# Scheduling Unrelated Parallel Machines
Algorithms, Complexity, and Performance

**Dissertation**

von

Andreas Wotzlaw

Schriftliche Arbeit zur Erlangung des Grades
eines Doktor der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Paderborn, Juni 2006

# Acknowledgments

My acknowledgments go here to all persons who supported my research during the past three years. First and foremost, I would like to thank my advisor, Prof. Dr. Burkhard Monien, whose constant guidance, patience, and good example gave me a great support in the process of scientific work. It was him who introduced me to the field of scheduling problems and showed me in many helpful discussions how to improve my work and suggested in which direction my research should go. His research group with its friendly and relaxed atmosphere always provided a convenient and creative research environment to me. I also thank him for giving me the opportunity of a graduate fellowship.
I would also like to thank Prof. Dr. Friedhelm Meyer auf der Heide who has been my secondary advisor in the PaSCo graduate school. Additionally, I would like to thank Martin Gairing, Dr. Thomas Lücking, and Dr. Manuel Rode with whom I cooperated closely during the course of working on my PhD thesis. They provided me with a critical insight into various scientific problems and were a great help in understanding many theoretical issues concerning my and their work.
I am indebted to Dr. Robert Elsässer, Henning Meyerhenke, Dr. Ulf-Peter Schroeder, and Volker Winzenick for reading and commenting on parts of this work.
I thank all my colleagues for many discussions on scientific and non-scientific topics, and for technical and administrative support. To name those who I did not mention yet: Ulrich Ahlers, Bernard Bauer, Yvonne Bleischwitz, Dr. Rainer Feldmann, Sven Grothklags, Dr. Georg Kliewer, Dr. Ulf Lorenz, Dr. Tomáš Plachetka, Marion Rohloff, Thomas Sauerwald, Stefan Schamberger, Dr. Meinolf Sellmann, Thomas Thissen, Karsten Tiemann, Tobias Tscheuschner, and Oleksiy Znamenshchykov.
My research was made possible by the grant from the DFG graduate school No. 693 ”Scientific Computation: Application-oriented Modeling and Development of Algorithms” hosted by the Paderborn Institute for Scientific Computation (PaSCo) at the University of Paderborn. Here my work was supported well beyond financial coverage by the interdisciplinary environment of a graduate school with people from several different countries. At this place my special thanks go to the whole team of the graduate school, and especially to Prof. Dr. Michael Dellnitz, Dr. Valentina Damerow, Dr. Martin Otto, and Dr. Robert Preis.
Last but not least, I would also like to thank my family whose continuous support greatly helped me to finish this work.

Paderborn, June 2006

Andreas Wotzlaw

# Contents

# 1

# Introduction

## 1.1 Scheduling Problems

*Scheduling* is a form of decision-making which plays a crucial role in manufacturing as well as in service systems and industries. In the current competitive environment with rapidly changing conditions, effective scheduling has become a necessity for survival in the marketplace. Companies have to meet production deadlines and shipping dates committed to the customers and a failure to do so may result in a significant loss of money and customer confidence.

Generally speaking, scheduling concerns the allocation of limited *resources* to *tasks* over *time*. It is a *decision-making* process whose goal is the optimization of one or more *objectives* subject to some *constraints*.

The resources and tasks may take many forms. For instance, the resources may be machines in a workshop, runways at an airport, crews of a construction site, processing units in a computing environment, etc. The corresponding tasks may take the following forms: operations in a production process, take-offs and landings at an airport, stages of a construction project, executions of computer programs. Furthermore, each task may have a different processing time, priority level (weight), earliest possible starting time, and due date. The objective may also take many forms. One possible objective is the minimization of the completion time of the last task (we call it the minimization of makespan) and another is the minimization of the number of tasks completed after the committed due dates.

It may not be clear yet what impact schedules have on given objectives. The following question arises: Does it pay to invest time and money in constructing a good schedule rather than taking an arbitrary one? The answer is that usually the choice of a schedule has a *significant* impact on the system performance and the invested costs.

Scheduling began to be taken seriously in manufacturing at the beginning of the last century with the work of Henry Gantt [54, 55] and other pioneers who followed his ideas. However, it took many years for the first scheduling publications to appear in the operations research literature. Some of the first publications appeared in the early 1950s and contained results by W. E. Smith [159], S. M. Johnson [93], and J. R. Jackson [88]. During the 1960s a significant amount of work was done on dynamic programming and integer programming formulations of scheduling problems. After Richard Karp published his famous paper on complexity theory [97], the research in the 1970s focused mainly on the complexity hierarchy of scheduling problems. In the 1980s several different directions were pursued in theory and practice with an increasing amount of attention paid to stochastic scheduling problems [139]. Also, as personal computers started to be widely used by manufacturing facilities, scheduling systems were being developed for generating usable schedules in practice. This kind of system design and development was and still is being done by computer scientists, operations research analysts, and industrial engineers.

Scheduling can be difficult from both a theoretical and an implementation point of view. The types of theoretical difficulties are similar to those encountered in other branches of combinatorial optimization. Thus, theoretical results (e.g., solution methods, performance analysis, etc.) obtained for other optimization problems can be applied at least partially to the scheduling problems. Unfortunately, the difficulties encountered during the implementation process are of a completely different kind. They are related to the modeling of the real-world scheduling problems and the retrieval of information. Roughly speaking, one can say that every real-life scheduling problem is different, and thus requires strong problem-oriented modeling which in most cases has to be done from scratch. For a good introduction into the scheduling theory and its applications we refer the reader to, e.g., [18, 19, 22, 139].

## 1.2 Applications

Scheduling is an important decision-making tool for most manufacturing and service industries, as well as for most information-processing environments where it can have a major impact on the productivity of a process.

In manufacturing the purpose of scheduling is to minimize the production time and costs by telling a production facility what to make, when, with which staff, and on which equipment. Similarly, scheduling in service industries, such as airlines and public transport, aims at maximizing the efficiency of the operations and reducing their costs.

Modern computerized scheduling tools greatly outperform older manual scheduling methods. They provide the production scheduler with powerful graphical interfaces which can be used to visually optimize real-time work loads in various stages of the production (see, e.g, [140]). Pattern recognition used here allows the software to automatically create scheduling opportunities which might not be apparent without this view into the data. For example, an airline might wish to minimize the number of airport gates required for its aircraft, in order to reduce costs, and scheduling software can allow the planners to see how this can be done by analyzing time tables, aircraft usage, or the flow of passengers.

Companies use backward and forward scheduling to plan their human and material resources. Backward scheduling is planning the tasks from the due date to determine the start date and/or any changes in capacity required, whereas forward scheduling is planning the tasks from the start date to determine the shipping date or the due date. For practical examples we refer the reader to, e.g., [134, 137, 145, 165, 173].

In the following we give two examples which illustrate the role of the scheduling process in different real-life situations.

### 1.2.1 Scheduling of Tasks in a CPU

Scheduling is a key concept in computer multitasking and multiprocessing operating system design as well as in real-time operating system design. It refers to the way processes are assigned priorities and the time that Central Processing Unit (CPU) devotes to them. The processes may be represented by, e.g., different programs or parts of them. This assignment is carried out by a piece of software known as a scheduler. In general-purpose operating systems the goal of the scheduler is to *balance* processor loads on all processors and prevent any process from either monopolizing the processor or being starved for resources. In real-time environments, such as devices for automatic control in industry, e.g, robotics, the scheduler must also ensure that processes can meet deadlines. This is crucial for keeping the system stable.

In general, the exact processing times of tasks are not known in advance. However, the distribution of random processing times may be known in advance, including their expected values and their variances. They can easily be computed, for example, from the statistics collected during previous executions, or obtained earlier with similar applications. Additionally, every task usually has a certain priority factor (weight). The operating system typically allows users to specify the priority of each task. In this case, the objective is to minimize, for instance, the expected sum of weighted completion times for all tasks.

To avoid the situation where relatively short tasks remain in the system for a long time while waiting for much longer tasks with a higher priority to be completed, the operating system cuts the tasks into smaller pieces. It rotates then these slices on the CPU so that during a given interval, the CPU spends some amount of time processing portions of all the tasks. Thus, if the processing time of one of the tasks is very short, it will be able to leave the system relatively quickly.

An interruption of the processing of a task is referred to as a *preemption*. It is clear that the optimal policy in such an environment makes heavy use of preemptions.

### 1.2.2 Gate Assignments at an Airport

A huge number of difficult scheduling problems can be found in airline industry. Here however, we do not go deeply into details since it is beyond the scope of this work. In the following we give only a simple example based on the discussion from [139].

We consider an airline terminal at a major airport. There are dozens of gates and hundreds of airplanes arrive and depart daily. The gates are not identical and neither are the planes. Additionally, some gates are at locations with more space where large planes can be accommodated more easily, whereas the other gates may be located in such a way that bringing in the planes may be quite difficult.

Planes arrive and depart according to a certain schedule. However, this schedule is a subject to a significant amount of randomness which may be caused by the weather condition or events at other airports. While a plane remains at a gate, arriving passengers have to leave the plane, the plane has to be serviced, and departing passengers have to be boarded. Here, the scheduled departure time can be viewed as a due date. However, if a plane cannot land at the next airport because of anticipated congestion at the scheduled time, the plane is not allowed to take off and the boarding time is postponed. In such a case the plane could remain at a gate for an extended period, and thus prevent other planes from using the gate.

The scheduler has to assign planes to gates in such a way that the assignment is physically feasible and that it optimizes a number of objectives. In particular, he has to assign planes to suitable gates that have to be free at the respective arrival time. The objectives may include the minimization of work for airline personnel and the minimization of airplane delays.

Note that in this scenario the planes are equivalent to tasks with release dates and due dates, whereas the gates are equivalent to resources.

## 1.3 Deterministic Parallel Machine Scheduling Models

### 1.3.1 Preliminaries and Assumptions

A bank of machines in parallel is a setting that is important from both theoretical and practical points of view. From the theoretical viewpoint it is a generalization of the single machine (see, e.g., [139]). From the practical point of view it is important because the occurrence of resources in parallel is common in the real-world. Moreover, techniques for parallel machines are often used in decomposition procedures for multistage systems.

In the thesis, we consider only *deterministic* scheduling problems with machines in parallel that is problems for which the number of tasks (jobs), the number of parallel machines, and all job characteristics (like, e.g., processing times, sets of allowed machines for each job) are known in advance. Throughout the whole thesis we assume that each processing time is either a positive integer or infinity. We restrict our attention only to *non-preemptive* scheduling where a job, once assigned to a particular machine, may not be removed until it has finished its execution. It is assumed that the jobs are *independent*, i.e., they do not require scheduling in accordance with any precedence constraints.

Furthermore, we restrict the objective of the optimization problems considered here only to the minimization of the makespan (an exact definition is given later). When dealing with parallel machines, the makespan becomes an objective of significant interest. In practice one often has to deal with the problem of balancing the load on machines in parallel and by minimizing the makespan the scheduler ensures a good balance of the load.

One can consider scheduling parallel machines as a two-step process. Firstly, one has to determine which jobs are to be allocated to which machines, and secondly, one has to specify the sequence of the jobs allocated to each machine. With the makespan objective and no precedence constraints only the allocation process is important.

### 1.3.2 Tractability of Deterministic Scheduling Problems

In the early days of scheduling the research on various scheduling problems mainly consisted of classifying scheduling problems into easy (i.e., polynomially solvable) and hard (i.e., $\mathcal{NP}$-hard) ones. Nowadays researchers have become interested in better understanding of hard scheduling problems. Consequently, a much finer classification of these hard problems has been established [153]. One very active branch of research classifies hard scheduling problems according to their *tractability* or *approximability*.

Non-preemptive parallel scheduling problems tend to be difficult to solve. The vast majority of them is $\mathcal{NP}$-hard already for the case with a fixed number of machines, as we show later in this section. Thus, it is unlikely (in fact unless $\mathcal{P} = \mathcal{NP}$) that there exists a polynomial-time algorithm for computing a minimal makespan.

A standard way of dealing with $\mathcal{NP}$-hard problems is not to search for an optimal solution, but to search for *near-optimal* solutions. An algorithm that returns near-optimal solutions is called an *approximation algorithm*. If it runs in polynomial time, then it is called a *polynomial-time* approximation algorithm. An approximation algorithm that returns a near-optimal solution which is at most a factor $\rho$ above the optimum (where $\rho > 1$ is some fixed real number) is called a *$\rho$-approximation algorithm*, and the value $\rho$ is called the *worst-case performance guarantee* or the *approximation factor*. A family of $(1 + \varepsilon)$-approximation algorithms over all $\varepsilon > 0$ with polynomial running times is called a *polynomial time approximation scheme* or PTAS, for short. If the time complexity of a PTAS is also polynomially bounded in $1/\varepsilon$, then it is called a *fully* polynomial time approximation scheme or FPTAS, for short. With respect to relative performance guarantees, an FPTAS is essentially the strongest possible polynomial-time approximation result that we can derive for an $\mathcal{NP}$-hard problem. For surveys on polynomial-time approximation algorithms for scheduling we refer the reader to [20, 80].

Another possibility of solving difficult scheduling problems is to consider *heuristic* algorithms in hope of providing near-optimal results. However, by using heuristics, we usually have no guarantee for the quality of solution, or for the computation time, or for both.

### 1.3.3 Selected Deterministic Scheduling Problems

In this section we present in detail three most important deterministic non-preemptive scheduling problems on parallel machines. To this end, we give their exact descriptions and discuss the best approximation results obtained for these problems so far. Here and later throughout the thesis, we use the standard, three-field notation introduced by Graham et al. [73] to denote in a short way each considered scheduling problem.

The goal of every scheduling problem described below is to schedule without preemption a set $J$ of $n$ independent jobs on a set $M$ of $m$ parallel machines such that the makespan is minimized. For each job $i \in J$ we are given its processing time on every machine it can be assigned to. Again, we assume that each processing time is a positive integer or $\infty$. We define a matrix of processing times $\mathbf{P}$ in the natural way. To be more specific, in these problems we want to find an *assignment $\alpha$* according to Definition 1.3.1.

**Definition 1.3.1** *An assignment of jobs to machines is defined by a function $\alpha : J \mapsto M$. We denote $\alpha(i) = j$ if job $i$ is assigned to machine $j$.*

For any assignment $\alpha$, the *load* $\delta_j(\mathbf{P}, \alpha)$ on machine $j$ for a matrix of processing times $\mathbf{P}$

is the sum of processing times for the jobs that are assigned to machine $j$, i.e., for which $\alpha(i) = j$. We omit $\mathbf{P}$ in the notation of $\delta_j$ if $\mathbf{P}$ is clear from the context.

We define the *makespan* of an assignment $\alpha$ for a matrix of processing times $\mathbf{P}$, denoted by $C_{\max}(\mathbf{P}, \alpha)$, as the maximum load on a machine that is

$$C_{\max}(\mathbf{P}, \alpha) = \max_{j \in M} \delta_j(\alpha).$$

Additionally, by $C_{\max}(\mathrm{A})$ we denote the makespan of *any* assignment computed by algorithm A for matrix $\mathbf{P}$. Associated with matrix $\mathbf{P}$ is the *optimum makespan*, denoted by $C^*_{\max}(\mathbf{P})$, which is the least possible makespan of a valid assignment $\alpha$ that is

$$C^*_{\max}(\mathbf{P}) = \min_{\alpha} C_{\max}(\mathbf{P}, \alpha).$$

**Scheduling on identical parallel machines.** First, we consider the scheduling problem with identical machines denoted by Graham et al. [73] as $P||C_{\max}$. Here, the processing time of job $i \in J$ on any machine from $M$ is equal to $p_i$. This problem is of interest because minimizing the makespan has the effect of balancing the load over the various machines, what is important in practice. It is easy to see that already $P||C_{\max}$ with two machines is $\mathcal{NP}$-hard in the ordinary sense since PARTITION [56] polynomially reduces to it. Here, each of both machines represents a partition and the jobs are the items which we want to divide evenly into these two partitions.

During the last couple of decades many approximation algorithms and heuristics have been developed for the $P||C_{\max}$ problem. Perhaps the earliest and simplest result on the worst-case performance of *list scheduling* LS (see Section 4.1) is given by Graham [71]:

$$C_{\max}(\mathrm{LS})/C^*_{\max}(\mathbf{P}) \leq 2 - \frac{1}{m}.$$

The algorithm which he considered processes the jobs sequentially in an arbitrary order. It assigns each job to a machine whose load at the moment of assignment is smallest.

The *longest processing time* first (LPT) rule assigns at time $t = 0$ the $m$ largest jobs to $m$ machines. After that, whenever a machine is freed, the largest unscheduled job is put onto the machine. This heuristic tries to place the shorter jobs toward the end of the schedule where they can be used for balancing the loads. Thus, if the jobs are selected in LPT order the worst-case bound can be considerably improved, as it was shown by Graham in [72]:

$$C_{\max}(\mathrm{LPT})/C^*_{\max}(\mathbf{P}) \leq \frac{4}{3} - \frac{1}{3m}.$$

Furthermore, he showed that the approximation factor of LPT for two identical machines is $\frac{7}{6}$. Seiden, Sgall and Woeginger [154] proved that this is tight, i.e., LPT has the best possible approximation factor for the problem. For three machines, they gave a general worst-case bound of $\frac{1}{6}(1 + \sqrt{37}) \approx 1.18$. The running time of the LPT algorithm is $O(n \log n + n \log m)$.

More complicated algorithms were designed by Sahni [149] which can be used to obtain results as close to optimum as desired, but their running time grows rapidly as the desired accuracy is increased. They are exponential in $m$ whereas Graham's algorithm is not. More specifically, Sahni presented a family of algorithms $A_\varepsilon$ with $O(n(n^2/\varepsilon)^{m-1})$ running time which satisfy

$$C_{\max}(A_\varepsilon)/C^*_{\max}(\mathbf{P}) \leq 1 + \varepsilon.$$

Note that for any fixed $m$, the family of algorithms $A_\varepsilon$ becomes a PTAS. Later, Hochbaum and Shmoys [81] gave a better PTAS for $P||C_{\max}$ which runs in $O((n/\varepsilon)^{1/\varepsilon^2})$ time.

A somewhat better algorithm, called *multifit* (MF) and based on a completely different principle, was given by Coffman at al. [21]. The analysis of this algorithm was later improved by Friesen [49]. The idea behind MF is to find (by binary search) the smallest bin capacity such that a set of $m$ bins can still accommodate all jobs when the jobs are taken in order of non-increasing $p_i$ and each job is placed into the first bin which it fits. The set of jobs in the $j$-th bin will be processed by machine $j$. If $k$ packing attempts are made, the algorithm (denoted by $\text{MF}_k$) runs in time $O(n \log n + kn \log m)$ and satisfies

$$C_{\max}(\text{MF}_k)/C^*_{\max}(\mathbf{P}) \leq 1.2 + \frac{1}{2^k}. \tag{1.1}$$

Interestingly, Friesen gave an example in [49] showing that there exists an instance of the scheduling problem for which $C_{\max}(\text{MF}_k)/C^*_{\max}(\mathbf{P}) = \frac{13}{11} = 1.\overline{18}$. Note that for large $n$, the running times required for the LPT algorithm and for the $\text{MF}_k$ algorithm are dominated by the $O(n \log n)$ term of the initial sort operation. Furthermore, observe that by using binary search, no more than seven iterations of $\text{MF}_k$ are necessary to produce a schedule whose finish time is less than or equal to $(1.2 + 0.01)$ times optimal.

The bound given in (1.1) was further improved by Langston [107] who analyzed a modification of the MF algorithm using the same weighting function techniques as Friesen. The worst-case bound of his modified MF algorithm (MMF) satisfies

$$C_{\max}(\text{MMF})/C^*_{\max}(\mathbf{P}) \leq \frac{72}{61} \approx 1.18 < \frac{13}{11}.$$

**Scheduling on uniformly related parallel machines.** Now, we consider the scheduling problem on uniformly related (or just related, for short) machines denoted by Graham et al. [73] as $Q||C_{\max}$. Here, we are given a set of $n$ independent jobs with sizes $p_i$ that are to be executed on $m$ non-identical machines. These machines run at different speeds $s_j$. More precisely, if job $i$ is processed on machine $j$, it takes time $p_i/s_j$ to be completed. Note that this model is a generalization of the previous one with identical machines.

Liu and Liu [114] studied numerous questions dealing with related machines. Graham [71, 72] generalized the LPT algorithm to make it applicable for the $Q||C_{\max}$ problem. This natural extension works as follows. It assigns each job, in order of non-increasing size $p_i$, to a machine on which it will be completed soonest, i.e., it assigns job $i$ to machine $j$ for which $\delta_j + p_i/s_j$ is minimized. Here, $\delta_j$ is the load on machine $j$ just before the assignment of job $i$. For the general case Graham showed

$$C_{\max}(\text{LPT})/C^*_{\max}(\mathbf{P}) \leq 2 - \frac{2}{m+1}.$$

Additionally, Gonzales et al. [70] gave examples for which $C_{\max}(\text{LPT})/C^*_{\max}(\mathbf{P})$ approaches $\frac{3}{2}$ as $m$ tends to infinity. For two related machines, they showed that for any speed ratio

$q \geq 1$, the approximation factor of the LPT algorithm is at most $\frac{1}{4}(1 + \sqrt{17}) \approx 1.28$. Here, $q$ is the ratio between the speed of the faster machine and the speed of the slower machine. Recently, this case was investigated by Epstein and Favrholdt [41]. They gave the exact approximation factor of LPT in function of speed ratio $q$.

For a general setting of $m$ related machines, Friesen [50] proved that the approximation factor of the LPT algorithm satisfies

$$1.52 \leq C_{\max}(\text{LPT})/C^*_{\max}(\mathbf{P}) \leq \frac{5}{3}.$$

The LPT algorithm was also investigated by Dobson [35]. He claimed to improve the upper bound to $\frac{19}{12} = 1.58\overline{3}$. Unfortunately, his proof does not seem to be complete [41].

At the same time, Friesen and Langston [51] modified further the *multifit* algorithm by Coffman et al. [21] for the case with related parallel machines and proved by a minimal counterexample that its worst-case performance is substantially better than that of the LPT algorithm, with a worst-case bound of

$$C_{\max}(\text{MF})/C^*_{\max}(\mathbf{P}) < 1.4.$$

In the same work they gave a worst case example for which the solution is a little more than 1.341 times worse than optimal. Thus, we conclude that $1.341 < C_{\max}(\text{MF})/C^*_{\max}(\mathbf{P}) < 1.4$ for the $Q||C_{\max}$ problem.

The first PTAS for $Q||C_{\max}$ was given by Hochbaum and Shmoys [82]. Since the problem is strongly $\mathcal{NP}$-complete, their results are the best possible in the sense that if there were an FPTAS for this problem, then $\mathcal{P} = \mathcal{NP}$. Their approximation algorithm is based on a decision procedure which tests if there exists a schedule for a given problem instance where all jobs are completed by time $T$. Thus, the decision problem can be viewed as a bin-packing problem with variable bin sizes. The minimum $T$ is computed by a simple binary search procedure. The overall running time of the algorithm is $O((\log m + \log(3/\varepsilon))(m/\varepsilon)(n/\varepsilon)^{1/\varepsilon^2})$.

**Scheduling on unrelated parallel machines.** As the last deterministic scheduling problem here we consider the problem of scheduling a set $J$ of $n$ independent jobs on a set $M$ of $m$ *unrelated* machines without preemption. The processing time of job $i$ on machine $j$ is denoted by $p_{ij}$. Following Graham's notation, the problem is denoted by $R||C_{\max}$. For analysis purposes, we define $U = \max_{i\in J, j\in M}\{p_{ij} \neq \infty\}$, and denote by $A$ the number of pairs $(i, j)$ with $p_{ij} \neq \infty$. The problem can be formulated as the following mixed integer program (MIP):

$$\begin{array}{lll} \text{MIP:} & \min & T \\ & \text{s.t.} & \sum_{i\in[n]} p_{ij}x_{ij} \leq T, \qquad \forall\, j \in [m] \\ & & \sum_{j\in[m]} x_{ij} = 1, \qquad \forall\, i \in [n] \\ & & x_{ij} \in \{0, 1\}, \qquad \forall\, i \in [n], j \in [m] \end{array} \tag{1.2}$$

Here, $x_{ij}$ is a 0/1 assignment variable which is equal to 1 (respectively to 0) if job $i$ is assigned (respectively not assigned) to machine $j$, i.e., for which $\alpha(i) = j$. The objective is to minimize a non-negative variable $T$ which corresponds to the makespan. The first set of $m$ constraints associated with machine $j$ ensures that the load $\delta_j$ on every machine $j$ is at most $T$. The next $n$ constraints associated with jobs mean that every job $i$ has to be completed. The last integrality constraints ensure that each job is assigned to one and only one machine.

Using the complexity hierarchy of deterministic scheduling problems, scheduling unrelated parallel machines belongs to the most difficult scheduling problems. It is easy to see that it is a generalization of both previously presented problems, and hence it belongs also to the class of $\mathcal{NP}$-hard problems.

Within many manufacturing environments (but not only, see for more examples Section 1.2), there are often groups of similar workstations that have a wide variety of similar equipment with differing performance characteristics. This can be explained by the fact that they either have been purchased for slightly differing products or are at different technological levels. Consequently, there are often banks of parallel machines that may not be identical to each other. When the machines are not identical and cannot be completely correlated by simple rate adjustments, they are said to be unrelated. In such a case, this environment is classified as *unrelated parallel machines environment*, and thus can be modeled as a $R||C_{\max}$ problem. Here again, the minimization of makespan has the effect of balancing the load over all machines.

Because of the importance of this problem for both the theoretical and the practical research on scheduling and planning, we have decided to dedicate this thesis *only* to this problem. Throughout the rest of this work, we concentrate especially carefully on this problem while discussing the various algorithms designed to solve it, their worst-case complexities, and practical efficiencies in solving large-scale scheduling problems.

Motivated by this and in contrast to the two previous cases, we present detailed related work on the $R||C_{\max}$ problem and several of its direct extensions including recent advances in the next separate section.

## 1.4 Related Work on Unrelated Parallel Machines

There exists a large amount of literature on scheduling independent jobs on unrelated parallel machines with the makespan minimization as the objective. In this section, we first concentrate very carefully on the existing results for the generic $R||C_{\max}$ problem, and then give several results developed for more specific cases of the scheduling problem with processing time characteristics like, e.g., precedence constraints, release dates, or processing costs.

### 1.4.1 Results for Generic $R||C_{\max}$

Horowitz and Sahni [84] presented a non-polynomial-time dynamic programming algorithm to compute a schedule with minimum makespan. They gave also the first FPTAS to approximate an optimum schedule with minimum makespan for the case when the number of unrelated machines $m$ is fixed. They proved that, for any $\varepsilon > 0$, an $(1 + \varepsilon)$-approximate solution can be computed in $O(nm(nm/\varepsilon)^{m-1})$ time, which is polynomial in both $n$ and $1/\varepsilon$ if

$m$ is fixed. However, for the case where the number of machines is specified as a part of the problem instance, an FPTAS is unlikely to exist. Lenstra et al. [112] also gave an approximation scheme for the problem with running time bounded by the product of $(n+1)^{m/\varepsilon}$ and a polynomial of the input size. Although for a fixed $m$ their algorithm is not fully polynomial, it has a much smaller space complexity than the one in [84].

Lenstra et al. [112] proved that unless $\mathcal{P} = \mathcal{NP}$, there is no polynomial-time approximation algorithm for the optimum schedule to the $R||C_{\max}$ problem with approximation factor less than $\frac{3}{2}$. They also presented a polynomial-time 2-approximation algorithm for $R||C_{\max}$. This algorithm computes first an optimal *fractional* (or preemptive) solution and then uses rounding to obtain a schedule for the discrete problem with approximation factor 2 (see Section 2.1.2). Shmoys and Tardos [157] generalized this technique to obtain the same approximation factor for the generalized assignment problem. Furthermore, they generalized the rounding technique to hold for any fractional solution (see Section 2.1.4 for more details). Recently, Shchepin and Vakhania [156] introduced a new rounding technique which yields an improved approximation factor of $2 - \frac{1}{m}$. This is so far the best approximation result for this problem.

The fractional unrelated scheduling problem can also be formulated as a *generalized maximum flow* problem, where the network is defined by the scheduling problem and the capacity of some edges, that corresponds to the makespan, is minimized (see Section 3.1.2 for more details). This generalized maximum flow problem is a special case of *linear programming* (LP). Using techniques of Kapoor and Vaidya [95] and by exploiting the special structure of the problem, an optimum fractional solution can be found with the interior point algorithm of Vaidya [166] in time $O(|E|^{1.5}|V|^2 \log(U))$.

In contrast to the linear programming methods, the aforementioned generalized maximum flow problem can also be solved with a purely *combinatorial* approach. Here, the makespan minimization is done by binary search which contributes a factor $\log(nU)$ to the running time. Computing generalized flows has a rich history going back to Dantzig [25]. The first combinatorial algorithms for the generalized maximum flow problem were exponential time augmenting path algorithms by Jewell [91] and Onaga [132]. Truemper [164] showed that the generalized maximum flow problem and the *minimum cost flow* problem are closely related. More specifically, he transformed a generalized maximum flow problem into a minimum cost flow problem by setting the cost of an edge to be the logarithm of the gain from the generalized maximum flow problem (see Section 3.1.2 for more details). Goldberg et al. [64] designed the first polynomial-time combinatorial algorithms for the generalized maximum flow problem. Their algorithms were further refined and improved by Goldfarb, Jin, and Orlin [65] and later by Radzik [144]. Radzik's algorithm is so far the fastest combinatorial algorithm for this problem with a running time of $O(|E|\,|V|(|E| + |V| \log |V|) \log U)$. In order to minimize makespan, this algorithm has to be called at most $O(\log(nU))$ times.

