1$" and "$0x_{1,4} + x_{2,4} + 0x_{3,4} = 1$" are satisfied, then constraint "$1x_{2,1} + 0 \cdot 3x_{2,2} + 2x_{2,3} + 1x_{2,4} + 0 \cdot 3x_{2,5} + s_1 = 2$" cannot be satisfied as $x_{2,1} + 0 + 2 \cdot 1 + 1 + 0 + s_1 \geq 3$. So, $LP_1(P, d = 2)$ is infeasible. So the algorithm sets $L = d + 1 = 3$. Now, this process is repeated again using $U = 4$ and $L = 3$.

$$\begin{aligned}
\text{minimize} \quad & 0 && \text{(Testing feasibility)}\\
\text{subject to} \quad & x_{1,1} + x_{2,1} + x_{3,1} = 1\\
& x_{1,2} + x_{2,2} + x_{3,2} = 1\\
& x_{1,3} + x_{2,3} + x_{3,3} = 1\\
& x_{1,4} + x_{2,4} + x_{3,4} = 1\\
& x_{1,5} + x_{2,5} + x_{3,5} = 1\\
& 5x_{1,1} + 2x_{1,2} + 3x_{1,3} + 3x_{1,4} + 2x_{1,5} + s_1 = 3\\
& 1x_{2,1} + 3x_{2,2} + 2x_{2,3} + 1x_{2,4} + 3x_{2,5} + s_2 = 3\\
& 2x_{3,1} + 2x_{3,2} + 3x_{3,3} + 4x_{3,4} + 1x_{3,5} + s_3 = 3\\
& x_{1,1} = 0\\
& x_{3,4} = 0\\
& x_{i,j} \geq 0, \ for\ j = 1,2,3,4,5, \ i = 1,2,3.
\end{aligned}$$

The procedure sets $d = \lfloor (L+U)/2 \rfloor = \lfloor (3+4)/2 \rfloor = 3$. So the algorithm next tests if $LP_1(P,t)$ is feasible when $t = d = 3$. The LP $LP_1(P,3)$ can be shown to be feasible. So the algorithm assigns $d_{best} = 3$, and sets $U = 3$. Since $U = L = 3$, the algorithm proceeds to the next step of the algorithm with $d_{best} = 3$. The method finds a BFS $(\tilde{\mathbf{x}}, s_1, \ldots, s_m)$ in $LP_1(P, t = d_{best})$. In our example, one such BFS $(\tilde{\mathbf{x}}, s_1, \ldots, s_m)$ in

$LP_1(P, t = d_{best} = 3)$ has each slack variable set to zero, and decision variables

$$\tilde{\mathbf{x}} = \begin{pmatrix} 0 & 0.5 & 0 & 0 & 1 \\ 1 & 0.2 & 0.2 & 1 & 0 \\ 0 & 0.3 & 0.8 & 0 & 0 \end{pmatrix}.$$

As expected, $LP_1(P, 3)$ does not use any entries of $P$, where $p_{i,j} > 3$.

Now, Algorithm 3.1.1 applies Construction 3.3.2 using the decision variables $\tilde{\mathbf{x}}$ of the BFS to build a bipartite graph $G(\tilde{\mathbf{x}}) = (M \cup J, E)$. Eight of the decision variables in the BFS are non-zero, so there are eight edges in $G(\tilde{\mathbf{x}})$. The bipartite graph $G(\tilde{\mathbf{x}})$ is shown in Figure 3.1.

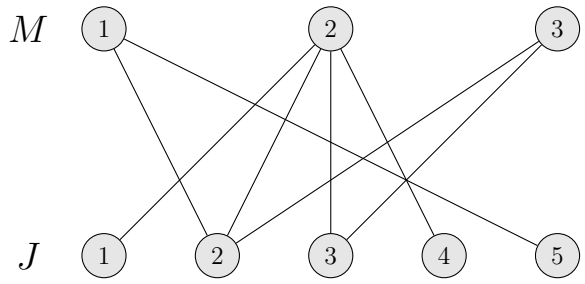



Figure 3.1: A bipartite graph $G(\tilde{\mathbf{x}}) = (M \cup J, E)$ built from Construction 3.3.2 for an example execution of the 2-approximation algorithm by Lenstra *et al.* [33].

Algorithm 3.1.1 considers each $j \in J$. If $deg_{G(\tilde{\mathbf{x}})}(j) = 1$ and $\{i, j\} \in E$, then job $j$ is scheduled on machine $i$. Upon scheduling each job, $E = E \setminus \{i, j\}$ and $J = J \setminus \{j\}$. In this example, jobs 1, 4, and 5 are assigned at this step. After this process, Algorithm 3.1.1 obtains a subgraph $G'(\tilde{\mathbf{x}})$. The bipartite graph $G'(\tilde{\mathbf{x}})$ is shown in Figure 3.2.

For this example, there is only one pseudotree. Algorithm 3.1.1 finds a matching $F$ that saturates all remaining job vertices in $G'(\tilde{\mathbf{x}})$. The first phase of this is applied by checking if any machine vertices have degree of one. Notice that machine vertex 1

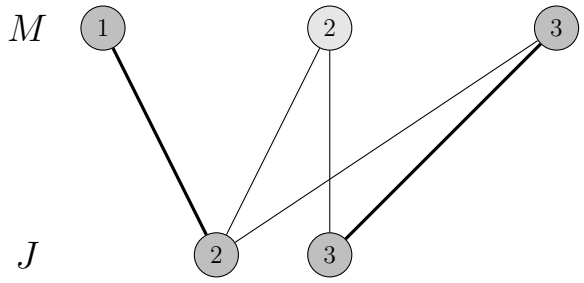



Figure 3.2: The subgraph $G'(\tilde{\mathbf{x}}) = (M \cup J', E')$ in the example execution of the 2-approximation algorithm by Lenstra *et al.* [33]. For our instance, a matching $F$ is found that saturates all remaining job vertices in $G'(\tilde{\mathbf{x}})$. Matching $F$ is shown with dark edges and shaded vertices.

has degree one, so it is matched with job vertex 2. Upon removing job vertex 2 and any of its edges, notice that either machine vertex remaining can be used to pair the last job vertex. In this example we match job 3 with machine 3. Since there are no more job vertices, the DFS procedure is not needed to match alternating edges. Matching $F$ consists of all remaining job vertices. For the final step, the algorithm assigns all remaining jobs based on the matching $F$, then returns the schedule shown in Figure 3.3, and terminates. The optimal makespan for this instance is three, and the makespan of this schedule is four. ■

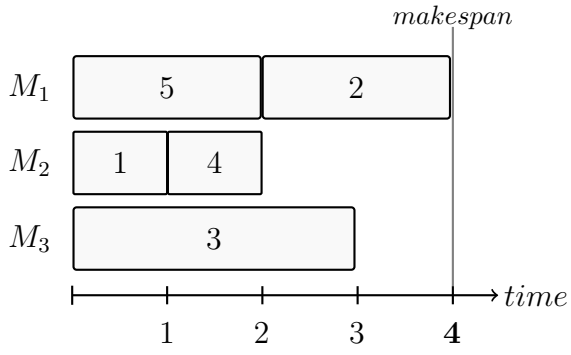



Figure 3.3: The schedule produced by the 2-approximation algorithm for $R||C_{max}$ by Lenstra *et al.* [33] when applied to an example instance.

# Chapter 4

# Results for Subclasses of $R||C_{max}$, and Related Scheduling Problems

In Chapter 2, we presented a brief summary of approximation algorithms and hardness of approximation results for $R||C_{max}$. Also, in Chapter 3, we gave a 2-approximation algorithm for $R||C_{max}$ by Lenstra *et al.* [33]. In this chapter, we present known results for subclasses of $R||C_{max}$, and scheduling problems that are related to $R||C_{max}$. First, we thoroughly describe algorithms for subclasses of $R||C_{max}$ that can be solved in polynomial time. Following this, we summarize known results for NP-hard subclasses of $R||C_{max}$ and some related intractable scheduling problems by machine environment.

## 4.1 Known Tractable Subclasses of $R||C_{max}$

We now present known results for subclasses of $R||C_{max}$ that can be solved in polynomial time. In this section we focus on three tractable results that are given by Lenstra *et al.* [33]. Three subclasses of $R||C_{max}$ for which we present polynomial-time algorithms are $R|p_{i,j} = 1|C_{max}$, $R|p_{i,j} \in \{1, \infty\}|C_{max}$, and $R|p_{i,j} \in \{1, 2\}|C_{max}$.

### 4.1.1 $R|p_{i,j} = 1|C_{max}$ and $R|p_{i,j} \in \{1, \infty\}|C_{max}$

In 1990, Lenstra *et al.* [33] investigated subclasses that have restricted processing times. To begin, the authors remarked two restricted cases that are stated without proof. We give these two remarks as Theorem 4.1.1 and Theorem 4.1.3.

**Theorem 4.1.1** (Lenstra *et al.* [33])**.** *$R|p_{i,j} = 1|C_{max}$ can be solved in polynomial time.*

To solve any instance of $R|p_{i,j} = 1|C_{max}$, apply the list scheduling algorithm presented in Theorem 1.5.1. Since each job completes in one time unit, such a greedy algorithm produces optimal schedules as there is no contention amongst the jobs. For this subclass, the algorithm can be improved to have time-complexity $\Theta(n)$. Instead of finding the machine with least load each time, cyclically assign jobs to machines.

**Example 4.1.2.** For example, consider the processing requirement matrix

$$P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Each job is assigned by the greedy algorithm to a machine with least load. For this instance, a schedule with an optimal makespan of two is produced. An assignment obtained by cyclically assigning jobs to machines is

$$\begin{pmatrix} \textcircled{1} & 1 & 1 & \textcircled{1} & 1 \\ 1 & \textcircled{1} & 1 & 1 & \textcircled{1} \\ 1 & 1 & \textcircled{1} & 1 & 1 \end{pmatrix},$$

where each circled entry corresponds to a job $j$ being assigned to a machine $i$, for $1 \leq i \leq 3$ and $1 \leq j \leq 5$. The schedule constructed by the greedy algorithm is shown in Figure 4.1.

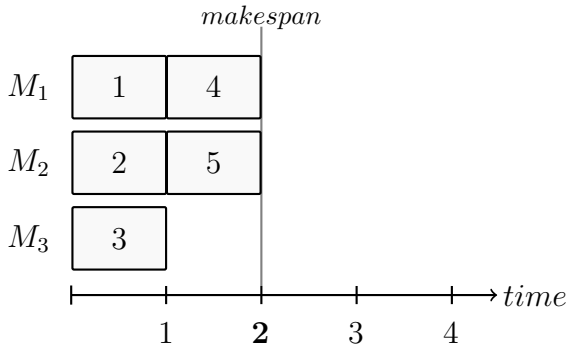



Figure 4.1: An optimal schedule for an example processing requirement matrix of $R|p_{i,j} = 1|C_{max}$.

■

Another subclass of $R||C_{max}$ that Lenstra *et al.* [33] state as polynomial–time solvable is $R|p_{i,j} \in \{1, \infty\}|C_{max}$. We describe an algorithm that solves $R|p_{i,j} \in \{1, \infty\}|C_{max}$, then provide an example. We state the following theorem without proof.

**Theorem 4.1.3** (Lenstra *et al.* [33])**.** *$R|p_{i,j} \in \{1, \infty\}|C_{max}$ can be solved in polynomial time.*

**Algorithm 4.1.4** Polynomial-time Algorithm for $R|p_{i,j} \in \{1, \infty\}|C_{max}$

```
1: procedure MakespanUPMsAssignment(P, m, n)
2:     Let X be a m × n matrix with zero entries;        ▷ m × n assignment matrix
3:     M_max := ∅;                                       ▷ Will contain a maximum matching
4:     feasible := false;                                ▷ Stays false until a feasible schedule is found
5:     n' := 0;                                          ▷ Number of jobs to be scheduled
6:     for i := 1; i ≤ m; i++ do                         ▷ Counting jobs that are not redundant
7:         counter := 0;
8:         for j := 1; j ≤ n; j++ do
9:             if p_{i,j} == ∞ then
10:                counter := counter + 1;
11:            end if
12:        end for
13:        if counter == m then
14:            n' := n' + 1;
15:        end if
16:    end for
17:    d := ⌈n'/m⌉;                                      ▷ Proposed deadline
18:    while ¬feasible do                                ▷ Do until a feasible schedule can be found
19:        Apply Construction 4.1.5 to obtain G = (M ∪ J, E);
20:        Let M_max be a maximum matching of G;
21:        if |M_max| < n' then
22:            d := d + 1;        ▷ Feasible schedule cannot be found yet, increment d
23:        else
24:            feasible := true;                         ▷ |M_max| = n'
25:        end if
26:    end while
27:    for each {k, j} ∈ M_max, where k ∈ M and j ∈ J do    ▷ Construct schedule
28:        Let i be the machine that corresponds to machine vertex k in Construction 4.1.5;
29:        X_{i,j} := 1;                                 ▷ Assign job j to machine i
30:    end for
31:    return X;                                         ▷ Return schedule
32: end procedure
```

Let us describe an algorithm that solves this problem in polynomial time. Pseudocode for this algorithm is provided as Algorithm 4.1.4. Assume the algorithm is provided an arbitrary instance of $R|p_{i,j} \in \{1, \infty\}|C_{max}$. If a job cannot be assigned

to any machine, then we say a job is *redundant*. The processing requirement matrix $P$ is checked for any redundant jobs. All redundant jobs are removed from $P$ by the algorithm, and, as a result, an $m \times n'$ processing requirement matrix is obtained, where $n' \leq n$. Denote the modified processing requirement matrix as $P'$. The algorithm assigns a deadline variable $d$ to be $d = \lceil n'/m \rceil$. The deadline $d$ represents the minimum makespan. Next, the algorithm applies Construction 4.1.5 to build a bipartite graph $G = (M \cup J, E)$ using processing requirement matrix $P'$.

**Construction 4.1.5.** Given a $m \times n'$ processing requirement matrix $P'$ with $p_{i,j} \in \{1, \infty\}$ and a deadline $d$, create a bipartite graph $G = (M \cup J, E)$, where $M$ consists of *machine vertices* and $J$ consists of *job vertices*. For each job in $P'$, create a job vertex. For each machine, create $d$ machine vertices. For each entry $p_{i,j} = 1$ in $P'$, create $d$ edges from a job vertex corresponding to job $j$ to each machine vertex corresponding to machine $i$. This correspondence of machine vertices with machines can be stored in a lookup table. Since $d \leq n'$, this construction takes a polynomial number of steps to build. ✠

Upon creating a bipartite graph $G$ by applying Construction 4.1.5, the algorithm finds a maximum matching $M_{max}$ of $G$. A maximum matching $M_{max}$ of $G$ can be found in polynomial time [19]. Next, a schedule is constructed using the maximum matching $M_{max}$. If there is an edge connecting a job vertex $j \in J$ and a machine vertex that corresponds to a machine $i$, then the algorithm schedules job $j$ on machine $i$. The algorithm does this for each edge in $M_{max}$. If a schedule containing all the non-redundant jobs is not found, there are not enough machine vertices to saturate all the job vertices in the matching. If such is the case, then $d$ is not the minimum

makespan and the makespan must be larger. If the algorithm does not saturate all the job vertices for a given value of $d$, then the makespan is larger than $d$.

If $|M_{max}| < n'$, the algorithm increases $d$ by one and repeats the previous step starting at applying Construction 4.1.5 to $P'$. A schedule is found in polynomial time, as $d \leq n' \leq n$. If $|M_{max}| = n'$, the algorithm returns the constructed schedule and terminates. A schedule is found with makespan $d$ by this procedure.

Let us apply this algorithm to an example instance of $R|p_{i,j} \in \{1, \infty\}|C_{max}$.

**Example 4.1.6.** Consider the processing requirement matrix

$$P = \begin{pmatrix} 1 & 1 & \infty & 1 & \infty \\ \infty & \infty & 1 & \infty & \infty \end{pmatrix}.$$

The algorithm begins by removing any redundant jobs that cannot be scheduled. Upon removing any redundant jobs, $n' = 4$, and

$$P' = \begin{pmatrix} 1 & 1 & \infty & 1 \\ \infty & \infty & 1 & \infty \end{pmatrix}.$$

In the next step, the algorithm assigns the deadline $d = \lceil n'/m \rceil = \lceil 4/2 \rceil = 2$. First, a bipartite graph $G = (M \cup J, E)$ is constructed as shown in Figure 4.2 based on the one entries in $P'$. There are $d = 2$ vertices representing each machine for $M$. A maximum matching $M_{max}$ is found for $G$ by the algorithm, and $|M_{max}| = 3$. Since $|M_{max}| < n' = 4$, the deadline $d$ is incremented by one, and the algorithm repeats the procedure once again for $d = 3$.

In this iteration, a bipartite graph $G = (M \cup J, E)$ is constructed as seen in Figure 4.3. Notice that there are $d = 3$ vertices representing each machine, and when a maximum matching $M_{max}$ is found by the algorithm, $|M_{max}| = 4$.

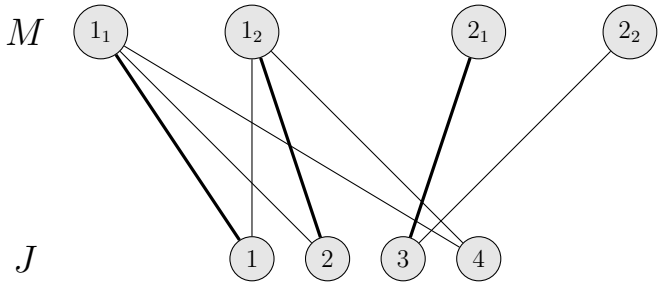



Figure 4.2: The bipartite graph $G = (M \cup J, E)$ constructed for $d = 2$ using the example $R|p_{i,j} \in \{1, \infty\}|C_{max}$ instance. The edges of a maximum matching $M_{max}$ of $G$ are shown with thick edges.

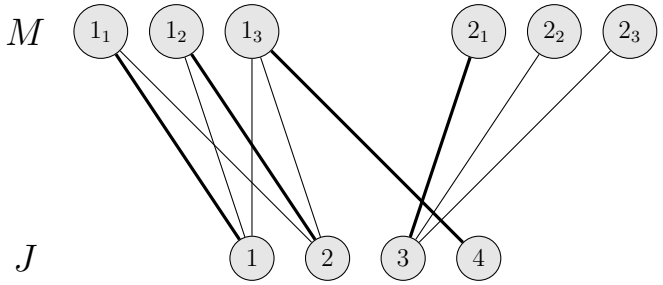



Figure 4.3: The bipartite graph $G = (M \cup J, E)$ construction for $d = 3$ using the example instance of $R|p_{i,j} \in \{1, \infty\}|C_{max}$. A maximum matching $M_{max}$ of $G$ is shown with thick edges.

Since $|M_{max}| = n' = 4$, a schedule is constructed based on $M_{max}$, then the algorithm terminates. Using the edges in maximum matching $M_{max}$, jobs 1, 2, and 3 are assigned to machine 1; and job 4 is assigned to machine 2. Such a schedule has makespan of three, which is optimal. The optimal schedule produced is shown in Figure 4.4.