There exist a number of fast FPTASs for computing a fractional solution to the scheduling problem [44, 45, 90, 141, 143, 162]. Using the rounding technique from [157], this leads to a $(2+\varepsilon)$-approximation for the discrete problem. The approximation schemes can be divided into those that approximate generalized maximum flows and those that directly address the scheduling problem. The generalized maximum flow packing algorithm by Fleischer and

Wayne [45], the recursive fat-path algorithm by Radzik [143], and the recursive rounded fat-path algorithm by Tardos and Wayne [162] compute a $(1 + \varepsilon)$-approximation for the generalized maximum flow. They all have running time $\widetilde{O}(\log \varepsilon^{-1}|E|(|E| + |V| \log \log U))$, where the $\widetilde{O}(\cdot)$ notation hides a factor polylogarithmic in $|V|$. Here again, an extra factor of $O(\log(nU))$ is needed for the makespan minimization by binary search. Plotkin et al. [141], and Jansen and Porkolab [90] gave approximation schemes that directly address the scheduling problem. The latter one has the faster running time of $O(\varepsilon^{-2}(\log \varepsilon^{-1})mn \min\{m, n \log m\} \log m)$. This algorithm combines two previous approaches: dynamic [84] and linear [112] programming. Note that for any fixed $m$ approximate solutions of any fixed accuracy can be computed in linear time.

Fishkin et al. [44], using grouping techniques for input data combined with dynamic programming, developed an FPTAS for the non-preemptive $R||C_{\max}$ problem. In particular, the algorithm classifies all machines for a given job as fast, slow, expensive, or cheap in order to aid in the decision of whether to schedule a job on a particular machine or not. Their algorithm runs in $O(n) + (\log m/\varepsilon)^{O(m)}$ time. Note again that for any fixed $m$ and accuracy $\varepsilon$ the running time is linear in $n$.

Unrelated machine scheduling is a very important problem from a practical point of view, and therefore many heuristics and exact methods have been proposed to solve it. Existing techniques used here range from combinatorial approaches with list scheduling [27, 85, 127] (see Section 4.1), partial enumeration [126] (see Section 4.2), branch-and-greed [160] (see Section 4.3) to integer programming with cutting planes [125] (see Section 5.1) and branch-and-price (see Section 5.2).

Local search methods have also been employed to solve the $R||C_{\max}$ problem. Glass et al. [58] evaluated the performance of gradient search, simulated annealing, tabu search, and genetic algorithms. In addition, Srivastava [161] employed a tabu search heuristic with hashing to reduce the computational requirements of the tabu search. These authors populated the initial neighborhood with solutions obtained by the algorithms of Ibarra and Kim (see Section 4.1.1).

Finding a discrete solution for the unrelated scheduling problem can be formulated as a generalized *unsplittable* maximum flow problem (see Section 3.3). Several authors [34, 101, 105] have studied the unsplittable flow problem for usual flow networks. Kleinberg [101] formulated the problem of finding a solution with minimum makespan for the *restricted* scheduling problem on identical machines as an unsplittable flow problem. In the restricted case of the scheduling problem, for each job $i$ a subset $M_i \subseteq M$ of machines is given on which it is allowed to be executed. Gairing et al. [52], by exploiting the special structure of the network, gave a 2-approximation algorithm for the restricted scheduling problem on related machines based on preflow-push techniques. They also designed an algorithm for computing a Nash equilibrium [129] for the restricted $P||C_{\max}$ problem.

## 1.4.2 Results for Extended $R||C_{\max}$

At the time of this work, there exist very few results from the research conducted on the $R||C_{\max}$ problem with processing time characteristics like, e.g., precedence constraints, release dates, or processing costs. This situation can be explained particularly by a much bigger complexity of these models. Nevertheless, Jansen and Porkolab [89] considered the makespan minimization problem with operating costs and presented an FPTAS (for fixed $m$) with running time $n(m/\varepsilon)^{O(m)}$. Later, Fishkin et al. [44] considered the problem of minimizing the objective function that is a weighted sum of makespan and the total costs. They gave an FPTAS (again for fixed $m$) with a better running time $O(n) + (\log m/\varepsilon)^{O(m^2)}$. Note that for the case with $c_{ij} = 0$ their approximation scheme can be used for the generic $R||C_{\max}$ problem. Armacost and Salem [8] developed a decomposition methodology to solve the scheduling problem with sequence dependent setups and machine eligibility restrictions. Herrmann et al. [79] considered scheduling unrelated parallel machines with precedence constraints. Lancia [106] investigated the unrelated machine scheduling problem on two machines and jobs with release dates and tails.

## 1.5 Our Contribution

In this section we discuss briefly the main contributions presented in this thesis. As it was already partially indicated in Section 1.3.1, the goal of our research is both the development of new efficient algorithms as well as the practical investigation on various algorithmic tools for solving deterministic scheduling problems with unrelated parallel machines. In our work we restrict our attention only to the $R||C_{\max}$ problem. This was motivated partially by its intractability and partially by its great importance to both the theoretical and the practical research (see Section 1.3.1).

**New combinatorial algorithm for $R||C_{\max}$.** The main theoretical result of this work is a new combinatorial algorithm, Approximation-Unsplittable-Truemper, for computing an assignment to the unrelated scheduling problem with makespan at most twice the optimum (see Chapter 3). We prove that a 2-approximate schedule can be computed in $O(m^2 A \log(m) \log(nU))$ time, where $A$ is the number of pairs $(i, j)$ with $p_{ij} \neq \infty$. This is better than the previously known best time bounds of Vaidya's interior point algorithm [166] and Radzik's combinatorial algorithm [144]. In particular, this is the first time that a combinatorial algorithm always beats the interior point approach for this problem.

An essential element of our approximation algorithm is the procedure Unsplittable-Blocking-Flow from [52]. This procedure was designed to solve the unsplittable maximum flow problem in a bipartite network, which is defined by the restricted scheduling problem on identical machines. In this thesis, the connection to flow is more tenuous. We solve an unsplittable flow problem in a *generalized* bipartite network (see Section 3.3), which is defined by the unrelated scheduling problem. The generalized flow problem can be transformed to a minimum cost flow problem. Our algorithm uses a Primal-Dual approach [3] combined with a gain scaling technique to obtain a polynomial running time. To compute a flow among the edges with zero reduced cost (i.e., a blocking flow), it uses the procedure Unsplittable-Blocking-Flow as a subroutine.

Given some candidate value for the makespan, our algorithm finds an approximate solution for the generalized unsplittable flow problem in the two-layered bipartite network. Throughout the whole execution, the algorithm always maintains an integral assignment of jobs to machines. Each assignment defines a partition of the machines into underloaded, balanced, and overloaded machines. The overloaded machines are *heavily* overloaded that is their load is at least twice as large as the candidate makespan.

The main idea of our algorithm is to utilize the existence of overloaded machines in conjunction with the fact that we are looking for an approximate integral solution. We use this idea twice. On the one hand this allows us to show an improved lower bound on the makespan of an optimum schedule and thus to overcome the $(1 + \epsilon)$ error usually induced by the gain scaling technique. On the other hand this is also used to reduce the number of outer iterations to $O(m \log m)$ which is the main reason for the substantial running time improvement.

Our algorithm is a generic minimum cost flow algorithm without any complex enhancements for generalized flow computation. Overloaded and underloaded machines are treated as sources and sinks, respectively. The height of a node is equal to its minimum distance to a sink. In our algorithm the admissible network, used for the unsplittable blocking flow computation, consists only of edges and nodes which are on shortest paths from overloaded machines with minimum height to underloaded machines. This modification in the Primal-Dual approach is important for showing the improved lower bound on the makespan of an optimum schedule.

The Approximation-Unsplittable-Truemper algorithm is simpler and faster than the previously known algorithms. For the unrelated scheduling problem we have replaced the classical two-step technique, i.e., computing first a fractional solution and rounding afterward (see Chapter 2), by a completely integral approach. Our algorithm takes advantage from addressing the approximation problem directly. In particular, this allows us to benefit from an unfavorable preliminary assignment. We feel that this may also be helpful in other applications (see Chapter 7).

Identifying the connection to network flows may be the key idea for obtaining new combinatorial (approximation) algorithms for problems for which solving the LP relaxation and rounding is currently the (only) alternative. Our techniques do not improve upon the approximation factor for the unrelated scheduling problem. However, we still expect more exciting improvements for other hard problems to which our technique is applicable.

**Column generation scheme for $R||C_{\max}$.** A big drawback of the two-step approach (see Section 2.1.1) is the enormous number of assignment variables in the linear programming (LP) relaxation of the scheduling problem. This situation becomes especially critical for large-scale problems. In the worst case, it cannot even be possible to state all variables of the problem explicitly as needed. Furthermore, in each iteration of the simplex algorithm, which is more often used to solve the LP relaxation than other techniques, we look explicitly for a non-basic variable to price out and enter the basis. This operation becomes too costly when the number of variables is large. Consequently, the overall performance of the application decreases dramatically. To overcome this unfavorable behavior, a *column generation* approach can be applied to solve the LP relaxation.

Column generation is nowadays a prominent method to cope with a huge number of variables. We apply it to solve the LP relaxation of the unrelated scheduling problem (see Section 2.2). The appealing idea of our algorithm is to work only with a reasonably small subset of variables, forming the, so called, restricted master problem. More variables are added only when needed. Here, each variable corresponds to a separate column in the constraint matrix of the problem.

Since the initialization of the restricted master problem is a very important issue for column generation solution method, we pay special attention to this crucial problem and develop a new hybrid initialization procedure for our algorithm (see Section 2.2.4 and Section 6.2.2). We observe that poorly chosen initial columns lead the algorithm astray, whenever they do not resemble the structure of a possible optimal solution at all. Furthermore, even an excellent initial integer solution could be detrimental to solving a linear program by column generation as our experiments show.

**Randomized two-step approach for $R||C_{\max}$.** Since the *randomized rounding* technique has proved to be a powerful tool often used to attack hard optimization problems, we have decided to develop a randomized version of the two-step approach, Randomized-Two-Step-Scheduling, for the $R||C_{\max}$ problem (see Section 2.3), too.

While studying randomization techniques, we are interested in efficient, polynomial-time approximation algorithms that deliver solutions within a provably good tolerance compared to the optimal solution. The randomized rounding technique can be applied to a class of $\{0, 1\}$ integer linear programs. It is a probabilistic method, i.e., for the existence of results, we prove that the solution to an (mixed) integer program satisfies a certain property (e.g., an upper bound for the solution) by showing that a randomly generated solution satisfies that property with a non-zero probability.

In our approach, we solve first an LP relaxation of the unrelated scheduling problem given in (1.2), and then use randomization to return from the relaxation to the original optimization problem. Here, the second phase of the algorithm can be repeated several times in order to improve the quality of solution.

**LPT-based heuristics for $R||C_{\max}$.** The LPT algorithm by Graham [72] was originally designed to schedule jobs on identical machines and later adapted to solve a more general scheduling problem on related machines (see Section 1.3.3 for details).

We have developed three simple adaptations of the LPT algorithm to the $R||C_{\max}$ problem (see Section 4.1.3). Each of the heuristics is accustomed to one of three different processing time characteristics. Here, we consider three various types of correlation which may exist between machines and jobs. In particular, we take into consideration the following models: (1) with no correlations between machines and jobs, (2) with correlated machines, and (3) with correlated jobs. All three models are motivated by their appearance in real-world applications.

**Branch-and-price algorithm for $R||C_{\max}$.** The *branch-and-price* technique is a powerful tool for solving hard optimization problems. It offers an interesting alternative for the general purpose mixed integer programming based on decomposition or cutting planes. Branch-and-price combines column generation with a branch-and-bound scheme. The first one usually produces at the root node of the search tree tight lower bounds which are further improved during the branching process. Branching also helps to generate integer solutions.

In Section 5.2.2 we present a new branch-and-price algorithm which we have developed for scheduling unrelated parallel machines. The algorithm is an extension of the generic branch-and-price method. It begins with a computation of the initial integer solution to the scheduling problem. To this end, we use a simple heuristic method (see Section 6.2.2). We proceed with the exploration of the branch-and-bound tree unless there is no subproblem to consider. In each phase of the tree exploration we compute an optimal solution to the current subproblem using a column generation algorithm (see Section 2.2). In the generic implementation of the algorithm, we choose for a subproblem the one with a currently smallest lower bound. In case the solution is not integral, two new subproblems are generated for some fractional variable of the current solution. On this variable we perform also the branch.

Our algorithm uses several heuristic extensions which help branch-and-price to find good integer solutions earlier (see Section 5.2.5). The acceleration scheme which we use is based on the cooperation between column generation and local search presented in [30]. The strength of this hybrid scheme is diversification by means of using different algorithms for solving the same problem. Branch-and-price benefits from local search which is more effective in finding feasible solutions. But in turn, local search benefits from branch-and-price which provides it with diverse initial solutions.

**Experimental study.** In Chapter 6 we present an experimental study on various algorithms for solving the $R||C_{\max}$ problem. We restrict our research only to the algorithms discussed in this work. Nevertheless, it results in eighteen different solution techniques. These algorithms represent, to our best knowledge, the present state-of-the-art of solution methods for this scheduling problem. They use almost all types of computational frameworks developed for this problem so far.

The experiments are motivated, firstly, by the importance of the unrelated scheduling problem in general, both for the theoretical and the practical research, and secondly, by the wish to evaluate in practice the Approximation-Unsplittable-Truemper algorithm and other algorithms which we have developed.

Using the preliminary test results, we are able to improve further our combinatorial algorithm what leads to a significant increase of its overall performance (see Section 6.2.2).

Our experiments show that the combinatorial algorithm, Approximation-Unsplittable-Truemper, has a clear advantage over all tested two-step approaches (see Section 6.3.1). In particular, for large test instances it delivers solutions of the same or better makespans than other approaches, and in most cases is much faster than the LP-based techniques. For large instances with correlated machines, however, the column generation approach is the fastest solution method. Moreover, our experiments show that the implementations of Approximation-Unsplittable-Truemper require much less operational memory than the two-step techniques and branch-and-bound methods, which makes them more efficient and easier to handle, especially when the size of the problem grows.

The comparisons with exact methods indicate that our combinatorial algorithm is outperformed by them when only the quality of solution is considered (see Section 6.3.2).

However, the time performance of exact methods based on the branch-and-bound approach is satisfactory only for very small problem instances. The good quality of solutions is in the vast majority of the test instances obtained due to very long computation times. For large instances we need to wait much longer for solutions whose quality would be comparable to the quality obtained in much shorter time by Approximation-Unsplittable-Truemper. Moreover, we want to point out a very good performance of our branch-and-price algorithm. Its performance is similar to that of the cutting plane algorithm [125] which is so far the best exact method proposed for the $R||C_{\max}$ problem. In our opinion it should not be difficult, but time-consuming to improve further our exact algorithm in order to beat the cutting plane approach (see Chapter 7).

Finally, the comparison tests with heuristics show in contrary to our expectations that the Approximation-Unsplittable-Truemper algorithm can be as efficient as the heuristic algorithms, and thus can successfully compete with them in solving difficult scheduling problems (see Section 6.3.3 for numerical results).

## 1.6 Bibliographical Notes

Some of the results described in this thesis have already been presented on international computer science conferences and are published in a preliminary form in the conference proceedings. In particular, the new combinatorial approximation algorithm, Approximation-Unsplittable-Truemper, for the $R||C_{\max}$ problem presented in details in Chapter 3, appeared in [53]. The column generation two-step approach presented in Section 2.2 and the computational results on 2-approximation algorithms for the unrelated scheduling problem, together with the improvements of the Approximation-Unsplittable-Truemper algorithm, have been published in a preliminary form in [127]. In [127] also one of the three adapted versions of the LPT algorithm from Section 4.1.3 has been briefly described.

There are also results in this work which have not yet been published. The randomized version of the two-step approach given in Section 2.3 and the extended branch-and-price method presented in Section 5.2 are two new algorithms for the $R||C_{\max}$ problem. Also the comprehensive experimental studies on different exact and heuristic methods for the unrelated scheduling problem presented in Section 6.3.2 and Section 6.3.3, respectively, are to our best knowledge the first of their kind and appear for the first time in this work.

# 2

# Two-Step Algorithms

We begin our discussion on the solution methods for the unrelated scheduling problem with the classical two-step approach. This technique was originally introduced by Potts [142] in 1985 to solve the $R||C_{\max}$ problem. His exponential-time heuristic was the first method which approximated the optimal solution with a factor of 2. Five years later Lenstra, Shmoys, and Tardos presented in their seminal work [112] the first polynomial-time version of the two-step approach for approximating the optimal solution to the $R||C_{\max}$ problem. Their, similar to Potts' heuristic, algorithm computes first an optimal fractional solution to the scheduling problem, and then it rounds the fractional solution in order to obtain an integer schedule. Because of their importance for the later analysis, we present in the first section a detailed description for both methods. Next, we give an overview on various algorithms which are used to solve the fractional scheduling problem; we concentrate there especially on their usability for solving large-scale integer problems and point out their main drawbacks. Afterward, we present three different rounding procedures. The first one, introduced by Lenstra et al. [112], computes a 2-approximative integral solution from a basic optimal solution to the fractional scheduling problem. The second one, developed by Shmoys and Tardos [157], can be applied to any fractional solution to the relaxed scheduling problem. Finally, we present briefly an improved optimal rounding algorithm which was recently presented by Shchepin and Vakhania [156]. Since the randomized rounding technique is a powerful tool often used to attack hard optimization problems, we present in the last section of this chapter our randomized version of the two-step approach for the $R||C_{\max}$ problem.

## 2.1 Generic Two-Step Approach

### 2.1.1 Algorithm by Potts

We begin with a description of a simple algorithm designed by Potts [142]. His heuristic method, called LPE (for Linear Programming and Enumeration), solves the unrelated scheduling problem in exponential time in the number of machines $m$. The objective is to minimize the makespan $T$ of the schedule. Consider the mixed integer program (MIP) that represents the problem of assigning jobs to machines as given in (1.2). Since $T$ is minimized in the objective function, at optimality $T$ represents the maximum processing time on any machine.

Suppose now that we relax the binary constraints on assignment variables $x_{ij}$ and require that $x_{ij} \geq 0$ for all $i \in [n]$ and $j \in [m]$, as given in (2.1). We call this linear program an LP *relaxation* of the integer program given in (1.2). A *basic* optimal solution $\mathbf{x}^*$ to (1.2) has the property that the number of positive variables is at most the number of rows in the constraint matrix, i.e., it is at most $m + n$. The other, non-basic variables, take value zero [25].

$$\begin{array}{lll} \text{LP:} & \min & T \\ & \text{s.t.} & \sum_{i\in[n]} p_{ij}x_{ij} \leq T, \qquad \forall\, j \in [m] \\ & & \sum_{j\in[m]} x_{ij} = 1, \qquad \forall\, i \in [n] \\ & & x_{ij} \geq 0, \qquad \forall\, i \in [n], j \in [m]. \end{array} \tag{2.1}$$

Since $T$ is always positive, at most $m + n - 1$ of $x_{ij}$ variables from $\mathbf{x}^*$ are positive. More specifically, it can be shown that each constraint of the first $m$ inequalities in (2.1) is satisfied as an equality. This implies that exactly $m+n-1$ of the assignment variables are basic. For more detailed explanation we refer to Schrijver [151]. Moreover, observe that every job $i$ has at least one positive variable $x_{ij}$ associated with it. Denote by $\bar{x}$ the number of jobs that are assigned in $\mathbf{x}^*$ to exactly one machine, i.e., $\bar{x} = |\{x_{ij} \,|\, x^*_{ij} = 1, \forall i \in [n]\}|$. Then, since one job can be split on at least two machines, it follows

$$n + m - 1 = \bar{x} + 2(n - \bar{x}) \Rightarrow \bar{x} = n - m + 1.$$

Is is easy to see that $\bar{x}$ increases when the number of machines on which a single job is split also increases. This implies that $\bar{x} \geq n - m + 1$. Thus, since we know that the total number of jobs is $n$, we conclude that at most $m - 1$ jobs have been split onto two or more machines in the optimal (fractional) assignment $\mathbf{x}^*$. Note that the optimal LP solution value $T$ is a lower bound on the the value of integral optimal solution $C^*_{\max}$.

Now we show how Potts' heuristic constructs a feasible schedule. To this end we need two phases. In the first phase, for all those jobs with integral assignments in the optimal LP solution $\mathbf{x}^*$ we assign them according to $\mathbf{x}^*$, i.e., $\alpha(i) := j$, if $x^*_{ij} = 1$. This first piece of the schedule is guaranteed to have a length that is at most the value of the LP solution, which is clearly at most $T$. For the remaining set of at most $m - 1$ jobs, by complete enumeration, we find an optimal assignment in time $O(m^{m-1})$. This optimal schedule on a partial of jobs is also guaranteed to have makespan at most $C^*_{\max}$. Thus, by concatenating these two partial schedules, we obtain the following result:

**Theorem 2.1.1 (Potts [142])** LPE *algorithm computes a schedule with makespan at most* $2C^*_{\max}$ *to the* $R||C_{\max}$ *problem, where* $C^*_{\max}$ *is the makespan of the optimal integer solution to the scheduling problem.*

Note that for a fixed number of $m$ the number of computational steps required by the heuristic is polynomial. However, for arbitrary $m$ an exponential number of steps is needed.

For practical purposes, however, the complete enumeration procedure can be replaced by a branch-and-bound algorithm. Such use of a bounding procedure to limit the search of the enumeration tree reduces the computational requirements for many problems. A branch-and-bound algorithm has the further advantage that, if desired, computation can be terminated before optimality is reached. Since the optimal solution is generated at an early stage of a branch-and-bound algorithm while the remaining computation verifies optimality, an early termination of the algorithm does not necessarily detract from its performance although the guarantee of Theorem 2.1.1 becomes invalid. For more discussion on branch-and-bound techniques we refer the reader to Chapter 5.

Potts also proved that for $m = 2$ the LPE heuristic has the best possible worst-case approximation factor of $(1 + \sqrt{5})/2$. For the case when $m \geq 3$, the bound of 2 is the *best* possible. To show this, he gave the following example:

**Example 2.1.1 (Potts [142])** *Consider a scheduling problem with m uniformly related machines. There are m jobs and* $p_{ij} = \frac{p_i}{q_j}$ *for* $i \in [m], j \in [m]$, *where* $q_1 = m - 1$ *and* $q_j = 1$ *for* $j = 2, \cdots, m$; $p_1 = (m-1)p$ *and* $p_i = p$ *for* $i = 2, \cdots, m$ *for any positive p. An optimal schedule is obtained by assigning job j to machine j for* $j \in [m]$, *giving* $T^* = p$. *A solution (not unique) of the* LP *relaxation is* $x_{11} = 0, x_{i1} = 1$ *(for* $i = 2, \cdots, m$*),* $x_{1j} = \frac{1}{m-1}$ *(for* $j = 2, \cdots, m$*) and* $x_{ij} = 0$ *(*$i, j = 2, \cdots, m$*). The partial schedule assigns jobs* $2, \cdots, m$ *to machine* 1 *and job* 1 *is a fractional job. Enumeration shows that job* 1 *is scheduled on machine* 1 *to give* $T_{\text{LPE}} = 2p$. *Therefore,* $T_{\text{LPE}}/T^* = 2$ *as required.*

### 2.1.2 Algorithm by Lenstra, Shmoys, and Tardos

The drawback of the LPE method by Potts is that the construction of the second phase potentially requires exponential-time in the number of machines. In consequence, this algorithm is polynomial-time only if the number of machines is fixed. Nevertheless, it seems very suggestive that only $m - 1$ of the jobs get split. It implies that the average number of jobs assigned to a machine in the second phase is less than one. If there were a direct way to assign at most one job to each machine in a manner such that each machine got a job which processing time is not *too* large for it, perhaps we could avoid the total enumeration process for the fractional jobs.

The algorithm by Lenstra, Shmoys, and Tardos [112] which we present now is motivated by this intuition. The algorithm is based on the following 2-relaxed decision procedure: Given a length of $T$, the procedure either correctly deduces that no schedule with a length of $T$ exists, or it constructs a schedule with makespan at most $2T$. It is then possible, using binary search on $T \in \left[\lceil T_g/m \rceil, T_g\right]$, to convert such a procedure into a 2-approximation algorithm. Here, $T_g$ is the makespan of a schedule constructed by a greedy algorithm which assigns each job to a machine on which it has a smallest processing time, i.e., $T_g \leq \sum_{i\in[n]} \min_{j\in[m]}\{p_{ij}\}$.

We present now the decision procedure in more detail. But first let us construct for a given schedule length $T$ the following sets of machine and job indices: for each job $i \in [n]$:

$$M_i(T) = \{j \in [m] \,|\, p_{ij} \leq T\},$$

and for each machine $j \in [m]$:

$$J_j(T) = \{i \in [n] \mid p_{ij} \leq T\}.$$

In other words, $M_i(T)$ denotes all the machines that can possibly process job $i$ in any schedule of length $T$, and $J_j(T)$ includes those jobs that can be processed on machine $j$ in any schedule of length $T$.

Consider a linear feasibility system, LFS, as given in (2.2). Notice that if a (possibly integral) schedule of length $T$ exists, then LFS is guaranteed to be feasible. Moreover, the basic feasible solution $\tilde{\mathbf{x}}$ to this system has the property that at most $m + n$ variables are positive. Therefore, by the same argumentation as for Potts' heuristic (see Section 2.1.1), we conclude that at most $m$ jobs get split (fractional) assignments. As before, we construct an approximate schedule in two phases. First, we assign each unsplit job to its proper machine, i.e., $\alpha(i) := j$ if $\tilde{x}_{ij} = 1$, and then we round the jobs with fractional assignments.

$$\text{LFS:}\quad \begin{array}{rcll} \sum\limits_{i \in J_j(T)} p_{ij}x_{ij} & \leq & T, & \forall\ j \in [m] \\ \sum\limits_{j \in M_i(T)} x_{ij} & = & 1, & \forall\ i \in [n] \\ x_{ij} & \geq & 0, & \forall\ i \in J_j(T), j \in [m] \\ x_{ij} & = & 0, & \forall\ i \notin J_j(T), j \in [m] \end{array} \tag{2.2}$$

By the construction of the sets $M_i(T)$ and $J_j(T)$, if $\tilde{x}_{ij} > 0$, then $p_{ij} \leq T$. So, if we can construct a matching of the fractional jobs to the machines in such a way that each job gets matched to a machine it is already partially assigned to, then we will have constructed an assignment of the fractional jobs that is guaranteed, as the partial schedule, to have length at most $T$. How can we be sure that such a matching exists? Fortunately, the structure of basic feasible solutions to LFS (they are represented by the vertices of the polytope defined by (2.2)) can be exploited to prove existence constructively, and thus to provide a simple way of producing the desired matching from the solution $\tilde{\mathbf{x}}$ to LFS. The following theorem together with the proof guarantees that such a matching always exists for any basic solution $\tilde{\mathbf{x}}$ to LFS and explains how it can be computed.

**Theorem 2.1.2 (Rounding Theorem by Lenstra et al. [112])** *Let* $\mathbf{P} = (p_{ij}) \in \mathbb{Z}_+^{m \times n}$ *be a matrix of processing times, and $T$ a positive integer. If the linear program* LFS$(\mathbf{P}, T)$ *given in (2.2) has a feasible solution, then any vertex $\tilde{\mathbf{x}}$ of the polytope induced by* LFS$(\mathbf{P}, T)$ *can be rounded to a feasible solution $\bar{\mathbf{x}}$ of the integer program* IP$(\mathbf{P}, T)$*, given by:*

$$\text{IP:}\quad \begin{array}{rcll} \sum\limits_{i \in J_j(T)} p_{ij}x_{ij} & \leq & 2T, & \forall\ j \in [m] \\ \sum\limits_{j \in M_i(T)} x_{ij} & = & 1, & \forall\ i \in [n] \\ x_{ij} & \in & \{0, 1\}, & \forall\ i \in J_j(T), j \in [m] \\ x_{ij} & = & 0, & \forall\ i \notin J_j(T), j \in [m] \end{array} \tag{2.3}$$

*and this rounding can be done in polynomial time.*

**Sketch of the Proof:** Let us construct a graph $G(\tilde{\mathbf{x}})$ which nodes are the jobs and machines of the problem instance and which edges $(i, j)$ correspond to all variables $\tilde{x}_{ij} > 0$. It is a well known fact that if the optimal solution $\tilde{\mathbf{x}}$ is basic, then the constructed graph will consist of a forest of trees and 1-trees (a tree plus one edge), in which job nodes and machines nodes alternate. We call such a graph *pseudoforest*. For the proof of this property of graph $G(\tilde{\mathbf{x}})$ we refer to the complete proof of this theorem or to [25]. Now, if we delete the job nodes with integral assignments (they have only one incident edge, i.e., $\tilde{x}_{ij} = 1$) from the graph, we will still have a forest which all leaves correspond to machine nodes. For each deleted job $i$ and corresponding incident edge $(i, j)$ we set $\bar{x}_{ij} = 1$. Note that the remaining fractional job nodes have at least two incident edges. Our task is to construct a matching in graph $G(\tilde{\mathbf{x}})$ in which every job gets matched. The matching will then correspond to the desired assignment.

We show now how such a matching can be constructed. For each of the 1-trees we first focus on the unique cycle in the tree. This cycle contains an equal number of alternating job and machine nodes. We arbitrarily orient the cycle in one direction and assign each job node to the machine node succeeding it on the cycle. By doing this each machine in the cycle receives exactly one job. We delete now all the nodes along the cycle. What now remains of graph $G(\tilde{\mathbf{x}})$ is a collection of trees, each containing at most one job leaf node. Note that upon the deletion of the cycles, job leaf nodes might be created. But there can be at most one such a leaf node per resulting tree.

We root each tree either at its unique job leaf node, or at an arbitrary node if no job leaf node exists. Afterward, for each tree we can assign each job node to one of its machine children. Since each machine node has a unique parent, it is guaranteed to receive at most one job in this assignment. Thus, we have completed with the construction of the desired matching.

Since to each machine is assigned at most one job, and since the processing time of the job is at most $T$, the schedule corresponding to this part of the assignment is guaranteed to have makespan at most $T$. Finally, we put the two pieces of the schedule (corresponding to the schedule of integral jobs from $\tilde{\mathbf{x}}$, and the schedule of rounded fractional jobs), and obtain a schedule with a length of at most $2T$. ■

The rounding procedure presented in the proof of Theorem 2.1.2, which can be applied to any assignment represented by a pseudoforest, takes $O(n + m)$ time. The following theorem summarizes our discussion on the polynomial-time two-step 2-approximation algorithm of Lenstra at al. In Algorithm 1 we present once more the main parts of this approach.

**Theorem 2.1.3 (Lenstra et al. [112])** *There is a 2-approximation algorithm for the minimum makespan problem on unrelated machines that runs in time bounded by a polynomial in the input size.*

The analysis of the algorithm cannot be improved to yield a better worst-case approximation factor. We show this with the following example.

**Example 2.1.2 (Lenstra et al. [112])** *Consider the following instance of the scheduling problem. There are $m^2 - m + 1$ jobs and $m$ identical machines. The first job takes $m$*

**Algorithm 1** Two-Step-Algorithm(**P**)

```
Input: matrix of processing times $\mathbf{P}$
Output: assignment $\alpha$
1: solve LP relaxation;
2: round the fractional solution $\tilde{\mathbf{x}}$;
3: return integral solution $\bar{\mathbf{x}}$;
```

*time units on all machines, and all other jobs take one unit on all machines. In the optimal solution the first job is assigned to one machine and m of the remaining jobs are assigned to each of the other machines. It results with a schedule of makespan m. Surprisingly, there exists no feasible fractional solution for $T < m$. Suppose that the vertex of* LFS($\mathbf{P}, T$) *for $T = m$ corresponds to the schedule where one unit of the job of length m and $m - 1$ unit-length jobs are assigned to each machine. Rounding this fractional solution produces a schedule of length $2m - 1$. Here, the choice of vertex has forced the inferior solution. No extension of the integral part of this solution will yield a better schedule.*

### 2.1.3 Solution Methods for LP Relaxation

We say now a few words about the methods which are used to solve the LP relaxation in the algorithms discussed in the previous subsections. As it was already mentioned, the second phase in these algorithms (the rounding phase) requires in general as an input a *basic* optimal solution to the relaxed MIP.

The *simplex* algorithm always generates a basic solution of this form [25]. However, this technique is not proved to be running in a polynomial time. Instead of simplex method, the *interior point algorithms* can be used, e.g., the ellipsoidal algorithm by Khachiyan [99] – the first polynomial-time algorithm for LP, the projective method by Karmarkar [96] – the first interior point algorithm efficient for practical usage, or so far the best interior point algorithm by Vaidya [166]. The last one needs time $O(m^{1.5}n^{3.5} \log U)$, where $U = \max\{p_{ij} < \infty\}$. There exists also a number of other specializations of the interior point technique (see, e.g., [15]) which can be applied to more specific applications. For a good survey on interior point algorithms we refer the reader to [28, 75].

For practical purposes, however, the LP problem would be solved by the simplex method rather than by some interior point algorithm. Because of the sparse structure of the constraint matrix in the LP relaxation, the number of simplex iterations (pivots) required to solve the problem is unlikely to be large.

**Indirect rounding.** Should an alternative interior point polynomial-time algorithm be applied to yield an optimal *non-basic* solution in which more than $n + m$ variables may take positive values, then an additional polynomial-time procedure (e.g., from [77, 141]) has to be applied to transform this solution into an optimal basic solution.

We describe now briefly the converting procedure by Plotkin, Shmoys, and Tardos [141] for the $P||C_{\max}$ problem. Any (fractional) assignment $\alpha$ defined by an optimal non-basic fractional solution $\mathbf{x}^*$ can be represented as a weighted bipartite graph $G(\mathbf{x}^*) = (J \cup M, E)$.

An edge $(i, j) \in E$ if and only if $x^*_{ij} > 0$ for $i \in J$ and $j \in M$. If $\mathbf{x}^*$ is a vertex of the polytope of the LP relaxation, then graph $G$ is called a pseudoforest, i.e., any connected component of the corresponding graph is either a tree or a tree plus one additional edge, so called 1-tree. The procedure converts graph $G(\mathbf{x}^*)$ into a pseudoforest. The following theorem gives the running time of the converting procedure. For a detailed proof we refer to [141]. Note that, since $|E| \leq nm$, the time to preprocess $\alpha$ for rounding is dominated by the time taken to find $\alpha$, i.e., to compute an optimal fractional solution $\mathbf{x}^*$ to the LP relaxation.

**Theorem 2.1.4 (Plotkin et al. [141])** *Let $\alpha$ be an assignment represented by the graph $G = (J \cup M, E)$. Then $\alpha$ can be converted in $O(m|E|)$ into another assignment $\beta$ of no greater length, where $\beta$ is represented by a pseudoforest.*

### 2.1.4 Direct Rounding

An interesting result in this regard was presented by Shmoys and Tardos in [157]. Originally they introduced a new rounding technique to solve the *generalized assignment problem*, but, as we will see, it can easily be adapted for the unrelated scheduling problem, too. The advantage of this new approach is that it does not rely on finding basic solutions to the LP relaxation and exploiting their special structure (i.e., they need to be pseudoforests). Instead, it uses a more sophisticated grouping and an assignment of fractional jobs to obtain a schedule for the fractionally assigned jobs. This more powerful algorithm builds on earlier work of Lin and Vitter [113], and Trick [163].

We describe now how the approach from [157] can be applied for our scheduling problem. Given *any*, not necessarily basic, optimal solution to the LP relaxation of (1.2), we will construct an integral assignment with a total makespan at most $2T$ where $T$ is the length of the fractional assignment.

Consider an optimal solution $\mathbf{x}^*$ to the LP relaxation. We construct a bipartite graph $G = (W \cup V, E)$ as follows. Set $W$ contains a node $w_i$ corresponding to each job $i \in J$. Extended set of machines, $V$, contains a *set* of nodes corresponding to each machine $j \in M$. The edges $(i, j)$ in the graph correspond to job-machine pairs which associated value $x^*_{ij}$ is positive. Each edge $(i, j)$ has associated *weight* $\hat{x}(i, j)$.

Let us first focus on a particular machine $j \in M$. Without lost of generality, we temporarily renumber the jobs so that $p_{1j} \geq p_{2j} \geq \cdots \geq p_{nj}$. Let $k_j = \left\lceil \sum_{i \in J} x^*_{ij} \right\rceil$. Then the nodes of $V$ corresponding to machine $j$ are $v_{j1}, \cdots, v_{jk_j}$. If $\sum_{i \in J} x^*_{ij} \leq 1$, then there is only one node $v_{j1} \in V$ corresponding to machine $j$. In this case, for each $x^*_{ij} > 0$ we include an edge $(v_{j1}, w_i)$ into $E$ and set $\hat{x}(v_{j1}, w_i) := x^*_{ij}$.

Otherwise, we find an index $q$ for which

$$\sum_{i=1}^{q-1} x^*_{ij} < 1 \leq \sum_{i=1}^{q} x^*_{ij}.$$

We add to $E$ edges from node $v_{j1}$ to job nodes $w_1, \cdots, w_{q-1}$ with weights $\hat{x}(v_{j1}, w_i) := x^*_{ij}$, for $i = 1, \cdots, q-1$. Additionally, we create an edge $(v_{j1}, w_q)$ with weight

$$\hat{x}(v_{j1}, w_q) := 1 - \sum_{i=1}^{q-1} x^*_{ij}.$$

These are the only edges to node $v_{j1}$. Observe that the sum of the weights of the edges incident to $v_{j1}$ is exactly equal to 1.

Now we consider node $v_{j2}$. If $x^*_{qj} > \hat{x}(v_{j1,w_q})$, we construct an edge $(v_{j2}, w_q)$ with weight $\hat{x}(v_{j2}, w_q) := x^*_{qj} - \hat{x}(v_{j1}, w_q)$. We then proceed with jobs $i > q$, i.e., those with smaller processing time on machine $j$, and construct edges incident to node $v_{j2}$ until a total of exactly one job is assigned to $v_{j2}$, and so forth. More precisely, for each $s = 2, \cdots, k_j - 1$, we find the minimum index $q_s$ such that

$$\sum_{i=1}^{q_s} x^*_{ij} \geq s.$$

We add to $E$ those edges $(v_{js}, w_i)$ for $i = q_s + 1, \cdots, q_s - 1$, and for each of these set $\hat{x}(v_{js}, w_i) := x^*_{ij}$. Furthermore, we add an edge $(v_{js}, w_{q_s})$ to $E$ with weight

$$\hat{x}(v_{js}, w_{q_s}) := x^*_{q_s j} - \sum_{i=1+q_{s-1}}^{q_s - 1} \hat{x}(v_{js}, w_i).$$

If $\sum_{i=1}^{q_s} x^*_{ij} > s$, then we include also an edge $(v_{j,s+1}, w_{q_s})$ into $E$, and set $\hat{x}(v_{j,s+1}, w_{q_s}) := x^*_{jq_s} - \hat{x}(v_{js}, w_{q_s})$.

**Example 2.1.3** *To better understand this construction, we give a simple example for unrelated scheduling problem with $n = 4$ jobs and $m = 3$ machines. Assume that the jobs have temporarily been renumbered according to non-increasing processing times for each machine. It does not have any influence on the generality of the construction. In Figure 2.1.4(a) we give a feasible optimal solution $\mathbf{x}^* = (x^*_{ij})$ to the* LP *relaxation of the unrelated scheduling problem. Figure 2.1.4(b) shows the constructed bipartite graph $G(V \cup W, E)$. There are three job nodes $w_1, \cdots, w_3$, one for each job, and two extended machine nodes for each machine, i.e., there are nodes $\{v_{11}, v_{12}\}$ for the first machine and nodes $\{v_{21}, v_{22}\}$ for the second machine. The edges in the graph $G$ are constructed according to the rules given above. The weight of each edge is equal to $\frac{1}{2}$. Observe that the total weight of edges incident to each machine node in $V$ is at most one, and is exactly one for all nodes in $V$ expect the last one for a given machine $j$.*

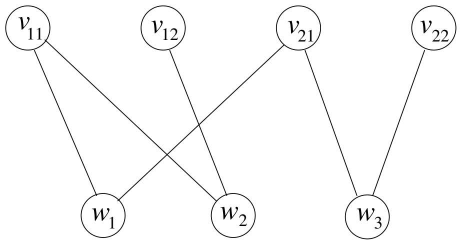
Next step in the direct rounding procedure is to construct an assignment of jobs to machines using the bipartite graph $G(V \cup W, E)$. First, we observe that the variables $\hat{x}(v, w)$ form a feasible, fractional solution to the following linear program:

$$\begin{cases} \sum_{j\in[m]} \sum_{s\in[k_j]} \hat{x}(v_{js}, w_i) = 1, & \forall\, i \in [n] \\ \sum_{i\in[n]} \hat{x}(v_{js}, w_i) \leq 1, & \forall\, s \in [k_j], j \in [m] \\ \hat{x}(v, w) \geq 0, & \forall\, v \in V, w \in W \end{cases} \tag{2.4}$$

It is a well known fact that a feasible LP of this form has integral extreme-point (or basic) solutions. More specifically, any solution $\bar{\mathbf{x}}$ to (2.4) is a convex combination of such integral basic solutions (see, e.g., [3]). It can by computed efficiently in $O(|V \cup W|^{1/2}|E|)$ with an

| | Job 1 | Job 2 | Job 3 |
|---|---|---|---|
| Machine 1 | $\frac{1}{2}$ | 1 | 0 |
| Machine 2 | $\frac{1}{2}$ | 0 | 1 |

(a) optimal solution $\mathbf{x}^*$



(b) graph $G(V \cup W, E)$

Figure 2.1: Example of a construction of a bipartite graph $G(V \cup W, E)$.

algorithm by Even and Tarjan [42] for the maximum flow problem in *unit* capacity *simple* networks, i.e., in which every arc has capacity one, and every node, except the source and sink nodes, has at most one in-coming or at most on out-going arc. Furthermore, it is typical that for a network whose all capacities, supply, and demand are integers, the optimal solution is also integer [3]. Hence, any such integral solution represents a matching in graph $G(V \cup W, E)$ in which every node from $W$ is matched.

We claim now that this matching, when interpreted as an assignment of jobs to machines, has makespan of at most $2T$. Let us focus on particular machine $j$. Since there are $k_j$ machine nodes associated with machine $j$, then at most $k_j$ jobs are assigned to machine $j$. We show now, in a similar way as in [80, 157], that the total processing time of jobs assigned to machine $j$ in the matching is no more than $T$ plus the total processing time of the fractional solution on machine $j$.

Let $p_{js}^{\min}$ and $p_{js}^{\max}$ denote respectively the minimum and the maximum processing times of jobs which nodes in $G(V \cup W, E)$ are adjacent to machine node $v_{js}$ for $s = 1, \cdots, k_j$. Note that $p_{js}^{\min} \geq p_{j,s+1}^{\max}$. The amount of processing time assigned to machine $j$ by the matching is bounded by

$$\sum_{s=1}^{k_j} p_{js}^{\max} = p_{j1}^{\max} + \sum_{s=2}^{k_j} p_{js}^{\max} \leq T + \sum_{s=1}^{k_j-1} p_{js}^{\min} \leq T + \sum_{s=1}^{k_j-1} \sum_{i:(v_{js},w_i)\in E} p_{ij}\hat{x}(v_{js}, w_i) \leq$$

$$\leq T + \sum_{s=1}^{k_j} \sum_{i:(v_{js},w_i)\in E} p_{ij}\hat{x}(v_{js}, w_i) \leq T + \sum_{i=1}^{n} p_{ij}x_{ij}^* \leq 2T,$$

where the last equality follows from the fact that the fractional load on machine $j$ is at most $T$. The following lemma summarizes the result for the $R||C_{\max}$ problem.

**Lemma 2.1.5** *The direct rounding by Shmoys and Tardos [157] can be applied to round any fractional solution to the $R||C_{\max}$ problem. Furthermore, if the length of the fractional schedule equals $T$, then the makespan of the rounded schedule is at most $2T$.*

### 2.1.5 Optimal Rounding

In Example 2.1.2 we have seen that there exists an instance of the unrelated scheduling problem for which the two-step algorithm by Lenstra, Shmoys, and Tardos [112] delivers a solution which is a factor of $2 - \frac{1}{m}$ within the optimum. This almost matches the approximation factor of 2 shown for their method.

Recently, Shchepin and Vakhania [156] presented a new approximation algorithm for scheduling unrelated machines which yields solutions with approximation factor of $2 - \frac{1}{m}$. Their approach is similar to the two-step heuristic used by Potts [142]. First, an LP relaxation of the unrelated scheduling problem is computed. As it was already mentioned in Section 2.1.1, an optimal fractional solution to that relaxation consists of at most $m - 1$ preemptions, i.e., jobs assigned to at least two machines. In the second step this fractional solution is rounded to an integral solution. Here, Shchepin and Vakhania use a completely new rounding procedure for which they show that it gives the best possible approximation factor which can be obtained using the rounding approach.

We describe now very briefly the rounding procedure which they use. Generally speaking, a fractional job cannot be rounded on a machine unless a fraction of at least $\frac{1}{m}$ of that job has been originally assigned to it by the fractional solution. Their algorithm constructs in time $O(m^2)$ a rounding in which no machine receives more than $\frac{m-1}{m}$ new job parts. This ensures that the makespan of the resulting integral assignment exceeds the makespan of the optimal fractional solution, $T$, by at most $\frac{m-1}{m} \cdot p_{\max}$. Here, $p_{\max}$ denotes the maximal processing time in the optimal fractional solution. Since $p_{\max} \leq T$, a $(2 - \frac{1}{m})$-approximation of the optimal integral solution is guaranteed.

As it is the case for other two-step methods, the total computation time of the approximation algorithm using the optimal rounding procedure is here also dominated by solving the LP relaxation (see Section 2.1.3 for more details). We refer to [156] for a complete description of this interesting algorithm. The following theorem summarizes the main result from [156].

**Theorem 2.1.6 (Shchepin and Vakhania [156])** *There exists a polynomial-time* $(2 - \frac{1}{m})$-*approximation algorithm for* $R||C_{\max}$.

We give now a simple example which shows that the optimal rounding procedure gives a best possible approximation factor which can be obtained with the rounding approach.

**Example 2.1.4** *Consider m identical machines and a single job with processing time p. We want to schedule this job without preemption on these machines. It is easy to see that the makespan of an optimal non-preemptive solution equals p. The makespan of the optimal fractional solution is* $\frac{1}{m}p$. *The difference between them is* $\frac{m-1}{m}p$. *Thus, the algorithm by Shchepin and Vakhania can assign the fractional job on any of m machines. This shows that the makespan of an optimal non-preemptive schedule may exceed the makespan of an optimal fractional solution by* $(1 - \frac{1}{m})p$.

## 2.2 Scheduling by Column Generation

### 2.2.1 Motivation.

A big drawback of the two-step approaches presented in the previous section is the enormous number of assignment variables in the LP relaxation of the scheduling problem. This situation becomes especially critical in large-scale problems.

Large applications require much more operational memory to represent the $R||C_{\max}$ problem, and thus they are harder to handle by the operating system throughout all computations. In the worst case it cannot even be possible to state all variables of the problem explicitly as needed. Furthermore, in each iteration of the simplex algorithm, which is more often used to solve the LP relaxation than other techniques, we look explicitly for a non-basic variable to price out and enter the basis. This operation becomes too costly when the number of variables is large. Consequently, the overall performance of the application decreases dramatically.

To overcome this unfavorable behavior, a *column generation* approach can be applied to solve the LP relaxation.

### 2.2.2 Preliminaries on Column Generation.

Dantzig-Wolfe decomposition and column generation approach, both devised originally for linear programs, are two closely related successful methodologies for the large-scale *integer programming*. The origins of column generation goes almost five decades back to Ford and Fulkerson [46]. They were the first who suggested dealing only implicitly with the variables of a multicommodity flow problem. Dantzig and Wolfe [26] pioneered this fundamental idea by developing a strategy to extend a linear program columnwise as needed in the solution process. This technique was first put into actual use by Gilmore and Gomory [59, 60] as a part of an efficient heuristic algorithm for solving the cutting stock problem. Column generation is nowadays a prominent method to cope with a huge number of variables. The embedding of column generation techniques within a linear programming based branch-and-bound framework, introduced by Desrosiers, Desrochers, and Soumis [32] for solving a vehicle routing problem under time window constraints, is the key step in the design of exact algorithms for a large class of integer programs.

**The method.** There are three necessary building blocks for every column generation based solution approach to the (mixed) integer programs:

(1) a *master problem*, MP, which is an original formulation to solve and acts as the control center to facilitate the design of natural branching rules and cutting planes[1],

(2) a *restricted master problem*, RMP, used to determine the currently optimal dual multipliers and to provide a lower bound at each node of the branch-and-bound tree, and

(3) a *pricing subproblem*, PSP, which explicitly reflects an embedded structure of the problem which we want to exploit.

Let us call the following linear program the master problem MP. Here, $I$ denotes a set of indices of all columns $\mathbf{a}_k$ from constraint matrix $A$.

$$\begin{aligned} \mathsf{MP}:\quad z^* = \quad & \min \quad \sum_{k\in I} c_k\lambda_k \\ & \text{s.t.} \quad \sum_{k\in I} \mathbf{a}_k^T\lambda \geq \mathbf{b} \\ & \qquad\quad \lambda_k \geq 0, \qquad \forall k \in I \end{aligned} \tag{2.5}$$

[1]In Chapter 5 we show how we exploit the original formulation to design a *branch-and-price* algorithm.

In each iteration of the simplex method we look for a non-basic variable to price out and enter the basis, i.e., in the *pricing step*, given the non-negative vector $\boldsymbol{\pi}$ of dual variables, we want to find:

$$k_{\min} = \arg\min_{k\in I}\{\bar{c}_k := c_k - \boldsymbol{\pi}^T\mathbf{a}_k\}$$

This explicit search becomes too costly operation when $|I|$ is huge. Instead, we work with a reasonably smaller subset $I' \subseteq I$ of column indices. It defines us a restricted master problem RMP. We evaluate then the reduced costs $\bar{c}_k$ by *implicit* enumeration. Assuming we have a feasible solution, let $\boldsymbol{\lambda}$ and $\boldsymbol{\pi}$ be a primal and a dual solution of the RMP, respectively. When columns $\mathbf{a}_k$ for $k \in I$ are implicitly given as elements of matrix $A$ and the cost coefficient $c_k$ can be computed from $\mathbf{a}_k$ via a function $c$, then the subproblem

$$\bar{\mathbf{c}}^* = \min_{\mathbf{a}\in A}\{c(\mathbf{a}) - \boldsymbol{\pi}^T\mathbf{a}\} \tag{2.6}$$

performs the pricing. If $\bar{\mathbf{c}}^* \geq 0$, then there is no negative $\bar{c}_k$ for $k \in I$, and the solution $\boldsymbol{\lambda}$ to the restricted master problem optimally solves the master problem as well [26]. Otherwise, we add to the RMP the column(s) with negative reduced costs derived from the optimal pricing subproblem solution, and repeat with re-optimization of the just extended RMP.

In what regards convergence, note that each column $\mathbf{a}_k \in A$ is generated at most once since no variable in an optimal RMP has negative reduced cost. When dealing with a finite set $A$ of columns, the column generation algorithm converges to the optimal solution. With the usual precautions against cycling of the simplex method, column generation is finite and exact.

**Lower and upper bounds.** It is worth to point out that we can make use of bounds produced by the column generation algorithm. Let $\bar{z}$ denotes the optimal objective value of the RMP. Note that by duality we have $\bar{z} = \boldsymbol{\pi}^T\mathbf{b}$. When the upper bound $\kappa \geq \sum_{k\in I}\lambda_k$ holds for an optimal solution of the master problem, we have not only an upper bound $\bar{z}$ on $z^*$ in each iteration, but also a lower bound: we cannot reduce $\bar{z}$ by more than $\kappa$ times the smallest reduced cost $\bar{c}^*$, thus

$$\bar{z} + \bar{c}^*\kappa \leq z^* \leq \bar{z}. \tag{2.7}$$

In the optimum of (2.5), $\bar{c}^* = 0$ for the basic variables, and the bounds close, i.e., $\bar{z} = z^*$. The lower bound in (2.7) is computationally cheap and readily available when (2.6) is solved to optimality.

Even though column generation originates from linear programming, its strengths unfold in solving integer programming problems. The simultaneous use of two concurrent formulations, the restricted master problem and the subproblem, their compactness and extensiveness allow very often for a better understanding of the problem at hand and simulates our inventiveness in what concerns, e.g., branching rules, as we will see in Chapter 5.

For a good survey on column generation we refer the reader to [115, 172]. Application of column generation to integer programs is discussed exhaustively in [170]. An insightful overview of the state-of-the-art in integer programming column generation, its methodologies and many applications can be found in [30].

### 2.2.3 Column-Generation-Scheduling Algorithm

We present now how a column generation approach can be applied to speed up the computations of the LP relaxation of the unrelated scheduling problem. Algorithm 2, which we use in our experiments presented in Chapter 6, is based on the ideas from [168] where a *branch-and-bound* approach combined with column generation is used to solve the $P||\sum_{i\in[n]} C_i$ problem where $C_i$ is the completion time of job $i$.

In our case, the master problem MP of the column generation scheme is defined as an LP relaxation of the $R||C_{\max}$ problem and is given in (2.8) in standard form [136]. It is easy to prove that both formulations, (2.1) and (2.8), are equivalent.

$$\begin{array}{lll} \text{LP:} & \min & \mathbf{c}^T\mathbf{x} \\ & \text{s.t.} & A\mathbf{x} = \mathbf{b} \\ & & \mathbf{x} \geq 0 \end{array} \tag{2.8}$$

where

$$A = \begin{bmatrix} p_{11} & & 0 & p_{21} & & 0 & & p_{n1} & & 0 & -1 & 1 & & 0 \\ & \ddots & & & \ddots & & & & \ddots & & \vdots & & \ddots & \\ 0 & & p_{1m} & 0 & & p_{2m} & & 0 & & p_{nm} & -1 & 0 & & 1 \\ 1 & \cdots & 1 & 0 & \cdots & 0 & & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 1 & \cdots & 1 & & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ & & & & & & \ddots & & & & & & & \\ 0 & \cdots & 0 & 0 & \cdots & 0 & & 1 & \cdots & 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \in \mathbb{Z}_{\geq -1}^{(m+n)\times(nm+1+m)}$$

$$\mathbf{x} = \begin{bmatrix} x_{11} \cdots x_{1m}\ x_{21} \cdots x_{2m} \cdots x_{n1} \cdots x_{nm}\ T\ s_1 \cdots s_m \end{bmatrix}^{\mathrm{T}} \in \mathbb{R}_{\geq 0}^{nm+1+m}$$

$$\mathbf{c} = \begin{bmatrix} 0 \cdots 0\ 1\ 0 \cdots 0 \end{bmatrix} \in \{0,1\}^{nm+1+m}$$

$$\mathbf{b} = \begin{bmatrix} 0 \cdots 0\ 1 \cdots 1 \end{bmatrix}^{\mathrm{T}} \in \{0,1\}^{m+n}$$

The appealing idea of column generation, as it was indicated already in the previous section, is to work only with a reasonably small subset of variables, forming the restricted master problem RMP. More variables are added only when needed. We define RMP by a subset $\mathbf{a}$ of columns from $A$, i.e., $\mathbf{a} = \{\mathbf{a}_k \in A \,|\, k \in I' \subseteq I\}$ where $\mathbf{a}_k$ is the $k$-th column from $A$ corresponding to variable $x_k \in \mathbf{x}$, and $I$ is a set of indices of all columns from $A$, $|I| = nm + 1 + m$. Note that the number of columns in RMP, $|I'|$, induces the same number of variables in RMP.

$$\begin{array}{llll} \text{PSP:} & \text{find} & k & \\ & \text{s.t.} & \bar{c}_k = c_k - \mathbf{a}_k^T\mu(\mathbf{a}) & < 0 \\ & & \mathbf{a}_k \in A, & \forall\, k \in I \setminus I' \end{array} \tag{2.9}$$

Let $x(\mathbf{a})$ and $\mu(\mathbf{a})$ be the primal and the dual optimal solution to the current RMP defined by $\mathbf{a}$, respectively. To determine if $x(\mathbf{a})$ is also the optimal solution for the MP, we solve the pricing subproblem PSP given in (2.9) where $\bar{c}_k$ denotes the reduced cost of variable $x_k$. If PSP returns no feasible $k$, i.e., the reduced costs of all variables from $I \setminus I'$ are non-negative, then the solution $x(\mathbf{a})$ to RMP optimally solves MP as well, and we are done. Otherwise,

**Algorithm 2** COLUMN-GENERATION-SCHEDULING($A, \mathbf{b}, \mathbf{c}$)

```
Input: constraints matrix $A$
       positive vectors $\mathbf{b}$ and $\mathbf{c}$
Output: assignment $\alpha$
 1: find initial set of columns $\mathbf{a}$;
 2: initialize $I'$ according to $\mathbf{a}$;
 3: $(x(\mathbf{a}), \mu(\mathbf{a})) :=$ solve RMP($\mathbf{a}, \mathbf{b}, \mathbf{c}$);
 4: compute reduced costs $\bar{c}_k$ from dual multipliers $\mu(\mathbf{a})$ and columns $\mathbf{a}_k \in A$, for $k \in I \setminus I'$;
 5: if $\min_{k \in I - I'} \bar{c}_k < 0$ then
 6:     generate new column(s) from $A$ and add them to $\mathbf{a}$;
 7:     update $I'$ and go to 2;
 8: end if
 9: round the fractional solution $x(\mathbf{a})$ into an integral solution $\bar{\mathbf{x}}$;
10: return integral solution $\bar{\mathbf{x}}$;
```

variable $x_k$ identified by PSP is added to RMP, and optimization of a just extended RMP is repeated.

### 2.2.4 Solution Methods for the Restricted Master Problem

Column generation is a primal method, i.e., it maintains primal feasibility and works toward dual feasibility. Therefore, it is natural only to monitor the dual solution in the course of the algorithm. In other words, the purpose of RMP is to provide dual variables. They are used later by the pricing subproblem PSP to compute reduced costs $\bar{\mathbf{c}}$ and to control the stopping criterion. When the algorithm terminates, we need to recover from the dual RMP solution a primal feasible solution (e.g., by employing the complementary slackness conditions [151] for linear programs).

**Solution methods.** Solving RMP by the simplex method leads to an optimal basis essentially chosen at random, whereas the application of an interior point method produces a solution in the relative interior of the optimal face [15]. Therefore, various interior point algorithms have been proposed, e.g., analytic and volumetric centers algorithms [40, 63], central paths [121], and central prices [63] methods. The computational use of various methods for obtaining integer solutions has been evaluated by Briant et al. in [17].

Extreme point dual solutions are immediately available when using the simplex method, and because of their *random* nature they may result in different, even complementary kinds of columns [170].

As it is the case for general linear programs depending on the application and the available solvers we do not know beforehand which of the several traditional ways of solving RMP will perform better. Useful comments on the suitability of primal, dual, and primal-dual simplex methods can be found in [108]. In presence of primal degeneracy the dual simplex may be preferred to the primal one. The *sifting* method[2] can be a reasonable tech-

[2]In Chapter 6 we show how this method perform for large-scale scheduling problems.

nique for large-scale problems [5, 15]. For some linear programs *barrier* methods [14] can prove most effective, although there is no possibility to use warm start as initialization.