■

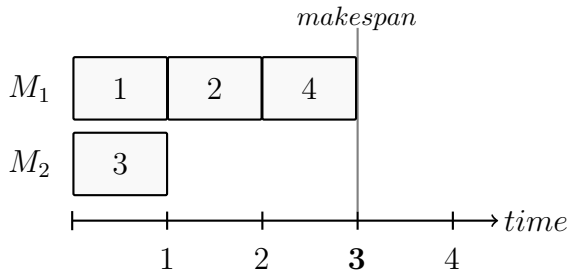



Figure 4.4: An optimal schedule for an example processing requirement matrix of $R|p_{i,j} \in \{1, \infty\}|C_{max}$.

### 4.1.2 $R|p_{i,j} \in \{1, 2\}|C_{max}$

The last subclass of $R||C_{max}$ for which we present tractable results in this chapter has processing times of lengths one and two. In 1990, Lenstra *et al.* [33] showed that $R|p_{i,j} \in \{1, 2\}|C_{max}$ can be solved in polynomial time. At the heart of their algorithm are two ideas. First, to pair jobs of length one together, then treat each paired job as a job of length two. Second, to then treat the remaining tasks as a partial scheduling problem with unit processing times, where $p_{i,j} = 2$.

Before we present the polynomial-time algorithm by Lenstra *et al.* for $R|p_{i,j} \in \{1, 2\}|C_{max}$, we consider a graph theoretic problem we call the maximum 2-1 constrained bipartite subgraph problem (maximum 2-1 CBS). Finding an optimal solution to a maximum 2-1 CBS instance is a core subroutine for "lumping" jobs of length one together in the algorithm.

First, we define a 2-1 constrained bipartite subgraph (2-1 CBS). Given a bipartite graph $G = (S \cup T, E)$, a *2-1 CBS* $H = (S' \cup T', E')$ is a subgraph of $G$, where $S' \subseteq S$, $T' \subseteq T$, $E' \subseteq E$, with $deg_H(s') = 2$ for $s' \in S'$, and $deg_H(t') = 1$ for $t' \in T'$. The graph $G$ does not need to be connected. Next, we give the *maximum 2-1 CBS problem*. Given a bipartite graph $G = (S \cup T, E)$, the goal of the maximum 2-1 CBS

problem is to find a 2-1 CBS $H = (S' \cup T', E')$, where the number of edges $|E'|$ is maximized. Let us give an example to illustrate this problem.

**Example 4.1.7.** Suppose we are given the bipartite graph $G = (S \cup T, E)$ shown on the left in Figure 4.5 as an instance of the maximum 2-1 CBS problem. As shown in the middle in Figure 4.5, one possible feasible solution could be a subgraph $H = (S' \cup T', E')$, as every vertex $s' \in S' \subseteq S$ has degree two and every $t' \in T' \subset T$ has degree one. Finally, an optimal solution for our instance can be $H^* = (S^* \cup T^*, E^*)$, as shown in Figure 4.5 on the right.

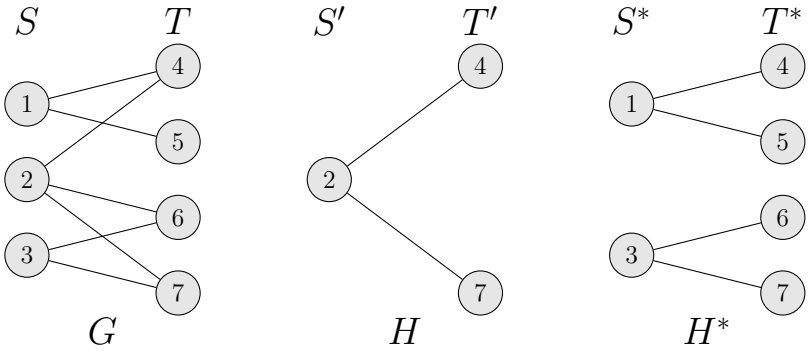



Figure 4.5: Three bipartite graphs $G$, $H$, and $H^*$; the input graph $G = (S \cup T, E)$ given as an instance of the maximum 2-1 CBS problem (left); a feasible solution $H = (S' \cup T', E')$ for the given instance (middle); an optimal solution $H^* = (S^* \cup T^*, E^*)$ for the maximum 2-1 CBS instance (right).

■

We next show that the maximum 2-1 CBS problem is polynomial-time solvable. The algorithm we present employs a construction that is based on one for degree-constrained subgraphs by Gabow [12], and one for perfect $b$-matchings presented by Schrijver [40] and Tucker [44].

**Theorem 4.1.8** (Gabow [12], Schrijver [40], Tucker [44])**.** *The maximum 2-1 CBS*

*problem can be solved in polynomial time.*

*Proof.* First, let us describe a polynomial-time algorithm for the maximum 2-1 CBS problem, then show its correctness. Pseudocode for this algorithm is given as Algorithm 4.1.9. Assume we are given an arbitrary bipartite graph $G = (S \cup T, E)$. Using $G$, construct a graph $G_2$ by invoking Construction 4.1.10. This construction creates $G_2$ in polynomial time. Note that the number of vertices and number of edges in the new graph $G_2$ are $2|S| + |T| + 2|E|$, and $4|E| + |S|$, respectively.

**Construction 4.1.10.** Assume we are given a bipartite graph $G = (S \cup T, E)$, where $S = \{1, 2, \ldots, |S|\}$ and $T = \{1, 2, \ldots, |T|\}$. Create a graph $G_2$ as follows. For each $s \in S$, create two vertices $s_1$ and $s_2$, and include an edge between these two vertices. Also, create a vertex $t_1$ for each $t \in T$. Next, for every $e = \{s, t\} \in E$, create two vertices $e_s$ and $e_t$, and include an edge $\{e_s, e_t\}$. Also, add edges $\{s_1, e_s\}$, $\{s_2, e_s\}$, and $\{e_t, t_1\}$. ※

**Example 4.1.11.** Continuing Example 4.1.7, we apply Construction 4.1.10 with bipartite graph $G = (S \cup T, E)$, which is shown on the left in Figure 4.6. As a result graph $G_2$ is obtained. We show this graph on the right in Figure 4.6. ■

Next, the algorithm finds a maximum matching $M_{max}$ in $G_2$, which can be done in polynomial time [10; 36]. In particular, a maximum matching in $G_2$ can be found in $\Theta((|S||T|)^{3/2})$ steps. Let us consider the size of the maximum matching before continuing with our algorithm. Since the size of a maximum matching in $G_2$ cannot be more than the number of edges in $G_2$,

$$|M_{max}| \leq 4|E| + |S| \leq 4|S||T| + |S| \in \Theta(|S||T|).$$

**Algorithm 4.1.9** Polynomial-time Algorithm for the Maximum 2-1 CBS problem

1: **procedure** MAXIMUM2-1CBS($G = (S \cup T, E)$)
2: $M_{max} := \varnothing$; ▷ Will contain a maximum matching
3: $M_{2-1} := \varnothing$; ▷ Will contain the edges of a 2-1 CBS
4: Apply Construction 4.1.10 to obtain $G_2$;
5: Let $M_{max}$ be a maximum matching of $G_2$;
6: **for each** $s \in S$ **do** ▷ Construct 2-1 CBS
7: **if** $deg_G(s) \geq 2$ **then**
8: $v := 0$;
9: $w := 0$;
10: **for each** $\{s, z\} \in E$ **do** ▷ Consider each edge incident to vertex $s$
11: **if** $(v == 0) \vee (w == 0)$ **then**
12: $b_1 := \{s_1, \{s_1, z\}_s\}, \{\{s_1, z\}_z, z\} \in M_{max}$?; ▷ In $M_{max}$?
13: $b_2 := \{s_2, \{s_2, z\}_s\}, \{\{s_2, z\}_z, z\} \in M_{max}$?; ▷ Also in $M_{max}$?
14: **if** $(b_1 \vee b_2)$ **then**
15: **if** $(v == 0)$ **then**
16: $v := z$;
17: **else**
18: $w := z$;
19: **end if**
20: **end if**
21: **end if**
22: **end for**
23: **if** $(v > 0) \wedge (w > 0)$ **then**
24: $M_{2-1} := M_{2-1} \cup \{s, v\} \cup \{s, w\}$; ▷ Found a pair of edges
25: **end if**
26: **end if**
27: **end for**
28: Let $H = (S' \cup T', E' = M_{2-1})$ be an induced subgraph of $M_{2-1}$, formed by the vertices at the end points of each edge in $M_{2-1}$, and edges $M_{2-1}$; ▷ Can be done by checking each edge of $M_{2-1}$
29: **return** $H$; ▷ Return the subgraph $H$
30: **end procedure**

Let $M_{2-1}$ be a set of edges created by the algorithm to produce an induced subgraph $H = (S' \cup T', E' = M_{2-1})$. Assume there are vertices $v, w, z \in T$ in $G$, where $v \neq w$. Consider each $s \in S$. Then, if there is a pair of edges $\{s, v\}, \{s, w\} \in E$, such that $\{s_1, \{s, v\}_s\}$, $\{\{s, v\}_v, v_1\}$, $\{s_2, \{s, w\}_s\}$, $\{\{s, w\}_w, w_1\} \in M_{max}$, the

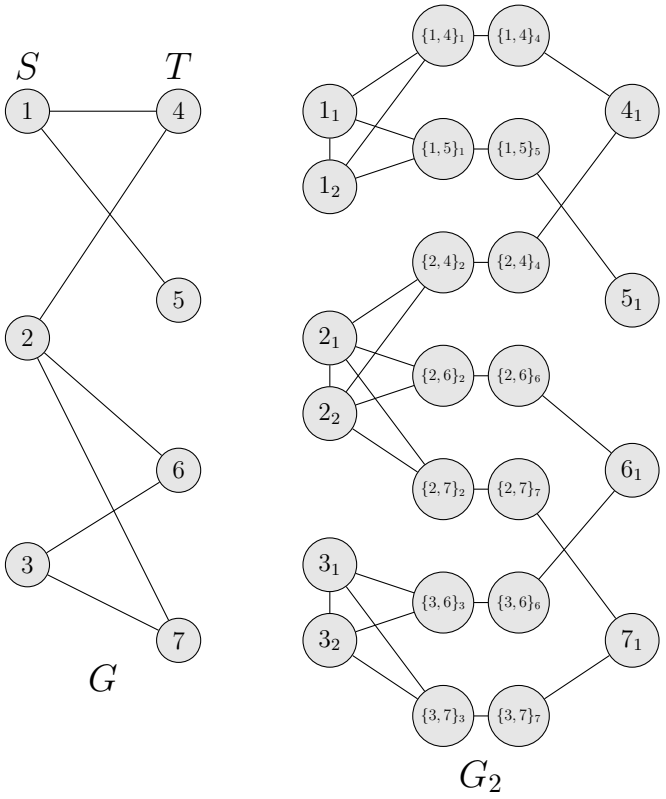



Figure 4.6: Given the bipartite graph $G = (S \cup T, E)$ from Example 4.1.7 (left), we give the new graph $G_2$ produced by applying Construction 4.1.10 (middle).

algorithm adds edges $\{s, v\}$ and $\{s, w\}$ to $M_{2-1}$ (lines 10–25 in Algorithm 4.1.9). The algorithm checks this condition for each $s \in S$ by considering each edge $\{s, z\} \in E$ in $G$, and observing if the edge satisfies either $\{s_1, \{s, z\}_s\}, \{\{s, z\}_z, z_1\} \in M_{max}$ or $\{s_2, \{s, z\}_s\}, \{\{s, z\}_z, z_1\} \in M_{max}$. This requires checking each edge in the maximum matching $M_{max}$. Keep track of $z$ each time when the test is passed as $v$, then $w$. Note that no more than two edges incident to some $s \in S$ can satisfy either condition, because maximum matching $M_{max}$ can saturate at most two vertices representing $s$ in $G_2$; namely, $s_1$ and $s_2$. Though this is not required, if the $deg_G(s) < 2$, one can proceed to the next vertex in $S$. Upon checking each $s \in S$, return the induced

subgraph $H$ of $M_{2-1}$. This algorithm terminates in polynomial time. If the input graph $G$ is given as an adjacency list, this algorithm has worst-case time complexity of $O(|S||T||E|)$.

**Example 4.1.12.** Following from Example 4.6, a maximum matching in $G_2$ is found as shown on the left in Figure 4.7. Next, the algorithm checks each $s \in S$ in $G$, and attempts to find two vertices $v, w$ as we have described. When $s = 1$ and $s = 3$, two such vertices were successfully found. With $s = 1$, $v = 4$ and $w = 5$. Also, when $s = 3$, $v = 6$ and $w = 7$. So edges $\{1, 4\}$, $\{1, 5\}$, $\{3, 6\}$, and $\{3, 7\}$ were added to $M_{2-1}$. We show the resulting 2-1 CBS $H = (S' \cup T', E')$ on the right as thick edges and shaded vertices in Figure 4.7.

■

To complete our proof, we verify that $H$ from the algorithm is indeed a 2-1 CBS, and that the number of edges $|E'| = |M_{2-1}|$ in $H$ is maximized. Consider both the set of edges $M_{2-1}$ and the induced graph $H = (S' \cup T', E' = M_{2-1})$. To form $M_{2-1}$, the algorithm picked either zero or two edges that are incident to each vertex $s \in S$ in $G$. So, for each vertex $s' \in S'$, $deg_H(s') = 2$. Next, by the definition of a matching, at most one edge in $M_{max}$ saturates each vertex $t_1$ in $G_2$, for $t \in T$. As a consequence, for each $t' \in T'$, $deg_H(t') = 1$, and $H$ is a 2-1 CBS.

Suppose the number of edges $|E'| = |M_{2-1}|$ is not of maximum size. Then there is a 2-1 CBS $H^* = (S^* \cup T^*, E^*)$, where $|E^*| > |M_{2-1}|$. As a consequence, $H^*$ contains as many vertices from $s \in S$ as possible, and $|S^*| > |S'|$. Consider the graph $G_2$ constructed from bipartite graph $G = (S \cup T, E)$ using Construction 4.1.10. Each edge in $M_{2-1}$ was added because there were two vertices $v$ and $w$, such that

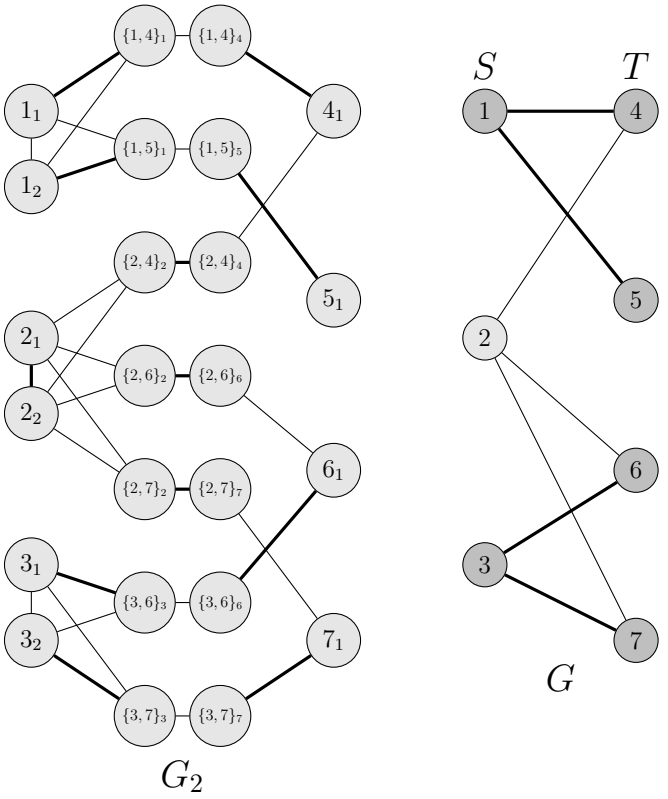



Figure 4.7: Upon building graph $G_2$ from bipartite graph $G = (S \cup T, E)$ by applying Construction 4.1.10, a maximum matching $M_{max}$ is found in $G_2$ (left). The maximum matching $M_{max}$ is shown with thick edges in $G_2$. As carried out by the algorithm, a 2-1 CBS $H = (S' \cup T', E')$ is found in $G$. Thick edges and shaded vertices show $H$ in $G$ (right). The subgraph $H$ is a maximum 2-1 CBS.

$\{s_1, \{s, v\}_s\}, \{\{s, v\}_v, v_1\}, \{s_2, \{s, w\}_s\}, \{\{s, w\}_w, w_1\} \in M_{max}$. Construct a matching $M^*_{max}$ with two steps. First, consider each pair $\{u, v\}, \{u, w\} \in E^*$, and include edges $\{u_1, \{u, v\}_u\}$,$\{\{u, v\}_v, v_1\}$ and $\{u_2, \{u, w\}_u\}$,$\{\{u, w\}_w, w_1\}$ in $M^*_{max}$, where $u \in S^*$. Second, match as many vertices as possible that remain in $G_2$ and include the resulting edges in $M^*_{max}$. Since $M_{max}$ is a maximum matching in $G_2$, $|M^*_{max}| \leq |M_{max}|$. Next, decompose $M_{max}$ and $M^*_{max}$ in the following manner. For any $s \in S$, consider an induced subgraph $Z(s) = (Z(s)_V, Z(s)_E)$ formed by edges $\{s_1, s_2\}$ and all vertex-disjoint paths from $s_1$ or $s_2$ to each vertex $t_1$ in $G_2$, where $t$ is a neighbour of $s$ in $G$.

Decompose the edges of $M_{max}$ and $M^*_{max}$ into sets we denote $M(s)_{max}$ and $M^*(s)_{max}$, where each set contains the edges of $M_{max}$ and $M^*_{max}$ found in $Z(s)$, respectively. Clearly, $M_{max} = \bigcup_{s \in S} M(s)_{max}$ and $M^*_{max} = \bigcup_{s \in S} M^*(s)_{max}$. As at most one edge can be matched with one vertex $t_1$ that corresponds to $t \in T$, $\bigcap_{s \in S} M(s)_{max} = \varnothing$ and $\bigcap_{s \in S} M^*(s)_{max} = \varnothing$.

Suppose we found a maximum matching $M_{Z(s)}$ in $Z(s)$. We prove that if $deg_G(s) \geq 2$, $|M_{Z(s)}| = deg_G(s) + 2$. Observe that $K_{2,deg_G(s)}$ is a subgraph of $Z(s)$, and that the length of the longest vertex-disjoint path in any $Z(s)$ including one of $s_1$ or $s_2$ to some $t_1$ is at most 3. Two vertices can be matched along two such paths, so four edges will be matched. Next, remove the vertices at the end points of each matched edge as we cannot match any edges incident to such vertices. This leaves $deg_G(s) - 2$ connected components, each containing at most three vertices. Since only one edge can be included in the maximum matching for each connected component, $deg_G(s) - 2$ additional edges are matched. So, $|M_{Z(s)}| = deg_G(s) + 2$.

Observe that the only vertices that intersect between any $Z(s)$ and $Z(g)$ when $s \neq g$ correspond to each $t \in T$. If an edge is matched with some $t_1$ in $Z(g)$, and $t_1$ is a vertex of $Z(s)$, then this vertex cannot be matched. So we consider a maximum matching in $Z(s)$ when there are some vertices corresponding to $t \in T$ removed due to being matched in some other $Z(g)$. Notice that a matching of size $deg_G(s) + 2$ can only be found as in our explanation before when there are two vertex-disjoint paths from $s_1$ and $s_2$ to some $t_1$ and $q_1$ without the first path traversing through $s_2$ and the second path traversing through $s_1$, where $t, q \in T$ and $q \neq t$. When this is not the case, the number of edges matched is at most $deg_G(s) + 1$, as there exists at most

one vertex-disjoint path from some $s_i$ to $t_1$ without traversing through some $s_j$ in $G_2$, where $i \neq j$. This matching is found by matching the edge connecting to $s_1$ and $s_2$, then considering the remaining vertices. If the vertices at the end points of the matched edge are removed, we are left with $deg_G(s)$ connected components, each with at most three vertices. In each connected component, at most $deg_G(s)$ vertices can be matched with an edge. When no path exists from some $s_i$ to $t_1$ without including $s_j$ (where $i \neq j$), $deg_G(s)+1$ edges are still matched. This is because the edge connecting $s_1$ and $s_2$ can be matched, and $2 \cdot deg_G(s)$ vertices remain, each with an edge connected between them that are matched. Thus, if a maximum matching were found in $G_2$, edges would be matched so that as many $Z(s)$ in $G_2$ have a maximum matching of size $deg_G(s) + 2$, and as few $Z(s)$ with maximum matching of size $deg_G(s) + 1$ as possible.

When a vertex $s \in S$ is included in a 2-1 CBS by the algorithm and $deg_G(s) \geq 2$, two vertex-disjoint paths with two matched edges each were found in $G_2$, and correspond to edges in $M_{max}$. But, $M_{max}$ is a maximum matching, and $|M^*_{max}| \leq |M_{max}|$. This implies either $E^* = M_{2-1}$ or $E^*$ has a vertex $s^*$, where $deg_{H^*}(s^*) > 2$; which violates the definition of a 2-1 CBS. So, the 2-1 CBS $H$ produced by the algorithm is a maximum 2-1 CBS. Therefore, the maximum 2-1 CBS problem is polynomial-time solvable. □

Now we present the polynomial-time algorithm by Lenstra *et al.* [33] for $R|p_{i,j} \in \{1, 2\}|C_{max}$. This algorithm employs a decision procedure that pairs as many jobs of length one on the machines while respecting a proposed deadline. The idea is to group as many jobs of length one in pairs on machines as if each pair were a job that

takes two time units. Then all remaining jobs are assigned for two time units while attempting to continue to respect the proposed deadline. If the deadline cannot be met, the proposed deadline is incremented by one, then the decision procedure repeats. We present their result in a more general form to show its significance with respect to other subclasses of $R||C_{max}$.

**Theorem 4.1.13** (Lenstra *et al.* [33], Theorem 6)**.** *Let $p \in \mathbb{Z}^+$. $R|p_{i,j} \in \{p, 2p\}|C_{max}$ is polynomial-time solvable.*

*Proof.* Consider an arbitrary processing requirement matrix $P$, where $p_{i,j} \in \{p, 2p\}$. Since $p \mid 2p$ and $p \in \mathbb{Z}^+$, create a new processing requirement matrix $P'$ with $p_{i,j} \in \{1, 2\}$ by dividing each element of $P$ by $p$. Clearly such a transformation can be done in $\Theta(mn)$ steps. If we find an optimal schedule for $P'$, then its job assignments correspond to an optimal schedule for $P$. We give pseudocode for an algorithm that finds an optimal schedule for $P'$ in polynomial time as Algorithm 4.1.14.

We wish to find a feasible schedule $A$ with minimum makespan. The algorithm performs a decision procedure in which a deadline variable $d$ is incremented until an optimal schedule is found. The procedure begins by setting $d := 1$. Then, the algorithm constructs a bipartite graph $G = (S \cup T, E)$ based on one of two cases depending on the parity of $d$. Consider the following two cases.

1. **Case ($d = 2k$):** Apply Construction 4.1.15 using $P'$ to create a bipartite graph $G = (S \cup T, E)$.

   **Construction 4.1.15.** Construct a bipartite graph $G = (S \cup T, E)$, where each machine has $k$ vertices in $S$, and each job corresponds to one vertex in $T$. For

**Algorithm 4.1.14** Polynomial-time Algorithm for $R|p_{i,j} \in \{1,2\}|C_{max}$

**procedure** MAKESPANUPMS12($P', m, n$)
  Let $X$ be a $m \times (n+m)$ matrix with zero entries; ▷ $m \times (n+m)$ assignment matrix
  $P := P'$; ▷ Storing a copy of the processing requirement matrix
  $d := 1$; ▷ Proposed deadline
  $feasible := false$; ▷ Stays false until a feasible schedule is found
  **while** $\neg feasible$ **do** ▷ Do until a feasible schedule can be found
    $P' := P$; ▷ Reset the processing requirement matrix
    $X := 0_{m\times(m+n)}$; ▷ Ensure all the entries of the matrix are zero
    **if** ($d$ mod 2 == 1) **then** ▷ Case 2: $d$ is odd
      Apply Construction 4.1.16 to create processing requirement matrix $P'$;
    **end if**
    Apply Construction 4.1.15 to obtain $G = (S \cup T, E)$;
    Call **Maximum2-1CBS**($G$) to get $H = (S' \cup T', E')$; ▷ Max 2-1 CBS;
    **for each** $\{k, j\} \in E'$, where $k \in S'$ and $j \in T'$ **do** ▷ Construct schedule
      Let $i$ be the machine that corresponds to machine vertex $k$ in Construction 4.1.15;
      $X_{i,j} := 1$; ▷ Assign job $j$ to machine $i$
    **end for**
    Assign each unscheduled job for two time units with $X$ while respecting $d$;
    Let $L$ be the makespan of the schedule for assignment matrix $X$;
    **if** $((d \bmod 2 == 0) \wedge (L > d)) \vee ((d \bmod 2 == 1) \wedge (L > d+1))$ **then**
      $d := d + 1$; ▷ Feasible schedule cannot be found yet, increment $d$
    **else**
      $feasible := true$; ▷ We have a feasible schedule
    **end if**
  **end while**
  Let $X'$ be a $m \times n$ submatrix comprised of the first $n$ columns of $X$;
  **return** $X'$; ▷ Return schedule
**end procedure**

every job, add an edge that connects from a job vertex corresponding to job $j$ to a machine vertex corresponding to machine $i$ if $p'_{i,j} = 1$, where $1 \leq i \leq m$. Keep track of what machine each machine vertex corresponds to in a lookup table. ✠

2. **Case ($d = 2k - 1$):** If $d = 2k - 1$ (i.e., $d$ is odd), then create a modified $P'$

by applying Construction 4.1.16. Now, consider case 1 (when $d = 2k$) with this modified instance of $P'$.

**Construction 4.1.16.** Assume we are given a processing requirement matrix $P'$. Create $m$ dummy jobs $n+1$, $n+2$, ..., $n+m$, where each corresponds to one machine. For $1 \leq i \leq m$, if a dummy job corresponds to machine $i$, then $p_{i,n+i} = 1$, and is assigned two time units in $P'$ for every other machine. ✠

Upon applying the appropriate case, the algorithm has a bipartite graph $G = (S \cup T, E)$. The graph $G$ is an instance of the maximum 2-1 CBS problem. By Theorem 4.1.8, the maximum 2-1 CBS problem can be solved in polynomial time. Solve this instance of the maximum 2-1 CBS problem, and obtain a maximum 2-1 CBS $H = (S' \cup T', E')$. Using the subgraph $H$, we build a schedule $A$. Consider each edge in $H$. If there is an edge from $T$ corresponding to job $j$ to a machine vertex corresponding to machine $i$, then schedule job $j$ on machine $i$ for one time unit. Next, schedule all remaining jobs for two time units greedily while respecting the makespan $d$. If a schedule $A$ can be produced without the makespan exceeding $d$, then the deadline $d$ is met. If the makespan is not $d$, increase $d$ by one, and repeat the process described until a feasible schedule $A$ that has makespan $d$ is produced. If $d = 2k$, return the schedule $A$ and terminate. Next we consider if there was a feasible schedule for $d = 2k - 1$.

Returning to the case $d = 2k - 1$, dummy jobs are created and then included in $P'$. If the modified instance met the deadline, remove all the dummy jobs in $A$. Since the dummy jobs in $A$ correspond to pairings of unit time jobs in $P'$ that appeared in $H$, there is at least one dummy job that could be scheduled for one time unit on each

machine. Removing the dummy jobs produces a schedule with makespan $(2k-1)$ for $P'$. The algorithm then returns the feasible schedule $A$, and terminates.

This procedure finds a feasible schedule with minimum makespan as this procedure finds a feasible schedule with makespan at most $d$, if it exists. For $P'$, the largest value the makespan $d$ can be is $\lceil 2n/m \rceil$. So this algorithm terminates in polynomial time after at most $\lceil 2n/m \rceil$ applications of this decision procedure. Since the schedule contains all $n$ jobs and each job is assigned a processing time of either one or two time units, this is a feasible schedule for $P'$. In turn, the same assignments to $P'$ applied to $P$ yields an optimal schedule. Therefore, for any $p \in \mathbb{Z}^+$, this algorithm solves $R|p_{i,j} \in \{p, 2p\}|C_{max}$ in polynomial time. □

We now apply the algorithm described in Theorem 4.1.13 to an example $R|p_{i,j} \in \{1, 2\}|C_{max}$ instance.

**Example 4.1.17.** Consider

$$P = \begin{pmatrix} 2 & 2 & 1 & 2 & 2 \\ 1 & 2 & 2 & 1 & 1 \\ 2 & 2 & 2 & 1 & 1 \end{pmatrix}.$$

Since $p_{i,j} \in \{1, 2\}$ for our instance, the algorithm divides all entries of $P$ by $p = 1$, then define this as $P'$. Next, the procedure sets the deadline $d$ to one.

Since $d = 1$, the algorithm creates $m = 3$ dummy jobs and includes them in $P'$. Each dummy job corresponds to one machine and is scheduled to take one time unit on that particular machine. Otherwise, $p_{i,j} = 2$. After including the dummy jobs in

the processing requirement matrix,

$$P' = \begin{pmatrix} 2 & 2 & 1 & 2 & 2 & 1 & 2 & 2 \\ 1 & 2 & 2 & 1 & 1 & 2 & 1 & 2 \\ 2 & 2 & 2 & 1 & 1 & 2 & 2 & 1 \end{pmatrix}.$$

Next, using $P'$, Construction 4.1.15 is applied, and a bipartite graph $G = (S \cup T, E)$ as shown in Figure 4.8 is constructed. Notice that $k = 1$ as $d = 2k - 1$.

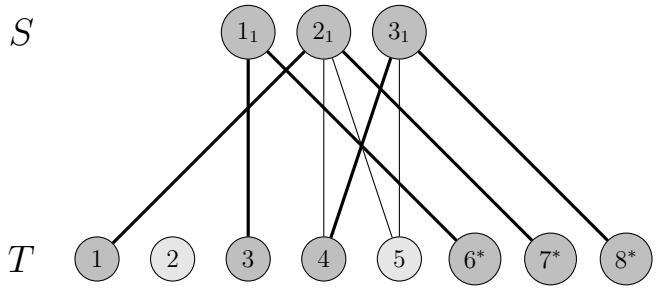



Figure 4.8: A bipartite graph $G = (S \cup T, E)$ constructed by the algorithm for $d = 1$. A subgraph $H$ obtained by solving the maximum 2-1 CBS problem for $G$ has edges shown with thick edges, and vertices are a darker shade of grey. The dummy jobs of $T$ are indicated by an asterisk beside the job number.

A maximum 2-1 CBS $H$ is found in $G$. Consider the edges of $H$. For one time unit each, the algorithm assigns jobs 3 and 6 to machine 1; jobs 1 and 7 to machine 2; and jobs 4 and 8 to machine 3. After such, jobs 2 and 5 are greedily scheduled for two time units. This produces a schedule with makespan of four. Since $d = 2 \neq 4$, the algorithm increments $d$ by one, then repeats the decision procedure.

Next, $d = 2$, so the algorithm doesn't include dummy jobs, and

$$P' = \begin{pmatrix} 2 & 2 & 1 & 2 & 2 \\ 1 & 2 & 2 & 1 & 1 \\ 2 & 2 & 2 & 1 & 1 \end{pmatrix}.$$

As $d = 2k$, $k = 1$, and Construction 4.1.15 is applied to $P'$ to produce a bipartite graph. The bipartite graph $G = (S \cup T, E)$ built by the algorithm is shown in Figure 4.9.

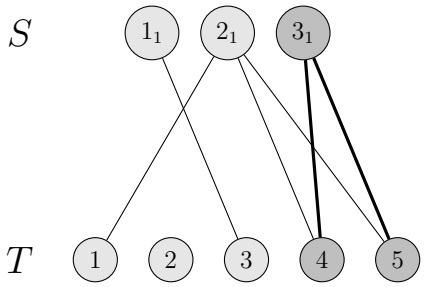



Figure 4.9: A bipartite graph $G = (S \cup T, E)$ construction for the example instance by the algorithm when $d = 2$. A maximum 2-1 CBS $H$ in $G$ has edges shown with thick edges, and the vertices that are a darker shade of grey.

After finding a maximum 2-1 CBS $H$ in $G$, the algorithm schedules jobs 4 and 5 for one time unit each on machine 3. Also, jobs 1, 2, and 3 are greedily scheduled for two time units each. Such a schedule will have makespan of four, which is not two. So $d = 3$, and the odd case is considered once again.

Like $d = 1$, when $d = 3$, the algorithm includes $m = 3$ dummy jobs in the processing requirement matrix $P'$. Thus, the algorithm considers

$$P' = \begin{pmatrix} 2 & 2 & 1 & 2 & 2 & 1 & 2 & 2 \\ 1 & 2 & 2 & 1 & 1 & 2 & 1 & 2 \\ 2 & 2 & 2 & 1 & 1 & 2 & 2 & 1 \end{pmatrix}.$$

Since $d = 3 = 2k - 1$, $k = 2$. So when the bipartite graph is constructed by the algorithm in this iteration, there are $k = 2$ machine vertices representing each machine as shown in Figure 4.10. The algorithm then constructs a bipartite graph $G = (S \cup T, E)$ using $P'$ by applying Construction 4.1.15. Then, a maximum 2-1

CBS $H$ is found in $G$.

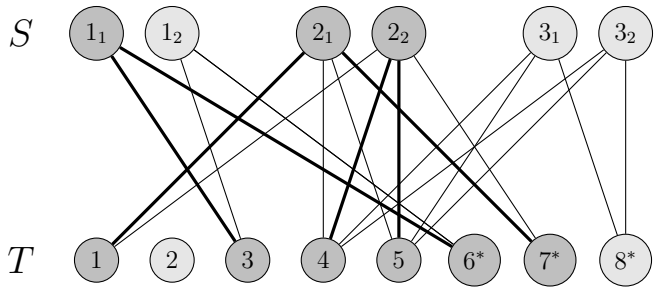



Figure 4.10: A bipartite graph $G = (S \cup T, E)$ construction for $d = 3$ in using example instance of $R|p_{i,j} \in \{1, 2\}|C_{max}$. A subgraph $H$ from finding a maximum 2-1 CBS in $G$ has edges shown as thick edges, and vertices are a darker shade of grey. The dummy jobs of $T$ are indicated by an asterisk beside the job number.

Using the edges of $H$, jobs 3 and 6 are assigned to machine 1; jobs 1, 4, 5 and 7 are assigned to machine 2; for one time unit each. Following this, jobs 2 and 8 are scheduled greedily while respecting the deadline. As seen on the left in Figure 4.11, a schedule containing dummy jobs with makespan of four is produced. When the dummy jobs are removed, a schedule with optimal makespan $d = 3$ is obtained. This schedule is shown on the right in Figure 4.11. ■

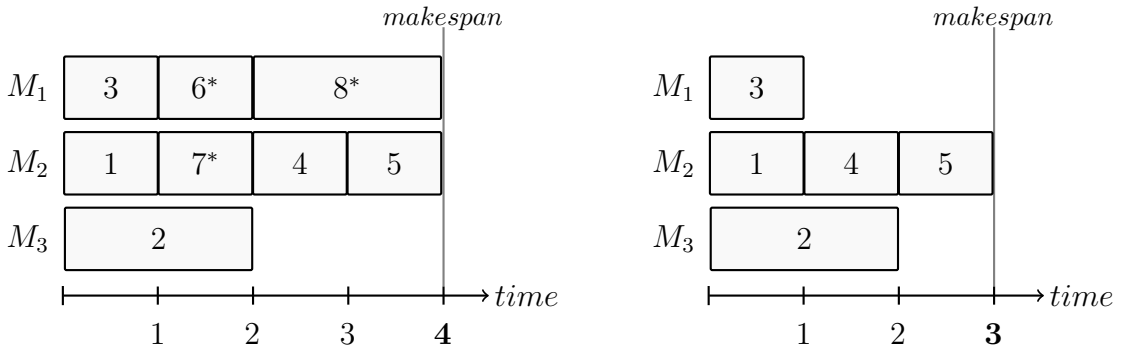



Figure 4.11: The schedule produced when $d = 3$ (i.e., $d$ is odd, so dummy jobs are included then treated as $d = 4$) before removing dummy jobs (on the left) and final schedule (on the right) after removing all the dummy jobs. The schedule on the right is a schedule with minimum makespan $d = 3$.

## 4.2 Some NP-hard Subclasses of $R||C_{max}$ and Related Scheduling Problems

In this section, we briefly summarize a variety of results for NP-hard scheduling problems. We focus on special cases of $R||C_{max}$, and related problems. We begin by considering scheduling problems on identical parallel machines, uniformly related parallel machines, then unrelated parallel machines. One particular subclass of $R||C_{max}$ we give in-depth results for has processing times of constant lengths $p$ and $q$, where $q \neq 2p$. We present a proof shown originally by Lenstra *et al.* [33] that says this subclass of $R||C_{max}$ is NP-hard.

### 4.2.1 Identical Parallel Machines ($P||C_{max}$)

Recall that $P||C_{max}$ denotes the makespan problem on identical parallel machines. The first approximation algorithm results for this subclass of $R||C_{max}$ were by Graham [16]. Through two papers, Graham considered $P|prec|C_{max}$ and $P||C_{max}$. Originally in 1966, Graham [15] solved this problem using directed graphs with a partial ordering to design a greedy 2-approximation algorithm called the list scheduling algorithm. In Section 1.5, we presented this algorithm for $P||C_{max}$. Graham [16] also described this solution with another set of analyses in his 1969 results. Based on his 1966 greedy algorithm, Graham used the strategy of assigning tasks by arranging jobs from longest execution time to the shortest execution time. He showed this simple modification of the job inputs yields an approximation ratio of 4/3. This is called the *longest processing time first (LPT) algorithm.* Graham [16] also developed one of

the first known PTAS for $R||C_{max}$. In 1976, Sahni [39] reduced Graham's result to a FPTAS.