**Initialization.** The initialization of RMP is a very important issue for column generation solution method. Poorly chosen initial columns lead the algorithm astray whenever they do not resemble the structure of a possible optimal solution at all. Even an excellent initial integer solution could be detrimental for solving a linear program by column generation [170].

The well known simplex *first phase* can be applied here. Artificial variables, one for each constraint, penalized by a "big $M$" cost are kept in RMP to ensure feasibility in a branch-and-bound algorithm. A smaller $M$ gives a tighter upper bound on the dual variables and may reduce the effect of producing irrelevant columns [30].

Another option is a warm start from primal solutions obtained in earlier similar runs [5]. However, the best results are obtained when both estimations of the primal and the dual solutions are used [37].

## 2.3 Randomized Two-Step Approach

Randomization has proved to be a powerful technique in finding approximate solutions to difficult problems in the combinatorial optimization. The main focus of this section is put on the usage of *randomized rounding* to solve hard scheduling problems. In this approach, one solves an LP relaxation of a problem, and then uses randomization to return from the relaxation to the original optimization problem.

The section is organized as follows. First, we give basic information and most important theoretical results concerning randomized rounding. Afterward, we discuss our application of these techniques to solve the unrelated scheduling problem.

### 2.3.1 Preliminaries

In the last decades the randomization in approximation algorithms for difficult combinatorial problems has witnessed a great amount of attention. Probabilistic search methods like, e.g., *simulated annealing* (see [169] for a detailed survey on simulated annealing and its applications), have enjoyed a considerable success in solving large instances of various optimization problems. While studying randomization techniques, we are interested in *efficient* polynomial-time approximation algorithms that deliver solutions within a *provable good* tolerance to the optimal solution. The only randomness in the performance guarantee must stem from the randomization in the algorithm itself, and not due to any probabilistic assumptions in the input instance [128].

We make use of a technique called *randomized rounding* [145]. This technique is applicable to a class of $\{0, 1\}$ integer linear programs. It is a probabilistic method, i.e., for the existence of results we prove that the solution to an (mixed) integer program satisfies a certain property by showing that a randomly generated solution satisfies that property with non-zero probability. All the algorithms using randomized rounding follow a common paradigm. They first formulate the problem as an integer program. Next, some constraints in this integer program are relaxed in order to solve the relaxation efficiently. Finally, randomization is used to restore the relaxed constraints.

There are several advantages offered by randomness to computation. See further the books by Alon, Spencer, and Erdös [4], Kleinberg and Tardos [102], and by mentioned already Motwani and Raghavan [128] for various aspects and applications of randomized computations. Randomized computation was several times applied to improve approximation algorithms for scheduling unrelated machines. New randomized fully *linear* time approximation schemes for the $R||C_{\max}$ and generalized assignment problem, both with fixed number of machines, are given in [38]. A new class of randomized approximation algorithms for scheduling unrelated parallel machines with the average weighted completion time objective and individual job release dates is presented in [152]. The algorithms there, like our, use the two-step approach, but in contrast to our technique, the randomized rounding was applied there indirectly, and so was the analysis.

## 2.3.2 Application of Randomized Rounding to $R||C_{\max}$

We show now how randomized rounding can be used to construct in a simple way a randomized two-step approach for scheduling unrelated parallel machines.

Our problem can be readily formulated as a mixed integer program as it was already given in (1.2). There, each variable $x_{ij} \in \{0, 1\}$ denotes whether or not job $i$ is assigned to machine $j$. The objective is to minimize a positive integer $T$ corresponding to the makespan. Thus, $T$ is the upper bound for load $\delta_j$ on each machine $j$. We solve the scheduling problem using a randomized two-step approach. As it is shown in Algorithm 3, we first compute an optimal fractional solution $\mathbf{x}^*$ to an LP relaxation of (1.2). The solution to this relaxation can be found efficiently in polynomial time. See Section 2.1.3 for a detailed discussion on various solution methods for solving (2.1).

**Algorithm 3** RANDOMIZED-TWO-STEP-SCHEDULING(**P**)

```
Input: matrix of processing times $\mathbf{P}$
Output: integral solution $\mathbf{x}$
 1: solve LP-relaxation to the $R||C_{\max}$ problem defined by $\mathbf{P}$;
 2: let $\mathbf{x}^* \in [0,1]^{nm}$ be the fractional solution to the LP-relaxation;
 3: for $i \in [n]$ do
 4:     let $a$ be a random number generated uniformly from $(0,1]$;
 5:     define $x^*_{i0} := 0$;
 6:     $j := \{k \in [m] \mid \sum_{l=0}^{k-1} x^*_{il} < a \le \sum_{l=0}^{k} x^*_{il}\}$;
 7:     $x_{ij} := 1$;
 8:     $x_{ik} := 0$, for $k \in [m] \setminus j$;
 9: end for
10: return integral solution $\mathbf{x}$;
```

Let the optimum (fractional) value of $x_{ij}$ be denoted by $x^*_{ij}$. Furthermore, let $T^*$ be the optimum makespan obtained from the LP solution. Note that $T^*$ is a lower bound on the best possible integer optimum makespan. In the second phase of Algorithm 3 we seek to use this fractional solution to obtain an integer solution to (1.2). As already indicated, we do this by means of randomization, i.e., for each job $i$, set $x_{ij}$ to one with probability $x^*_{ij}$. More specifically, we generate first a uniformly distributed random number $a$ from $(0, 1]$. Afterward we assign job $i$ to machine $j$ such that $j = \{k \in [m] \mid \sum_{l=0}^{k-1} x^*_{il} < a \leq \sum_{l=0}^{k} x^*_{il}\}$. Note that by this we have

$$\mathbf{Pr}[x_{ij} = 1] = x^*_{ij} = \sum_{l=0}^{j} x^*_{il} - \sum_{l=0}^{j-1} x^*_{il},$$

where $x^*_{i0} = 0$. This is equivalent to the probability that the value of random variable $a$ lies in $(\sum_{l=0}^{j-1} x^*_{il}, \sum_{l=0}^{j} x^*_{il}]$. The choice is done in an exclusive manner, i.e., for each job $i$, exactly one of the $x_{ij}$ is set to one; the rest is set to zero. This random choice is made independently for each job $i \in [n]$.

Observe that the second (i.e., randmization) phase of Algorithm 3 can be repeated several times in order to improve the solution.

# 3

# UNSPLITTABLE-TRUEMPER Algorithm

In this chapter we present a new combinatorial algorithm, APPROXIMATION-UNSPLITTABLE-TRUEMPER [53], for scheduling unrelated parallel machines. The algorithm is based on generalized and unsplittable network flows and computes schedules with approximation factor of 2 within the best worst-case running time know so far. It is worth to note that this is the first time that a fully combinatorial approach for solving unrelated scheduling problem always beats the interior point algorithm of Vaidya [166] applied to this problem. Our algorithm solves a minimum cost unsplittable flow problem in a generalized bipartite network defined by the unrelated scheduling problem. In the following we give a complete characterization of our technique. We begin first with a short introduction into the theory of network flows. To this end, special attention is given to the generalized and unsplittable network flows. Among other things we show their direct connection to other scheduling problems. In the next section we give a detailed description of the transformation upon the APPROXIMATION-UNSPLITTABLE-TRUEMPER algorithm builds. In addition to this, we present an adaptation of the UNSPLITTABLE-BLOCKING-FLOW algorithm [52] to our scheduling problem. We use it later as a subroutine in our scheduling algorithm. Finally, we present the approximation algorithm itself together with the theoretical results proving its correctness and running time. In the last section we compare the running time of our combinatorial algorithm with the so far fastest algorithms for scheduling unrelated parallel machines.

## 3.1 Scheduling with Generalized Maximum Flows

### 3.1.1 Definitions and Notation

The *generalized maximum flow* problem is a generalization of the ordinary maximum flow problem, where each edge $(i, j)$ in the network has some positive *gain* factor $\mu_{ij}$. If $f_{ij}$ units of flow are sent from node $i$ to node $j$ along edge $(i, j)$, then $\mu_{ij} f_{ij}$ units arrive at node $j$.

More specifically, let $G = (V, E)$ be a directed graph of the generalized flow problem and $s$ and $t$, source and sink node, respectively. Each edge from $E$ has a gain factor associated with it. The gains are defined by a gain function $\mu : E \mapsto \mathbb{R}_{>0}$. We assume that the gain function is antisymmetric, i.e.,

$$\mu_{ij} = \frac{1}{\mu_{ji}}, \ \forall (i,j) \in E.$$

Furthermore, there is a nonnegative capacity function $u : E \mapsto \mathbb{R}_{\geq 0}$ defined on the edges. A *generalized flow* $f : E \mapsto \mathbb{R}$ is a function on the edges that satisfies following types of constraints

- capacity constraints:
$$f_{ij} \leq u_{ij}, \ \forall (i,j) \in E,$$
- flow conservation constraints:
$$\sum_{(j,i)\in E} \mu_{ji} f_{ji} - \sum_{(i,j)\in E} f_{ij} = 0, \ \forall i \in V \setminus \{s,t\},$$
- generalized antisymmetry constraints:
$$f_{ij} = -\mu_{ji} f_{ji}, \ \forall (i,j) \in E.$$

When all constraints except the constraints of the second type are satisfied by flow $f$, then we call $f$ a *generalized pseudoflow*. If $\mu_{ij} > 1$, then $(i,j)$ is a *gain* arc; if $\mu_{ij} < 1$, then $(i,j)$ is a *loss* arc. The gain of a path (cycle) is a product of the gains of arcs on the path (cycle). The value of the generalized flow $f$ is defined as the amount of flow into the sink. Among all generalized flows of maximum value, the goal here is to find one that minimizes the flow out of the source.

In a generalized minimum cost flow problem we are additionally given a real-valued cost function on edges $c : E \mapsto \mathbb{R}$. Without loss of generality, we assume that the costs are antisymmetric, i.e.,

$$c_{ij} = -c_{ji}, \ \forall (i,j) \in E.$$

The goal in the generalized minimum cost flow problem is to find a minimum-cost (optimal) generalized flow $f$ in the input network.

Given a generalized pseudoflow $f$, the definitions of *residual capacities* $r_{ij}$, *residual graph* $G_f$, and *excess* and *deficit* nodes are the same as for the ordinary pseudoflows. Here however, for the sake of completeness we give briefly their definitions. For details we refer the reader to Ahuja et al. [3].

**Definition 3.1.1** *We define the residual network $G_f$ with respect to a given flow $f$ as follows. We replace each arc $(i,j)$ in the original network $G$ by two arcs $(i,j)$ and $(j,i)$, such that:*

- *the arc $(i,j)$ has cost $c_{ij}$ and residual capacity $r_{ij} = u_{ij} - f_{ij}$, and*
- *the arc $(j,i)$ has cost $-c_{ij}$ and residual capacity $r_{ji} = f_{ij}$.*

*The residual network $G_f$ consists only of arc with a positive residual capacity.*

**Definition 3.1.2** *We define the imbalance of node i as the node inflow minus node outflow, i.e.,*

$$e(i) = \sum_{j:(j,i)\in E} x_{ji} - \sum_{j:(i,j)\in E} x_{ij}, \ \forall i \in V.$$

*If the inflow is less then the outflow, i.e., $e(i) < 0$, then we say that node i is a* deficit *node. Otherwise, if $e(i) > 0$, then we say that node i is an* excess *node. If the inflow equals outflow, then we say that node i is a* balanced *node.*

### 3.1.2 Application of Generalized Flows to Scheduling Problems

The generalized flow problem and closely related *generalized circulation problem* (see [141] for the definition) have been studied extensively by many researchers, especially in the 1960s and the early 1970s. For a good overview we refer to the following papers: [39, 45, 65, 62, 64, 109, 131, 132, 133, 144, 162, 164]. Generalized flows can be used to model many situations which are impossible to be expressed using standard network flows, see, e.g., [61, 109, 141]. The gain factors can be used to represent the fact that we loose some fraction of the commodity while transporting it (e.g., due to damage, evaporation, theft, etc.) or that the commodity that enters an arc is transformed into a different commodity before leaving it. The later application is often used to model manufacturing processes or financial operations like, e.g., currency exchange.

The fractional version of the scheduling problem can easily be converted into a generalized maximum flow problem [141]. In order to check whether a fractional schedule of length $T$ exists, one can construct a bipartite graph with nodes representing jobs and machines and introduce an edge from machine node $j$ to job node $i$ with gain $1/p_{ij}$, if $p_{ij} \leq T$. There is a source node which is connected to all machine nodes with edges of unit gain and capacity $T$, and the job nodes are connected to a sink node with edges of unit gain and unit capacity. A generalized flow in this network that results in an excess of $n$ units at the sink node corresponds to a solution of the fractional scheduling problem. On the other hand, if the maximum excess that can be generated at the sink node is below $n$, then the fractional scheduling problem is infeasible, i.e., the current value of $T$ is too small. To find the best length $T$, we use binary search on $T \in [1, nU]$ where $U = \max\{p_{ij} < \infty\}$.

The generalized maximum flow problem is in many aspects similar to the ordinary minimum cost flow problem where each arc has a cost per unit of flow in addition to the capacity. This relationship was first observed by Onaga [133], and then studied in detail by Truemper [164]. Both problems can be viewed as problems of transporting a commodity from a producer to a consumer. Intuitively, the only difference between them lies in the method of payment for shipping costs. In the case of minimum cost flows, these costs are paid with money; in the case of the generalized flows – with the commodity itself. This similarity is not only intuitive. The LP duals of these problems, as well as the optimality conditions (what we show later), can be expressed in very similar forms [64].

In Truemper's construction the cost $c_{ij}$ for each arc in the minimum cost flow problem is defined as the logarithm of the gain, $\log \mu_{ij}$, from the generalized maximum flow problem.

Unfortunately, such a way of defining the arc costs results only in pseudo-polynomial algorithms. To obtain polynomial running time, the costs must be *integer*. In order to transform the generalized maximum flow problem into a minimum cost flow problem with integral arc costs, a gain rounding technique has to be applied (see, e.g., [162]). To this end, the gains are rounded down to integer powers of some base $b > 1$. More specifically, a rounded gain $\gamma_{ij}$ of each arc $(i, j) \in G$ is defined as

$$\gamma_{ij} = b^{c_{ij}}, \text{ where } c_{ij} = \left\lfloor \log_b \mu_{ij} \right\rfloor.$$

Antisymmetry of gains is maintained by setting $\gamma_{ij} = 1/\gamma_{ji}$ and $c_{ij} = -c_{ji}$. The cost of arc $(i, j)$ in the resulting minimum cost flow problem equals now $c_{ij}$. The PRIMAL-DUAL approach of Ford and Fulkerson [48] for minimum cost flows can be used here to compute a generalized maximum flow as it is shown in [162]. Although the generic PRIMAL-DUAL approach is pseudo-polynomial, when combined with the gain scaling technique it runs in a polynomial time. The algorithm works with node potential function $\pi : V \mapsto \mathbb{R}^+$ and the reduced costs $c^\pi_{ij}$ which are defined for each residual arc $(i, j) \in G_f$ as follows

$$c^\pi_{ij} = c_{ij} - \pi(i) + \pi(j).$$

The PRIMAL-DUAL approach preserves the *reduced cost optimality condition*, i.e., $c^\pi_{ij} \geq 0$, for each edge $(i, j)$ in the residual network. Because of the rounding, an optimum solution of the minimum cost flow problem gives only a $(1 + \epsilon)$-approximation of the generalized (fractional) maximum flow problem. Though, using techniques of Shmoys and Tardos from [157], the fractional solution can be transformed to an integral solution (see Section 2.1.4). This approach leads to a $(2 + \epsilon)$-approximation algorithm for the scheduling problem.

## 3.1.3 Optimality Theorem and Flow Decomposition Lemma

We give now some well known theoretical results which are helpful to understand the analysis of the APPROXIMATION-UNSPLITTABLE-TRUEMPER algorithm given in the next sections. We begin with the theorem stating the optimality of the solution to the minimum cost flow problem in terms of the reduced costs of the arcs.

**Theorem 3.1.1 (Ford and Fulkerson [47])** *A feasible solution $f^*$ is an optimal solution of the minimum cost flow problem if and only if some set of node potentials $\pi$ satisfies the following reduced cost optimality conditions:*

$$c^\pi_{ij} \geq 0, \quad \forall (i, j) \in G_{f^*}.$$

Another useful result regarding ordinary network flows is the decomposition theorem. It states that any feasible flow can be viewed as a collection of flows along paths and cycles.

**Theorem 3.1.2 (Ford and Fulkerson [47])** *Every path and cycle flow has a unique representation as non-negative network flows. Conversely, every non-negative arc flow $f$ can be represented as a path and a cycle flow (though not necessarily unique) with the following two properties:*

*(a) every directed path with positive flow connects a deficit node to an excess node, and*

*(b) at most $|V| + |E|$ paths and cycles have nonzero flow; out of these, at most $|E|$ cycles have nonzero flow.*

For the case of generalized flows, Gondran and Monoux [69] presented a version of the decomposition theorem for generalized network flows. They showed how any feasible generalized flow can be decomposed into at most $|E|$ components, each being a positive *elementary* generalized flow. They defined six types of elementary generalized flow. For more details we refer to [64, 69]. For our case, however, we do not refer to that theorem directly but present our own version which we use later in the analysis. Recall that $M$ and $J$ denotes the set of machines and the set of jobs for the $R||C_{\max}$ problem, respectively.

**Lemma 3.1.3 (Decomposition lemma)** *Let $f$ and $g$ be two generalized feasible flows in $G = (J \cup M, E)$. Then $g$ equals $f$ plus fractional flow:*

*(a) on some directed cycles in $G_f$, and*

*(b) on some directed paths in $G_f$ with end points in $M$ and with the additional property that no end point of some path is also the starting point of some other path.*

**Proof:** Let $h = g - f$. Set $\tilde{E} = \{(i, j) \mid h_{ij} > 0, (i, j) \in E_f\} \cup \{(j, i) \mid h_{ij} < 0, (i, j) \in E_f\}$. We show first that $\tilde{E} \subset E_f$. Let $g_{ij}$ and $f_{ij}$ be the flows on arc $(i, j)$ in $G_g$ and $G_f$, respectively, and $u_{ij}$ its capacity. Then if

$$\begin{aligned} g_{ij} > f_{ij} \quad &\Rightarrow \quad f_{ij} < u_{ij} \Rightarrow (i, j) \in E_f \text{ , and if} \\ g_{ij} < f_{ij} \quad &\Rightarrow \quad f_{ij} > 0 \Rightarrow (j, i) \in E_f. \end{aligned}$$

Let $\tilde{u}_{ij} = h_{ij}$ if $(i, j) \in \tilde{E}$, and $\tilde{u}_{ji} = -h_{ij}$ if $(j, i) \in \tilde{E}$. Since, by the assumption, $f$ and $g$ are feasible flows, they both send all supply from job nodes in $J$ to machine nodes in $M$. By this fact it is easy to see that for each job node $i \in J$,

$$\sum_{(i,j)\in\tilde{E}} h_{ij} - \sum_{(j,i)\in\tilde{E}} \mu_{ji} h_{ji} = 0,$$

holds, and thus flow $h$ fulfills the flow conservation constraints on all job nodes from $J$.

We will now decompose the flow in $\tilde{E}$. If there exists a cycle in the graph induced by $\tilde{E}$, let us choose some node $u \in M$ on the cycle and determine the maximal flow $x$ which can be pushed from $u$ along this cycle without conflicts with the capacity constraints. Subtract this flow from the flow in $\tilde{E}$. Since $x$ was chosen to be maximal, at least one edge is saturated, and thus deleted from $\tilde{E}$. Note that by performing this flow subtraction, we maintain the flow conservation constraints on all job nodes $J$. Let us proceed this way as long as there are cycles in the graph induced by $\tilde{E}$. Afterward, $\tilde{E}$ defines a directed acyclic graph.

Now choose a path connecting a node $v$ of zero in-degree to a node of zero out-degree, determine the maximum allowed flow on this path and delete this flow from $\tilde{E}$. Note that at least one edge is deleted from $\tilde{E}$. Furthermore, by the flow conservation constraint, any job node $i \in J$ with zero out-degree has also zero in-degree. This implies that $v \in M$. Let us proceed this way until there are no edges in $\tilde{E}$. The way the paths are chosen guarantees that no end point of some path is also a starting point of some other path. ■

## 3.2 Unsplittable Network Flows

In the unsplittable flow problem we are given a directed or undirected network $G = (V, E)$ with edge capacities defined by a function $u : E \mapsto \mathbb{R}^+$, and a set $K = \{(s_i, t_i) \mid i \in [k]\}$ of $k$ terminal pairs (or commodity requests) with demand $d_i$ each. A feasible solution for this problem is a subset $S \subseteq K$ of the requests such that the demand of each request in $S$ is satisfied by a flow on a *single* path in $G$ and the capacity flow constraints are fulfilled. The objective is to maximize the cardinality of $S$.

It is an easy observation that already a single source unsplittable flow problem without costs contains several well-known $\mathcal{NP}$-complete problems as special cases, e.g., Partition, Bin Packing, and even scheduling parallel machines with makespan objective [112]. Moreover, an interesting special case of the generalized assignment problem considered by Shmoys and Tardos [157] can also be modeled as a single source unsplittable minimum cost flow problem [34, 103]. If we consider the problem with costs, we obtain the Knapsack problem as a special case. The well known maximum edge-disjoint paths problem is also a special case of the unsplittable flow problem where all demands and capacities are equal to one [76].

This combination of routing and bin packing problems makes the various versions of the unsplittable flow problem particularly difficult. Therefore, the best one can hope for, unless $\mathcal{N} = \mathcal{NP}$, is to find good approximate solutions. Approximation algorithms for the unsplittable flow problem and its special cases have been considered in many other works, e.g, [9, 34, 100, 104, 105, 111, 158, 162]. Kleinberg [100] provided a comprehensive background on these problems. Most of the approximation algorithms begin with an LP relaxation of the problem (i.e., instead of using a single path, a commodity is shipped along multiple paths) and then round the solution in a suitable way to obtain an approximate solution for the unsplittable flow problem.

One of the motivations for the unsplittable flow problem is the problem of allocation of bandwidth for traffic with different bandwidth requirements in heterogeneous networks.

In our setting we use unsplittable flows to obtain integral solution to the unrelated scheduling problem. Our model can be explained by the following example in which the restricted scheduling problem on identical machines is a special case of the unsplittable flow problem.

**Example 3.2.1** *We have to schedule $n = 3$ jobs on $m = 3$ parallel machines without preemption. Job $i$ can only be processed on a subset $M(i) \subseteq M$ of machines. The processing time of job $i$ is $p_i$ on any of the admissible machines from $M(i)$, and the goal is to minimize the makespan of the schedule. To formulate this problem as an unsplittable flow problem, we create a node for each machine and for each job, respectively, and a source node $s$. Afterward, we connect the source node to all machine nodes with edges of capacity $T$, and then connect each machine node to the job nodes it can process with edges of infinite capacity (see Figure 3.1). Let each node representing job $i$ be a terminal for commodity $i$ with demand $d_i$ equal to the processing time $p_i$. It is easy to see that this unsplittable flow problem is feasible if and only if there exists a feasible schedule of makespan at most $T$.*

Among other results Lenstra et al. [112] and Shmoys and Tardos [157] showed that if there exists a fractional solution to the unsplittable flow problem from Example 3.2.1, then

$M = \{1,2,3\}$
$J = \{1,2,3\}$

$M(1) = \{1,2,3\}$
$M(2) = \{1,3\}$
$M(3) = \{2,3\}$

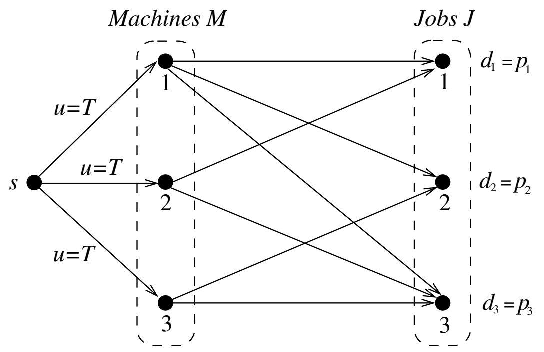


Figure 3.1: Transformation of the restricted scheduling problem into a single-source unsplittable flow problem.

there exists a feasible schedule of makespan $T + \max\{p_i\}$. The two constructions are different. The first one assumes that the fractional solution is an extreme point of the corresponding polyhedron. The latter one does not possess this restriction.

## 3.3 Transformation of the $R||C_{\max}$ Problem into Generalized Flow Problem

In our approach we formulate the unrelated scheduling problem as a generalized maximum flow problem. However, we use a different construction from the construction presented by Plotkin et al. [141] (cf. Section 3.1.2). In particular, we construct a bipartite graph $G = (V, E)$ with nodes representing jobs and machines. The number of nodes in $G$ is $n + m$. There is a directed edge from a job node $i$ to a machine node $j$ if job $i$ can be processed on machine $j$, i.e., $p_{ij} < \infty$. It has unit capacity and gain $\mu_{ij} = p_{ij}$ if $p_{ij} \leq T$. Here, $T$ is a positive control parameter which value is determined by binary search. Each job node $i$ has supply equal to 1. A generalized flow $f$ in $G$ is a solution to the fractional version of the scheduling problem if and only if in $f$ all supplies are sent to the machines. In this case we call $f$ a *feasible* flow. A generalized flow in such a network creates excesses on the machine nodes. An excess on machine $j$ corresponds to the load on machine $j$. We define $\delta_j(\mathbf{P}, f)$ to be the load on machine $j$ under the generalized flow $f$ with gains defined by $\mathbf{P}$.

If we require that the supply of each job is sent to *exactly* one machine, then we get an integral solution to the scheduling problem. In this case, we call flow $f$ a generalized *unsplittable* flow, and $f$ is equivalent to an assignment $\alpha$, i.e., assigning job $i$ to machine $j$ corresponds to sending one unit of flow (the whole supply) along edge $(i, j)$. By the construction of the bipartite graph, the assignment $\alpha$ has the property that each job $i$ is assigned to a machine $j$ for which $p_{ij} \leq T$. In the following, we call such an assignment *$T$-feasible*. We omit the term $T$-feasible if it is clear from the context. We are interested in finding a generalized unsplittable flow $f$ such that the maximum excess over all machines is at most $2T$. This is not always possible, however, if we cannot find such a flow for a given value of $T$, we can still derive the lower bound for the optimum, $C^*_{\max}(\mathbf{P}) \geq T + 1$.

In order to solve this generalized unsplittable flow problem, we follow the construction from Section 3.1 and formulate the problem as a minimum cost unsplittable flow problem. We set the value of the logarithm base for the gain rounding $b$ to $(1 + \frac{1}{m})$. If $(i, j)$ is an edge from job node $i$ to machine node $j$, then the cost $c_{ij}$ and the rounded gain $\gamma_{ij}$ are defined as follows

$$c_{ij} = \lfloor \log_b(p_{ij}) \rfloor, \quad \text{and} \quad \gamma_{ij} = b^{c_{ij}}.$$

For any path $W$ in graph $G$, we define the gain of $W$ as $\gamma(W) = \prod_{(i,j)\in W} \gamma_{ij}$. In the same way we define the gain $\gamma(K)$ for some cycle $K$ in graph $G$. In the following, we denote $\mathbf{C} = (c_{ij})$ and $\mathbf{\Gamma} = (\gamma_{ij})$. In order to solve the minimum cost flow problem, we use the well known PRIMAL-DUAL approach [3].

For a positive integer $T$, a $T$-feasible assignment $\alpha$, and a matrix of processing times $\mathbf{P}$, we define the residual network $G_\alpha(T)$. By Definition 3.3.1, an edge which starts on machine node $j$ and ends on job node $i$ means that job $i$ is assigned to machine $j$. The definition ensures the correctness of the assignment, i.e., in a feasible assignment there can be only one edge between a given machine node and job node, regardless of its direction.

**Definition 3.3.1** *Let $\alpha$ be an assignment and $T \in \mathbb{N}$. We define a directed bipartite graph $G_\alpha(T) = (V, E_\alpha(T))$ where $V = M \cup J$ and each machine is represented by a node in $M$, whereas each job defines a node in $J$. Furthermore, $E_\alpha = E^1_\alpha \cup E^2_\alpha$ where*

$$\begin{aligned} E^1_\alpha &= \left\{(j,i) \,|\, j \in M, i \in J, \alpha(i) = j, p_{ij} \le T\right\}, \text{ and} \\ E^2_\alpha &= \left\{(i,j) \,|\, j \in M, i \in J, \alpha(i) \neq j, p_{ij} \le T\right\}. \end{aligned}$$

*We write $G_\alpha$ for the case that $T \ge \max_{i\in J, j\in M}\{p_{ij} < \infty\}$.*

By the construction of the bipartite graph $G$ and by Definition 3.3.1 the gain factors of the edges in the residual network $G_\alpha(T)$ are defined as follows:

$$\begin{aligned} \mu_{ij} &= p_{ij}, \text{ for all } (i,j) \in E^2_\alpha, \text{ and} \\ \mu_{ji} &= \frac{1}{p_{ij}}, \text{ for all } (j,i) \in E^1_\alpha. \end{aligned}$$

Let $G_f$ be the residual network corresponding to generalized flow $f$. A generalized unsplittable flow $f$ is called *feasible*, if $f$ defines an assignment $\alpha$.