### 4.2.2 Uniformly Related Parallel Machines ($Q||C_{max}$)

We now focus on the scheduling problem $Q||C_{max}$, the makespan problem on uniformly related parallel machines. For $Q||C_{max}$, a job $j$ with processing time $p_j$ takes $s_i p_j$ time units to complete for any machine $i$, where $s_i \in \mathbb{Z}^+$ is the speed of machine $i$, $1 \leq i \leq m$ and $1 \leq j \leq n$. This subclass of $R||C_{max}$ has applications in scheduling homogeneous computer systems, where processors may vary in speed but jobs process identically. Often jobs are queued in such computer systems, so some researchers have focused their efforts on $Q|prec|C_{max}$ for this application. Note that $Q||C_{max}$ is a special case of $Q|prec|C_{max}$, where each vertex of the directed acyclic graph (DAG) $G$ is isolated. For the following results, assume there is a partial order for the jobs.

In 1999, Chudak and Shmoys [4] considered both $Q|prec|C_{max}$, and $Q|prec|\sum_j w_j C_j$ which was introduced by Hall *et al.* [18]. Chudak and Shmoys improved a previously known $O(\sqrt{m})$-approximation algorithm by Jaffe [21] for the makespan problem for uniformly related parallel machines. Based on Graham's list scheduling algorithm [15] and a relaxed mixed integer program, Chudak and Shmoys developed an algorithm that minimizes the makespan based on the number of distinct speeds of each machine $K$. The authors developed a $O(\log(m))$-approximation algorithm, where $K = O(\log(m))$ was the number of distinct machine speeds. Also, the authors extended the algorithmic results to $Q|prec|\sum_j w_j C_j$. As a remark, Chudak

and Shmoys also investigated the scheduling problem $Q|prec, r_j|\sum_j w_j C_j$.

In 2001, Chekuri and Bender [3] developed a different $O(\log(m))$-approximation algorithm for $Q|prec|C_{max}$. Though inspired by many of the ideas of Chudak and Shmoys [4], the graph-theoretic combinatorial algorithm by Chekuri and Bender was based on finding longest length chains in a DAG. Their algorithm has $O(n^3)$ worst-case time complexity. The algorithm they designed is based on maximal chain decompositions, where they define a chain to be all the jobs assigned to a particular uniformly related parallel machine. This procedure exploits their lower bound for building a maximal chain decomposition algorithmically in $O(n^3)$ worst-case time. Also, if a problem instance is given with its maximal chain decomposition, then this algorithm, as well as the algorithm by Chudak and Shmoys [4], executes in $O(n\log(m))$ worst-case time. Chekuri and Bender also considered $Q|prec, r_j|C_{max}$ and $Q|chains, r_j|C_{max}$. If release dates are included with each job, the modified algorithm still executes in $O(n^3)$ worst-case time, and the approximation ratio remains $O(\log(m))$. When precedence constraints are restricted to chains (i.e., $Q|chains, r_j|C_{max}$), Chekuri and Bender showed that this same procedure is a 6-approximation algorithm.

For $Q|prec|C_{max}$ and $Q|prec|\sum_j w_j C_j$, Chekuri and Bender [3] conjectured that there exists an algorithm with constant approximation ratio. In recent years, other researchers have developed polynomial-time approximation schemes, and $k$-approximation algorithms for less general variations $Q|prec|C_{max}$. These new algorithms [23; 29] are close to a constant approximation ratio. To this date, there is no known $k$-approximation algorithm for $Q|prec|C_{max}$ with a constant approximation factor.

### 4.2.3 Other Known NP-hard Subclasses

Based on the hardness of approximation results and the best known approximation factor of two for $R||C_{max}$, researchers have focused on intractable subclasses of $R||C_{max}$. Recall that Theorem 4.1.13 by Lenstra *et al.* [33] says $R|p_{i,j} \in \{p, q\}|C_{max}$ when $q = 2p$ is polynomial-time solvable. The authors also considered all other two-valued integer instances of $R||C_{max}$ as well. Lenstra *et al.* [33] showed that $R|p_{i,j} \in \{p, q\}|C_{max}$, where $q \neq 2p$ is NP-hard. Their proof uses a polynomial-time reduction from the $q$-dimensional matching ($q$DM) problem, which is known to be NP-complete [14].

**Problem 4.2.1** ($q$-Dimensional Matching ($qDM$))**.** *Let $q > 2$. Let there be $q$ disjoint sets $A_1 = \{a_{1,1}, \ldots, a_{1,n'}\}, \ldots, A_q = \{a_{q,1}, \ldots, a_{q,n'}\}$, and a family of $q$-sets $F = \{T_1, \ldots, T_{m'}\}$, where $|T_i \cap A_j| = 1$ for $i = 1, \ldots, m'$, and for $j = 1, \ldots, q$. Does $F$ contain a matching? That is, does $F$ contain a subfamily $F' \subseteq F$, where $|F'| = n'$ and $\bigcup_{T_i \in F'} T_i = \bigcup_{t=1}^{n'} A_t$?*

**Theorem 4.2.2** (Lenstra *et al.* [33], Theorem 7)**.** *$R|p_{i,j} \in \{p, q\}|C_{max}$, where $q \neq 2p$, is NP-hard.*

*Proof.* Without loss of generality, assume $gcd(p, q) = 1$, and $q \neq 2p$. In order to show the makespan problem for UPMs when $p_{i,j} \in \{p, q\}$ is NP-hard, we show its decision problem counterpart is NP-complete. Consider the following decision problem.

**Problem 4.2.3** (MAKESPANUPM-$\{p, q\}$)**.** *Let there be a $m \times n$ processing requirement matrix $P$, where each $p_{i,j} \in \{p, q\}$, and $q \neq 2p$. Does there exist a schedule with makespan at most $d$?*

We wish to show that MAKESPANUPM-$\{p, q\} \in$ NP-complete by showing MAKESPANUPM-$\{p, q\} \in$ NP, and $qDM \leq_P$ MAKESPANUPM-$\{p, q\}$. Given a schedule, one can find the completion time of the machine that finishes last in a polynomial number of steps. Thus, MAKESPANUPM-$\{p, q\} \in$ NP. Next, we wish to show a reduction from the $q$DM problem to MAKESPANUPM-$\{p, q\}$.

Suppose we are given an arbitrary instance $I = (A_1, \ldots, A_q, F, n', m') \in qDM$, if $m' \geq n'$, use Construction 4.2.4 to get an instance $I' = (m, n, P, d) \in$ MAKESPANUPM-$\{p, q\}$ in polynomial time. If $m' < n'$, create a trivial "no" instance of MAKESPANUPM-$\{p, q\}$.

**Construction 4.2.4.** Given an arbitrary $qDM$ instance, create an $m \times n$ processing requirement matrix $P$, with $m = m'$ machines and $n = qn' + p(m' - n')$ jobs that consist of $qn'$ *element jobs* and $p(m' - n')$ *dummy jobs*. Next, let $d = pq$. Each machine corresponds to a $q$-set of the family $F$, and the element jobs correspond to the elements contained over all $q$ disjoint sets $A_1, \ldots, A_q$. An element job takes $p$ time units on a machine if a $q$-set corresponding to a machine contains an element represented by the element job. If that is not satisfied, an element job is processed in $q$ time units. Each dummy job takes $q$ time units on every machine. Clearly, this is an instance of MAKESPANUPM-$\{p, q\}$, and can be constructed in polynomial time.

Next, we need to show that there is a matching in a $qDM$ instance $I$ if and only if there is a schedule with makespan at most $d = pq$ for $I'$, when $m' \geq n'$. First, let us show that if there is a matching then there is a schedule with makespan at most $pq$. If there is a matching, then exactly $n'$ of the machines are scheduled with $q$ jobs that

take $p$ time units, then the remaining $m' - n'$ machines each schedule $p$ dummy jobs for $q$ time units. Element jobs that take $p$ time units in such a schedule correspond to elements contained in exactly $n'$ $q$-sets of $F$. Such a schedule has makespan at most $pq$.

Second, let us show that if there is a schedule with makespan at most $d = pq$, then there is a matching. Suppose there exists a schedule that has makespan at most $pq$, but doesn't contain a matching, implying that at least one element is not covered by $n'$ $q$-sets of $F$. If a schedule containing all $n$ jobs has makespan at most $pq$, then there is no idle time for any of the machines. Either a machine is scheduled with $q$ element jobs that take $p$ time units each, or with $p$ dummy jobs, each with duration of $q$ time units. But, at least one element job must take $q$ time units. A schedule with makespan at most $pq$ cannot be produced as at least one machine will cause at least $(q - p)$ idle time for the remaining machines and have makespan strictly larger than $pq$. By contradiction, such a schedule cannot exist. Thus, if there is a schedule with makespan at most $pq$, then there is a matching.

Since there is a matching if and only if there is a schedule with makespan at most $pq$, an instance $I'$ is a "yes" instance only if there is a schedule with makespan at most $pq$. If $m' < n'$, or there is no schedule with makespan at most $pq$, then it is a "no" instance of MAKESPANUPM-$\{p, q\}$. Thus, $qDM \leq_P$MAKESPANUPM-$\{p, q\}$, and MAKESPANUPM-$\{p, q\} \in$ NP-complete.

Therefore, $R|p_{i,j} \in \{p, q\}|C_{max}$ when $q \neq 2p$ is NP-hard.

□

As a consequence of Theorem 4.2.2, given any two integral values $p \leq q$,

$R|p_{i,j} \in \{p, q\}|C_{max}$ is NP-hard.

To help understand the decision procedure of Theorem 4.2.2 and application of Construction 4.2.4, let us consider an example.

**Example 4.2.5.** To begin, suppose we are given a 4DM (i.e., $q = 4$) instance, where

$A_1 = \{1, 2, 3, 4, 5\}, A_2 = \{6, 7, 8, 9, 10\}, A_3 = \{11, 12, 13, 14, 15\}, A_4 = \{16, 17, 18, 19, 20\},$

and

$F = \{\{1, 6, 11, 16\}, \{2, 7, 12, 17\}, \{3, 8, 15, 19\}, \{3, 8, 13, 19\}, \{4, 9, 14, 20\}, \{5, 10, 15, 18\}\}.$

For this example, $m' = 6$ and $n' = 5$. Let us apply Construction 4.2.4 to create a MAKESPANUPM-$\{1, 4\}$ instance where $p = 1$ and $q = 4$. As $m = m' = 6$ and $n = 4n' + p(m' - n') = 4 \cdot 5 + 1(6 - 5) = 21$, create a $6 \times 21$ processing requirement matrix $P$ consisting of twenty element jobs, and one dummy job. Applying the conditions for the processing requirement matrix, we obtain matrix $P$ with entries

$$\begin{pmatrix}
\textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & 4 \\
4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 \\
4 & 4 & 1 & 4 & 4 & 4 & 4 & 1 & 4 & 4 & 4 & 4 & 4 & 4 & 1 & 4 & 4 & 4 & 1 & 4 & \textcircled{4} \\
4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 \\
4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & 4 & \textcircled{1} & 4 \\
4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & 4 & 4 & \textcircled{1} & 4 & 4 & \textcircled{1} & 4 & 4 & 4
\end{pmatrix}.$$

Each circled entry above in the processing requirement matrix is an assignment of a job to a machine. The makespan of such a schedule is $pq = 4$. Notice that the 4-sets found in the 4DM instance that consist of a matching correspond to the element jobs scheduled to take one time unit each. Since there is a matching if and only if there is a schedule with makespan at most $pq$, this is a "yes" instance. ■

Werner and Vakhania [45] in 2012 presented a $q$-absolute approximation algorithm for $R|p_{i,j} \in \{p, q\}|C_{max}$. The $q$-absolute approximation algorithm developed by Werner and Vakhania is comprised of two phases. First, it uses the polynomial-time algorithm by Lenstra *et al.* [33] for $R|p_{i,j} \in \{1, 2\}|C_{max}$ to create an initial schedule. In its second phase, it solves a LP to reassign jobs from the assignments in phase one to guarantee their result. To this date, no $k$-approximation algorithm has been presented for $R|p_{i,j} \in \{p, q\}|C_{max}$ from our investigation of the literature. In Section 6.2.1, we investigate this subclass and $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$.

Researchers have also focused on other restricted cases of $R||C_{max}$ and related scheduling problems, discovering approximation algorithms with better approximation factors [1; 8; 9; 11; 30; 31; 41; 43; 45]. Some researchers have focused on multiple objectives or enforcing precedence constraints. For instance, in 2005, Kumar *et al.* [30] used randomized rounding to yield a bicriteria $(2T, 3C/2)$-approximation algorithm for $R||\sum_j w_j C_j$ with the goal of minimizing the makespan (where the optimal makespan is $T$), and weighted completion times (where the optimal weighted completion time is $C$) simultaneously. Also, Kumar *et al.* [31] in 2007 considered $R|forest|C_{max}$. For this instance, the authors developed an $O(\log^2(n)/\log(\log(n)) \cdot \lceil \log(min\{p_{max}, n\})/\log(\log(n)) \rceil)$-approximation algorithm,, where $p_{max}$ is the maximum total processing time along a path in the forest. Also, Kumar *et al.* discovered the same approximation ratio for $R|forest|\sum_j w_j C_j$.

Researchers have also considered subclasses, that restrict jobs to certain machines. Ebenlendr *et al.* [8; 9] considered a subclass of $R||C_{max}$ called *graph balancing*. This is $R|p_{i,j} \in \{p_j, \infty\}|C_{max}$, where every job has a choice of being assigned on at most two

machines. By using LP techniques such as rounding, Ebenlendr *et al.* [8; 9] designed a 1.75-approximation algorithm. In 2010, Svensson [43] investigated a more general subclass, $R|p_{i,j} \in \{p_j, \infty\}|C_{max}$. Svensson developed a polynomial-time algorithm that has an approximation ratio of 1.9412. The performance guarantee of 1.9412 for this algorithm is obtained through linear programming.

When considering subclasses of $Rm||C_{max}$, PTASs have been discovered [1; 11; 22; 33]. Efraimidis and Spirakis [11] in 2006 developed several improved PTASs using a rounding technique called combinatorial randomized rounding, including for $Rm||C_{max}$. Some other problems the authors considered are the generalized assignment problem, load balancing, two-cost objective scheduling, and, lastly, assigning each machine one specific load. In 2012, a PTAS for the makespan problem for UPMs, where we have a fixed number of types of machines scheduling $D$-dimensional jobs, was developed by Bonifaci and Wiese [1]. For this PTAS, the number of machines are fixed, $D$ is fixed, and machines of the same type are identical. The authors used linear programming techniques to formulate their $(1 + \epsilon)$-approximation algorithm, for any $\epsilon > 0$.

# Chapter 5

# A New Polynomial-Time Algorithm for Certain Subclasses of $R||C_{max}$

In this chapter, we present a polynomial-time algorithm for certain restricted assignment instances of $R||C_{max}$. By Theorem 4.1.3, $R|p_{i,j} \in \{1, \infty\}|C_{max}$ is polynomial-time solvable We extend these results to solve $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ in polynomial time, where $\omega$ is a fixed positive integer. Then, we generalize this subclass to $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with an initial load for each machine. Our algorithm for $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads employs matching techniques.

## 5.1 $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with Initial Loads

Recall Theorem 4.1.3 states $R|p_{i,j} \in \{1, \infty\}|C_{max}$ is polynomial-time solvable. Assume $\omega \in \mathbb{Z}^+$ is a constant. What if we generalized this problem to $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$, a subclass of $R||C_{max}$ when processing times are restricted to $p_{i,j} \in \{\omega, \infty\}$? This generalization can be solved in polynomial time by changing each entry that takes $\omega$ time units to one time unit in the processing requirement matrix $P$, then applying the algorithm described for $R|p_{i,j} \in \{1, \infty\}|C_{max}$ in Section 4.1. So we obtain the next corollary from Theorem 4.1.3.

**Corollary 5.1.1.** *Let $\omega \in \mathbb{Z}^+$ be fixed. $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ is polynomial-time solvable.*

We now use this generalized problem to define a simple, but pragmatic scheduling problem. Let there be $m$ machines, and $n$ jobs to be scheduled. For each machine $i$, assume there is an *initial load* of $\mu_i$ time units to complete before any jobs may be scheduled, where $\mu_i \geq 0$, and $1 \leq i \leq m$. A job $j$ can be scheduled on machine $i$ to take $p_{i,j} \in \{\omega, \infty\}$ time units, where positive integer $\omega$ is fixed. Assume that processing requirement matrix $P$ has at least one positive integer in each column; if not, it contains redundant jobs. For this problem, the goal is to find a feasible schedule with minimum makespan that includes each initial load on the machines. We call this *the makespan problem on UPMs with initial loads and processing times $p_{i,j} \in \{\omega, \infty\}$, where $\omega \geq 1$.* This problem is denoted $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads. The $R|p_{i,j} \in \{w, \infty\}|C_{max}$ problem is the special case of $R|p_{i,j} \in \{w, \infty\}|C_{max}$ with initial loads where each $\mu_i = 0$.

This problem is common in homogeneous computer systems with similar tasks and speeds. Very often, homogeneous systems execute unrelated start-up processes that are predetermined before applications may be scheduled. For example, given four cores, one machine may be booting an operating system that takes a number of time units, while the remaining three may have applications already scheduled that have given completion times. Suppose that, at this point, a scheduler for the entire system of cores needs to schedule a set of similar-length processes that can be scheduled on particular machines from a batch script to complete amongst all four cores. It would be ideal if the tasks in the batch script need not be delayed any further than necessary.

Consider an example instance of $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads. Fix $\omega = 3$. Suppose we have four machines with initial loads shown in Table 5.1, and

$$P = \begin{pmatrix} 3 & 3 & \infty & \infty & \infty \\ \infty & 3 & 3 & 3 & 3 \\ 3 & \infty & 3 & \infty & 3 \\ \infty & 3 & 3 & 3 & \infty \end{pmatrix}.$$

| machine $i$ | $\mu_i$ |
|---|---|
| 1 | 2 |
| 2 | 5 |
| 3 | 6 |
| 4 | 2 |

Table 5.1: Initial loads of each machine in an instance of $R|p_{i,j} \in \{3, \infty\}|C_{max}$ with initial loads.

If one applied the same method that yields Corollary 5.1.1, it is possible to obtain a feasible schedule with makespan of twelve. Unfortunately, this is not optimal. The

optimal makespan for this instance is eight. Using matching techniques, we show that $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads, where $\omega \geq 1$, is polynomial-time solvable.

**Theorem 5.1.2.** *Let $\omega \in \mathbb{Z}^+$ be fixed. $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads can be solved in polynomial time.*

*Proof.* We wish to describe an algorithm to solve this problem in polynomial time, then prove its correctness. Our algorithm is outlined as pseudocode in Algorithm 5.1.4. Let constant $\omega \geq 1$. Assume the algorithm is given an arbitrary list of initial loads $\mu_1, \ldots, \mu_m$, and an $m \times n$ processing requirement matrix $P$, where $p_{i,j} \in \{\omega, \infty\}$. We use a proposed deadline $d'$ to keep track of the makespan of the schedule tested in each iteration. Since the minimum makespan cannot be less than the largest initial load, initially set $d' := max_{1 \leq z \leq m}\{\mu_z\}$.

Next, the procedure calculates how much available time is on each machine based on the proposed deadline $d'$. The intention is to place as many jobs of length $\omega$ on the machines as possible such that the makespan is at most $d'$. If $\sum_{i=1}^{m} \left( \lfloor (d' - \mu_i)/\omega \rfloor \right) < n$, the makespan cannot be respected, because not all $n$ jobs of length $\omega$ can be scheduled even if all $n$ jobs could be scheduled on any machine. If this is the case, restart this procedure with $d' := d' + 1$. Otherwise, the algorithm applies Construction 5.1.3 to build a bipartite graph $G$ using the availability for jobs of length $\omega$ time units on each machine.

**Construction 5.1.3.** Create a bipartite graph $G = (M \cup J, E)$ consisting of a set $M$ of *machine vertices* and a set $J$ of *job vertices*. For each machine $i$, create $\lfloor (d' - \mu_i)/\omega \rfloor$ machine vertices to represent machine $i$. Create $n$ job vertices, one for corresponding to each job. The correspondence between machines and machine

vertices can be stored in a lookup table. For each $p_{i,j} = \omega$ of processing requirement matrix $P$, include edges from job vertex $j$ to every machine vertex of machine $i$. This construction can be created in polynomial time. ✠

Next, the procedure finds a maximum matching $M_{max}$ in $G$ using any polynomial-time maximum matching algorithm for bipartite graphs [19]. If $|M_{max}| < n$, restart the algorithm after incrementing $d'$ by one. If $|M_{max}| = n$, then all $n$ job vertices are saturated. For each edge $\{j, v\} \in M_{max}$, where $j \in J$, schedule job $j$ on the machine that corresponds to machine vertex $v$. Upon returning the schedule, the algorithm terminates. Such a schedule has makespan of $d'$, and is feasible.

Recall that the algorithm starts by setting deadline $d' = max_{1 \leq z \leq m}\{\mu_z\}$. For any instance, the makespan is at most $\omega n + max_{1 \leq z \leq m}\{\mu_z\}$. When the algorithm terminates, the number of times $d'$ is increased is at most

$$(d' - max_{1 \leq z \leq m}\{\mu_z\}) \leq \omega n \in \Theta(n).$$

Thus, the algorithm finds a feasible schedule in polynomial time.

Suppose $d'$ is not the optimal makespan. Then, there exists a schedule $S'$ with makespan $d^*$, such that $d' > d^*$. Consider the algorithm when the proposed deadline is $d^*$. At this stage, $\sum_{i=1}^{m} \left( \lfloor (d^* - \mu_i)/\omega \rfloor \right) < n$, or $|M_{max}| < n$. If $S'$ exists and is feasible, then $\sum_{i=1}^{m} \left( \lfloor (d^* - \mu_i)/\omega \rfloor \right) \geq n$, and $|M_{max}| = n$. But, this violates the definition of a matching of a graph as a machine vertex in the matching would have more than one incident edge from a job vertex. That implies the makespan of $d^*$ could not have been respected for schedule $S'$, and $S'$ cannot exist. Thus, $d^* = d'$.

Therefore, $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads can be solved in polynomial time. □

**Algorithm 5.1.4** Polynomial-time Algorithm for $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads

**procedure** MAKESPANUPMSASSIGNMENTINITLOADS($P, m, n, \mu$)
  Let $X$ be a $m \times n$ matrix with zero entries; ▷ $m \times n$ assignment matrix
  $d' := max_{1 \leq i \leq m}\{\mu_i\}$; ▷ Proposed deadline
  $M_{max} := \varnothing$; ▷ Will contain a maximum matching
  $feasible := false$; ▷ Stays false until a feasible schedule is found
  **while** $\neg feasible$ **do** ▷ Do until a feasible schedule can be found
    **if** $\sum_{i=1}^{m} \left( \lfloor (d' - \mu_i)/\omega \rfloor \right) \geq n$ **then** ▷ Is there enough time available yet?
      Apply Construction 5.1.3 to obtain $G = (M \cup J, E)$;
      Let $M_{max}$ be a maximum matching of $G$;
    **end if**
    **if** $|M_{max}| < n$ **then**
      $d' := d' + 1$; ▷ Feasible schedule cannot be found yet, increment $d'$
    **else**
      $feasible := true$; ▷ $|M_{max}| = n$
    **end if**
  **end while**
  **for each** $\{v, j\} \in M_{max}$, where $v \in M$ and $j \in J$ **do** ▷ Construct schedule
    Let $i$ be the machine that corresponds to machine vertex $v$ in Construction 5.1.3;
    $X_{i,j} := 1$; ▷ Assign job $j$ to machine $i$
  **end for**
  **return** $X$; ▷ Return schedule
**end procedure**

We now apply the algorithm described in our proof for Theorem 5.1.2 on the example we gave earlier.

**Example 5.1.5.** Recall that we fixed $\omega = 3$. Let there be $m = 4$ machines with initial loads in Table 5.1, and

$$P = \begin{pmatrix} 3 & 3 & \infty & \infty & \infty \\ \infty & 3 & 3 & 3 & 3 \\ 3 & \infty & 3 & \infty & 3 \\ \infty & 3 & 3 & 3 & \infty \end{pmatrix}.$$

In the algorithm, we begin by setting proposed deadline

$$d' = max\{\mu_1, \mu_2, \mu_3, \mu_4\} = max\{2, 5, 6, 2\} = 6.$$

Next, the algorithm calculates the available time based on deadline $d'$ for each machine and how many machine vertices to represent each machine will be needed for the bipartite graph construction. We show these calculations in Table 5.2.

| machine $i$ | $\mu_i$ | $d' - \mu_i$ | Machine vertices for machine $i$ |
|---|---|---|---|
| 1 | 2 | $6 - 2 = 4$ | $\lfloor (d' - \mu_1)/\omega \rfloor = \lfloor 4/3 \rfloor = 1$ |
| 2 | 5 | $6 - 5 = 1$ | $\lfloor (d' - \mu_2)/\omega \rfloor = \lfloor 1/3 \rfloor = 0$ |
| 3 | 6 | $6 - 6 = 0$ | $\lfloor (d' - \mu_3)/\omega \rfloor = \lfloor 0/3 \rfloor = 0$ |
| 4 | 2 | $6 - 2 = 4$ | $\lfloor (d' - \mu_4)/\omega \rfloor = \lfloor 4/3 \rfloor = 1$ |

Table 5.2: Determining the number of machine vertices when $d' = 6$ in the example instance with initial machine loads.

Observe that $\sum_{i=1}^{4} \Big( \lfloor (6 - \mu_i)/3 \rfloor \Big) = 2 < 5$. This means the makespan of $d' = 6$ could not be respected, so the procedure increases the proposed deadline to $d' = 7$. The algorithm then repeats the same task of calculating the available time for jobs of length three on each machine. We show these calculations for $d' = 7$ in Table 5.3, and we would obtain the same outcome as the previous iteration when $d' = 6$. We leave this iteration as an exercise to the reader, and proceed to $d' = 8$.

| machine $i$ | $\mu_i$ | $d' - \mu_i$ | Machine vertices for machine $i$ |
|---|---|---|---|
| 1 | 2 | $7 - 2 = 5$ | $\lfloor (d' - \mu_1)/\omega \rfloor = \lfloor 5/3 \rfloor = 1$ |
| 2 | 5 | $7 - 5 = 2$ | $\lfloor (d' - \mu_2)/\omega \rfloor = \lfloor 2/3 \rfloor = 0$ |
| 3 | 6 | $7 - 6 = 1$ | $\lfloor (d' - \mu_3)/\omega \rfloor = \lfloor 1/3 \rfloor = 0$ |
| 4 | 2 | $7 - 2 = 5$ | $\lfloor (d' - \mu_4)/\omega \rfloor = \lfloor 5/3 \rfloor = 1$ |

Table 5.3: Calculating the number of machine vertices for each machine when $d' = 7$ in the example.

The algorithm now determines the available load on each machine for $d' = 8$ as shown in Table 5.4. Next, the algorithm applies Construction 5.1.3 to produce

a bipartite graph $G = (M \cup J, E)$ as shown in Figure 5.1. This bipartite graph consists of five job vertices, and five machine vertices. Of the machine vertices, two correspond to machine 1; one corresponds to machine 2; and the last two machine vertices represent machine 4.

| machine $i$ | $\mu_i$ | $d' - \mu_i$ | Machine vertices for machine $i$ |
|---|---|---|---|
| 1 | 2 | $8 - 2 = 6$ | $\lfloor (d' - \mu_1)/\omega \rfloor = \lfloor 6/3 \rfloor = 2$ |
| 2 | 5 | $8 - 5 = 3$ | $\lfloor (d' - \mu_2)/\omega \rfloor = \lfloor 3/3 \rfloor = 1$ |
| 3 | 6 | $8 - 6 = 2$ | $\lfloor (d' - \mu_3)/\omega \rfloor = \lfloor 2/3 \rfloor = 0$ |
| 4 | 2 | $8 - 2 = 6$ | $\lfloor (d' - \mu_4)/\omega \rfloor = \lfloor 6/3 \rfloor = 2$ |

Table 5.4: Computing the number of machine vertices that represent each machine when $d' = 8$ in the example $R|p_{i,j} \in \{3, \infty\}|C_{max}$ with initial loads instance.

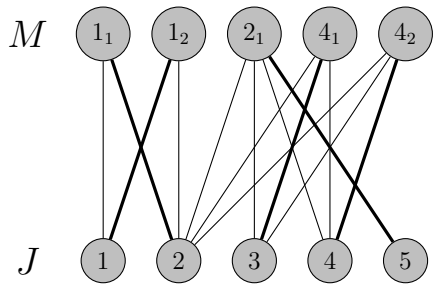



Figure 5.1: A bipartite graph $G = (M \cup J, E)$ by the algorithm for $d' = 8$ using Construction 5.1.3 on an example instance of $R|p_{i,j} \in \{3, \infty\}|C_{max}$ with initial loads. A maximum matching $M_{max}$ found in $G$ is shown with darker edges and vertex labels. Such a matching saturates all the job vertices of $J$.

After finding a maximum matching $M_{max}$ in $G$, we observe that the matching contains edges incident to every job vertex, and $|M_{max}| = 5$. Thus, we can schedule all $n = 5$ jobs, with makespan is eight. Based on the edges of the maximum matching, jobs 1 and 2 are assigned to machine 1, job 5 is scheduled on machine 2, and jobs 3 and 4 are assigned to machine 4. The algorithm returns an optimal schedule, then terminates. ■

As a final remark for this chapter, $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads can be modelled as a set of restricted assignment instances. Consider the $m \times m$ matrix

$$C = \begin{pmatrix} \mu_1 & \infty & \dots & \infty \\ \infty & \mu_2 & \dots & \infty \\ \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \mu_m \end{pmatrix},$$

where each $\mu_i \geq 0$, $0 \leq i \leq m$. Then, let $m \times n$ processing requirement matrix $P'$ be from an instance of $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$. Now consider the processing requirement matrix $P = (C\ P')$. Let us show that any instance $(P, m, n)$ of this form can be solved in polynomial time. For $1 \leq i \leq m$, let $\mu_i$ be the initial load for machine $i$. Next, use $P'$ as the processing requirement matrix for the $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads instance. Thus, we can solve in polynomial time such restricted assignment instances as a corollary to Theorem 5.1.2.

**Corollary 5.1.6.** *Consider the* $m \times m$ *matrix*

$$C = \begin{pmatrix} \mu_1 & \infty & \dots & \infty \\ \infty & \mu_2 & \dots & \infty \\ \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \mu_m \end{pmatrix},$$

*where* $\mu_i \geq 0$, *for* $1 \leq i \leq m$. *Also, let there be a* $m \times n$ *processing requirement matrix* $P'$ *of* $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$. *Then, the makespan problem on UPMs with processing requirement matrix* $P = (C\ P')$, *can be solved in polynomial time.*

# Chapter 6

# New Results for NP-hard Subclasses of $R||C_{max}$

In our investigation into subclasses of $R||C_{max}$, we also focused on subclasses with three-valued processing times. That is, only three types of job lengths are allowed as entries in a processing requirement matrix. We present new intractability results in this area, and approximation algorithms with approximation factors on par with or better than the best known approximation ratio for $R||C_{max}$ of two. Finally in Section 6.2 we present some NP-hard subclasses of $R||C_{max}$ and approximation algorithms for certain subclasses that can violate the hardness of approximation of $R||C_{max}$. Though the existence of such approximation algorithms for certain subclasses of $R||C_{max}$ does not imply $\mathsf{P} = \mathsf{NP}$, they may provide insight into what properties may lead to an intractable scheduling problem. To this date, such subclasses of $R||C_{max}$ and their approximation algorithms do not seem to have been discussed in the literature. At the end of this chapter, we give two conjectures as future work.

## 6.1 $R|p_{i,j} \in \{1,2,4\}|C_{max}$

In order to determine the computational complexity of $R|p_{i,j} \in \{1,2,4\}|C_{max}$, we will prove that $R|p_{i,j} \in \{1,2,4\}|C_{max}$ is NP-hard.

**Theorem 6.1.1.** *$R|p_{i,j} \in \{1,2,4\}|C_{max}$ is NP-hard.*

*Proof.* Consider the makespan problem on UPMs when processing times are restricted to $p_{i,j} \in \{1,2,4\}$. We show $\{1,2,4\}$-MAKESPANUPM is NP-complete in order to prove the makespan problem with restricted processing time of $p_{i,j} \in \{1,2,4\}$ is NP-hard. Consider the following decision problem.

**Problem 6.1.2** ($\{1,2,4\}$-MAKESPANUPM)**.** *Let $\mu \in \mathbb{Z}^+ \cup \{0\}$. Given an $m \times n$ processing requirement matrix $P$ with processing times $p_{i,j} \in \{1,2,4\}$, does there exist a schedule with makespan at most $\mu$?*

If we are given a schedule, we can check in polynomial time if the makespan of the schedule is at most $\mu$. So $\{1,2,4\}$-MAKESPANUPM $\in$ NP. We will reduce from the $q$DM problem when $q = 4$. Recall that $m'$ and $n'$ are the number of 4-sets in a family $F$ and number of elements in each of the 4 disjoint sets, respectively. When $m' \geq n'$ for a $4DM$ instance, we define a construction that takes a $4DM$ problem instance and constructs an instance of $\{1,2,4\}$-MAKESPANUPM in polynomial time.

**Construction 6.1.3.** Given an instance $I = (A_1, \ldots, A_q, F, n', m')$ of the $4DM$ problem, where $m' \geq n'$, we construct from $I$ an instance $I' = (P, m, n, \mu)$ of $\{1,2,4\}$-MAKESPANUPM in polynomial time. Create a $m \times n$ processing requirement matrix $P$, where $m = m'$ and $n = 4n' + m' - n'$. There are two types of jobs: dummy jobs, and element jobs. A *dummy job* takes 4 time units on every machine. For

$1 \leq i \leq m'$, an *element job* takes 1 time unit on machine $i$ if element $a_{s,t} \in T_i$ of $F$ and 2 time units otherwise, where $1 \leq s \leq 4$ and $1 \leq t \leq n'$. Let $4n'$ of the jobs be element jobs and $(m' - n')$ of the jobs be dummy jobs. Assign $\mu = 4$. Such a processing requirement matrix $P$ consists of elements $p_{i,j} \in \{1, 2, 4\}$. ✠

**Example 6.1.4.** To demonstrate Construction 6.1.3, we present an example. Suppose we wanted to build a $\{1, 2, 4\}$-MAKESPANUPM instance from a $4DM$ problem instance. Let $m' = 3$, $n' = 2$. Also, let there be four disjoint sets $A_1 = \{1, 2\}$, $A_2 = \{3, 4\}$, $A_3 = \{5, 6\}$, $A_4 = \{7, 8\}$, and a family of three 4-sets

$$F = \{T_1, T_2, T_3\} = \{\{1, 3, 5, 7\}, \{2, 3, 5, 7\}, \{2, 4, 5, 8\}\}.$$

We now apply Construction 6.1.3 to our instance $(A_1, A_2, A_3, A_4, F, 3, 2) \in 4DM$. Build an $m \times n$ processing requirement matrix $P$, where

$$m = m' = 3,$$

and

$$n = 4n' + (m' - n') = 9.$$

There will be $4n' = 8$ element jobs, and $(m' - n') = 1$ dummy jobs. The one dummy job takes 4 time units on every machine. An element job takes 1 time unit on machine $i$ if its corresponding element is in a 4-set $T_i \in F$. Otherwise, an element job takes 2 time units on a machine. For instance, $1 \in T_1$, so element job 1 takes 1 time unit on machine 1. Element job 1 takes 3 time units on each other machine

as $1 \notin T_2$ and $1 \notin T_3$. Set $\mu = 4$. So processing requirement matrix

$$P = \begin{pmatrix} 4 & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\ 4 & 2 & 1 & 1 & 2 & 1 & 2 & 1 & 2 \\ 4 & 2 & 1 & 2 & 1 & 1 & 2 & 2 & 1 \end{pmatrix}.$$

There does not exist a schedule with makespan at most four for $P$. It is worth noting that the $4DM$ problem instance we provided does not have a matching. ■

Consider an arbitrary instance $I \in 4DM$, where $m' \geq n'$. Apply Construction 6.1.3 to $I$ to produce an instance $I' \in \{1, 2, 4\}$-MAKESPANUPM. In the next step of our proof, we demonstrate that there is a matching $F'$ in $I$ if and only if there exists a schedule with makespan at most $\mu$ for $I'$. First, we wish to show that if there is a matching $F'$ in $I$, then there is a schedule with makespan at most 4 for instance $I'$. If there is a matching, then $|F'| = n'$ and $\bigcup_{T_i \in F'} T_i = \bigcup_{b=1}^{4} A_b$. After creating $I'$ by applying Construction 6.1.3 with $I$, there are $n = 4n' + (m' - n')$ jobs. Each 4-set in $F'$ corresponds to a machine with 4 element jobs that can be scheduled for one time unit each. This means exactly $n'$ of the machines are each scheduled with 4 element jobs. So each machine processing element jobs completes in 4 time units. As there are $(m' - n')$ dummy jobs, exactly $(m' - n')$ of the machines are assigned one dummy job since one machine available for each dummy job. Since each dummy job can be assigned to take 4 time units, the makespan of the schedule produced is at most 4. Thus, if there is a matching, then there exists a schedule with makespan at most 4 using the processing requirement matrix $P$ of $I'$.

Next, we show that if we are given $I' = (P, m, n, \mu)$ and it has a schedule with makespan at most $\mu = 4$, then there is a matching $F'$ in instance $I \in 4DM$. Suppose

there exists a schedule that has a makespan at most 4, and does not correspond to a matching. Thus, there is at least one element job that is scheduled to take 2 time units. Since the makespan of the schedule is at most 4, an element job cannot be scheduled along with a dummy job on the same machine. Exactly $(m' - n')$ dummy jobs must be scheduled on $(m' - n')$ machines; one dummy job on each machine. No other jobs can be scheduled on these machines while respecting the makespan 4. All that remain are $4n'$ element jobs to be scheduled on $n'$ machines. The $4n'$ element jobs must be scheduled so that the processing time of each job is 1 time unit. So 4 element jobs are assigned to each of the $n'$ remaining machines. But, if one element job takes 2 time units, at least one machine has load greater than 4, which violates our assumption. Thus, if there exists a schedule with makespan at most 4, then there is a matching. Therefore, assuming $m' \geq n'$, there is a matching with instance $I \in 4DM$ if and only if there is a schedule with makespan at most 4 for instance $I' \in \{1, 2, 4\}$-MAKESPANUPM.