To be complete with the definition of the generalized minimum cost flow problem we still need to define the source and the sink node in the graph. To this end, we partition the set of machines $M$ with respect to their loads $\delta_j(\alpha)$ and parameter $T$ into three subsets as follows:

**Definition 3.3.2** *Let $T \in \mathbb{N}$ and $\alpha$ be a $T$-feasible assignment. We partition the set of machines $M$ into three subsets:*

$$\begin{aligned} M^-(\alpha) &= \{j \,|\, \delta_j(\mathbf{P}, \alpha) \le T\} \\ M^0(\alpha) &= \{j \,|\, T + 1 \le \delta_j(\mathbf{P}, \alpha) \le 2T\} \\ M^+(\alpha) &= \{j \,|\, \delta_j(\mathbf{P}, \alpha) \ge 2T + 1\} \end{aligned}$$

In our setting, the nodes from $M^-$ can be interpreted as sink nodes whereas the nodes from $M^+$ as source nodes. Furthermore, we call the machines from $M^+$ *overloaded* and the machines from $M^-$ *underloaded*.

## 3.4 Unsplittable Blocking Flows

Our approximation algorithm makes use of the Unsplittable-Blocking-Flow algorithm introduced in [52]. Unsplittable-Blocking-Flow was designed for a restricted scheduling problem on identical machines. Here, each job $i$ has some weight $p_i$ and is only allowed to use a subset $M(i) \in M$ of the machines. This is a special case of the unrelated scheduling problem where $p_{ij} = p_i$ if $j \in M(i)$ and $p_{ij} = \infty$ otherwise. Given an integer $T$ and a $T$-feasible assignment $\alpha$, Unsplittable-Blocking-Flow$(\alpha, T)$ computes a $T$-feasible assignment $\beta$ in which there is no path from $M^+(\beta)$ to $M^-(\beta)$ in $G_\beta(T)$.

In the following we present a version of the Unsplittable-Blocking-Flow algorithm adapted to the setting of our problem, i.e., it runs on processing times $p_{ij}$ and pushes jobs only along edges from some subgraph $G^0_\alpha(T)$ of graph $G_\alpha(T)$. The subgraph $G^0_\alpha(T)$ has the property that each job node has exactly one incoming edge (from the machine node to which it is assigned by $\alpha$) and at least one outgoing edge. We give a formal definition of subgraph $G^0_\alpha(T)$ in Section 3.5.2. In the following we also call this modified algorithm Unsplittable-Blocking-Flow. These adaptations do not influence the correctness and the running time of the Unsplittable-Blocking-Flow algorithm. Algorithm 4 shows the structure of the adapted algorithm.

The Unsplittable-Blocking-Flow algorithm reassigns jobs so that the loads of underloaded machines from $M^-$ never decrease, the loads of overloaded machines from $M^+$ never increase, and machines from $M^0$ stay in $M^0$. It receives as input an assignment $\alpha$, a graph $G^0_\alpha(T) = (E^0_\alpha, V^0_\alpha)$, a matrix of processing times $\mathbf{P}$ and a positive integer $T$ (see Algorithm 4). When Unsplittable-Blocking-Flow terminates, we have computed a new assignment $\beta$ having the property that in $G^0_\beta(T)$ there is no path from $M^+(\beta)$ to $M^-(\beta)$.

We give now a more detailed description of the algorithm. Unsplittable-Blocking-Flow is controlled by a height function $h : V \mapsto \mathbb{N}_0$ with

$$h(j) = \text{dist}_{G^0_\alpha}(j, M^-), \forall j \in V^0_\alpha.$$

Here, the distance is defined as the number of edges. Observe that $h$ induces a *levelgraph* on $G^0_\alpha(T)$. We call an edge $(u, v) \in G^0_\alpha(T)$ *admissible* with respect to the height function $h$ (or just admissible if it is clear from the context) if $h(u) = h(v) + 1$. In an admissible path all edges are admissible. For each node $j \in V$ with $0 < h(j) < \infty$, let $S(j)$ be the set of successors of node $j$ that is the set of nodes to which $j$ has an admissible edge, i.e.,

$$S(j) = \left\{ i \in V \mid (j, i) \in E^0_\alpha \text{ and } h(j) = h(i) + 1 \right\}.$$

Let $s(j)$ be the first node on list $S(j)$.

Now we give a definition of a *helpful* machine which is crucial for the understanding of Lemma 3.4.1. This lemma is a result from [52] adapted for the unrelated scheduling problem. It gives properties of the machine nodes lying on a directed path which begins with a helpful machine. These properties are important for the further analysis.

**Definition 3.4.1** *A machine $j \in V$ is called helpful with respect to some integer $T$ and some $T$-feasible assignment $\alpha$, if $h(j) < \infty$ and $\delta_j(\alpha) \geq T + 1 + p_{s(j),j}$.*

**Algorithm 4** Unsplittable-Blocking-Flow($\alpha, G^0_\alpha(T), \mathbf{P}, T$)

```
Input: assignment $\alpha$, graph $G^0_\alpha(T)$
       matrix of processing times $\mathbf{P}$
       positive integer $T$
Output: assignment $\beta$
 1: compute height $h$ for each node in $G^0_\alpha(T)$;
 2: while $M^- \neq \emptyset$ and $\exists j \in M^+$ with $h(j) < \infty$ do
 3:     $d := \min_{j \in M^+}(h(j))$;
 4:     while $\exists$ admissible path from $j \in M^+$ with $h(j) = d$ to $M^-$ in graph $G^0_\alpha(T)$ do
 5:         choose some helpful machine $v \in M$ of minimum height $h$;
 6:         push jobs along a helpful path defined by $v$;
 7:         update $\alpha$, $G^0_\alpha(T)$, $M^+$, $M^-$;
 8:     end while
 9:     recompute $h$;
10: end while
11: return assignment $\alpha$;
```

Observe that a machine $j \in M^+$ is *always* helpful since only jobs $i$ with $p_{ij} \leq T$ are assigned to it.

**Lemma 3.4.1** *Let $v_0$ be a helpful machine of minimum height. Then there exists a sequence $v_0, \ldots, v_r$ where $s(v_i) = v_{i+1}$, for all $0 \leq i \leq r-1$, $v_{2i} \in M$, for all $0 \leq i \leq r/2$ and $v_{2i+1} \in J$, for all $0 \leq i < r/2$ with the following properties:*

*(a)* $(v_i, v_{i+1}) \in E^0_\alpha$ *and* $h(v_i) = h(v_{i+1}) + 1$

*(b)* $\delta_{v_0} \geq T + 1 + p_{s(v_0),v_0}$

*(c)* $T + 1 \leq \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} - p_{s(v_{2i}),v_{2i}} \leq 2T,\ \forall 0 < i < r/2$

*(d)* $\delta_{v_r} + p_{s(v_{r-2}),v_r} \leq 2T$

**Proof:** By Definition 3.4.1, $0 < h(v_0) < \infty$ and thus there exists a path from $v_0$ to a machine in $M^-$ that defines the height of $v_0$. On this path (a) must hold. Furthermore, condition (b) follows directly from the definition of a helpful machine.

Note that a machine $j \in M^+$ is helpful if $h(j) < \infty$. So if we start a path with a helpful machine of minimum height, then all machine nodes $v_2, v_4, \ldots, v_{r-2}$ belong to $M^0$ and $v_r$ may belong to $M^0$ or $M^-$. Therefore

$$\delta_{v_{2i}} \leq 2T, \forall 0 < i \leq r/2.$$

Furthermore, none of these nodes is helpful what implies that

$$\delta_{v_{2i}} < T + 1 + p_{s(v_{2i}),v_{2i}}, \forall 0 < i \leq r/2.$$

There are two cases to consider now. If $\delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} \leq 2T$, then $r = 2i$ and condition (d) holds. On the other hand, since $T \geq p_{s(v_{2i}),v_{2i}}$

$$\begin{array}{rcl} \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} & \geq & 2T + 1 \\ \Rightarrow \quad \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} - p_{s(v_{2i}),v_{2i}} & \geq & 2T + 1 - p_{s(v_{2i}),v_{2i}} \\ \Rightarrow \quad \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} - p_{s(v_{2i}),v_{2i}} & \geq & T + 1, \end{array}$$

what proves the lower bound given in (c). To proof the upper bound, note that $v_{2i}$ is not helpful for all $0 < i \leq r/2$. It follows that

$$\begin{array}{rcl} \delta_{v_{2i}} & < & T + 1 + p_{s(v_{2i}),v_{2i}} \\ \Rightarrow \quad \delta_{v_{2i}} - p_{s(v_{2i}),v_{2i}} + p_{s(v_{2i-2}),v_{2i}} & \leq & T + p_{s(v_{2i-2}),v_{2i}} \\ \Rightarrow \quad \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} - p_{s(v_{2i}),v_{2i}} & \leq & 2T, \end{array}$$

proving the upper bound given by (c). This completes the proof of the lemma. ■

We call the sequence of nodes $v_0, \ldots, v_r$ defined by Lemma 3.4.1 a *helpful path*. Observe that if we reassign jobs according to a helpful path then:

(1) we reassign jobs only according to admissible edges,

(2) the load on machine $v_0$ decreases but not below $T + 1$,

(3) all machines $v_{2i}$ with $0 < i < r/2$ stay in $M^0$, and

(4) the load on machine $v_r$ increases but not above $2T$.

The Unsplittable-Blocking-Flow algorithm terminates when $M^- = \emptyset$ or when $h(j) = \infty$ for all machines $j \in M^+$. It works in phases. Before the first phase starts, the height function $h$ is computed as the distance of each node in graph $G^0_\alpha$ to a node in $M^-$. When computing $h$ we also collect the set of admissible edges with respect to $h$. In each phase, first the minimum height $d = h(j)$ of a machine $j \in M^+$ is computed. Inside a phase, we do not update the height function but we successively choose a helpful machine $v$ of minimum height, and then we push jobs along the helpful path induced by $v$ and adjust the assignment accordingly. Each job push is equivalent to the reassignment of that job. In order to update $G^0_\alpha$, we have to change the direction of two arcs for each job push. The newly created edges are not admissible with respect to $h$. To update $M^+$ and $M^-$, it suffices to check the load on the first and the last node of the helpful path. The phase ends when no further admissible path from a machine $j \in M^+$ with $h(j) = d$ to some machine in $M^-$ exists in the levelgraph defined by the admissible edges with respect to the height function $h$. Before the new phase starts we recompute $h$ and we check whether we have to start a new phase or not. In [52] it was shown that there are at most $m$ phases and the running time of each phase is dominated by the computation of the height function $h$ what can be done by *breadth-first-search* in time $O(A)$ where $A$ is the number of edges in the admissible graph $G^0_\alpha$.

Lemma 3.4.2 and Theorem 3.4.3 are derived from [52] and describe the properties of the Unsplittable-Blocking-Flow algorithm that are used in the next section in the discussion of the Unsplittable-Truemper algorithm. Their proofs are direct generalizations of the corresponding proofs in [52]. For details on the complete analysis we refer to [52].

**Lemma 3.4.2 (Gairing et al. [52])** *Let $\beta$ be the assignment computed by* Unsplittable-Blocking-Flow*$(\alpha, G^0_\alpha(T), \mathbf{P}, T)$. Then*

*(a)* $j \in M^-(\alpha) \Rightarrow \delta_j(\mathbf{P},\beta) \geq \delta_j(\mathbf{P},\alpha)$

*(b)* $j \in M^0(\alpha) \Rightarrow T + 1 \leq \delta_j(\mathbf{P},\beta) \leq 2T$

*(c)* $j \in M^+(\alpha) \Rightarrow \delta_j(\mathbf{P},\beta) \leq \delta_j(\mathbf{P},\alpha)$.

**Proof:** The Unsplittable-Blocking-Flow algorithm pushes only jobs along a helpful path that is defined by a helpful machine $v$ of minimum height $h(v)$. Lemma 3.4.1 shows that by doing this we never add a machine to $M^-$. Furthermore, a machine in $M^-$ can only be the last machine in such a helpful path. But this machine only receives load which implies case (a). On the other hand, a machine $v \in M^+$ can only be the first machine in such a helpful path, case (c) follows. Condition (b) also follows directly from Lemma 3.4.1. ■

Lemma 3.4.2 ensures that the loads of the machines from $M^-$ never decrease, the loads of machines from $M^+$ never increase, and machines from $M^0$ stay in $M^0$.

Let $G^0_\alpha(T)$ be the subgraph of $G_\alpha(T)$. Let $\beta$ be the assignment computed by Unsplittable-Blocking-Flow$(\alpha, G^0_\alpha(T), \mathbf{P}, T)$. During this call jobs were reassigned by pushing them along the edges of $G^0_\alpha(T)$. We define $G^0_\beta(T)$ as the graph that results from $G^0_\alpha(T)$ after these reassignments.

**Theorem 3.4.3 (Gairing et al. [52])** *The* Unsplittable-Blocking-Flow$(\alpha, G^0_\alpha(T), \mathbf{P}, T)$ *algorithm takes running time $O(mA)$ and computes an assignment $\beta$, having the property that there is no path from $M^+(\beta)$ to $M^-(\beta)$ in $G^0_\beta(T)$.*

## 3.5 Approximation Algorithm

In the following we present the Approximation-Unsplittable-Truemper algorithm. This fully combinatorial approach is based on generalized and unsplittable network flows. The main part of our approximation algorithm constitutes the Unsplittable-Truemper algorithm [53]. In contrast to other algorithms, Unsplittable-Truemper always maintains an integral solution, i.e., an unsplittable flow in the bipartite network defined by the scheduling problem. By allowing some gap for the machine loads we loose a factor of 2 in the quality of solution. Also the scaling technique discussed in Section 3.3, which we use to obtain the polynomial running time, introduces some error. However, the special structure of Unsplittable-Truemper allows us to compensate this error by doing a more careful lower bound analysis on $C^*_{\max}(\mathbf{P})$. We terminate the computations as soon as we get this better lower bound on $C^*_{\max}(\mathbf{P})$. This approach improves also the running time. This and other theoretical results concerning the correctness and running time of Unsplittable-Truemper are given in the second part of this section. In the last subsection, we show how Unsplittable-Truemper can be used to compute an assignment $\alpha$ where $C_{\max}(\mathbf{P}, \alpha) \leq 2 \cdot C^*_{\max}(\mathbf{P})$.

### 3.5.1 Algorithm Unsplittable-Truemper

We formulate the unrelated scheduling problem as a generalized minimum cost unsplittable flow problem with rounded gain factors as described in Section 3.3. Truemper [164] showed that the generalized network flows and the minimum cost network flows are closely related. More specifically, he proved that the well-known Primal-Dual approach [3] for minimum cost flows can be used to compute a generalized maximum flow. In honor of Trumeper, Tardos and Wayne [162] named their algorithm after Truemper. In order to solve our generalized unsplittable flow problem we use a similar approach. Since we also use ideas by Truemper, we have decided to call our algorithm Unsplittable-Truemper.

The Unsplittable-Truemper algorithm maintains the reduced cost optimality condition (see Theorem 3.1.1) what is implied by the usage of the Primal-Dual approach [3]. In our setting this means that the algorithm does not create negative cost cycles in the residual network of the generalized unsplittable flow problem. Unsplittable-Truemper works in phases what is represented by the `while`-loop of Algorithm 5. In order to maintain the reduced cost optimality condition, Unsplittable-Truemper iteratively computes an admissible graph $G^0_\alpha(T)$, which we define below, and then uses an adapted version of Unsplittable-Blocking-Flow to compute a blocking flow in this admissible graph. While the costs $c_{ij}$ in Unsplittable-Truemper refer to the rounded processing times $p_{ij}$, the algorithm itself operates on the original processing times $p_{ij}$ given by $\mathbf{P}$. More specifically, it uses the original processing times $\mathbf{P}$ to compute machine loads $\delta_j(\alpha)$ for a given assignment $\alpha$. It is important to note that both the costs $c_{ij}$ as well as the original processing times $p_{ij}$ are integer. By Theorem 3.4.3 we know that there exists no path from a machine in $M^+$ to a machine in $M^-$ in an admissible graph $G^0_\alpha(T)$ after termination of Unsplittable-Blocking-Flow. The Unsplittable-Truemper algorithm leaves the `while`-loop when it can either derive a *good* lower bound on $C^*_{\max}(\mathbf{P})$ (see Theorem 3.5.3) or it has found an assignment $\alpha$ with empty set of overloaded machines $M^+$ (see the termination conditions of the `while`-loop of Algorithm 5). More details are given later in the subsection concerning the analysis of the algorithm.

We now describe our algorithm in more detail. Unsplittable-Truemper starts with an initial assignment $\alpha$. In $\alpha$ each job $i \in J$ is assigned to some machine $j \in B(i)$ where its processing time is minimum, i.e.,

$$B(i) = \{j \in M \mid p_{ij} \leq p_{ik}, \forall k \in M\}.$$

Arc capacities are given by $\mathbf{P}$ whereas arc costs are given by $\mathbf{C}$. Furthermore, Unsplittable-Truemper gets as input an integer $T$ which is large enough so that assignment $\alpha$ is $T$-feasible. In our setting, it means that $T \geq \max_{i\in[n]} \min_{j\in[m]}\{p_{ij}\}$. Assignment $\alpha$ and integer $T$ define us a graph $G_\alpha(T)$ as given in Definition 3.3.1, and the partitions of machines are constructed as described in Definition 3.3.2. At any time of the computations Unsplittable-Truemper maintains a feasible assignment that is all jobs are *always* assigned to some machine. If a job gets unassigned from a machine (during *push* operation of Unsplittable-Blocking-Flow, see Algorithm 4), it is immediately assigned to some other machine.

The Unsplittable-Truemper algorithm computes iteratively shortest path distances $d(u)$ from each node $u \in M \cup J$ to the set of sinks $M^-$ with respect to the reduced costs $c^\pi_{ij}$.

**Algorithm 5** Unsplittable-Truemper($\alpha, \mathbf{P}, \mathbf{C}, T$)

```
Input: assignment $\alpha$
       matrix of processing times $\mathbf{P}$ and matrix of edge costs $\mathbf{C}$
       positive integer $T$
Output: assignment $\beta$
 1: let $G_\alpha(T)$ be a bipartite graph induced by matrix $\mathbf{P}$, assignment $\alpha$ and parameter $T$;
 2: $\pi := 0$;
 3: while $\exists\, j \in M^+$ with a path to $k \in M^-$ in $G_\alpha(T)$ and $\forall u \in M^+ : \pi(u) < \log_b(m)$ do
 4:     compute shortest path distances $d(\cdot)$ from all nodes to the set of sinks $M^-$ in $G_\alpha(T)$
            with respect to the reduced costs $c^\pi_{ij}$;
 5:     $\pi := \pi + d$;
 6:     compute set $M^+_{\min} \subseteq M^+$ of machines with minimum distance to a node in $M^-$ with
            respect to the costs $c_{ij}$;
 7:     compute admissible graph $G^0_\alpha(T) \subseteq G_\alpha(T)$ consisting only of edges on shortest
            paths from $M^+_{\min}$ to $M^-$ in $G_\alpha(T)$;
 8:     $\alpha :=$ Unsplittable-Blocking-Flow($\alpha, G^0_\alpha(T), \mathbf{P}, T$);
 9:     update $G_\alpha(T)$;
10: end while
11: return assignment $\alpha$;
```

Afterward, the vector of potentials $\pi$ is updated such that all arcs on shortest paths to the sinks $M^-$ have zero reduced costs. For each node $u \in M$, $\pi(u)$ never decreases. Furthermore, note that after the update of potentials $\pi$, $\pi(u)$ holds the minimum distance from $u$ to $M^-$ for each node $u$ with respect to the costs $c_{ij}$.

**Definition 3.5.1** *We define $M^+_{\min}$ as the set of machines from $M^+$ with minimum distance to a node in $M^-$ with respect to the costs $c_{ij}$.*

**Definition 3.5.2** *We define the admissible graph $G^0_\alpha(T)$ as a subgraph of graph $G_\alpha(T)$, consisting only of edges on shortest paths from $M^+_{\min}$ to $M^-$ in $G_\alpha(T)$.*

Note that $M^+_{\min}$ consists of all machines $u \in M^+$ where $\pi(u)$ is minimum. Furthermore, note that $G^0_\alpha(T)$ includes only arcs with zero reduced costs. We will see later that these two definitions are essential for our algorithm.

After the admissible graph $G^0_\alpha(T)$ is computed, Unsplittable-Blocking-Flow is applied to it. It reassigns jobs from the admissible graph such that after Unsplittable-Blocking-Flow terminates there is no longer a path from a machine in $M^+_{\min}$ to a machine in $M^-$ in the admissible graph $G^0_\alpha(T)$. Hence, $\min\{\pi(u) \,|\, u \in M^+\}$ must increase in the next iteration of the `while`-loop of Algorithm 5. The residual network $G_\alpha(T)$ is then updated accordingly. The `while`-loop terminates when there exists no machine from $M^+$ with a path to a machine from $M^-$ in $G_\alpha(T)$ or there exists a machine $u \in M^+$ with $\pi(u) \geq \log_b(m)$.

## 3.5.2 Correctness and Running Time

We now analyze the correctness and the running time of UNSPLITTABLE-TRUEMPER. The main result of this part is Theorem 3.5.3. It gives the worst case running time and shows that a call to UNSPLITTABLE-TRUEMPER($\alpha, \mathbf{P}, \mathbf{C}, T$) terminates if either $M^+(\alpha) = \emptyset$ or there exists a machine $u \in M^+$ with $\pi(u) \geq \log_b(m)$. From the first case we know that $C_{\max}(\mathbf{P}, \alpha) \leq 2T$ for a given $T$. We will see that in the latter case we can take some advantage from an assignment $\alpha$ which is still unfavorable, i.e., for which $M^+(\alpha) \neq \emptyset$ holds.

The reduced cost optimality condition, $c^\pi_{ij} \geq 0$, holds for all $(i, j) \in E_\alpha(T)$ during the whole computation. This fact is ensured by the usage of the PRIMAL-DUAL approach. It also implies that $\gamma(K) \geq 1$ for each cycle $K$ in $G_\alpha(T)$ since

$$\gamma(K) = \prod_{(i,j)\in K} \gamma_{ij} = \prod_{(i,j)\in K} b^{c_{ij}} = b^{\sum_{(i,j)\in K} c_{ij}} = b^{\sum_{(i,j)\in K} c^\pi_{ij}} \geq b^0 = 1.$$

This property does not necessarily hold for every path $W$ in $G_\alpha(T)$. Lemma 3.5.1 is of crucial importance for the analysis. It shows that $\gamma(W) \geq 1$ holds for every path $W$ connecting some node from $M^+(\alpha)$ to any other node from $M$ in $G_\alpha(T)$. For proving this result we need the special structure of the admissible graph $G^0_\alpha(T)$ given by Definition 3.5.2, i.e., that $G^0_\alpha(T)$ is defined only by shortest paths from nodes in $M^+_{\min}$ to nodes in $M^-$.

**Lemma 3.5.1** UNSPLITTABLE-TRUEMPER *maintains the property that for each path $W$ in $G_\alpha(T)$ from any machine in $M^+$ to any other machine in $M$ we have $\gamma(W) \geq 1$.*

**Proof:** We show that the claim is an invariant of the algorithm. The property holds at the beginning since each job $i$ is assigned to a machine $j \in B(i)$. Assume the claim holds at some time of the execution of UNSPLITTABLE-TRUEMPER. We will show that after the next job reassignment the claim still holds. Our algorithm reassigns only jobs on shortest paths from $M^+_{\min}$ to $M^-$. For any two nodes $u, v \in V$ denote $W_{uv}$ as a path from $u$ to $v$ in $G_\alpha(T)$ where $\gamma(W_{uv})$ is minimum. If no such path exists, define $\gamma(W_{uv}) = \infty$. Let $j$ be any machine from $M^+_{\min}$ and let $i$ be any job on a shortest path from $j$ to $M^-$ as given in Figure 3.2(a).

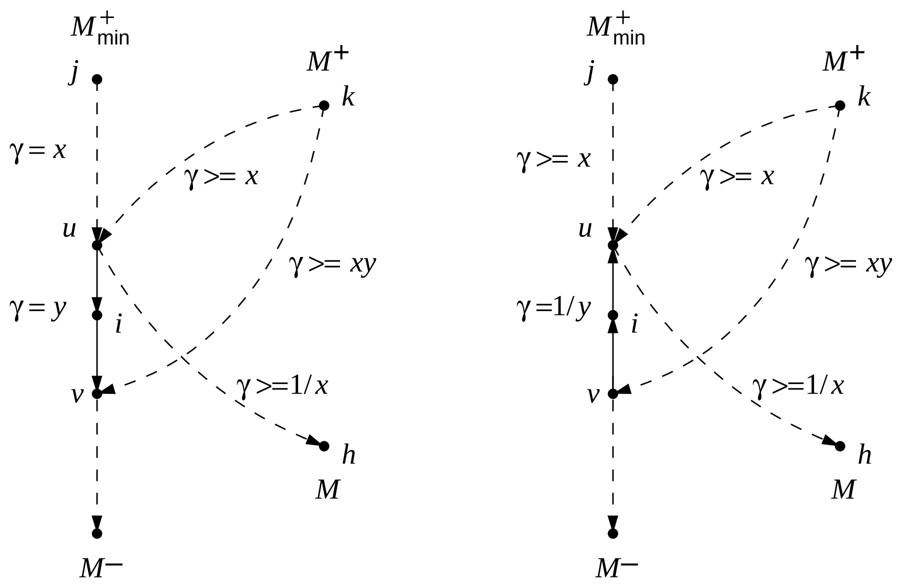
We may assume that $i$ gets reassigned from some machine $u$ to some machine $v$ (see Figure 3.2(b)). Define $y = \gamma(W_{uv})$ and $x = \gamma(W_{ju})$. Note that this implies that $\gamma(W_{jv}) = xy$. Let $k$ be any machine from $M^+$ and consider any path from $k$ to some other machine $h \in M$. Since $j$ has minimum distance to $M^-$, we know that $\gamma(W_{ku}) \geq x$ and $\gamma(W_{kv}) \geq xy$, otherwise $j \notin M^+_{\min}$. By reassigning $i$, $\gamma(W_{uv})$ can not decrease. Only $\gamma(W_{vu})$ on the path from $v$ to $u$ might decrease. If $\gamma(W_{vu})$ does not decrease, the claim follows immediately.

So assume $\gamma(W_{vu})$ decreased (see Figure 3.2 (b)). However, then $\gamma(W_{vu})$ is defined by the new path $(v \to i \to u)$ and thus $\gamma(W_{vu}) = \frac{1}{y}$. Now consider path $W_{kh}$. If $W_{kh}$ does not go through $(v, i, u)$, then $\gamma(W_{kh})$ did not change. However, if $W_{kh}$ uses $(v \to i \to u)$, then

$$\gamma(W_{kh}) = \gamma(W_{kv}) \cdot \gamma(W_{vu}) \cdot \gamma(W_{uh}) \geq x \cdot \gamma(W_{uh}).$$

Since $j$ has a path to $u$ with $W_{ju} = x$ and $\gamma(W_{jh}) \geq 1$, it follows that $\gamma(W_{uh}) \geq \frac{1}{x}$. Thus, $W_{kh} \geq 1$. This completes the proof of the lemma. ■

The following lemma will be used to derive a lower bound on $C^*_{\max}(\mathbf{P})$.



(a) before the push (b) after the push

Figure 3.2: Before and after the push from machine $j$ to some machine in $M^-$ the gain of each path from any machine in $M^+$ to any machine in $M$ is at least 1.

**Lemma 3.5.2** *Let $(G, \mathbf{\Gamma})$ denote a generalized maximum unsplittable flow problem defined by network $G$ and matrix of processing times $\mathbf{\Gamma}$. Let $f$ be a generalized feasible unsplittable flow in $(G, \mathbf{\Gamma})$, and let $s, t \in \mathbb{R}^+$. Suppose $\forall u \in M : \delta_u(\mathbf{\Gamma}, f) \geq s$, and $\exists \hat{u} \in M : \delta_{\hat{u}}(\mathbf{\Gamma}, f) \geq s + t$, and for each cycle $K$ in $G_f$, $\gamma(K) \geq 1$. If on every path $W$ in $G_f$ from $\hat{u}$ to any other machine $u \in M$, $\gamma(W) \geq 1$, then $C^*_{\max}(\mathbf{\Gamma}) \geq s + \frac{t}{m}$.*

**Proof:** Let $f^*$ be an optimum generalized fractional flow in $(G, \mathbf{\Gamma})$ and define $\tilde{f} = f^* - f$. Consider the cycle/path decomposition of $\tilde{f}$ according to Lemma 3.1.3. Recall that in this cycle/path decomposition no end node of some path is also a starting node of some other path. Note that $\gamma(K) \geq 1$ for any cycle $K$. This implies that pushing flow along any cycle $K$ does not decrease the load on any of its machines. By pushing flow along a path $W$, only the load of the starting node of $W$ can decrease. The load on the inner nodes of $W$ does not change and the last node receives load. Since $\gamma(W) \geq 1$, the increase in the load of the end node is not smaller than the decrease of the load of the starting node. Together with the facts that $\delta_u(\mathbf{\Gamma}, f) \geq s$ for all $u \in M$ and that there exists $\hat{u} \in M$ with $\delta_{\hat{u}}(\mathbf{\Gamma}, f) \geq s + t$, we can imply that $C^*_{\max}(\mathbf{\Gamma}) \geq s + \frac{t}{m}$. ■

**Theorem 3.5.3** *The* Unsplittable-Truemper *algorithm takes time $O(m^2 A \log(m))$. Furthermore, if* Unsplittable-Truemper$(\alpha, \mathbf{P}, \mathbf{C}, T)$ *terminates with $M^+ \neq \emptyset$ then $C^*_{\max}(\mathbf{P}) \geq T + 1$.*

**Proof:** We first show that Unsplittable-Truemper terminates after at most $O(m \log(m))$ iterations of the `while`-loop. In the first place consider one iteration. Recall that $\pi(u)$ for each node $u \in M$ never decreases. Furthermore, Unsplittable-Blocking-Flow terminates if and only if there is no path from $M^+_{\min}$ to $M^-$ in the residual network. According to Lemma 3.4.2, it does not add nodes to $M^+$. Afterward, either $M^+_{\min} = \emptyset$ or the distance $d(u)$ with respect to the reduced costs $c^{\pi}_{ij}$ from any node $u \in M^+_{\min}$ to a sink is at least 1. If $M^+_{\min} = \emptyset$, then in the next iteration $M^+_{\min}$ is defined by a new set of nodes from $M^+$ with larger potential $\pi$. Hence, in each case $\pi(u)$ for $u \in M^+_{\min}$ increases at least by one with each iteration. The algorithm terminates if there exists a node $u \in M^+$ with $\pi(u) \geq \log_b(m)$. Note that

$$\log_b(m) = \frac{\log_2(m)}{\log_2(b)} = \frac{\log_2(m)}{\log_2(1 + 1/m)} = O(m \cdot \log(m)).$$

Thus, at most $O(m \log(m))$ iterations of the `while`-loop are possible. The running time of one iteration of the `while`-loop is dominated by the running time of Unsplittable-Blocking-Flow. By Theorem 3.4.3, this takes $O(mA)$ running time. Finally, the total running time of Unsplittable-Truemper is $O(m^2 A \log(m))$. This completes the proof of the running time.