If $m' \geq n'$, then apply Construction 6.1.3 to obtain instance $I' = (P, m, n, \mu = 4)$. Our construction $I'$ is an instance of $\{1, 2, 4\}$-MAKESPANUPM and can be produced in polynomial-time from any instance $I \in 4DM$. If there is a schedule with makespan at most 4, then it is a "yes" instance. A "no" instance is one where either $m' < n'$, or there is no schedule with makespan at most 4. Thus, $4DM \leq_P \{1, 2, 4\}$-MAKESPANUPM, and $\{1, 2, 4\}$-MAKESPANUPM is NP-complete. Therefore, $R|p_{i,j} \in \{1, 2, 4\}|C_{max}$ is NP-hard. □

Below are two direct consequences of Theorem 6.1.1. Let positive integers $p$ and $q$ be constants, where $1 < p < q$. The first of these two results follows from the proof

if 2 is replaced with $p$, and 4 is replaced with $q$.

**Theorem 6.1.5.** *Let $p, q \in \mathbb{Z}^+$ be fixed, where $1 < p < q$. $R|p_{i,j} \in \{1, p, q\}|C_{max}$ is* NP*-hard.*

**Corollary 6.1.6.** *Let $p, q, r \in \mathbb{Z}^+$ be fixed, where $p < q < r$. $R|p_{i,j} \in \{p, q, r\}|C_{max}$ is* NP*-hard.*

### 6.1.1 2-approximation algorithm

Next, we present an approximation algorithm for $R|p_{i,j} \in \{1, 2, 4\}|C_{max}$.

**Theorem 6.1.7.** *There is a 2-approximation algorithm for $R|p_{i,j} \in \{1, 2, 4\}|C_{max}$.*

*Proof.* Pseudocode for our algorithm can be found in Algorithm 6.1.8. Given any $m \times n$ processing requirement matrix $P$ when $p_{i,j} \in \{1, 2, 4\}$, create a copy of the processing requirement matrix $P'$ where all entries are only ones and twos by taking all entries that equal four and replacing such entries with two. Next, apply to $P'$ the polynomial time algorithm for $R|p_{i,j} \in \{1, 2\}|C_{max}$ described in Theorem 4.1.13. Upon completion of this procedure, assign all the jobs in $P$ as they were assigned in $P'$. This terminates in polynomial time and produces a feasible solution for $R|p_{i,j} \in \{1, 2, 4\}|C_{max}$.

Let us show that this algorithm has an approximation factor of two. Let us consider the assignment matrix of a schedule $S$ produced by the algorithm. Suppose $\alpha$ is a machine that finishes last in the schedule $S$. Since the algorithm applies an procedure that solves optimally an instance when $p'_{i,j} \in \{1, 2\}$ after constructing $P'$,

$$\sum_{j=1}^{n} p'_{\alpha,j} x_{\alpha,j} = OPT(P').$$

Each job that is assigned two time units in processing requirement matrix $P'$ is either of length two or four in processing requirement matrix $P$. So

$$OPT(P') \leq OPT(P),$$

and

$$\sum_{j=1}^{n} p_{\alpha,j}x_{\alpha,j} \leq \sum_{j=1}^{n} (2p'_{\alpha,j})x_{\alpha,j} = 2 \cdot \sum_{j=1}^{n} p'_{\alpha,j}x_{\alpha,j} \leq 2 \cdot OPT(P).$$

Thus, when we consider any schedule $S$ produced by our algorithm for a given instance,

$$\sum_{j=1}^{n} p_{\alpha,j}x_{\alpha,j} \leq 2 \cdot OPT(P).$$

Therefore, this is a 2-approximation algorithm.

**Algorithm 6.1.8** 2-approximation algorithm for $R|p_{i,j} \in \{1,2,4\}|C_{max}$

```
1: procedure MakespanUPMOneTwoFour(P, m, n)
2:     Let X be a m × n matrix with zero entries;        ▷ m × n assignment matrix
3:     Let P' be a m × n matrix with zero entries; ▷ m × n processing requirement matrix
4:     for i := 1 to m do                  ▷ Construct processing requirement matrix P'
5:         for j := 1 to n do
6:             if p_{i,j} ≥ 2 then
7:                 p'_{i,j} := 2;
8:             else
9:                 p'_{i,j} := 1;
10:            end if
11:        end for
12:    end for
13:    X := MakespanUPMs12(P', m, n);                ▷ Optimal schedule for P'
14:    return X;                                     ▷ Return feasible schedule for P
15: end procedure
```

□

Consider the following tight example. Create a $m \times n$ processing requirement matrix $P$, where $n = m$. For $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, m$, if $i = j$, then

let $p_{i,j} = 4$. Otherwise, let $p_{i,j} = 2$. Applying the 2-approximation algorithm to $P$ produces a modified processing requirement matrix $P'$, where each $p'_{i,j} = 2$. As a result, it is possible for this algorithm to pick each diagonal entry, which yields a schedule with makespan of four. The optimal schedule has makespan of two.

## 6.2 $R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$, and $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$

Scheduling *short* or *long* jobs on certain machines can arise in multiprocessor scheduling. So a subclass of $R||C_{max}$ of pragmatic interest has processing times that are restricted to $p_{i,j} \in \{1, 2, \infty\}$. We know that $R|p_{i,j} \in \{1, \infty\}|C_{max}$ and $R|p_{i,j} \in \{1, 2\}|C_{max}$ are polynomial-time solvable by Theorem 4.1.3 and Theorem 4.1.13, respectively. What about when we consider arbitrary processing requirement matrices with processing times $p_{i,j} \in \{1, 2, \infty\}$? We show $R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$ is NP-hard. This subclass is closely related to the hardness of approximation of $R||C_{max}$ proved by Lenstra *et al.* [33].

**Theorem 6.2.1** (Lenstra *et al.* [33], Theorem 5)**.** *$R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$ is* NP*-hard.*

*Proof.* In order to prove the makespan problem on UPMs when $p_{i,j} \in \{1, 2, \infty\}$ is NP-hard, let us show its decision problem variant is NP-complete. Consider the following decision problem.

**Problem 6.2.2** (MAKESPANUPM-$\{1, 2, \infty\}$)**.** *Given a $m \times n$ processing requirement matrix $P$, where $p_{i,j} \in \{1, 2, \infty\}$. Does there exist a schedule with makespan at most $d$?*

Clearly MAKESPANUPM-$\{1, 2, \infty\} \in$ NP, because one can find the load of a machine that completes last in polynomial time. We show that there is a polynomial-time reduction between the $3DM$ problem and MAKESPANUPM-$\{1, 2, \infty\}$.

Suppose we are given an arbitrary instance $I$ of the 3DM problem. Begin by applying Construction 2.2.4 to $I$. If a trivial "no" instance is constructed, ensure the processing requirement matrix has processing times that consist of $p_{i,j} \in \{1, 2, \infty\}$. Let $d = 5$. Next, include *slack jobs*. A slack job is one that can be assigned to only one machine and no others (i.e., the other machines are set to take $\infty$ time units). Include a slack job for each machine that takes one time unit, then an additional slack job for each machine that takes two time units. This procedure includes an additional $2m'$ jobs and takes a polynomial number of steps. Call the resulting scheduling instance $I'$. It is clear that $I' \in$ MAKESPANUPM-$\{1, 2, \infty\}$. Since Construction 2.2.4 can be performed in polynomial time, this modified construction can be built in polynomial time. Next, let us prove a result that links a $3DM$ to the schedules produced.

Consider a schedule $S$ produced by our procedure. Notice two slack jobs are assigned to each machine. Ignore all the slack jobs in $S$ and the load of each machine decreases by three time units. If we ignore the slack jobs of $S$, then we need to show there is a matching if and only if $S$ has makespan of at most two. By Lemma 2.2.5, when we consider the whole schedule of $S$, there is a matching if and only if $S$ has makespan of at most five as each machine must execute exactly three time units of slack jobs. Thus, there is a matching if and only if there exists a schedule with makespan at most five.

If there is a schedule with makespan at most five, then it is a "yes" instance. Also,

if there is no schedule with makespan at most five, then $I'$ is a "no" instance as it does not correspond to a matching. Thus, $3DM \leq_P$ MAKESPANUPM-$\{1, 2, \infty\}$, and MAKESPANUPM-$\{1, 2, \infty\} \in$ NP-complete. Therefore, $R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$ is NP-hard.

□

As a consequence of Theorem 6.2.1, we obtain the following computational complexity result.

**Theorem 6.2.3.** *Let* $p, q \in \mathbb{Z}^+$*, where* $p \leq q$*.* $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$ *is* NP*-hard.*

At this stage, we could attempt to design an approximation algorithm for subclass $R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$. One method that could be taken is to combine both approaches used in the polynomial-time algorithms of $R|p_{i,j} \in \{1, 2\}|C_{max}$ and $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads. We break the proposed algorithm into two major steps. The first step is to pick jobs of length one using a modified processing requirement matrix by pretending entries containing $p_{i,j} = \infty$ are $p_{i,j} = 2$ instead. Upon building this, solve it as an instance of $R|p_{i,j} \in \{1, 2\}|C_{max}$ using the algorithm by Lenstra *et al.* [33] given in Theorem 4.1.13. Now enter step two, and reassign a job length of one to all the jobs of length two that were not assigned. Taking the original processing requirement matrix, form a new processing requirement matrix with columns of the remaining jobs. Change all entries in this new matrix that are $p_{i,j} = 1$ to $p_{i,j} = 2$; call the processing requirement matrix $K$. Let initial load $\mu_i$ be the load of machine $i$ after the assignment of jobs of length one in the first step. Apply our algorithm for $R|p_{i,j} \in \{2, \infty\}|C_{max}$ with initial loads to assign all the remaining jobs using $K$. This is a 3-approximation algorithm for $R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$. Can we

obtain a better approximation ratio? Yes, we can. We find a better $k$-approximation algorithm in Theorem 6.2.4 by focusing on $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$. It is important to note that no $k$-approximation algorithm for $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$ with $k < 2$ exists in literature to this date.

### 6.2.1 $(q/p)$-approximation algorithm for $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$

To open this section, we prove one of our main results in this thesis.

**Theorem 6.2.4.** *There is a $(q/p)$-approximation algorithm for $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$, where $p \leq q$.*

*Proof.* First, we describe the approximation algorithm, then we demonstrate its approximation ratio. The pseudocode for our algorithm is given by Algorithm 6.2.5.

Suppose we are given an arbitrary $m \times n$ processing requirement matrix $P$. Create another $m \times n$ processing requirement matrix $\Psi$ as follows. For each entry $p_{i,j} \neq \infty$, let $\psi_{i,j} = 1$, and $\psi_{i,j} = \infty$ otherwise. Clearly, this can be done in polynomial time. Notice that $\Psi$ is an instance of the makespan problem on UPMs when processing times are restricted to $\psi_{i,j} \in \{1, \infty\}$. Apply the algorithm described for Theorem 4.1.3 using $\Psi$ as the processing requirement matrix. Denote the schedule produced by this process as $S_{\{1,\infty\}}$. Next, using $S_{\{1,\infty\}}$, schedule job $j$ to machine $i$ for $p_{i,j}$ time units if job $j$ is assigned to machine $i$ in $S_{\{1,\infty\}}$. All jobs are then scheduled in polynomial time, and is a feasible schedule using processing requirement matrix $P$. Therefore, the following algorithm terminates in polynomial-time and produces a feasible schedule for any given instance of $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$.

We consider two cases, first when $p = q$, then the case when $q > p$. For $q = p$,

the last step of the algorithm only extends the schedule by a factor of $p$, and consist of only jobs of length $p$. This means only jobs of length one are being replaced by jobs of length $p$. By Theorem 4.1.3, the algorithm produces a schedule with optimal makespan when $p = q$.

Next, consider when $q > p$. Notice that each job is either of length $p$ or length $q$ when assignd to machines, and $q/p > 1$. So to determine the quality of a schedule produced by the algorithm, rescale the jobs assigned to be of lengths 1 and $q/p$, respectively. By considering this transformed schedule, the makespan of the original schedule is only $p$ times larger. Let $P'$ contain entries $p'_{i,j} \in \{1, q/p, \infty\}$. Call the transformed schedule $S'$.

Let us consider the assignment of jobs by the algorithm. Also, consider an optimal schedule $S^*$ that has undergone the same scaling as $S'$. Such an optimal schedule has makespan $(OPT(P)/p)$. Define optimal assignments of $S^*$ to be $x^*_{i,j} = 1$ only if job $j$ is assigned to machine $i$ in schedule $S^*$, and $x^*_{i,j} = 0$ otherwise. Let $\alpha$ and $\beta$ each be a machine that completes last for schedules $S'$ and $S^*$, respectively.

The makespan of schedule $S'$ is

$$\sum_{j=1}^{n} p'_{\alpha,j} x_{\alpha,j} = \sum_{\substack{\text{j=1,} \\ p_{\alpha,j} = p}}^{n} x_{\alpha,j} + \sum_{\substack{\text{j=1,} \\ p_{\alpha,j} = q}}^{n} \Big( (q/p) x_{\alpha,j} \Big).$$

Since $q/p > 1$, we have

$$\sum_{\substack{\text{j=1,} \\ p_{\alpha,j} = p}}^{n} x_{\alpha,j} + (q/p) \cdot \sum_{\substack{\text{j=1,} \\ p_{\alpha,j} = q}}^{n} x_{\alpha,j} \leq (q/p) \cdot \sum_{j=i}^{n} x_{\alpha,j}.$$

For an optimal schedule using processing requirement matrix $P$, let machine $\phi$ be

a machine that is assigned the most jobs. Clearly,

$$\sum_{j=1}^{n} x^*_{\phi,j} \leq \sum_{j=1}^{n} p'_{\beta,j} x^*_{\beta,j} = OPT(P)/p.$$

Recall that the algorithm constructs a processing requirement matrix $\Psi$, where $\psi_{i,j} \in \{1, \infty\}$. An optimal schedule for $\Psi$ is found by the algorithm described in Theorem 4.1.3. Each job in schedule $S'$ is assigned by the resulting assignments for $\Psi$. Then, the algorithm produces a schedule for which the maximum number of jobs assigned to a machine is minimized, and

$$\sum_{j=1}^{n} x_{\alpha,j} \leq \sum_{j=1}^{n} x^*_{\phi,j}.$$

As a consequence,

$$(q/p) \cdot \sum_{j=i}^{n} x_{\alpha,j} \leq (q/p) \cdot \sum_{j=i}^{n} x^*_{\phi,j} \leq (q/p) \cdot (OPT(P)/p).$$

Thus, when we consider a schedule produced by the algorithm,

$$\sum_{i=1}^{n} x_{i,j} p_{i,j} \leq (q/p) \cdot OPT(P).$$

Therefore, we have presented a $(q/p)$-approximation algorithm for $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$.

□

Here we give a tight example for our $(q/p)$-approximation algorithm presented in Theorem 6.2.4. Create a $m \times 1$ processing requirement matrix, where $m \geq 3$. Let $p_{1,1} = p$, $p_{2,1} = q$, and all remaining entries be $p_{i,j} = \infty$. The algorithm can assign job 1 to machine 2, though a schedule with optimal makespan has job 1 assigned to machine 1. The schedule produced has makespan of $q$, but the schedule with optimal makespan has makespan of $p$.

**Algorithm 6.2.5** $(q/p)$-approximation algorithm for $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$

```
1: procedure MakespanUPMAssignmentPQ(P, m, n)
2:     Let X be a m × n matrix with zero entries;        ▷ m × n assignment matrix
3:     Let Ψ be a m × n matrix with zero entries;  ▷ m × n processing requirement
   matrix
4:     for i := 1 to m do                ▷ Construct processing requirement matrix Ψ
5:         for j := 1 to n do
6:             if p_{i,j} ≠ ∞ then
7:                 ψ_{i,j} := 1;
8:             else
9:                 ψ_{i,j} := ∞;
10:            end if
11:        end for
12:    end for
13:    X := MakespanUPMsAssignment(Ψ, m, n);      ▷ Optimal schedule for Ψ
14:    return X;                                  ▷ Return feasible schedule for P
15: end procedure
```

Observe that the obtained approximation ratio for this algorithm depends entirely on the values the processing requirement matrix entries take. This means that if values are close to one another, we can produce schedules with near optimal solutions. This subclass of $R||C_{max}$ need not follow the hardness of approximation results for the general problem. For example, there is a $(5/4)$-approximation algorithm for $R|p_{i,j} \in \{4, 5, \infty\}|C_{max}$, and $5/4 < 3/2$.

### 6.2.2 2-approximation algorithm for $R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$

Notice that the $(q/p)$-approximation algorithm gives us an approximation factor of two for the sub-instance of the makespan problem on UPMs when $p_{i,j} \in \{1, 2, \infty\}$, as $q = 2$ and $p = 1$.

**Theorem 6.2.6.** *There is a* 2*-approximation algorithm for* $R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$.

### 6.2.3 $(q/p)$-approximation algorithm for $R|p_{i,j} \in \{p, q\}|C_{max}$

Observe that $R|p_{i,j} \in \{p, q\}|C_{max}$ is NP-hard, because $R|p_{i,j} \in \{p, q\}|C_{max}$ when $q \neq 2p$ is NP-hard. We can modify our $(q/p)$-approximation algorithm from Theorem 6.2.4 to be a simple greedy algorithm due to a way our approximation algorithm assigns jobs for these instances. Observe that the processing requirement matrix $\Psi$ produced by the algorithm for any instance of this form contains only unit processing times. But, this means $\Psi$ is a processing requirement matrix for $R|p_{i,j} = 1|C_{max}$. So we can bypass constructing $\Psi$, and modify our $(q/p)$-approximation algorithm as follows. For $j = 1, \ldots, n$, schedule job $j$ on machine $((j-1) \bmod m) + 1$. We will refer to this variation as the *greedy* $(q/p)$*-approximation algorithm*, and its pseudocode is given as Algorithm 6.2.8. Since assigning jobs in this manner also gives an optimal assignment for $\Psi$ when each $\psi_{i,j} = 1$, our argument for Theorem 6.2.4 still holds. Clearly, this $(q/p)$-approximation algorithm terminates in $\Theta(n)$ steps. As a consequence we obtain the following theorem.

**Theorem 6.2.7.** *There is a* $(q/p)$*-approximation algorithm for* $R|p_{i,j} \in \{p, q\}|C_{max}$*. Greedy* $(q/p)$*-approximation algorithm terminates in* $\Theta(n)$ *steps.*

### 6.2.4 Generalizing to Certain Multiple-Valued Instances of $R||C_{max}$

What if we generalized our results from Section 6.2.1 for restricted two-valued instances to restricted multiple-valued instances between two fixed integral values? Let $p, q \in \mathbb{Z}^+$ be fixed, where $p \leq q$. Define $A(p, q) = \{a \in \mathbb{Z}^+ \mid p \leq a \leq q\}$ .

**Algorithm 6.2.8** $(q/p)$-approximation algorithm for $R|p_{i,j} \in \{p, q\}|C_{max}$

```
1: procedure MakespanUPMPQ(P, m, n)
2:     Let X be a m × n matrix with zero entries;        ▷ m × n assignment matrix
3:     i := 1;
4:     for j := 1 to n do                                ▷ Cyclically assign jobs to machines
5:         X_{i,j} = 1
6:         i := i + 1;
7:         if i > m then
8:             i := 1;
9:         end if
10:    end for
11:    return X;                                         ▷ Return feasible schedule for P
12: end procedure
```

Consider the subclass $R|p_{i,j} \in A(p, q) \cup \{\infty\}|C_{max}$. That is, given $p, q \in \mathbb{Z}^+$ and $p \leq q$, let there be a processing requirement matrix $P$ with either positive integer entries $p \leq p_{i,j} \leq q$, or $p_{i,j} = \infty$. This subclass of the makespan problem on UPMs is NP-hard as when processing times $p_{i,j} \in \{p, q, \infty\}$, the problem is NP-hard by Theorem 6.2.3. In this section, we consider $R|p_{i,j} \in A(p, q) \cup \{\infty\}|C_{max}$, and then $R|p \leq p_{i,j} \leq q|C_{max}$, a NP-hard special case when $P$ consists only of positive integral entries inclusively between $p$ and $q$.

**Theorem 6.2.9.** *Let $p, q \in \mathbb{Z}^+$ be constants, where $p \leq q$. There is a $(q/p)$-approximation algorithm for $R|p_{i,j} \in A(p, q) \cup \{\infty\}|C_{max}$.*

*Proof.* Assume there are constants $p, q \in \mathbb{Z}^+$, and $p \leq q$. Given an arbitrary processing requirement matrix $P$, apply the $(q/p)$-approximation algorithm given in Theorem 6.2.4. Since this algorithm terminates in polynomial time, what remains to be shown is that the approximation factor remains $(q/p)$ for subclass $R|p_{i,j} \in A(p, q) \cup \{\infty\}|C_{max}$.

Next, list the entries of $A(p, q)$ as $A(p, q) = \{z_1, \ldots, z_{|A(p,q)|}\}$. For $b = 1, 2, \ldots, |A(p, q)|$,

$p \leq z_b \leq q$. Using a similar technique as in our proof for Theorem 6.2.4, introduce a processing requirement matrix $P'$ that is $P$ with rescaled job lengths $p'_{i,j} \in \{z_1/p, z_2/p, \ldots, z_{|A(p,q)|}/p, \infty\}$. Denote an optimal schedule for $P$ as $S^*$, where $x^*_{i,j} = 1$ if job $j$ is scheduled on machine $i$. Let $\alpha$ be a machine that completes last when we apply the $(q/p)$-approximation algorithm of Theorem 6.2.4. Call the schedule produced $S$, and denote its corresponding schedule with $P'$ schedule $S'$. The makespan of schedule $S'$ is

$$\sum_{i=1}^{n} p'_{\alpha,j} x_{\alpha,j} = \sum_{b=1}^{|A(p,q)|} \sum_{\substack{\text{j=1,} \\ p_{\alpha,j} = z_b}}^{n} \Big( (z_b/p) x_{\alpha,j} \Big).$$

Each $z_b \leq q$ and $q/p > 1$, so

$$\sum_{b=1}^{|A(p,q)|} \sum_{\substack{\text{j=1,} \\ p_{\alpha,j} = z_b}}^{n} \Big( (z_b/p) x_{\alpha,j} \Big) \leq (q/p) \cdot \sum_{j=i}^{n} x_{\alpha,j}. \tag{6.1}$$

Once we have inequality (6.1), we can employ a similar argument as the proof for Theorem 6.2.4. In an optimal schedule for $P$, let machine $\phi$ be a machine that is assigned the most jobs. Denote machine $\beta$ as a machine that completes last for schedule $S^*$, where $S^*$ is an optimal schedule. Observe

$$\sum_{j=1}^{n} x^*_{\phi,j} \leq \sum_{j=1}^{n} p'_{\beta,j} x^*_{\beta,j} = OPT(P)/p.$$

Our algorithm employs the procedure described for Theorem 4.1.3 on $\Psi$. The assignment of jobs for schedules $S$ and $S'$ are determined by the assignments made for $\Psi$. So, the algorithm produces a feasible schedule for which the maximum number of jobs assigned to each machine is minimized. Then it follows that

$$\sum_{j=1}^{n} x_{\alpha,j} \leq \sum_{j=1}^{n} x^*_{\phi,j},$$

and

$$(q/p) \cdot \sum_{j=i}^{n} x_{\alpha,j} \leq (q/p) \cdot \sum_{j=i}^{n} x^*_{\phi,j} \leq (q/p) \cdot (OPT(P)/p).$$

Thus, the makespan of schedule $S$ is

$$\sum_{i=1}^{n} x_{i,j} p_{i,j} \leq (q/p) \cdot OPT(P).$$

Therefore, this is a $(q/p)$-approximation algorithm for $R|p_{i,j} \in A(p,q) \cup \{\infty\}|C_{max}$.

□

Notice that for $b = 1, \ldots, |A(p,q)|$, each $p \leq z_b \leq q$ in our proof. Any integral values between $p$ and $q$ can be used, and our $(q/p)$-approximation algorithm can still guarantee an approximation factor of $q/p$. So for $R|p \leq p_{i,j} \leq q|C_{max}$, we can employ the greedy $(q/p)$-approximation algorithm described as Algorithm 6.2.8, which completes in linear time.

**Theorem 6.2.10.** *There is a* $(q/p)$*-approximation algorithm for* $R|p \leq p_{i,j} \leq q|C_{max}$ *that terminates in* $\Theta(n)$ *worst-case time.*

Let us give an application of Theorem 6.2.10 for general instances of $R||C_{max}$. We can apply Theorem 6.2.10 when a processing requirement matrix has entries close enough in value to obtain better approximation results than the best known approximation algorithms for $R||C_{max}$. Let $\tau \in \mathbb{Q}^+$ be fixed and $p, q \in \mathbb{Z}^+$, where $p \leq q$. Given an arbitrary processing requirement matrix $P$, let $p$ and $q$ be its smallest and largest integral valued entries, respectively. Let $\tau = 2$. If $q \leq 2p$, then we apply our approximation algorithm to obtain a schedule with makespan at most $(q/p) \cdot OPT(P)$ with greater efficiency. If $q > 2p$, then apply one of the known 2-

approximation algorithms [13; 33; 42]. Likewise, if we consider other subclasses of $R||C_{max}$, other approximation factors can be guaranteed by changing $\tau$.

**Example 6.2.11.** Let us consider an example of the $(q/p)$-approximation algorithm. Consider a processing requirement matrix

$$P = \begin{pmatrix} 2 & 2 & \infty & 3 & 2 & \infty \\ 3 & \infty & 3 & \infty & 3 & \infty \\ 3 & \infty & 2 & \infty & 2 & \infty \\ \infty & 3 & \infty & 3 & 2 & 2 \end{pmatrix}.$$

The algorithm begins by constructing another processing requirement matrix $\Psi$, where $\psi_{i,j} \in \{1, \infty\}$:

$$\Psi = \begin{pmatrix} 1 & 1 & \infty & \textcircled{1} & 1 & \infty \\ \textcircled{1} & \infty & 1 & \infty & \textcircled{1} & \infty \\ 1 & \infty & \textcircled{1} & \infty & 1 & \infty \\ \infty & \textcircled{1} & \infty & 1 & 1 & \textcircled{1} \end{pmatrix}.$$

Next, the $(q/p)$-approximation algorithm solves optimally $\Psi$ by applying the polynomial-time algorithm in Theorem 4.1.3. Upon completion, this algorithm yields deadline $d = 2$, and produces a feasible schedule for $P$ using the assignment of jobs shown by circled entries in $\Psi$ above. This schedule has makespan of six, while the optimal makespan is four. The algorithm then terminates. ■

# Chapter 7

# Conclusions

We have explored many aspects of $R||C_{max}$. Though there has been little progress for improving the best known approximation factor of two for approximation algorithms of $R||C_{max}$, researchers found improved approximation algorithms for subclasses of $R||C_{max}$. In our investigation, we found a polynomial-time algorithm for $R|p_{i,j} \in \{\omega, \infty\}|C_{max}$ with initial loads. Also, we proved subclasses such as $R|p_{i,j} \in \{1, 2, 4\}|C_{max}$, $R|p_{i,j} \in \{p, q, r\}|C_{max}$, and $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$ are NP-hard. In response, we gave a 2-approximation algorithms for $R|p_{i,j} \in \{1, 2, 4\}|C_{max}$ and $R|p_{i,j} \in \{1, 2, \infty\}|C_{max}$. We gave a $(q/p)$-approximation algorithm for subclasses such as $R|p_{i,j} \in \{p, q, \infty\}|C_{max}$, and $R|p \leq p_{i,j} \leq q|C_{max}$. In pursuit of understanding the relationship between the tractable and intractable instances, we emphasized the relationship between matchings and $R||C_{max}$. Additionally, we demonstrated that not all subclasses of $R||C_{max}$ follow the general hardness of approximation results. In conclusion, we have contributed new results to theoretical computer science, and combinatorial optimization.

As a consequence of this work, an open problem stands for the NP-hard subclass $R|p_{i,j} \in A(p,q) \cup \{\infty\}|C_{max}$. The fact that $R|p_{i,j} \in A(p,q) \cup \{\infty\}|C_{max}$ does not apply to the hardness of approximation result for $R||C_{max}$ may indicate that a PTAS exists. Is there a PTAS for $R|p_{i,j} \in A(p,q) \cup \{\infty\}|C_{max}$? If there is no PTAS for this subclass, are there any non-trivial hardness of approximation results for $R|p_{i,j} \in A(p,q) \cup \{\infty\}|C_{max}$? Since there exists a $(q/p)$-approximation algorithm for the NP-hard subclass $R|p_{i,j} \in A(p,q) \cup \{\infty\}|C_{max}$, the hardness of approximation results for $R||C_{max}$ do not apply when $q/p < 3/2$.

# Appendix A

# Linear Programming

The following appendix is to prepare readers for any concepts in linear programming that may be presented in this thesis. In this appendix, the concept of a linear program (LP) is given; a system of non-negative linear inequalities with the goal of minimizing or maximizing an objective function. We discuss some forms LPs can take, summarize some computational complexity results relating to LPs, and the concept of a basic feasible solutions for a LP. Following this, introduced are integer programs (IP), a natural way to formulate many optimization problems of interest to theoretical computer science. At the end of this chapter, we discuss the integrality gap of an IP.

We recommend reading Section 1.2 on optimization problems before this appendix, because we adopt the terminology used in Section 1.2 when describing optimization problems.

# A.1 Linear Programs

At times, we employ linear programming to assist in solving optimization problems. Before defining what a linear program (LP) is, we provide an example. Consider the following optimization problem we call the Perfect Ice Cream Sundae Problem. Imagine you are the owner of an ice cream parlour called *Yummy's Ice Cream Treats*. Suppose you are given $\rho$ toppings, and taste value $\tau \in \mathbb{R}^+$. For $1 \leq i \leq \rho$, each topping $i$ has a cost $c_i \in \mathbb{R}^+$ in cents per gram ($\not{c}/g$), and a yumminess factor $y_i \in \mathbb{R}^+$, each set by the owner based on the performance of various ice cream sundaes served. First, assume that all sundaes served are the same size, so the amount of each topping used is of concern. Next, define, for each $1 \leq i \leq \rho$, an amount of topping $x_i$ grams to be placed on the sundae, where $x_i \geq 0$. Develop the perfect ice cream sundae that satisfies the following constraints while minimizing the cost of the sundae:

- The sum of the yumminess factors is at least the taste value $\tau$. This constraint ensures that the quality of taste based on your data is splendid.
- The amount of toppings used for each sundae is at most 3 grams. You boast how your sundaes only have 3 grams of toppings while maintaining a splendid flavour.

For this problem, denote an instance $I = (\tau, (c_1, \ldots, c_\rho), (y_1, \ldots, y_\rho))$, and for $\mathbf{x} = (x_1, \ldots, x_\rho)$, $c(I, \mathbf{x})$ as an objective value for the objective function $c$. Consider the following example instance of the Perfect Ice Cream Sundae Problem with $\rho = 2$.

**Example A.1.1.** Let $\tau = 3.5$. In Table A.1, we give the costs and the yumminess factor of each topping.

| $i$ | **Topping** | **Cost** ($c_i$) | **Yumminess Factor** ($y_i$) |
|---|---|---|---|
| 1 | Fudge | 3 ¢/$g$ | 5 |
| 2 | Sprinkles | 2 ¢/$g$ | 7 |

Table A.1: An example table of toppings for an instance of the Perfect Ice Cream Sundae Problem.

There are many feasible solutions to this problem. Selecting $x_1 = 2$ grams, $x_2 = 1$ gram is a feasible solution for this instance. The optimal solution that minimizes $c((3.5, (3, 2), (5, 7)), (x_1, x_2)) = 3x_1 + 2x_2$ is when $x_1 = 0$ grams, and $x_2 = 0.5$ grams. In this case, it seems a simple sundae with a dash of sprinkles minimizes their costs while maintaining the quality of taste. ■

The Perfect Ice Cream Sundae Problem is an example of an minimization problem we can formulate as linear program (LP). One LP formulation $LP_{ic}$ of this problem:

$$\text{minimize} \sum_{i=1}^{\rho} c_i x_i \tag{$LP_{ic}$}$$

$$\text{subject to} \sum_{i=1}^{\rho} y_i x_i \geq \tau \tag{A.1}$$

$$\sum_{i=1}^{\rho} x_i \leq 3 \tag{A.2}$$

$$x_i \geq 0, \text{ for } i = 1, 2, \ldots, \rho. \tag{A.3}$$

In this LP, we call:

- $\sum_{i=1}^{\rho} c_i x_i$ the *objective function*,
- inequalities (can be equalities) A.1 – A.2 the *constraints*,
- inequalities A.3 the *non-negativity* constraints.

A *feasible solution* $(x_1, \ldots, x_\rho)$ to $LP_{ic}$ is one that satisfies all the constraints and non-negativity constraints. The *objective value* is the output of the objective function for a given feasible solution $\mathbf{x}$. An *optimal solution* $(x_1^*, \ldots, x_\rho^*)$ for $LP_{ic}$ is feasible, and has minimum objective value. Notice that the objective function, and constraints are comprised of linear terms with respect to variables $x_1, \ldots, x_\rho$. In $LP_{ic}$, we assumed there were two constraints, $\rho$ non-negativity constraints, and $\rho$ variables.

With a better understanding of some of the terminology we need, let us give the general linear programming problem. Let $A$ be a $v \times h$ real-valued matrix, where each row corresponds to the coefficients of linear terms on the LHS of each constraint. Call matrix $A$ a *constraint matrix*. We assume that the rows of $A$ are linearly independent. Also, let $\mathbf{b}$ and $\mathbf{c}$ be real-valued column vectors of length $v$ and $h$, respectively. The LP problem is to find a feasible solution $\mathbf{x}$ (a column vector of length $h$) so that $\mathbf{c}^T\mathbf{x}$ is minimized, subject to a system of inequalities or equalities called constraints, while each $\mathbf{x} \geq 0$. As a convention, we will write $\mathbf{x}$, $\mathbf{c}$, and $\mathbf{b}$ horizontally when being described as a tuple, but will properly use them as column vectors otherwise. An important note is that maximizing $\mathbf{c}^T\mathbf{x}$ is equivalent to minimizing $-\mathbf{c}^T\mathbf{x}$, so we do not need to consider LPs that state their goal as "maximize". An LP is called *feasible* if there exists a feasible solution to the LP. Also, if the set of feasible solutions for the LP is empty, then the LP is called *infeasible*. When we discuss our constraints as a system of linear inequalities of any form, an LP is said to be in *general form*. If constraints of the system take the form $A\mathbf{x} = \mathbf{b}$, with goal of minimization, and valid non-negativity constraints, we say the LP is in *standard form*. That is, the standard

form of an LP is

$$\text{minimize } \mathbf{c}^T\mathbf{x}$$

$$\text{subject to } A\mathbf{x} = \mathbf{b}$$

$$\mathbf{x} \geq 0.$$

Though we can formulate LPs in general form, we present LPs in standard form, because any LP can be written in standard form. Standard form is useful when we discuss basic feasible solutions in Section A.3. Next, we describe how to convert any LP in general form into a LP in standard form. After, we show how to give a LP in standard form for our instance of the Perfect Ice Cream Sundae Problem from Example A.1.1.

To turn a general form LP into a standard form LP, replace all inequality constraints with equality constraints as follows. Given an inequality constraint of the form "$\sum_{j=1}^{h} a_{i,j}x_j \leq b_i$", introduce a *slack variable* $s_i \geq 0$, and write instead "$\sum_{j=1}^{h} a_{i,j}x_j + s_i = b_i$". A slack variable represents a compensatory amount to enforce equality. Likewise, we can introduce a *surplus variable* $s_i \geq 0$ when an inequality constraint is of the form "$\sum_{j=1}^{h} a_{i,j}x_j \geq b_i$", and write as a substitute "$\sum_{j=1}^{h} a_{i,j}x_j - s_i = b_i$". Similar to a slack variable, a surplus variable models excess numeric contribution in case the original constraint did not meet with equality. Finally, include all slack and surplus variables in the objective function, each with coefficient of value 0. As a byproduct of these two possible transformations, a LP in standard form is produced that models the original LP in general form.

Let's put everything together now in an example to help demonstrate the procedure of transforming a general form LP to a standard form LP and give how we

defined the linear programming problem.

**Example A.1.2.** In Example A.1.1, we presented an instance of the "Perfect" Ice Cream Sundae Problem. Using our formulation $LP_{ic}$, we give the problem as a LP in general form.

$$\text{minimize } 3x_1 + 2x_2 \qquad (LP_{ic}\text{-general})$$

$$\text{subject to } 5x_1 + 7x_2 \geq 3.5 \qquad (A.4)$$

$$x_1 + x_2 \leq 3 \qquad (A.5)$$

$$x_1 \geq 0,\ x_2 \geq 0. \qquad (A.6)$$

Following our definition of the LP problem, $\mathbf{c} = (3, 2)$, $\mathbf{b} = (3.5, 3)$. We now put $LP_{ic}$-general in standard form. Introduce a surplus variable $s_1 \geq 0$ for inequality A.4, and slack variable $s_2 \geq 0$ for inequality A.5. Then, we obtain $LP_{ic}$-standard:

$$\text{minimize } 3x_1 + 2x_2 + 0s_1 + 0s_2 \qquad (LP_{ic}\text{-standard})$$

$$\text{subject to } 5x_1 + 7x_2 - s_1 = 3.5 \qquad (A.7)$$

$$x_1 + x_2 + s_2 = 3 \qquad (A.8)$$

$$x_1 \geq 0, x_2 \geq 0,\ s_1 \geq 0, s_2 \geq 0. \qquad (A.9)$$

The constraint matrix $A$ for this standard form LP is

$$A = \begin{pmatrix} 5 & 7 & -1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix}.$$

So the goal is to find what non-negative values for $(x_1, x_2, s_1, s_2)$, so that

$$\begin{pmatrix} 5 & 7 & -1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} 3.5 \\ 3 \end{pmatrix},$$

and the objective value

$$\begin{pmatrix} 3 & 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \end{pmatrix} = 3x_1 + 2x_2$$

is minimized. ■

To conclude this section, we next introduce the idea of unboundedness for a LP. We say that a LP is *unbounded* if the objective function of the LP is not bounded from above. When a LP is unbounded, variables in the LP can always be reassigned to yield a better objective value without finding an optimal solution. Unbounded LPs can be feasible, but do not have an optimal solution.