We now show the lower bound on $C^*_{\max}(\mathbf{P})$. Let $\beta$ be the assignment computed by a call to Unsplittable-Truemper($\alpha, \mathbf{P}, \mathbf{C}, T$). In the construction of graph $G_\alpha(T)$, we only have edges for processing times not greater than $T$ (see Definition 3.3.1). Thus, we do not assign job $i$ to machine $j$ with $p_{ij} \geq T + 1$. If in an optimum assignment a job $i$ is assigned to a machine $j$ with $p_{ij} \geq T + 1$, then $C^*_{\max}(\mathbf{P}) \geq T + 1$ follows immediately. Therefore, in the following we can assume that in an optimum assignment each job $i \in J$ is assigned only to machine $j \in M$ with $p_{ij} \leq T$.

Note that Unsplittable-Truemper maintains the reduced cost optimality condition for rounded gains $\mathbf{\Gamma}$. Since for each edge $(i, j)$, we have $\gamma_{ij} \leq p_{ij} \leq b\,\gamma_{ij}$. It follows that:

$$\delta_u(\mathbf{P},\beta) \leq b \cdot \delta_u(\mathbf{\Gamma},\beta), \ \text{for all } u \in M.$$

Since Unsplittable-Truemper has terminated, there is either no longer a path from $M^+$ to $M^-$ in the residual graph $G_\beta(T)$ or there exists a machine $u \in M^+$ with $\pi(u) \geq \log_b(m)$. In the following we consider these two cases.

Case I: $\neg\exists$ path from $M^+$ to $M^-$ in $G_\beta(T)$.
Define $\widetilde{M}$ as the set of machines still reachable from $M^+$ in $G_\beta(T)$. The load of jobs assigned to a machine from $\widetilde{M}$ can not be distributed to the other machines. For each machine $j \in \widetilde{M}$, we have $\delta_j(\mathbf{P},\beta) \geq T + 1$ and therefore $\delta_j(\mathbf{\Gamma},\beta) \geq \frac{1}{b}(T + 1)$. Furthermore, since $M^+ \neq \emptyset$, there exists machine $v$ with $\delta_v(\mathbf{P},\beta) \geq 2T + 1$. This implies that $\delta_v(\mathbf{\Gamma},\beta) \geq \frac{1}{b}(2T + 1)$.

Since the Unsplittable-Truemper algorithm maintains the reduced cost optimality condition for the rounded gains, we have $\gamma(K) \geq 1$ for any cycle $K$ in $G_\beta(T)$. By Lemma 3.5.1, we know that $\gamma(W) \geq 1$ for each path from any machine in $M^+$ to any other machine in $M$. Applying Lemma 3.5.2 to the machines in $\widetilde{M}$ with the matrix of processing times $\mathbf{\Gamma}$ proves the lower bound on $C^*_{\max}(\mathbf{\Gamma})$:

$$C^*_{\max}(\mathbf{\Gamma}) \geq \frac{1}{b}\left(T + 1 + \frac{T}{m}\right).$$

Since $b = (1 + \frac{1}{m})$, the expression $\frac{1}{b}(T + 1 + \frac{T}{m}) > T$ holds. Putting all these information together, we obtain that

$$C^*_{\max}(\mathbf{P}) \geq C^*_{\max}(\mathbf{\Gamma}) \geq \frac{1}{b}\left(T + 1 + \frac{T}{m}\right) > T.$$

Since $C^*_{\max}(\mathbf{P})$ is integer, we get $C^*_{\max}(\mathbf{P}) \geq T + 1$.

Case II: $\exists u \in M^+$ with $\pi(u) \geq \log_b(m)$.
Unsplittable-Truemper maintains the reduced cost optimality condition $c^{\pi}_{ij} \geq 0$ for all $(i, j) \in E_f$. For any path $W$ from node $u$ to some node $v$ in $E_f$, $c^{\pi}(W) = c(W) - \pi(u) + \pi(v) \geq 0$ holds. Now, $\pi(v) = 0$ holds for $v \in M^-$, and this implies

$$c(W) \geq \pi(u) \geq \log_b(m),$$

and therefore

$$\gamma(W) = b^{c(W)} \geq m.$$

Now assume that $C^*_{\max}(\mathbf{P}) \leq T$ and recall that $\delta_u(\mathbf{P},\beta) \geq 2T + 1$. Let $(G, \mathbf{P})$ and $(G, \mathbf{\Gamma})$ denote the generalized maximum unsplittable flow problem defined by network $G$ and matrix of processing times $\mathbf{P}$ and $\mathbf{\Gamma}$, respectively. Let $f$ be the generalized flow in $(G, \mathbf{P})$ that corresponds to assignment $\beta$ and let $f^*$ be an optimum generalized fractional flow in $(G, \mathbf{P})$. Define $\tilde{f} = f^* - f$. Note that $\tilde{f}$ is a generalized flow in $(G, \mathbf{P})$. However, $\tilde{f}$ is also a generalized flow in $(G, \mathbf{\Gamma})$. Define $\Delta_u(\mathbf{P}) = \delta_u(\mathbf{P},\beta) - \delta_u(\mathbf{P}, f^*)$. Since $u \in M^+$, $\Delta_u(\mathbf{P})$ is positive. $\Delta_u(\mathbf{P})$ is the amount of flow that is sent from machine $u$ to the other machines by $\tilde{f}$ in $(G, \mathbf{P})$. Define $\Delta_u(\mathbf{\Gamma})$ as the amount of flow that $\tilde{f}$ sends out of $u$ in $(G, \mathbf{\Gamma})$. It holds that

$$\Delta_u(\mathbf{\Gamma}) \geq \frac{1}{b}\Delta_u(\mathbf{P}).$$

Consider the cycle/path decomposition of $\tilde{f}$ according to Lemma 3.1.3. Pushing flow along any cycle $K$ does not decrease the load on any of its machines since $\gamma(K) \geq 1$. Because $C^*_{\max}(\mathbf{P}) \leq T$ and $\delta_v(\mathbf{P},\beta) \geq T + 1$ for all machines $v \in M^+ \cup M^0$, $\tilde{f}$ must send flow from all machines $v \in M^+ \cup M^0$ to machines in $M^-$. Recall that in the cycle/path decomposition no end node of some path is also a starting node of some other path. Since each machine from $M^0 \cup M^+$ is the starting node of some path of the cycle/path decomposition, it cannot be the end point of some other path. Thus, in the cycle/path decomposition of $\tilde{f}$ in $(G, \mathbf{\Gamma})$, a total flow of at least

$$\Delta_u(\mathbf{\Gamma}) \geq \frac{1}{b}\Delta_u(\mathbf{P}) = \frac{1}{b}\left(\delta_u(\mathbf{P},\beta) - \delta_u(\mathbf{P}, f^*)\right) \geq \frac{1}{b}\left(\delta_u(\mathbf{P},\beta) - C^*_{\max}(\mathbf{P})\right) \geq \frac{1}{b}(T + 1)$$

is sent on paths from machine $u$ to the machines in $M^-$.

However, since $\gamma(W) \geq m$ for every path $W$ from $u$ to some machine in $M^-$, the machines in $M^-$ will receive at least $\frac{1}{b}(T+1) \cdot m$ flow by $\tilde{f}$ in $(G, \mathbf{\Gamma})$. There are at most $m - 1$ machines in $M^-$. Hence, there exists a machine $s \in M^-$ that receives at least $\frac{1}{b}(T + 1) \cdot \frac{m}{m-1}$ flow by $\tilde{f}$ in $(G, \mathbf{\Gamma})$. Thus:

$$\delta_s(\mathbf{\Gamma}, f^*) \geq \frac{1}{b}(T + 1) \cdot \frac{m}{m - 1}.$$

Note that $\frac{1}{b} \cdot \frac{m}{m-1} > 1$ for $b = (1 + \frac{1}{m})$. Thus, we have

$$C^*_{\max}(\mathbf{P}) \geq \delta_s(\mathbf{P}, f^*) \geq \delta_s(\mathbf{\Gamma}, f^*) \geq \frac{1}{b}(T + 1) \cdot \frac{m}{m - 1} > T + 1.$$

This is a contradiction to our assumption that $C^*_{\max}(\mathbf{P}) \leq T$. Hence, $C^*_{\max}(\mathbf{P}) \geq T + 1$. This completes the proof of the theorem. ■

**Algorithm 6** APPROXIMATION-UNSPLITTABLE-TRUEMPER(**P**)

```
Input: matrix of processing times $\mathbf{P}$
Output: assignment $\beta$
 1: compute matrix $\mathbf{C}$ of rounded down costs $c_{ij}$;
 2: $l := \max_{i\in[n]} \min_{j\in[m]}\{p_{ij}\}$;
 3: $u := \sum_{i\in[n]} \min_{j\in[m]}\{p_{ij}\}$;
 4: while $l + 1 \neq u$ do
 5:     $T := \lceil \frac{l+u}{2} \rceil$;
 6:     compute initial assignment $\alpha$;
 7:     let $M^+$ be the set of overloaded machines induced by assignment $\alpha$;
 8:     $\alpha :=$ UNSPLITTABLE-TRUEMPER($\alpha, \mathbf{P}, \mathbf{C}, T$);
 9:     if $M^+ = \emptyset$ then
10:         $u := T$;
11:     else
12:         $l := T$;
13:     end if;
14: end while
15: return assignment $\alpha$;
```

### 3.5.3 APPROXIMATION-UNSPLITTABLE-TRUEMPER Algorithm

We show now how we use the UNSPLITTABLE-TRUEMPER algorithm to approximate a schedule with minimum makespan. We perform series of calls to the algorithm UNSPLITTABLE-TRUEMPER($\alpha, \mathbf{P}, \mathbf{C}, T$) where by a binary search on $T \in [l, u]$ we identify the smallest $T$ such that a call to the UNSPLITTABLE-TRUEMPER($\alpha, \mathbf{P}, \mathbf{C}, T$) returns an assignment with $M^+ = \emptyset$. Here, $l$ and $u$ represent the lower and the upper bound for $C^*_{\max}(\mathbf{P})$, respectively. Note that the upper bound for $u$ is $nU$. Exact values for $l$ and $u$ are given in Algorithm 6 which shows the overall structure of the APPROXIMATION-UNSPLITTABLE-TRUEMPER algorithm which we use for the approximation of the optimum schedule.

Afterward we have identified the value of parameter $T$ such that a call to UNSPLITTABLE-TRUEMPER($\alpha, \mathbf{P}, \mathbf{C}, T$) returns an assignment where $M^+ \neq \emptyset$, and a call to UNSPLITTABLE-TRUEMPER($\alpha, \mathbf{P}, \mathbf{C}, T + 1$) returns with $M^+ = \emptyset$.

**Theorem 3.5.4** UNSPLITTABLE-TRUEMPER *can be used to compute a schedule $\alpha$ with*

$$C_{\max}(\mathbf{P}, \alpha) \leq 2 \cdot C^*_{\max}(\mathbf{P})$$

*in time $O(m^2 A \log(m) \log(nU))$.*

**Proof:** We use UNSPLITTABLE-TRUEMPER as described above. Let $\beta_1$ be the assignment returned by UNSPLITTABLE-TRUEMPER($\alpha, \mathbf{P}, \mathbf{C}, T$) where $M^+ \neq \emptyset$, and let $\beta_2$ be the assignment returned by UNSPLITTABLE-TRUEMPER($\alpha, \mathbf{P}, \mathbf{C}, T + 1$) where $M^+ = \emptyset$. From $\beta_1$ it follows, by Theorem 3.5.3, that $C^*_{\max}(\mathbf{P}) \geq T + 1$ and in $\beta_2$ we have that $C_{\max}(\mathbf{P},\beta_2) \leq 2(T + 1)$. Thus, $C_{\max}(\mathbf{P},\beta_2) \leq 2 \cdot C^*_{\max}(\mathbf{P})$. It remains to show the running time of $O(m^2 A \log(m) \log(nU))$. By Theorem 3.5.3, one call to UNSPLITTABLE-TRUEMPER takes time $O(m^2 A \log(m))$. The binary search contributes a factor $\log(nU)$. This completes the proof of the theorem. ■

## 3.6 Comparison of Running Times

We compare now the running time of the Approximation-Unsplittable-Truemper algorithm with the so far fastest algorithms of Vaidya [166] and Radzik [144]. Both of the former approaches have been designed to solve the fractional generalized maximum flow problem (see Section 3.1.2) on a graph with node set $V$ and edge set $E$. Rounding the fractional solution yields a 2-approximation.

In our comparison we consider the following three techniques for computing a 2-approximation schedule to the $R||C_{\max}$ problem. For each method we give its worst case running time. There exist:

- interior point approach for generalized (fractional) flow problem and rounding [166]

$$O(|E|^{1.5}|V|^2 \log(U)),$$

- combinatorial algorithm for generalized (fractional) flow problem and rounding [144]

$$O(|E|\,|V|(|E| + |V| \log |V|) \log U \log(nU)), \text{ and}$$

- the integer combinatorial approach presented in this thesis

$$O(m^2 A \log(m) \log(nU)).$$

To compare these bounds, note first that in the bipartite network defined by the scheduling problem $A = |E| = O(nm)$ and $|V| = n + m$. The Approximation-Unsplittable-Truemper algorithm is linear in $A$. It clearly outperforms the previous algorithms if $n + m = o(A)$. In the case $A = \Theta(n + m)$, the integer combinatorial algorithm is better by a factor of $\Omega(\frac{\sqrt{n+m}}{\log(n)\log(m)})$ than Vaidya's algorithm and by a factor of $\Omega(\log U)$ faster than Radzik's algorithm. This is the first time that a combinatorial algorithm always beats the interior point approach for this problem. The heuristics [125, 160, 167] consider instances where $A = \Theta(nm)$. In this case our algorithm outperforms both former approaches by a factor almost linear in $n$.

The $(1+\varepsilon)$-approximation algorithms for the generalized maximum flow problem in [45, 143, 162] have all running time $\widetilde{O}(\log \varepsilon^{-1}|E|(|E| + |V| \log\log U))$ where the $\widetilde{O}(\cdot)$ notation hides a factor polylogarithmic in $|V|$ (cf. Section 1.4). Again, an extra factor of $O(\log(nU))$ is needed for the makespan minimization by binary search. This running time is not always better than ours. The fastest FPTAS that directly addresses the unrelated scheduling problem is due to Jansen and Porkolab [90] and has $O(\varepsilon^{-2}(\log \varepsilon^{-1})mn \min\{m, n \log m\} \log m)$ running time (cf. Section 1.4). Clearly, for constant $\varepsilon$ this algorithm is faster than our algorithm. However, for $\varepsilon$ in the order of $\frac{1}{m}$ and $\log(U) = O(n)$, their running times become comparable. We want to recall here that the $(1+\varepsilon)$-approximation algorithms for the generalized maximum flow problem followed by rounding yield a $(2 + \varepsilon)$-approximation for the unrelated scheduling problem.

— 4 –

# Heuristic Approaches for $R||C_{\max}$

In this chapter we present various heuristic algorithms for the problem of scheduling unrelated parallel machines. In contrast to already discussed methods, here, only several techniques possess well-known approximation factors and run in polynomial time. Generally speaking, they have been developed for real-life problems where the guarantees for the quality of solution and for the running time are often a matter of trade-off between the simplicity and high efficiency on the one hand, and the reliability and predictability on the other hand. Nevertheless, many computational results show that they are more efficient than most of the $\rho$-approximation algorithms mentioned in the previous chapters. In Chapter 6 we present a comprehensive experimental study comparing the efficiency of various heuristics for scheduling unrelated machines with different 2-approximation algorithms.

We begin our presentation of the heuristics for $R||C_{\max}$ with the list scheduling algorithms. They were the very first methods, considered already in the 1950s, for solving deterministic scheduling problems. All algorithms from this group work with a list of appropriate sorted jobs. They process the jobs sequentially, in order given by the list, and assign each of them to a machine according to some specific rule. Among other methods, we present here also our three adaptations of the well-known LPT algorithm by Graham [72]. In the second section of the chapter we concentrate on partial enumeration techniques. As we can see there, they mainly consist in solving an MIP subproblem with fewer binary assignment variables as in the original problem. Finally, in the last section we present a large neighborhood improvement procedure developed for $R||C_{\max}$ by Sourd [160]. We begin there with a presentation of a branch-and-greed scheme which was developed in the late 1990s to explore partially enumeration trees of large-scale optimization problems. Since a simple search in an enumeration tree is often not sufficient for many problems to provide solutions of satisfying quality, various post-optimization procedures using a branch-and-greed scheme have been proposed to improve the quality of solutions. In contrast to local improvement procedures like, e.g., taboo search or simulated annealing, they are able to search larger neighborhoods of a current solution, and thus are usually more efficient to find a better solution. In the last section of this chapter we present such post-optimization procedure given by Sourd [160] for solving the $R||C_{\max}$ problem.

## 4.1 List Scheduling Methods for $R||C_{\max}$

We begin our discussion on heuristic algorithms for the minimization of makespan in scheduling unrelated parallel machines problem with the presentation of *list scheduling* algorithms. These very simple methods were the first proposed for this problem. Ibarra and Kim [85] and Davis and Jaffe [27] analyzed several list scheduling heuristics for the $R||C_{\max}$ problem and for some of its special cases.

The main idea behind the list scheduling algorithms is similar to that of the LPT algorithm by Graham [72] which we have already presented in Section 1.3. What have these algorithms in common is the usage of a *list* for $n$ independent jobs which have to be assigned without preemption to the set of $m$ parallel machines. Depending on the structure of the algorithm itself, the list of jobs is first sorted either non-increasingly or increasingly with respect to the processing times $p_{ij}$, or in accordance to some heuristic. Afterward, in a single traverse of the list the algorithm assigns each job to some machine according to some specific rule. In most cases the total running time of these algorithms is dominated by the time needed for sorting the list of jobs.

Hence, the only things which differ these methods from each other are the way the jobs are sorted and the rule according to which the jobs are assigned. In the rest of this section we concentrate mainly on these differences.

### 4.1.1 $m$-approximation Algorithms

We present first several simple heuristics given by Ibarra and Kim [85] in 1977. Originally, they designed five algorithms, each of which was guaranteed to be at most $m$ times worse than the optimal solution in the worst case. In addition, three of them were proved to be *exactly m* times worse than optimal in the worst case. The forth algorithm was left as an open problem – its effectiveness was shown to be between 2 and $m$ times worse than optimal. We give now a brief description for the first four of them. We assume that at the beginning all jobs are saved in a list $L$ in an arbitrary order.

**Algorithm A:** Schedule job $i$ from job list $L$ on a machine that minimizes its completion time.

**Algorithm B:** For each job $i \in L$, let $p_i^{\min} = \min_{j\in[m]}\{p_{ij}\}$. Sort list $L$ non-increasingly according to $p_i^{\min}$, and then call Algorithm A to schedule list $L$.

**Algorithm C:** For each job $i \in L$, let $p_i^{\max} = \max_{j\in[m]}\{p_{ij}\}$. Sort list $L$ non-increasingly according to $p_i^{\max}$, and then call Algorithm A to schedule list $L$.

**Algorithm D:** After having scheduled $k$ jobs, the algorithm schedules a job (from among the remaining $n - k$ jobs) which gives the least completion time.

Note that algorithms B and C reduce to the LPT scheduling algorithm which is at most $\frac{4}{3} - \frac{1}{3m}$ worse than the optimum for the case with identical machines $P||C_{\max}$ (cf. Section 1.3.3). It is obvious that Algorithm A runs in time $O(nm)$. Evaluating $p_i^{\min}$ or $p_i^{\max}$ for all jobs in $L$ takes $O(nm)$ time, and sorting of list $L$ takes $O(n \log n)$ time. It follows that algorithms B and

C have the worst case time complexity of $O(n \max\{m, \log n\})$. For Algorithm D, scheduling a job $i$ after $k$ jobs have been scheduled takes $O(m(n-k))$ time. Thus, the running time of the algorithm is $O(mn^2)$. In [27] is presented an example which indicates that Algorithm D is at least $(1 + \log m)$ time worse than optimal in the worst case. For detailed proofs of the worst case bounds for the approximation factors we refer to [27, 85].

### 4.1.2 $2\sqrt{m}$-approximation Algorithm

The idea of list scheduling was continued later by several researchers. Davis and Jaffe [27] devised a simple and useful algorithm which we would like to present in the following. In Section 5.2 we describe how we use it for computation of an initial solution in our branch-and-price method.

We describe now the approximation algorithm. Consider an assignment function $\alpha$ as given by Definition 1.3.1. We associate a starting time function $s$ with a given assignment $\alpha$. Intuitively, for each job $i \in [n]$, the value $s(i)$ represents the time at which machine $\alpha(i)$ begins to process job $i$. Formally, a starting function $s$ is a map $s : [n] \mapsto \mathbb{R}_{\geq 0}$ satisfying for each machine $j \in [m]$ the following two conditions:

(a) at most one job is being executed at any time on machine $j$, and

(b) if $0 \leq t < \sum_{i:\alpha(i)=j} p_{ij}$, then at least one job is being executed at time $t$ on machine $j$.

The value $s(i)$ is called the starting time of job $i$. Job $i$ is being processed on machine $j$ at time $t$ provided $j = \alpha(i)$ and $s(i) \leq t < s(i) + p_{ij}$. Note that the first condition forces all jobs to be processed sequentially and the second condition prevents any idle period on any machine.

We define for each job $i \in [n]$ its *minimal processing time* as $p_i^{\max} = \min_{j\in[m]}\{p_{ij} < \infty\}$. Furthermore, let us define the *efficiency* of machine $j$ for job $i$ as $ef_{ij} = \frac{p_i^{\max}}{p_{ij}}$. Note that the maximum efficiency is one.

The structure of the approximation algorithm is given by Algorithm 7. It generates an assignment function $\alpha$ and a starting time function $s$ for a given matrix of processing times $\mathbf{P}$. The algorithm is an adaptation of the list scheduling algorithm. It maintains $m$ lists, each one of $n$ jobs sorted in non-increasing order of $ef_{ij}$. The algorithm proceeds by finding the machine which first completes the jobs it has already been assigned and assigning to that machine the next job on the list. The algorithm varies from the simple list scheduling algorithms (see the previous subsection) in two respects. First, a separate list of jobs is constructed for each machine $j$ in order to reflect their different efficiencies on different jobs. Second, if a machine is very inefficient for the next job on the list, the job is not assigned to this machine and, as a consequence, this machine is deactivated.

The algorithm terminates when there is no unassigned job remaining. Note that at the termination some machine is still active because the machine that has the best time on the last job assigned could never have been deactivated. Since each iteration of the `while`-loop either assigns a job to or deactivates a machine, the algorithm terminates after $n + m$ iterations.

**Algorithm 7** DAVIS-AND-JAFFE-LIST-SCHEDULING(**P**)

**Input:** processing times matrix **P**

**Output:** assignment $\alpha$

```
 1: $p_i^{\max} := \min_{j\in[m]}\{p_{ij}\}$, for all $i \in [n]$;
 2: $ef_{ij} := \frac{p_i^{\max}}{p_{ij}}$, for all $i \in [n]$ and $j \in [m]$;
 3: create for each $j \in [m]$, a list $L_j$ of $n$ jobs sorted in non-increasing order of $ef_{ij}$;
 4: $\delta_j := 0$, for all $j \in [m]$;
 5: mark all machines as *active*;
 6: **while** $\exists$ unassigned job **do**
 7:     $j := \arg\min\{\delta_k \,|\, k \in [m], k \text{ is active}\}$;
 8:     let $i$ be the next unassigned job on list $L_j$;
 9:     **if** $\neg\exists\, i$ **or** $ef_{ij} < \frac{1}{\sqrt{m}}$ **then**
10:         mark machine $j$ as *inactive*;
11:     **else**
12:         $\alpha(i) := j$;
13:         $s(i) := \delta_j$;
14:         $\delta_j := \delta_j + p_{ij}$;
15:     **end if**
16: **end while**
17: **return** assignment $\alpha$;
```

The assignment $\alpha$ and starting time function $s$ determined by Algorithm 7 satisfy the following three conditions:

(1) $ef_{ij} \geq \frac{1}{\sqrt{m}}$, for $j = \alpha(i)$,

(2) if $s(i) > s(k)$, then $ef_{ij} \leq ef_{kj}$, for $j = \alpha(k)$ and $i, k \in [n]$,

(3) if $\sum_{i:\alpha(i)=j} p_{ij} < s(k)$, then $ef_{kj} < \frac{1}{\sqrt{m}}$.

Intuitively, the first condition indicates that a job is processed on a machine only if the machine is efficient for this job, i.e., its efficiency is at least $\frac{1}{\sqrt{m}}$. The second condition ensures that if a job $k$ is assigned at an earlier time than job $i$, then it must be that machine $\alpha(k)$ is more efficient for $k$ than for $i$. Finally, the third condition prevents a machine from stopping as long as it is enough efficient for some unassigned job.

The fact that the algorithm satisfies the first condition follows immediate from the way the jobs are assigned. Condition (2) follows from the fact that if $s(i) > s(k)$, then the fact that job $k$ is assigned to $\alpha(k)$ implies that $k$ must have at least as high an efficiency on $\alpha(k)$ as job $i$. Condition (3) also follows immediately from the way the jobs are assigned.

To analyze the running time, note that the sorting of a single list requires time $O(n \log n)$ for each machine. This results in total of $O(mn \log n)$ time for sorting. Consider the total time spent on iterations in which machine $j$ is chosen in step (7) of Algorithm 7. Determining if a given job $i$ is unassigned requires time $O(1)$ if that information is stored as a bit array. Thus, steps (8)-(15) of all iterations in which a particular machine $j$ is chosen require

**Algorithm 8** LPT-SCHEDULING-VER1(**P**)

```
Input: processing times matrix P
Output: assignment $\alpha$
1: $\delta_j := 0$, for all $j \in [m]$;
2: while $\exists$ unassigned job do
3:     let $i$ be an unassigned job with a maximal processing time;
4:     find machine $j$ such that $j = \arg\min_{k\in[m]}\{p_{ik} + \delta_k\}$;
5:     $\delta_j := \delta_j + p_{ij}$;
6:     $\alpha(i) := j$;
7: end while
8: return assignment $\alpha$;
```

time at most $O(n)$. If a data structure is maintained which keeps loads $\delta_j$ sorted, than $m + n$ iterations of step (7) require at most time $O((m + n)\log m)$. Thus the total running time is dominated by the initial sorting and is at most $(mn \log n)$.

The following theorem summarizes the main results. For a detailed analysis of the upper bound for the worst case performance we refer the reader to [27].

**Theorem 4.1.1 (Davis and Jaffe [27])** *Let $\beta$ be an optimal assignment for an instance of the $R||C_{\max}$ problem. Then Algorithm 7 computes in time $O(nm \log n)$ an assignment $\alpha$ for which*

$$\frac{makespan(\alpha)}{makespan(\beta)} \leq 2.5\sqrt{m} + 1 + \frac{1}{2\sqrt{m}}.$$

Together with the proof of the approximation factor, the authors gave an example for which the algorithm is as best as $2\sqrt{m}$ times worse than optimal. This indicates that their analysis was tight up to a factor of 1.25. This algorithm is best possible (up to a constant factor) among algorithms using a certain restricted class of heuristics.

By doing a more careful analysis, they achieved a worst-case performance bound to a constant factor. They showed that the approximation factor equals $(2m - 2 - \varepsilon)/(\varepsilon + \sqrt{m})$ for $\varepsilon > 0$. Note that this bound approaches $2\sqrt{m} - 2/\sqrt{m}$ as $\varepsilon$ gets smaller.

### 4.1.3 LPT-based Algorithms

As it was already indicated in the introduction to this chapter, we present now three adaptations of the LPT algorithm by Graham [72] to the unrelated scheduling problem. This algorithm was originally designed to schedule jobs on identical machines and later made applicable for solving a more general scheduling problem on uniform machines. We recall that in its generic version the LPT algorithm first sorts the jobs in non-increasing order of processing times, and then sequentially assigns each job to a machine on which it will be completed soonest.

Our three simple adaptations of the LPT algorithm are accustomed to three different real-world situations. In the first situation we consider the case in which the processing times are uncorrelated, i.e., there is no direct relations between machines and jobs. Here, Algorithm 8 can be applied. Observe that the actual load on any machine $j$ is saved by $\delta_j$.

**Algorithm 9** LPT-SCHEDULING-VER2(**P**)

**Input:** processing times matrix **P**
**Output:** assignment $\alpha$

```
1: $\delta_j := 0$, for all $j \in [m]$;
2: $b_i := \min\{p_{ij} \mid j \in [m]\} - 1$, for all $i \in [n]$;
3: while $\exists$ unassigned job do
4:     let $i$ be an unassigned job with a maximal $b_i$;
5:     find machine $j$ such that $j = \arg\min_{k\in[m]}\{p_{ik} + \delta_k\}$;
6:     $\delta_j := \delta_j + p_{ij}$;
7:     $\alpha(i) := j$;
8: end while
9: return assignment $\alpha$;
```

**Algorithm 10** LPT-SCHEDULING-VER3(**P**)

**Input:** processing times matrix **P**
**Output:** assignment $\alpha$

```
1: $\delta_j := 0$, for all $j \in [m]$;
2: $a_j := \min\{p_{ij} \mid i \in [n]\} - 1$, for all $j \in [m]$;
3: while $\exists$ unassigned job do
4:     find job $i$ such that $i = \arg\max_{k\in[n]}\{p_{kj} - a_j \mid k\, is\, unassigned, j \in [m]\}$;
5:     find machine $j$, such that $j = \arg\min_{k\in[m]}\{p_{ik} + \delta_k\}$;
6:     $\delta_j := \delta_j + p_{ij}$;
7:     $\alpha(i) := j$;
8: end while
9: return assignment $\alpha$;
```

In the second case the jobs are correlated, i.e., processing time of job $i$ on machine $j$ can be expressed as $p_{ij} = b_i + d_{ij}$, where $b_i$ is a mean value of processing time of job $i$ and $d_{ij}$ is its deviation. To solve such problem we have developed Algorithm 9. Here, before the jobs are being assigned, a mean value $b_i$ is computed for each job $i \in [n]$. Afterward, the jobs are processed in non-increasing order of the mean value $b_i$. In the last case we consider a problem with correlated machines. The processing times of each job can now be formulated as $p_{ij} = a_j + d_{ij}$, where, similar to the previous case, $a_j$ is a mean value of machine performance for machine $j$, and $d_{ij}$ is its deviation. For this case we have developed Algorithm 10. Note that now the jobs are processed in non-increasing order of deviation $d_{ij}$.

## 4.2 Partial Enumeration Technique

We present now three heuristic algorithms based on partial enumeration for the $R||C_{\max}$ scheduling problem. All of them were designed by Mokotoff and Jimeno [126]. Starting with the LP relaxation of the MIP formulation of the unrelated scheduling problem given in (1.2), they try to find a near optimal solution. Basically, the heuristics consist in analyzing the instance characteristics and considering the integrality of some subset of assignment variables. All three algorithms are based on the methodology described first by Dillenberger et al. in [33] and used later by Mansini and Speranza [116].

We first describe the technique that these three algorithms have in common and then we give a comprehensive description of each of them. The proposed algorithms mainly consist in solving a MIP subproblem with fewer binary assignment variables than the original one. The main procedure checks whether or not the optimum solution of the LP relaxation is an integral vector. If it is the case, then the main algorithm stops. Otherwise, a subset of the assignment variables has to be eliminated.

**Extended LP relaxation.** Using the MIP formulation of the $R||C_{\max}$ problem given in (1.2), we formulate the following *extended* LP relaxation as follows:

$$\begin{array}{lll}\text{LP:} & \min\ T & \\ & \text{s.t.}\ \sum_{i\in[n]} p_{ij}x_{ij} \le T, & \forall\ j \in [m] \\ & \sum_{j\in[m]} p_{ij}x_{ij} \le T, & \forall\ i \in [n] : p_i^{\max} > \frac{1}{m}\sum_{k\in[n]} p_k^{\min} \\ & \sum_{j\in[m]} x_{ij} = 1, & \forall\ i \in [n] \\ & x_{ij} \ge 0, & \forall\ i \in [n], j \in [m] \\ & T \ge LB & \end{array} \tag{4.1}$$

To get an LP relaxation solution near the integral optimum, a lower bound $LB$ for $T$ is computed. The most simple lower bound is the solution of the relaxation which we obtain by assuming that the machines are identical and each job $i$ may be processed in its minimum processing time. The corresponding $LB$ is thus

$$LB = \max\left\{\left\lceil \frac{1}{m}\sum_{i\in[n]} p_i^{\min}\right\rceil; \max_{i\in[n]}\{p_i^{\min}\}\right\}, \tag{4.2}$$

where $p_i^{\min} = \min_{j\in[m]}\{p_{ij}\}$. This lower bound can be further improved by adding to the LP relaxation of the original problem the non-overlapping constraints introduced by Lawler and Labetoulle [110] for the scheduling problem $R|\,pmtn\,|C_{\max}$. These constraints are represented in (4.1) by the second set of inequalities (at most $n$). Here, $p_i^{\max} = \max_{j\in[m]}\{p_{ij} < \infty\}$. Additionally, the new constraints ensure that no fractional job is processed in parallel. The proof of the property that only jobs $i$ for which $p_i^{\max} > \frac{1}{m}\sum_{k\in[n]} p_k^{\min}$ have to be considered in the extended LP relaxation we give later in Section 5.1.5. The fact that the additional constraints ensure that no preempted job is processed simultaneously is proved indirectly [110]. In the following, $\mathbf{x} \in \mathbb{R}^{nm}_{\ge 0}$ denotes a vector of assignment variables for (4.1) whereas $\bar{\mathbf{x}}$ denotes the solution to (4.1).

**MIP subproblem.** We describe now the main idea behind this heuristic approach. From the optimum solution $\bar{\mathbf{x}}$ of the LP relaxation given in (4.1), a list $L$ of assignment variables $x_{ij}$ is built. All variables $x_{ij}$ with positive values in the solution $\bar{\mathbf{x}}$ are placed at the beginning of the list. Afterward, the rest of the assignment variables is ordered according to a certain

**Algorithm 11** PARTIAL-ENUMERATION-VER1-SCHEDULING(**P**)

```
Input: processing times matrix P
Output: integer solution $\bar{\mathbf{x}}$
1: $LB$ := COMPUTE-LOWER-BOUND(P);
2: compute optimal solution $\bar{\mathbf{x}}$ to extended relaxation LP(P, $LB$);
3: if $\bar{\mathbf{x}} \notin \{0,1\}^{mn}$ then
4:     $S := \{x_{ij} \,|\, \bar{x}_{ij} > 0, i \in [n], j \in [m]\}$;
5:     $\bar{\mathbf{x}}$ := BRANCH-AND-BOUND(P, $LB$, $S$);
6: end if
7: return integer solution $\bar{\mathbf{x}}$;
```

criterion and saved to the list after the positive variables. If there are $k$ assignment variables with positive values in the solution $\bar{\mathbf{x}}$, a new MIP problem is constructed with $s \geq k$ assignment variables from the list $L$. Thus, the MIP subproblem to be solved has $mn - s$ binary variables less than the original MIP.

The methodology used by Mokotoff and Jimeno is similar to the two-step approaches of Hariri and Potts [83], and Potts [142]. However, there is one essential difference. While in [83, 142] a partial schedule is built by the solution to the not extended LP relaxation, here only the LP solution data is used to decide which variables should be eliminated from the MIP formulation (cf. Section 2.1.1).

Two relevant issues have to be decided on when designing the algorithm. We need to specify, firstly, the ordering criterion to the list of the subset of non-basic variables, and secondly, the number of binary variables $s$ which shall be kept in the new MIP subproblem. The criterion chosen to list and sort out the non-basic assignment variables leads to different heuristics. Following [126] we consider here two criteria:

(1) sorting out the variables in non-decreasing order according to the simplex reduced costs, and

(2) sorting out the variables in non-increasing order according to the Davis and Jaffe's efficiency index.

The efficiency index defined by Davis and Jaffe [27] was already presented in the previous section. Here, for the convenience, we repeat its definition. The efficiency index $ef_{ij}$ of job $i$ on machine $j$ is equal to the ratio $\frac{\min_{k\in[m]}\{p_{ij}\}}{p_{ij}}$.

The number of variables to be kept could be fixed in different ways. It results in more than one heuristic. One simple way is to fix the value of $s$ as the maximum number of assignment variables for which the MIP formulation is able to be solved within a reasonable amount of computer operational memory and time. By setting the value of parameter $s$ to a value greater than or equal to the number of basic variables, we can guarantee that the MIP subproblem always has a feasible solution, however, not necessarily optimal.

**The algorithms.** We start with the most simple of the heuristics which structure shows Algorithm 11. It consists in solving the extended LP relaxation of the original problem as

**Algorithm 12** PARTIAL-ENUMERATION-VER2-SCHEDULING(**P**)

```
Input: processing times matrix $\mathbf{P}$
Output: integer solution $\bar{\mathbf{x}}$
1: $LB$ := COMPUTE-LOWER-BOUND($\mathbf{P}$);
2: compute optimal solution $\bar{\mathbf{x}}$ to extended relaxation LP($\mathbf{P}, LB$);
3: if $\bar{\mathbf{x}} \notin \{0, 1\}^{mn}$ then
4:     $L$ := LIST($\bar{\mathbf{x}}$);
5:     compute parameter $s$;
6:     $S$ := SELECT($L, s$);
7:     $\bar{\mathbf{x}}$ := BRANCH-AND-BOUND($\mathbf{P}, LB, S$);
8: end if
9: return integer solution $\bar{\mathbf{x}}$;
```

defined in (4.1) and only retains the assignment variables with positive value in the LP solution $\bar{\mathbf{x}}$. The assignment variables with zero value in solution $\bar{\mathbf{x}}$ (i.e., non-basic variables) are eliminated from the formulation. The new MIP subproblem is then solved with branch-and-bound algorithm, and the schedule is built according to the vector of the obtained solution. In Algorithm 11 the lower bound $LB$ is computed according to (4.2).

Algorithm 12, which improves the previous heuristic, uses also the solution $\bar{\mathbf{x}}$ to the extended LP relaxation. However, it keeps more assignment variables than only those with a positive value in the LP solution. List $L$ of all assignment variables is created by function LIST following one of the rules described in the previous paragraph. To select the assignment variables to the list $S$, we make use of the parameter $s$ which is defined to be larger than $k$, where $k$ is the number of assignment variables with positive value in the LP solution $\bar{\mathbf{x}}$. A new MIP subproblem is constructed, where $nm - s$ assignment variables are eliminated. Here, to choose the variables for the subproblem, we use function SELECT. The MIP subproblem is then solved and the obtained solution is built into the schedule.

The parameter $s$ can be set in one of the following ways [126]:

- a percentage $r$ of the size of the problem instance, i.e., $s = \lceil r\,n\,m \rceil$,
- a surplus percentage of $k$, i.e., $s = \lceil (1 + r)\,k \rceil$, or
- as the addition of $k$ plus a percentage of the number of assignment variables with zero value in the LP solution, i.e., $s = k + \lceil r\,(nm - k) \rceil$.

Here, $r \in [0, 1]$. To guarantee a minimum number of assignment variables in the MIP subproblem, the parameter $s$ could be restricted to $s \geq k + m + \lceil r\,n \rceil$.

The structure of the last heuristic which uses the ideas from this section is given by Algorithm 13. This approach is an iterative algorithm which uses in each iteration a different subset of assignment variables and during this iterative process saves the best solution found so far. The first $k$ assignment variables from the ordered list $L$ are always included in the MIP subproblem. The remainder of the $s - k$ variables to be retained are not the same in each iteration. When a variable included in the MIP subproblem, with a value of zero in the

LP solution, has a value of zero in that MIP subproblem solution, it is removed from the list of variables.

When there is no assignment variable with a value of zero in the MIP subproblem and with zero value in the LP solution, then the algorithm stops. We can also add other stopping criteria, e.g., a pre-fixed maximum number of iterations, or a pre-fixed maximum number of variables removed from the list.

**Algorithm 13** PARTIAL-ENUMERATION-VER3-SCHEDULING(**P**)

```
Input: processing times matrix $\mathbf{P}$
Output: integer solution $\bar{\mathbf{x}}$
 1: $LB$ := COMPUTE-LOWER-BOUND($\mathbf{P}$);
 2: $UB := m\,LB$;
 3: compute optimal solution $\bar{\mathbf{x}}$ to extended relaxation LP($\mathbf{P}, LB$);
 4: if $\bar{\mathbf{x}} \notin \{0,1\}^{mn}$ then
 5:     $L$ := LIST($\mathbf{x}$);
 6:     compute parameter $s$;
 7:     while stopping criterion not met do
 8:         $S$ := SELECT($L, s$);
 9:         $(\hat{\mathbf{x}}, T)$ := BRANCH-AND-BOUND($\mathbf{P}, LB, S$);
10:         if $T < UB$ then
11:             $UB := T$;
12:         end if
13:         $L := L \setminus \{x_{ij} \,|\, \bar{x}_{ij} = 0 \wedge \hat{x}_{ij} = 0\}$;
14:     end while
15: else
16:     return integer solution $\bar{\mathbf{x}}$;
17: end if
18: return integer solution $\hat{\mathbf{x}}$;
```

## 4.3 Large Neighborhood Improvement Procedures

The aim of this section is to present an approximation algorithm for the $R||C_{\max}$ problem based on a large neighborhood improvement procedure designed by Sourd [160]. In contrast to already discussed methods, it is based upon a partial and heuristic exploration of the search tree, which is used not only to build a solution but also to improve it by using a *post-optimization* procedure. The main part of this technique constitutes a partial exploration of an enumeration tree of the problem which we describe in details in the following subsection. Next, we show how this branch-and-greed based exploration procedure can further be speed up in order to provide quickly good solutions. Finally, we present the approximation heuristic for the unrelated scheduling problem proposed by Sourd which builds upon ideas from this section.

### 4.3.1 Branch-and-Greed Scheme

To begin with we present a general scheme called *branch-and-greed* which is used to explore partially an enumeration tree of the problem. It is based both on a branching scheme and on a greedy heuristic. Developing efficient heuristics to explore an enumeration tree is also of high interest in constraint programming [117]. We refer to Harvey and Ginsberg [78], Meseguer [122], Sourd [160], and Walsh [171] for more details on branch-and-greed tree exploration methods. For a good survey on very large-scale neighborhood search techniques we suggest the paper by Ahuja et al. [2].

**Enumeration tree.** We consider a combinatorial optimization problem $\Pi$. Let $S$ be the set of all feasible solutions of $\Pi$. We assume that $S$ is finite. Let $f$ be a real function on $S$. A minimization problem is to find a feasible solution $x \in S$, such that:

$$f(x) = \min_{x \in S} f(x).$$

Set $S$ of feasible solutions can be partitioned into subsets. These subsets can also be partitioned into subsets and so on, until all subsets have only one element. These partitions can be represented by a diagram called an *enumeration tree*. An enumeration tree $\mathcal{T}$ of $S$ possesses the following properties:

- each node $v$ corresponds to a subset $S(v)$ of $S$,
- the root node corresponds to $S$,
- the leaves correspond to the singletons of $S$ (single feasible solutions), and
- each node $v$ which is not a leaf has $k(v) \in (1, a]$ direct descendants whose corresponding subsets of $S$ make a partition of $S(v)$ into $a \geq 2$ subsets.

The subtree of $\mathcal{T}$ with root node $v$ is denoted by $\mathcal{T}(v)$, and $\mathcal{D}(v)$ is the set of the direct descendants of $v$.

**Exploration of enumeration tree.** We describe now a heuristic exploration of the enumeration tree presented by Sourd. Recall first that a leaf vertex of the enumeration tree corresponds to a feasible solution of problem $\Pi$. We assume that we have a heuristic $H_0$ to find a good solution in $S$. This heuristic is based on a function $\mathcal{H}$ that *a priori* evaluates the direct descendants of a node of the enumeration tree. If $v$ and $w$ are two direct descendants of the same node and $\mathcal{H}(v) < \mathcal{H}(w)$, then we say that a subtree $\mathcal{T}(v)$ is a priori more likely to contain an optimum than $\mathcal{T}(w)$. At each stage, starting at the root of the tree, $H_0$ moves onto the direct descendant that minimizes $\mathcal{H}$.

Let us note that the function $\mathcal{H}_0 : v \mapsto f(H_0(\mathcal{T}((v)))$ can now be considered as an a priori evaluation of the descendants of a node. Indeed, if $v$ and $w$ are two direct descendants of the same node and if $f(H_0(\mathcal{T}(v))) < f(H_0(\mathcal{T}(w)))$, we can imagine that the solutions in $\mathcal{T}(v)$ should be better than those in $\mathcal{T}(w)$. Thus, we obtain a new heuristic, $H_1$. At each stage, $H_1$ looks at all direct descendants of a given node, applies $H_0$ to each of the corresponding

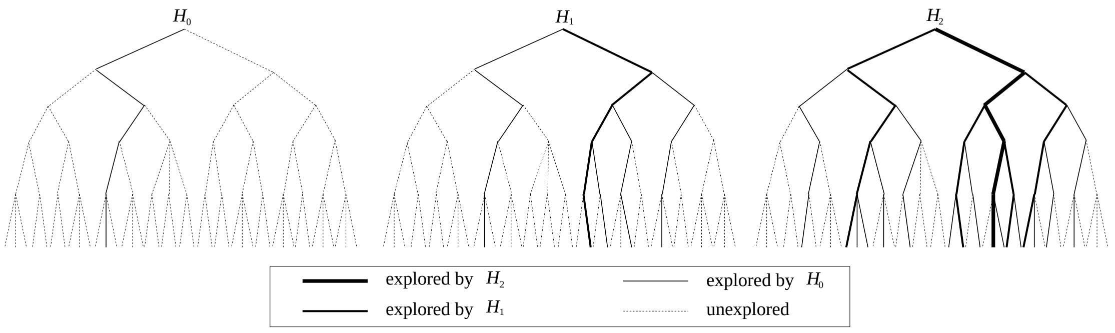


Figure 4.1: Branches of a search tree by $H_0$, $H_1$, and $H_3$ (based on [160]).

subtrees, evaluates the objective function $f$ at each leaf obtained in this $H_0$ search, and picks the one descendant with the lowest evaluation value.

The previous scheme can be generalized to create a heuristic $H_k$ by simply swapping $H_1$ for $H_k$, and $H_0$ for $H_{k-1}$ in the description of $H_1$. Hence, Algorithm 14 representing heuristic $H_k$ can be defined recursively with the a priori evaluation function $\mathcal{H}_{k-1} : v \mapsto f(H_{k-1}(\mathcal{T}(v)))$.

**Algorithm 14** $H_k(\mathcal{T})$

```
Input: enumeration tree $\mathcal{T}$
Output: single solution in $S(v)$
1: let $v$ be the root of $\mathcal{T}$;
2: if $v$ is a leaf then
3:     return the single solution contained in $S(v)$;
4: else
5:     let $w \in \mathcal{D}(v)$ that minimizes $\mathcal{H}_{k-1}$;
6:     return $H_k(\mathcal{T}(w))$;
7: end if
```

The definition of this algorithm shows that $H_k(\mathcal{T})$ finds a $k$-path from the root to one leaf. Each node $v$ on this $k$-path is the starting point of $(k-1)$-paths, and it corresponds to the partial exploration of the subtrees non-selected by $H_k$. Figure 4.1 shows the branches of a tree $\mathcal{T}$ explored by $H_0(\mathcal{T})$, $H_1(\mathcal{T})$ and $H_2(\mathcal{T})$. The 0, 1 and 2-paths in $\mathcal{T}$ are visible.

Note that all vertices explored by $H_k(\mathcal{T})$ are obviously explored by $H_{k+1}(\mathcal{T})$, thus $H_{k+1}$ gives a result at least as good as $H_k$. At each step of $H_k$, the subtree selected is first explored by $H_{k-1}$ when computing the minimum of $\mathcal{H}_{k-1}$, and then by $H_k$ when the algorithm descends to the selected node to build the end of the $k$-path. Hence, we note that some nodes are visited many times.

In order to get an upper bound on the number of nodes of the enumeration tree explored by $H_k$, we consider $(a, b)$-complete trees, i.e., type of trees where each node that is not a leaf has exactly $a$ descendants and the depth of each leaf is $b$.

Let $h_k(a, b)$ be the number of nodes in an $(a, b)$-complete tree visited by $H_k$. Each subtree of the root of an $(a, b)$-complete tree is $(a, b-1)$-complete, the selected subtree is explored by $H_k$, and $a-1$ other subtrees are just explored by $H_{k-1}$. Thus, following the analysis in [160] we have:

$$h_k(a,b) = 1 + h_k(a,b-1) + (a-1)h_{k-1}(a,b-1).$$

Obviously, $h_0(a,b) = 1+b$, and $h_k(a,0) = 1$. By induction we get that

$$h_k(a,b) - h_{k-1}(a,b) = (a-1)^k \binom{b+1}{k+1}.$$

And finally

$$h_k(a,b) = \sum_{i=0}^{k} (a-1)^i \binom{b+1}{i+1} = O(a^k b^{k+1}).$$

Note that for a fixed $k$ the number of vertices explored by $H_k$ is polynomially bounded. When $k = b$, we get $h_b(a,b) = \frac{a^{b+1}-1}{a-1}$, which is exactly the number of vertices of an $(a,b)$-complete tree. More generally, for any arbitrary tree, $H_k(\mathcal{T})$ explores the whole tree and therefore gives the optimal solution of $S$ if and only if $k$ is equal to or greater than the depth of $\mathcal{T}$. Observe that the exploration of the enumeration tree $\mathcal{T}$ makes sense only with heuristics $H_k(\mathcal{T})$ for $k \le b$.

Furthermore, observe that at each step of the algorithm, the selected subtree is explored twice, firstly by $H_{k-1}$ to minimize $\mathcal{H}_k$, and secondly by $H_k$ once it is selected. Therefore, in order to evaluate the time complexity of $H_k$, we have to replace $a-1$ by $a$ in the recurrent relation. Though, the worst-case time complexity is also $O(a^k b^{k+1})$.

**Improving the exploration.** We have just seen that for $k \ge 1$ the time complexity of $H_k$ depends greatly on $|\mathcal{D}(v)|$. Sourd gives two ideas to remove some nodes from $\mathcal{D}(v)$ that will be left unexplored. In both cases we assume that a solution $s$ has already been found since the beginning of the search. And at each step we remove from $\mathcal{D}(v)$ the nodes that either can be proved not to lead or cannot a priori lead to a better solution than $s$.

The previous paragraph shows that $H_k$ makes a partial exploration of the enumeration tree. Branch-and-bound concepts (see Section 5.1.3) can easily be used in this exploration. At each node $w$ in $\mathcal{T}$, a lower bound $LB = \min_{x \in S(w)} f(x)$ is computed. If we know a solution $s$ of $S$ such that

$$f(s) \le LB \le \min_{x \in S(w)} f(x),$$

then the optimal solution of $S$ cannot be in $S(w)$. Therefore, it is not necessary to explore $S(w)$, and thus $S(w)$ can be fathomed.

Following the discussion from [160] assume that $H_k$ is at node $v$ that is not a root of the initial search tree. Since $k \ge 1$, some leaves have already been explored. Let $s$ be the best solution found so far. Because $H_k$ chooses at each step the subtree where it has found the best solution when exploring each subtree with $H_{k-1}$, $s$ is necessary in $\mathcal{T}(v)$ and there is a node $w_s \in \mathcal{D}(v)$ such that $H_{k-1}(\mathcal{T}(w_s)) = s$. If $w \in \mathcal{D}(v)$ and $S(w)$ can be fathomed, then

$$\mathcal{H}_{k-1}(w) \ge \min_{x \in S(w)} f(x) \ge f(s) = \mathcal{H}_{k-1}(w_s)$$

and so

$$\min_{w \in \mathcal{D}(v)} \mathcal{H}_{k-1}(w) = \min_{w \in \mathcal{D}(v) \setminus \mathcal{F}(v)} \mathcal{H}_{k-1}(w)$$

where $\mathcal{F}(v) = \{w \in \mathcal{D}(v) \text{ such that } S(w) \text{ can be fathomed}\}$.

Hence, subtrees that are proved by the a priori function not to contain a better solution that the best found so far have not to be explored when computing $\min_{w \in \mathcal{D}(v)} \mathcal{H}_{k-1}(w)$. If our a priori function is good, then we can claim that some subtrees do not a priori contain good solutions. Thus, we can replace $\mathcal{D}(v)$ by

$$\overline{\mathcal{D}}(v) = \{w \in \mathcal{D}(v) \text{ such that } \mathcal{H}(w) \leq B(v)\}$$

where $B(v)$ is a problem-dependent value. Setting such a limit to the number of explored subtrees improves the efficiency of $H_k$ by decreasing remarkably the computation time with little effects on the quality of the obtained solution [160].

### 4.3.2 Post-optimization Procedure

The previous subsection shows that when $a$ and $b$ are large, $H_k$ cannot run within a reasonable time for $k$ being greater than 3 or 4. For computational results showing this unfavorable behavior see, e.g., [160, 171]. Thus, a simple partial search in an enumeration tree $H_k$ is often not sufficient for many problems to provide quickly good solutions. We present now several ideas given by Sourd to build improvement procedures based on a branch-and-greed approach. He called them *post-optimization* procedures.

Let $x$ be an element from $S$. Consider a set $\mathcal{V}(x)$ that contains $x$. We assume that we can find an enumeration tree $\mathcal{T}'$ and an a priori evaluation function $\mathcal{H}$ associated with $\mathcal{V}(x)$. Provided with this, we are able to build the algorithms $H_k$ based on $\mathcal{H}$. Therefore, for any integer $k$, $H_k(\mathcal{T}')$ is a *good* solution of $\mathcal{V}(x)$.

We consider $\mathcal{V}(x)$ as a *neighborhood* of $x$. The definition of such a neighborhood is problem-dependent (see, e.g., [2]). However, in most cases, $x$ can be viewed as a set of variables with assigned values. By relaxing some of the assignment constraints we usually get an efficient neighborhood for which an associated enumeration tree and an a priori evaluation function can easily be found.

In contrast to usual local improvement procedures (like, e.g., taboo search), $|\mathcal{V}(x)|$ may be exponentially large. Therefore, it is not possible to compute $f(v)$ for all $v \in \mathcal{V}(x)$. In order to solve this unfavorable situation, a heuristic $H_k$ is called to search for a better solution in this neighborhood. Algorithm 15 states the post-optimization procedure. Here, $x_0$ stands for the initial solution (it must be an element of $S$, e.g., $x_0 = H_{k_0}(\mathcal{T})$), $x^*$ is the best solution found so far, $l$ is the iteration counter, and $k$ represents the level of post-optimization.

The efficiency of post-optimization greatly depends on the choice of $\mathcal{V}(x)$. It can be chosen in a deterministic or randomized way. In both cases the neighborhood must be redefined with the beginning of each iteration – step (6). Symbol $\propto$ describes the relation between a new solution $x_{l+1}$ and the currently best solution $x^*$. If $f(x_{l+1}) < f(x^*)$, then $x_{l+1} \propto x^*$; and when $f(x_{l+1}) = f(x^*)$, other criteria may be used to decide whether $x_{l+1} \propto x^*$.

In [160] the following stopping criteria for the inner `repeat`-loop have been proposed:

**Algorithm 15** POST-OPTIMIZATION($x_0, k_0$)

```
Input: initial solution $x_0$
       initial level of post-optimization $k_0$
Output: better solution $x^*$
 1: $x^* := x_0$;
 2: $k := k_0$;
 3: $l := 0$;
 4: repeat
 5:     repeat
 6:         choose a neighborhood $\mathcal{V}(x^*)$ of $x^*$;
 7:         find $\mathcal{T}'$ associated with $\mathcal{V}(x^*)$;
 8:         $x_{l+1} := H_k(\mathcal{T}')$;
 9:         if $x_{l+1} \propto x^*$ then
10:             $x^* := x_{l+1}$;
11:         end if
12:         $l := l + 1$;
13:     until stopping criterion is met
14:     $k := k + 1$;
15: until stopping criterion is met
```

- $f(x^*)$ is equal to a lower bound of the problem,
- $l$ is equal to the maximum number of allowed iterations,
- $x^*$ has not been improved for a given number of iterations, or
- all the (deterministic) neighborhoods of $x^*$ have been tried without improvement.

The outer `repeat`-loop is generally stopped when $f(x^*)$ is equal to a lower bound of the problem, or $k$ is equal to the maximum number of allowed iterations. The reason why $k$ is increased within the post-optimization is motivated by the lower bound, i.e., the better $x^*$, the faster $H_k$. For example, at the beginning $H_1$ is usually sufficient to find better solutions than $x_0$. At the end, $H_{k_f}$ with $k_f \geq 2$ (typically $k_f = 5$ by [160]) is required to improve a good $x^*$. If we run $H_{k_f}$ already at the beginning of the post-optimization process, the solutions which are found with each iteration are better but computation time is much longer.

### 4.3.3 Approximation Algorithm for $R||C_{\max}$

We are now ready to present an approximation algorithm for the $R||C_{\max}$ problem. It has been designed by Sourd [160] and uses the post-optimization procedure discussed above. Computational experiments presented by the author of this method show that its efficiency is equivalent to that of the best local search heuristic for the unrelated scheduling problem published so far, i.e., to the taboo search method of Piersma and van Dijk [138]. In Chapter 6 we show how this heuristic performs in comparison with other techniques, especially with the combinatorial approximation algorithm, APPROXIMATION-UNSPLITTABLE-TRUEMPER.

In the following $S$ denotes the set of solutions to $R||C_{\max}$, any $x \in S$ is a feasible schedule, and $f(x)$ represents the makespan of schedule $x$.