## A.2 A Brief Note on Solving Linear Programs

Recall the problem instance of the Perfect Ice Cream Sundae Problem given in Example A.1.1 of Section A.1. We stated that the optimal solution for $LP_{ic}$-general was $\mathbf{x} = (0, 0.5)$. Consider the LP $LP_{ic}$-standard from Example A.1.2. The same optimal solution for picking the same toppings is $(x_1, x_2, s_1, s_2) = (0, 0.5, 0, 2.5)$. In

order to understand what makes this $\mathbf{x}$ an optimal solution for $LP_{ic}$-standard, let us first consider the possible feasible solutions for $LP_{ic}$-general.

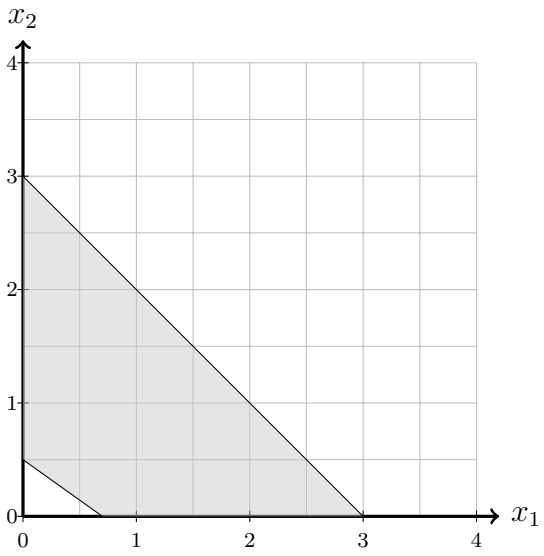



Figure A.1: A geometric plotting of the feasible region that corresponds to all the feasible solutions of $LP_{ic}$-general.

In order to be feasible, all the constraints need to be satisfied, such that each $x_1, x_2 \geq 0$. As seen in Figure A.1, this forms a closed region. We call such a region a *feasible region*, because all feasible solutions to $LP_{ic}$-general are found in it. Notice that the optimal solution sits at a corner of this feasible region. These corners solutions are called basic feasible solutions. We discuss basic feasible solutions in Section A.3.

Though we do not focus on linear programming itself in this thesis, it is fundamental to understand two points that need to be made about LPs. One, linear programs can be solved algorithmically. Two, there is a polynomial-time algorithm to solve any LP. The first known algorithm that solves any LP is the simplex method [6]. The simplex method traverses each of the corners of a feasible region to find an optimal

solution, which also exists at a corner if there is an optimal solution. Of the times we apply linear programming in this thesis, we do not use the simplex method. The simplex method in general is not a polynomial-time algorithm. In the worst case, the simplex method has an exponential running time [27]. Though the simplex method is not a polynomial-time algorithm, it performs well for many instances, and is often considered the "workhorse" for linear programming. In 1979, Khachiyan [26] developed the first polynomial-time algorithm for solving any LP. This algorithm is called the ellipsoid method. Though this result is theoretically significant, Karmarkar [24] in 1984 developed an algorithm that solves any LP in polynomial time, and is more efficient in practice than the ellipsoid method. Karmarkar's projection algorithm (or Karmarkar's algorithm) is an example of an interior-point method, that is, an algorithm that does not traverse the outer boundaries of a feasible region and, instead, finds a feasible solution that is optimal by traversing the inside of the feasible region.

## A.3 Basic Feasible Solutions of a Linear Program

Suppose we are given a LP in standard form with $v$ constraints and $h$ variables. Consider its $v \times h$ constraint matrix $A$ and some solution $\mathbf{x}$, so that $A\mathbf{x} = \mathbf{b}$. We say a solution $\mathbf{x}$ is *basic* if there are at most $v$ non-zero components of $\mathbf{x}$. A *basic feasible solution* (BFS) is a basic solution that is also a feasible solution for the given LP. When we consider a basic solution, we call $h - v$ of the variables that have value zero *non-basic variables*, and the remaining $v$ variables *basic variables*. A basic solution is called *non-degenerate* when there are exactly $v$ basic variables that are non-zero. Note that a basis $B$ for the column space of constraint matrix $A$ can be constructed

by taking linearly independent columns of $A$ that are indexed by each non-zero basic variable.

Consider the example instance we introduced for the Perfect Ice Cream Sundae problem in Section A.1. A solution $\mathbf{x} = (x_1, x_2, s_1, s_2) = (0.5, 0.5, 2.5, 2)$ is a feasible solution to the LP but is not a BFS, because there are more than two non-zero variables in $\mathbf{x}$. In Section A.2, we said that an optimal solution to the LP for this instance was $(x_1, x_2, s_1, s_2) = (0, 0.5, 0, 2.5)$. Observe that this is a (non-degenerate) BFS that occurs at a corner in the feasible region of the LP. This is no coincidence.

Since we have an understanding of what a BFS is, we can now provide a complete characterization for the existence of optimal solutions for any LP in standard form called the *Fundamental Theorem in Linear Programming.*

**Theorem A.3.1** (J. Matoušek and B. Gärtner [35], Theorem 4.2.3)**.** *Consider any arbitrary LP in standard form. Then the following is true:*

- *if there is no optimal solution for the LP, then the LP is unbounded or infeasible,*
- *if the LP is bounded and feasible, then there is an optimal solution for the LP,*
- *if there is an optimal solution, then there exists a BFS (see Section A.3) that is optimal.*

Finally, there is an important theorem in linear programming that follows from Theorem A.3.1 that says if the LP is bounded and feasible, then there is an optimal solution that occurs at one of the corners, and that this corner is the same as a BFS. In a standard form LP that is bounded, the number of corners is finite. Geometrically,

each corner when connected by edges from each constraint forms a convex polyhedron

$$P = \{\mathbf{x} \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0\},$$

the feasible region. We remind the reader that $A$ is the constraint matrix of a given LP. Each corner cannot be written as a convex combination of any two points in the feasible region, since $P$ is a convex hull. That is, given a corner $\mathbf{x}$ of $P$, there do not exist points $\mathbf{y}, \mathbf{z} \in P$ with $\mathbf{x} \neq \mathbf{y}$ and $\mathbf{x} \neq \mathbf{z}$, so that $\mathbf{x} = \lambda\mathbf{y} + (1 - \lambda)\mathbf{z}$ with $\lambda \in (0, 1)$. What we have been calling the corners of the feasible region are called *extreme points* of $P$. As we have previously stated, an extreme point of $P$ corresponds to a BFS of $P$. This correspondence is drawn from the linear independence of the columns indexed by non-zero basic variables in a BFS of $P$, and because an extreme point cannot be formed by a convex combination of different points in $P$. We now give this correspondence as a theorem, and we refer the reader to a proof of this result by Lau *et al.* [32].

**Theorem A.3.2** (Lau *et al.* [32], Theorem 2.1.5)**.** *Given a constraint matrix $A$, consider convex polyhedron $P = \{\mathbf{x} \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0\}$. The extreme points of $P$ are the BFSs of $P$.*

## A.4 Integer Programs, Linear Program Relaxations, and Integrality Gaps

To begin this section, let us define an Integer Program (IP). An IP takes the form,

$$\text{minimize } \mathbf{c}^T\mathbf{x}$$

$$\text{subject to } A\mathbf{x} \leq \mathbf{b}$$

$$\mathbf{x} \in \mathbb{Z}^h,$$

where $A$ is a $v \times h$ constraint matrix, $\mathbf{b}$ and $\mathbf{c}$ are real-valued column vectors of length $v$ and $h$, respectively and $\mathbf{x}$ is a column vector of length $h$ whose components all are integers (that correspond to what are often called *integrality constraints*). In many instances, $\mathbf{x} \in \{0,1\}^h$, because IPs can model numerous decisions in an optimization problem. That is, variables of this form can represent a yes/no choice with 1 or 0, respectively. Due to this, variables of this form are called *decision variables* and can model many NP-hard optimization problems. Unlike solving any linear programming problem, solving any general integer programming problem is NP-hard, because solving any general IP when $\mathbf{x} \in \{0,1\}^h$ is NP-hard [25]. In order to reconcile this, researchers have developed techniques for some IPs to find feasible solutions with an approximate objective value in polynomial time. One such technique we will focus on is LP relaxation of an IP, and rounding.

To illustrate the concepts throughout this section, we will use a continuing example, the *minimum vertex cover* problem. Let $G = (V, E)$ be a graph with vertices $V$ and undirected edges $E$. The goal is to find a subset of vertices $S \subseteq V$ such that every edge $e \in E$ is incident with a vertex in $S$ and $|S|$ is as small as possible. For example,

given a graph $G$ as shown in Figure A.2, the smallest vertex cover is of size two, and is $S = \{2, 6\}$. Every edge is incident with vertex 2 or vertex 6. The minimum vertex

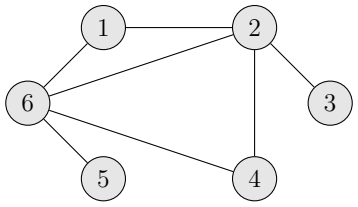



Figure A.2: A drawing of an input graph $G = (V, E)$ for the minimum vertex cover problem. The minimum vertex cover of $G$ is $S = \{2, 6\}$ as every edge is incident with either vertex 2 or 6.

cover problem can be formulated as an IP. Given a graph $G = (V, E)$, one such IP formulation is $IP_{VC}(G)$.

$$\text{minimize} \sum_{v \in V} x_v \tag{$IP_{VC}(G)$}$$

$$\text{subject to } x_u + x_v \geq 1, \forall \{u, v\} \in E \tag{A.10}$$

$$x_v \in \{0, 1\} \ , \forall v \in V \tag{A.11}$$

Observe that the above IP models the minimum vertex cover problem. Notice that for each vertex $v \in V$, $x_v = 1$ when vertex $v \in S$. The objective function of $IP_{VC}(G)$ minimizes the number of vertices included, and constraints A.10 ensure that at least one vertex in $S$ is at the endpoint of each edge. Given a feasible solution $\mathbf{x}$ to $IP_{VC}(G)$, the size of a vertex cover $|S| = \sum_{v \in V} x_v$.

Since we cannot solve this IP in polynomial time under the assumption that $\mathsf{P} \neq \mathsf{NP}$, maybe we can consider an "LP-counterpart" of $IP_{VC}(G)$ that can be solved in polynomial time. We can construct such a LP by relaxing the integrality constraints to be non-negativity constraints. This is called *relaxing* an IP, and the byproduct is

a LP called the *LP relaxation of an IP*. For instance, with $IP_{VC}(G)$, we can make a LP by replacing each $x_v \in \{0, 1\}$ with $x_v \geq 0$. We call this new LP $LP_{VC}(G)$.

$$\text{minimize} \sum_{v \in V} x_v \qquad (LP_{VC}(G))$$

$$\text{subject to } x_u + x_v \geq 1, \forall \{u, v\} \in E \qquad (A.12)$$

$$x_v \geq 0 \qquad , \forall v \in V \qquad (A.13)$$

Clearly, any feasible solution of $IP_{VC}(G)$ is a feasible solution for $LP_{VC}(G)$, but many feasible solutions for $LP_{VC}(G)$ are not feasible solutions for $IP_{VC}(G)$. So we have two problems. First, how can we obtain a feasible solution for $IP_{VC}(G)$ from an optimal solution of $LP_{VC}(G)$? Second, how "good" is the objective value of a feasible solution for $IP_{VC}(G)$ if we can find it? We can approach the second question by considering the integrality gap of an IP. Later we will give an example of how we could approach the first problem.

The *integrality gap* of an IP is the worst-case ratio for all instances between the optimal objective values of an IP and its relaxed LP counterpart. To be precise, given an instance $I$ of some minimization problem $\Pi$, the integrality gap of an IP is $max_I \frac{OPT(I)}{OPT_{LP}(I)}$, where $OPT_{LP}(I)$ is the optimal objective value of the LP relaxation for instance $I$. The integrality gap indicates a limit on how well a LP relaxation of an IP can approximate a feasible solution to the IP. If the integrality gap is found, researchers who design approximation algorithms employ techniques such as rounding to attempt to match the approximation factor with the integrality gap. In many cases, such an approximation algorithm can be designed from a proof for the integrality gap. For example, let us show that the integrality gap of $IP_{VC}(G)$ is two. The proof we

present is based on a proof given by Chekuri [2].

**Theorem A.4.1** (Chekuri [2])**.** *The integrality gap of $IP_{VC}(G)$ is at most two.*

*Proof.* Let graph $G = (V, E)$. Consider optimal solutions $\mathbf{x}^*$ and $\tilde{\mathbf{x}}$ for $LP_{VC}(G)$ and $IP_{VC}(G)$, respectively. Set $S = \{v \mid x_v^* \geq 1/2\}$. Since $\mathbf{x}^*$ is a feasible solution to $LP_{VC}(G)$, for any $\{u, v\} \in E$, $x_u^* + x_v^* \geq 1$. So, either $x_u^* \geq 1/2$, or $x_v^* \geq 1/2$, and at least $u$ or $v$ can be found in $S$. By definition, $S$ is a vertex cover, and $OPT(G) \leq |S|$. Now, let $OPT_{LP}(G) = \sum_{v \in V} x_v^*$. Then, $|S| \leq 2 \cdot OPT_{LP}(G)$. As a consequence, $OPT(G) \leq |S| \leq 2 \cdot OPT_{LP}(G)$. Therefore, $OPT(G)/OPT_{LP}(G) \leq 2$. □

To conclude this section, we describe a 2-approximation algorithm for the minimum vertex cover problem that is a consequence of the integrality gap of $IP_{VC}(G)$. Assuming we are given an some input graph $G$, first solve $LP_{VC}(G)$ to obtain $\mathbf{x}^*$. Construct a set $S$ as in the proof of Theorem A.4.1, then return $S$ as a vertex cover. It is straightforward to see that this is a 2-approximation algorithm from Theorem A.4.1.

# Bibliography

[1] V. Bonifaci and A. Wiese. Scheduling unrelated machines of few different types. *unpublished*, 2012. http://arxiv.org/abs/1205.0974.

[2] C. Chekuri. Vertex cover via LP, Jan. 2011. URL `https://courses.engr.illinois.edu/cs598csc/sp2011/Lectures/lecture_4.pdf`.

[3] C. Chekuri and M. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. *Journal of Algorithms*, 41:212–224, 2001.

[4] F. Chudak and D. Shmoys. Approximation algorithms for precedence–constrained scheduling problems on parallel machines that run at different speeds. *Journal of Algorithms*, 30:323–343, 2001.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press Cambridge, $2^{nd}$ edition, 2001.

[6] G. Dantzig. *Linear programming and extensions*. Princeton University Press, 1965.

[7] E. Davis and J. Jaffe. Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM (JACM)*, 28(4):721–736, 1981.

[8] T. Ebenlendr, M. Krčál, and J. Sgall. Graph balancing: A special case of scheduling unrelated parallel machines. In *Proceedings of the nineteenth annual ACM–SIAM Symposium on Discrete Algorithms (SODA)*, pages 483–490, 2008.

[9] T. Ebenlendr, M. Krčál, and J. Sgall. Graph balancing: A special case of scheduling unrelated parallel machines. *Algorithmica*, 68(1):62–80, 2014.

[10] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17 (3):449–467, 1965.

[11] P. Efraimidis and P. Spirakis. Approximation schemes for scheduling and covering on unrelated machines. *Theoretical Computer Science*, 359:400–417, 2006.

[12] H. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proceedings of the fifteenth annual ACM Symposium on Theory of Computing*, pages 448–456. ACM, 1983.

[13] M. Gairing, B. Monien, and A. Woclaw. A faster combinatorial approximation algorithm for scheduling unrelated parallel machines. *Theoretical Computer Science*, 380(1):87–99, 2007.

[14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman New York, 1979.

[15] R. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technological Journal*, 45(9):1563–1581, 1966.

[16] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[17] R. Graham, E. Lawler, J. Lenstra, and K. Rinnooy. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[18] L. Hall, A. Schulz, D. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.

[19] J. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[20] O. Ibarra and C. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM (JACM)*, 24(2):280–289, 1977.

[21] J. Jaffe. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 12:1–17, 1970.

[22] K. Jansen and L. Porkolab. Improved approximation schemes for scheduling unrelated parallel machines. *Mathematics of Operations Research*, 26(2):324–338, 2001.

[23] K. Jansen and R. Solis-Oba. Approximation schemes for scheduling jobs with chain precedence constraints. *International Journal of Foundations of Computer Science*, 21(1):27–49, 2010.

[24] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[25] R. Karp. *Reducibility Among Combinatorial Problems*. Springer, 1972.

[26] L. Khachiyan. A polynomial time algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[27] V. Klee and G. Minty. How good is the simplex algorithm? *Inequalities-III*, 1972.

[28] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 5th edition, 2012.

[29] C. Koulamas and G. Kyparisis. A modified LPT algorithm for the two uniform parallel machine makespan minimization problem. *European Journal of Operational Research*, 196(1):61–68, 2009.

[30] V. Kumar, M. Marathe, S. Parthasarathy, and A. Srinivasan. A unified approach to scheduling on unrelated parallel machines. *Journal of the ACM*, 56(5):28:1–28:31, 2009.

[31] V. Kumar, M. Marathe, S. Parthasarathy, and A. Srinivasan. Scheduling on unrelated machines under tree–like precedence constraints. *Algorithmica*, 55: 205–226, 2009.

[32] L. Lau, R. Ravi, and M. Singh. *Iterative Methods in Combinatorial Optimization*. Cambridge University Press, 2011.

[33] J. Lenstra, D. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1–3):259–271, 1990.

[34] L. Marchal. Scheduling on parallel machines, Oct. 2012. URL `http://graal.ens-lyon.fr/~lmarchal/scheduling/02.classical-P-machines-2012.pdf`.

[35] J. Matoušek and B. Gärtner. *Understanding and Using Linear Programming*. Springer, 2007.

[36] S. Micali and V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In $21^{st}$ *Annual Symposium on Foundations of Computer Science*, pages 17–27. IEEE, 1980.

[37] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Dover Publications, 1998.

[38] C. Potts. Analysis of a linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics*, 10(2):155–164, 1985.

[39] S. Sahni. Algorithms for scheduling independent tasks. *Journal of the Association for Computing Machinery*, 23(1):116–127, 1976.

[40] A. Schrijver. Min-max results in combinatorial optimization. *Mathematical Programming: The State of the Art – Bonn 1982*, pages 439–500, 1983.

[41] M. Serna and F. Xhafa. Approximating scheduling unrelated parallel machines in parallel. *Computational Optimization and Applications*, 21:325–338, 2002.

[42] E. Shchepin and N. Vakhania. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters*, 33:127–133, 2005.

[43] O. Svensson. Santa Claus schedules jobs on unrelated machines. In *STOC'11 Proceedings of the 43rd ACM Symposium on Theory of Computing*, pages 617–626, New York, 2011. ACM.

[44] A. Tucker. *Computer Science Handbook*. CRC press, 2004.

[45] F. Werner and N. Vakhania. Polynomial algorithms for scheduling jobs with two processing times on unrelated machines. In *Information Control Problems in Manufacturing*, volume 14, pages 93–97, 2012.

[46] D. Williamson and D. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

# Index