**The branch-and-greed approach.** In our problem each job has to be assigned without preemption to one of $m$ machines. In this setting a problem (or a subproblem) is divided into $m$ subproblems: we branch by selecting a non-assigned job and assigning it to one of $m$ machines. This results in a $(m, n)$-complete enumeration tree. The jobs are sorted non-increasingly according to the minimal processing time that is according to $p_i^{\min} = \min_{j \in [m]}\{p_{ij}\}$ for each $i \in [n]$. Each node of the tree corresponds to a partial schedule, i.e., some jobs are already assigned what defines the partial completion time $\delta_j$ (load) for each machine $j$. If we want to assign job $i$, we define an a priori evaluation function $\mathcal{H}$ as $\mathcal{H}(j) = \delta_j + p_{ij}$. Hence, $H_0$ is a list scheduling algorithm (see Section 4.1) based on the earliest completion time (ECT) rule. Therefore, we denote by $\mathsf{ECT}_k$ the $H_k$ heuristic for the level of post-optimization $k$. We refer to [160] for a discussion on other types of evaluation function $\mathcal{H}$.

**Example 4.3.1** *We give now a simple example which shows how the enumeration tree for some small instance of the $R||C_{\max}$ problem is explored. We consider here a problem instance with $n = 4$ jobs and $m = 2$ machines. The processing times are as follows:*

| | *Job $J_1$* | *Job $J_2$* | *Job $J_3$* | *Job $J_4$* |
|---|---|---|---|---|
| *Machine $M_1$* | *45* | *93* | *67* | *100* |
| *Machine $M_2$* | *99* | *96* | *29* | *65* |

*The jobs are processed sequentially according to the non-increasing order of minimal processing times. This results here with the following sequence of jobs: $J_2, J_4, J_1$ and $J_3$. To search the tree, we use heuristic $H_1$ with the evaluation function $\mathcal{H}$ defined as above. Figure 4.2 shows the branches of the search tree successively explored by $H_1$. The nodes of the tree represent the machines on each decision level. Each edge describes the assignment of a given job. Observe that each level of the tree corresponds to one job. The thickness of the edges indicates the heuristic, $H_1$ or $H_0$, which has traversed it already. The numbers below the leaves of the tree represent the lengths (makespans) of corresponding assignments. Note that every path from the root of the tree to any of its leaves defines a feasible assignment. The best solution found here by the heuristic $H_1$ has the length of 138.*

In the following we describe how the subsets of $S$ are explored. First, let $p_i^{\min} = \min_{j \in [m]}\{p_{ij}\}$ for every job $i \in [n]$. We denote by $S_\lambda$ the set of solutions to $R||C_{\max}$ for which $\sum_{j \in [m]} \delta_j \leq \lambda + \sum_{i \in [n]} p_i^{\min}$. Sourd [160] observed that a minimal solution to $S$ is often in a set $S_\lambda$ with a small $\lambda$. Therefore, we explore first these subsets of $S$. The search tree for $S_\lambda$ is build as it was described at the beginning of the paragraph. Branches that lead to solutions with $\sum_{j \in [m]} \delta_j > \lambda + \sum_{i \in [n]} p_i^{\min}$ are simply cut off, i.e., they are not explored.

Before we give the structure of the algorithm, we present the idea of post-optimization which is used here. An element $x \in S$ is defined by a set $\Delta(x)$ of pairs $(i, j)$, where one pair $(i, j)$ means that job $i$ is assigned to machine $j$. A subset $\Delta'(x)$ emerges from $\Delta(x)$ by

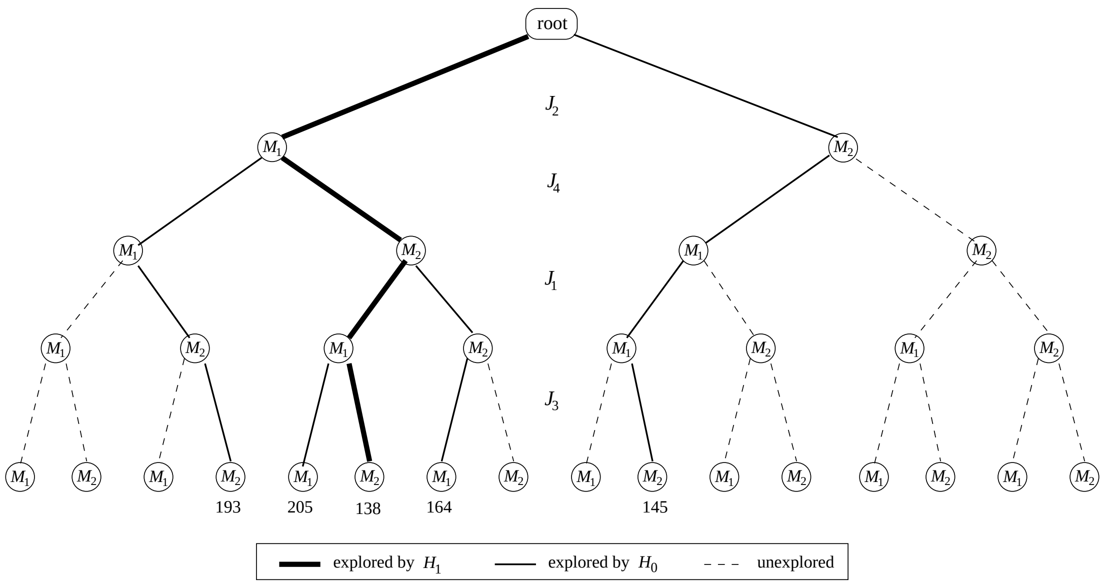


Figure 4.2: Successive exploration of the $(2,4)$-complete enumeration tree from Example 4.3.1 by heuristic $H_1$ ($\mathsf{ECT}_1$).

removing some pairs, i.e., by canceling assignments of some jobs. Then, the neighborhood of $x$ is defined as

$$\mathcal{V}(x) = \Delta'(x) \cup \{\overline{J} \times \{j \in [m]\}\}$$

where $\overline{J}$ is the set of not assigned jobs.

**Algorithm 16** Branch-And-Greed-Post-Optimization

```
1: compute some initial solution in $S_0$ with $\mathsf{ECT}_1$;
2: for $\lambda := 0$ to 10 do
3:     500 iterations of post-optimization with $\mathsf{ECT}_1$ in $S_\lambda$;
4: end for
5: for $k := 1$ to 5 do
6:     500 iterations of post-optimization with $\mathsf{ECT}_k$ in $S$;
7: end for
```

The structure of the post-optimization heuristic is given in Algorithm 16. At each step of post-optimization, the assignments of 10 jobs are canceled (we consider test instances with at least 10 jobs and 2 machines). The jobs are randomly selected on two machines. These two machines are also randomly selected, but the completion time, $\delta_j$, of one of the two machines must be maximal. If there are less than 10 jobs assigned to the two selected machines, the assignments of all jobs assigned to these machines are canceled. These jobs are then re-assigned using the heuristic $H_k$ described above. The relation $\propto$ that compares two schedules is defined in the following way. A schedule is better than other one if its completion time (makespan) is less. If two schedules have the same completion time $T$, the better is the one for which less machines complete at $T$. In case of equality, better is the one for which less machines complete at $T - 1$, and eventually the one that minimizes the total completion time, $\sum_{j\in[m]} \delta_j$, computed over all machines.

# Exact Solution Methods for $R||C_{\max}$

In this chapter we present several techniques for computing exact solutions to the unrelated scheduling problem with the minimization of makespan as the objective function. In particular, we describe here two *branch-and-bound* based methods developed for solving the $R||C_{\max}$ problem. Both of them are known to be powerful frameworks to solve hard combinatorial optimization problems. In the first section we discuss in detail a cutting plane algorithm designed by Mokotoff and Chrétienne [125] for the $R||C_{\max}$ problem. In addition to this we present the basic ideas behind a *branch-and-cut* scheme. This algorithmic technique uses cutting planes embedded in the branch-and-bound scheme to tighten the solution space of the LP relaxation of the scheduling problem. It is based on row generation which significantly improves the convergence of the optimization process. In the second section we present a new *branch-and-price* algorithm with heuristic extensions which we have developed for the $R||C_{\max}$ problem. Our approach combines column generation with the branch-and-bound scheme. The main idea of this technique is similar to that of branch-and-cut except that the branch-and-price algorithm focuses on column generation as a way to speed up the computations rather than on row generation. Common to both methods is the idea how the integer solution to the original problem is generated. Both algorithms apply branching to traverse the solution space of the problem given by the enumeration tree in order to find an optimal integer solution.

## 5.1 Cutting Planes and Branch-and-Cut Method

Cutting plane algorithms have turned out to be practically successful tools in combinatorial optimization, in particular when they are embedded in a branch-and-bound framework. Even though the implementation process of most branch-and-cut algorithms is rather more complicated in comparison to the implementation of many purely combinatorial algorithms, the high quality of solutions which they deliver (see Section 6.3.2) is a strong argument to overcome these technical difficulties. The aim of this section is to present a cutting plane scheme which combined with some heuristics leads to an exact algorithm for the $R||C_{\max}$

problem. We begin our discussion with an introduction into cutting plane algorithms. Here, we address carefully the main ideas behind every algorithm using cutting planes. Additionally, we give a short introduction into a branch-and-bound approach, and present several preliminary information about the branch-and-cut scheme which, when applied to the optimization process of (mixed) integer programs, significantly improves its convergence. Finally, we describe the cutting plane algorithm by Mokotoff and Chrétienne [125] for the $R||C_{\max}$ scheduling problem

## 5.1.1 Cutting Planes

Suppose we have to solve a linear optimization problem which set of constraints is too large to be represented explicitly in a computer memory, or too large to be handled by an LP-solver. In such a case we can still attempt to solve the problem using the following approach. We start with a small subset of constraints and compute an optimum solution subject to these constraints. Afterward, we check if any of the constraints which are not in the current LP is not satisfied. If such constraints are present, we add one or more of them to the current LP and resolve it. Otherwise, the current optimum solution also solves the original problem. This is the basic principle of the *cutting plane* approach. The name originates from the fact that the constraints added to the current LP cut off the current solution because it is infeasible for the original problem.

The essential ingredients of the solution method for integer combinatorial optimization problems which uses cutting planes are valid inequalities.

**Definition 5.1.1 (Jünger at al. [94])** *Given an integer programming formulation* $\min\{\mathbf{c}^T\mathbf{x} \mid A\mathbf{x} \le \mathbf{b}, \mathbf{x} \in \mathbb{Z}^n\}$ *of a combinatorial optimization problem, an inequality* $\mathbf{f}^T\mathbf{x} \le f_0$ *is called valid, if* $\mathbf{f}^T\bar{\mathbf{x}} \le f_0$ *for all feasible solutions* $\bar{\mathbf{x}} \in \{\mathbf{x} \mid A\mathbf{x} \le \mathbf{b}, \mathbf{x} \in \mathbb{Z}^n\}$.

If we know a class of valid inequalities, we have to be able to check if a constraint of this class is violated by a current solution, i.e., we must solve the following problem

**Definition 5.1.2 (General Separation Problem)** *Given a class of valid inequalities for a combinatorial optimization problem, and a vector* $\mathbf{y} \in \mathbb{R}^n$*, either prove that* $\mathbf{y}$ *satisfies all inequalities of this class, or find an inequality of this class which is violated by* $\mathbf{y}$.

An algorithm solving the general separation problem is called an *exact separation algorithm*. Unfortunately, exact algorithms are rarely known for classes of valid inequalities, or it can even be shown that the separation problem for a certain class of inequalities is $\mathcal{NP}$-hard. In this case we have to apply a *heuristic separation algorithm* which may find violated inequalities. However, if it fails, it is not guaranteed that no constraint of the class is violated.

## 5.1.2 Generic Cutting Plane Algorithm

We state now a generic cutting plane algorithm for solving an integer combinatorial optimization problem defined as $\min\{\mathbf{c}^T\mathbf{x} \mid A\mathbf{x} \le \mathbf{b}, \mathbf{x} \in \mathbb{Z}^n\}$.

**Algorithm 17** GENERIC-CUTTING-PLANE-ALGORITHM($A, \mathbf{b}$)

```
Input: constraint matrix $A$
       vector $\mathbf{b}$
Output: integer solution $\bar{\mathbf{x}}$
1: let $A' := \binom{I}{-I}$ and $\mathbf{b}' := \binom{1}{0}$;
2: compute an initial optimal solution $\bar{\mathbf{x}}$ to $\min\{\mathbf{c}^T\mathbf{x} \mid A'\mathbf{x} \leq \mathbf{b}', \mathbf{x} \in \mathbb{R}^n\}$;
3: while $\bar{\mathbf{x}}$ is not a feasible solution to the original problem do
4:     generate cutting plane $(\mathbf{f}, f_0)$, where $\mathbf{f} \in \mathbb{R}^n$, $f_0 \in \mathbb{R}$ such that $\mathbf{f}^T\bar{\mathbf{x}} > f_0$ and $\mathbf{f}^T\mathbf{y} \leq f_0$
           for all $\mathbf{y} \in \{\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \in \mathbb{Z}^n\}$;
5:     add inequality $\mathbf{f}^T\mathbf{x} \leq f_0$ to the constraint system $(A', \mathbf{b}')$;
6:     compute an optimal solution $\bar{\mathbf{x}}$ to $\min\{\mathbf{c}^T\mathbf{x} \mid A'\mathbf{x} \leq \mathbf{b}', \mathbf{x} \in \mathbb{R}^n\}$;
7: end while
8: return integral solution $\bar{\mathbf{x}}$;
```

Algorithm 17 takes into account that the number of inequalities in the integer program can be too large for explicit representation. In such a case, one can start with a trivial constraint system $\mathbf{0} \leq \mathbf{x} \leq \mathbf{1}$. The inequalities $A\mathbf{x} \leq \mathbf{b}$ are valid inequalities that can serve as cutting planes. However, an integral solution coming up in the course of the algorithm is only the incidence vector of a feasible solution of the optimization problem if no inequality of the system $A\mathbf{x} \leq \mathbf{b}$ is violated. Furthermore, the algorithm is only correct if all LPs in step (2) and (6) can be solved, and that in step (4) a cutting plane can be generated, and finally, that the algorithm terminates after finite number of iterations.

**General-purpose cutting planes.** We now address the question of how further cutting planes can be found if the optimum solution to the LP $\min\{\mathbf{c}^T\mathbf{x} \mid A'\mathbf{x} \leq \mathbf{b}'\}$ is not integral but no inequality of the system $A\mathbf{x} \leq \mathbf{b}$ is violated.

First of all, we can use classes of cutting planes that can be applied to any integer or mixed integer program. We call such cutting planes *general purpose* cutting planes [94]. They are not problem-specific and can be employed for the solution of every integer optimization problem.

The first cutting plane algorithms for integer and mixed integer programs were introduced by Gomory [66, 67, 68]. He also proved that these algorithms terminate with an optimal solution after a finite number of iterations. Unfortunately, it turned out that in practical experiments the Gomory cutting planes provide very weak cuts leading to numerical problems [94], and only rather small integer problems can be solved to optimality with these methods. Crowder, Johnson, and Padberg [23] use minimal cover inequalities and $(1, k)$-configurations for solving $0-1$ linear problems. These cutting planes are derived from facets of the polytope of the knapsack problem defined by each constraint of the $0-1$ optimization problem. Later, this approach was refined and generalized by Van Roy and Wolsey [147], and Hoffman and Padberg [83]. For mixed $0-1$ linear problems Balas et al. [10] generate cutting planes by a lift-and-project method. The computational results show that these cutting planes outperform Gomory cuts [94]. There are additionally Fenchel cuts for integer programs introduced by Boyd [16].

For solving combinatorial optimization problems, general cuts seem to be of limited use. Successful computational work, what we show later, relies on cutting planes designed for the particular problem. However, a cutting planes algorithm using *problem-specific* cutting planes often has to stop without finding an optimal solution. This can have two different reasons. Firstly, according to the well-know result by Karp and Papadimitriou [98], unless $\mathcal{NP}$ = co-$\mathcal{NP}$, for no $\mathcal{NP}$-hard optimization problem a complete linear description cannot be found. Secondly, even if a big class of facets is known, no efficient algorithm may be available for the solution of the exact separation problem of this class.

One can also design *hybrid* cutting plane algorithms which combine general purpose cutting planes and problem-specific cutting planes in the following way. When generating cutting planes, we first try to separate with a problem-specific, preferably facet defining, inequality. If this fails, we generate a general purpose cutting plane, e.g., a Gomory cut or a lift-and-project cut. Such hybrid algorithms were designed, e.g., by Miliotis et al. [124].

### 5.1.3 Solving to Optimality with Branch-and-Bound Algorithm

The cutting plane approach presented above does not necessarily solve a problem to optimality. We may get stuck at a solution which is not the incidence vector of a feasible solution to the optimization problem. In such a case, we can apply another basic algorithmic technique for solving hard optimization problems called *branch-and-bound*. This method was designed to solve mixed integer programs by Doig and Land [36], and Dakin [24] in the early 1960s.

Branch-and-bound is a divide-and-conquer approach trying to solve the original problem by splitting it into smaller problems for which upper and lower bounds are computed. The crucial part of an efficient branch-and-bound algorithm is the computation of lower bound for these subproblems. Here, the fundamental concept of LP relaxation is used.

Before we discuss the generic branch-and-bound algorithm (see Algorithm 18), we define some terminology. We call a bound *global*, if it is a bound for the original problem, and *local* if it is only valid for a subproblem. By solving the LP relaxation of the current subproblem, we obtain a local lower bound, $LB$, for the objective function value of the original problem. If the solution of the relaxation is feasible for the original problem and has smaller objective value than any feasible solution found so far, it is saved and the global upper bound, $UB$, is updated accordingly.

A branch-and-bound algorithm maintains a list of subproblems, $SP$, of the original problem, which is initialized with the original problem itself. For $0-1$ integer problems, $SP$ is often organized into binary tree. In each major iteration step the algorithm selects a subproblem $sp$ from $SP$, computes a lower bound $LB$ for this subproblem and tries to improve the upper bound $UB$. If $LB$ does exceed $UB$, then the current subproblem $sp$ is fathomed, because its (fractional) solution cannot be better than the best known feasible (integer) solution. Otherwise, we check if the optimal solution to the relaxation of the subproblem is a feasible solution for the original (integer) problem. If it is the case, we have solved the subproblem, and thus it is fathomed.

If the local $LB$ does not exceed the global $UB$ and no feasible (i.e., integer) solution has been found for the current subproblem, we perform a branching step by splitting the current

**Algorithm 18** GENERIC-BRANCH-AND-BOUND-ALGORITHM($A$, $\mathbf{b}$)

```
Input: constraint matrix $A$
       vector $\mathbf{b}$
Output: integer solution $\bar{\mathbf{x}}$
 1: initialize $SP$;
 2: compute global $UB$;
 3: while $SP \neq \emptyset$ do
 4:     take subproblem $sp$ from $SP$;
 5:     solve subproblem $sp$;
 6:     compute local $LB$;
 7:     if $LB \leq UB$ then
 8:         if $sp$ is feasible for the original problem then
 9:             fathom $sp$;
10:             $UB := LB$;
11:         else
12:             perform branching on $sp$;
13:             add new subproblems to $SP$;
14:         end if
15:     else
16:         fathom $sp$;
17:     end if
18: end while
19: return integral solution $\bar{\mathbf{x}}$;
```

subproblem into a collection of at least two new, mutually exclusive subproblems which union of feasible solutions contains all feasible solutions of the current subproblem. There exists a number of various branching rules, see, e.g., [94]. Some of them are presented later in this chapter.

If the list of subproblems $SP$ becomes empty, then the saved feasible solution with the objective value equal to $UB$ can be returned as the optimum solution. Important for the efficiency of a branch-and-bound algorithm is not only the quality of the relaxation technique, but also the quality of the generated feasible solutions, since otherwise the number of generated subproblems becomes rapidly very large.

### 5.1.4 Preliminaries on Branch-and-Cut Approach

We give now some important facts about the branch-and-cut algorithm for solving hard mixed integer optimization problems. The main difference in comparison to the classical branch-and-bound method is the usage of LP relaxations and the employment of problem-specific cutting planes at every node of the enumeration tree. This feature incurs several technicalities that make the design and implementation of branch-and-cut algorithms a non-trivial task.

The first combination of problem-specific valid inequalities and branch-and-bound methods can be found in Miliotis [123] for the traveling salesman problem. How to use the facet defining cutting planes and enhanced automatic cutting plane generation in combination with branch-and-bound was first considered by Grötschel, Jünger, and Reinelt [74] for the linear ordering problem. The term "branch-and-cut" has been introduced by Padberg and Rinaldi [135] for an algorithm solving the traveling salesman problem.

Since a detailed presentation of the branch-and-cut algorithm is beyond the scope of this work, we refer the reader to [94, 136, 151, 172] for good overviews on this method. Nevertheless, in the next subsection, we present some most important features of this method while discussing a cutting plane algorithm for the $R||C_{\max}$ problem given by [125]. Another interesting example of a branch-and-cut algorithm computing exact solutions to the $R||C_{\max}$ scheduling problem was presented by Martello, Soumis, and Toth [120]. They proposed lower bounds for the optimum based on Lagrangian relaxations [151] and some additive techniques using decomposition of the LP relaxation into $m$ independent knapsack problems. They also introduced new cuts which eliminate infeasible disjunctions of the cost function value, and proved that the bounds obtained through such cuts dominate the previous cuts. Finally, they presented a branch-and-bound algorithm which uses these new cuts to tighten the solution space.

### 5.1.5 Cutting Plane Algorithm for $R||C_{\max}$

We present now an exact cutting plane algorithm for the scheduling unrelated parallel machines problem given by Mokotoff and Chrétienne [125]. Their algorithm deals directly with the polyhedral structure of the $R||C_{\max}$ problem. As every cutting plane algorithm, it mainly consists of a constraint generation scheme by means of a separation procedure (cf. Section 5.1.1). This separation procedure checks whether or not the optimum solution of the current LP relaxation is an integral vector. If it is the case, the main algorithm stops. Otherwise some new valid inequalities are added to the current LP polyhedron and the optimization process is restarted.

For the MIP formulation of the $R||C_{\max}$ problem given by (1.2), the solution space $\mathcal{S}$ is defined by $(\mathbf{x}, T)$ such that

$$\begin{array}{rcll} \sum_{i\in[n]} p_{ij}x_{ij} & \leq & T, & \forall\ j \in [m] \\ \sum_{j\in[m]} x_{ij} & = & 1, & \forall\ i \in [n] \\ x_{ij} & \in & \{0,1\}, & \forall\ i \in [n], j \in [m] \\ T & > & 0. & \end{array}$$

In turn, when $\mathbf{x} \in \{0,1\}^{mn}$ is substituted with $\mathbf{x} \in \mathbb{R}^{mn}_{\geq 0}$, we get a polyhedron $\mathcal{P}$ that defines the LP relaxation of $\mathcal{S}$. Following the classical polyhedral combinatorics [130, 151], there exists a finite set of valid linear inequalities $A\mathbf{x} + DT \leq \mathbf{b}$ that defines the convex hull $\text{conv}(\mathcal{S})$ such that

$$\min\{T \mid (\mathbf{x}, T) \in \mathcal{S}\} = \min\{T \mid (\mathbf{x}, T) \in \mathcal{P}, A\mathbf{x} + DT \leq \mathbf{b}\}$$

The initial inequalities of $\mathcal{P}$ together with the valid inequalities $A\mathbf{x} + DT \leq \mathbf{b}$ define the *linear description* of $\mathcal{S}$ (according to classical results of Farkas, Weyl, and Minkowski;

see [151]). However, this set is too large and only a very small part of it is known [98]. Nevertheless, a partial linear description may often provide a powerful tool for the solution of the problem [94]. As indicated in the previous subsections, the key idea of the cutting plane technique is to develop an algorithm that would be capable of identifying a valid inequality (should one exist) which is not satisfied by the solution of the current LP relaxation. When no more cuts can be added, or when the number of iterations reached a given limit but the solution of the last LP is not integral yet, we have to apply a branch-and-bound algorithm to solve the problem. Observe that in this case, the solution value of the last LP relaxation provides an initial lower bound *LB* that may be good enough to efficiently reduce the number of nodes to be explored by the branch-and-bound algorithm applied afterward.

**Valid inequalities for conv($\mathcal{S}$).** Let $T^0 > 0$ and let $\mathcal{S}(T^0)$ denote the subset of points $(\mathbf{x}, T^0)$ which are in $\mathcal{S}$. Furthermore, let $\mathcal{I}$ be a family of valid inequalities for $\text{conv}(\mathcal{S}(T^0))$ and $\mathcal{P}(\mathcal{I})$ the current relaxation of $\text{conv}(\mathcal{S}(T^0))$ defined by the inequalities of $\mathcal{P}$ and the inequalities of $\mathcal{I}$. Let $(\mathbf{x}^0, T^0)$ be a point from $\mathcal{P}(\mathcal{I})$ that does not belong to $\mathcal{S}(T^0)$. We describe now how the upper bound flow model from [146] may lead to finding a valid inequality $I$ such that $\text{conv}(\mathcal{S}(T^0)) \subset \mathcal{P}(\mathcal{I} \cup \{I\})$ and $(\mathbf{x}^0, T^0) \notin \mathcal{P}(\mathcal{I} \cup \{I\})$.

For each machine $j \in [m]$, let $J_j = \{i \in [n] \,|\, x^0_{ij} > 0\}$ be the set of jobs assigned (at least fractionally) to machine $j$. If the machine constraint for machine $j$ is satisfied as equality by $(\mathbf{x}^0, T^0)$, then we call it *active*. Let $\Delta_j = \sum_{i \in J_j} p_{ij} - T^0$ be the *excess load* on machine $j$ and let $J'_j = \{k \,|\, p_{kj} > \Delta_j\}$ be the subset of jobs in $J_j$ whose processing time on machine $j$ is larger than $\Delta_j$. Note that the excess load on machine $j$ is positive if $\sum_{i \in J_j} p_{ij} > T^0$. Furthermore, for any $x \in \mathbb{R}$, we denote by $x^+$ the value $\max\{0, x\}$. The following theorem describes how we find valid inequalities. We give it without the proof. For details we refer the reader to [125].

**Theorem 5.1.1 (Mokotoff and Chrétienne [125])** *Let* $(\mathbf{x}^0, T^0)$ *be a point of* $\mathcal{P}(\mathcal{I})$ *and assume that the machine constraint for machine* $j$ *is active for* $(\mathbf{x}^0, T^0)$*. Then the linear inequality*

$$\sum_{i \in J_j} p_{ij} x_{ij} \leq T^0 - \sum_{i \in J_j} (p_{ij} - \Delta_j)^+ (1 - x_{ij}) \tag{5.1}$$

*is a valid inequality for* $\text{conv}(\mathcal{S}(T^0))$*. Moreover, if there is a job* $k \in J'_j$ *such that* $x^0_{kj} < 1$*, then the inequality is not satisfied by* $(\mathbf{x}^0, T^0)$*.*

**Exact algorithm for $R||C_{\max}$.** The result given by Theorem 5.1.1 can be used to design an exact cutting plane algorithm for the $R||C_{\max}$ scheduling problem. In the following we describe its structure which is given by Algorithm 19. But first we discuss the lower and upper bounds for the optimum, which we use in the algorithm.

A feasible initial (integer) solution is computed with the $2\sqrt{m}$-approximation algorithm of Davis and Jaffe [27]. The value of the makespan, $T_{DJ}$, obtained by the approximation algorithm is taken as the initial upper bound, i.e., $UB := T_{DJ}$. The total running time of the algorithm by Davis and Jaffe is $O(mn \log n)$. For details we refer to Section 4.1.

Now we consider the initial value of the lower bound *LB* for the optimum integer solution of the unrelated scheduling problem. The simplest lower bound is the solution of the relaxation which we obtain by assuming that the machines are identical and each job $i$ may be processed in its minimum processing time. Thus, the corresponding $LB_1$ is

$$LB_1 = \max\left\{\left\lceil \frac{1}{m}\sum_{i\in[n]} p_i^{\min}\right\rceil; \max_{i\in[n]}\{p_i^{\min}\}\right\},$$

where $p_i^{\min} = \min_{j\in[m]}\{p_{ij}\}$. This lower bound can be further improved by adding to the LP relaxation of the original problem the non-overlapping constraints introduced by Lawler and Labetoulle [110] for the scheduling problem $R|\,pmtn\,|C_{\max}$. Here, machine $j$ performs $x_{ij}$ percent of job $i$, where $x_{ij} \in [0,1]$ is the ratio of the time machine $j$ performs job $i$ to $p_{ij}$. The additional constraints are as follows:

$$\sum_{j\in[m]} p_{ij}x_{ij} \le T, \quad \forall i \in [n]. \tag{5.2}$$

The additional constraint of type (5.2) is not necessary for a job $i$ if

$$p_i^{\max} \le \frac{1}{m}\sum_{k\in[n]} p_k^{\min},$$

where $p_i^{\max} = \max_{j\in[m]}\{p_{ij} < \infty\}$. In this case we have

$$\begin{aligned}\sum_{j\in[m]} p_{ij}x_{ij} &\le \sum_{j\in[m]} p_i^{\max}x_{ij} = p_i^{\max} \le \tfrac{1}{m}\sum_{k\in[n]} p_k^{\min} = \\ &= \tfrac{1}{m}\sum_{k\in[n]}\sum_{j\in[m]} p_k^{\min}x_{kj} \le \tfrac{1}{m}\sum_{k\in[n]}\sum_{j\in[m]} p_{kj}x_{kj} = T^*,\end{aligned}$$

where $T^*$ is the optimal solution value of the LP relaxation without additional constraints of type (5.2). We call the LP relaxation with additional constrains preventing job overlapping an *extended* LP, and its optimal (fractional) solution we denote by $T_2$. Hence, the improved lower bound $LP$ is now

$$LB = \max\{LB_1; \lceil T_2\rceil\}. \tag{5.3}$$

In Algorithm 19 the function COMPUTE-LOWER-BOUND computes the maximum value of the lower bound according to (5.3). The next function COMPUTE-UPPER-BOUND returns the upper bound on the minimum makespan which is equal to the makespan of the schedule obtained by the approximation algorithm by Davis and Jaffe. Each iteration of the `loop` in Algorithm 19 is associated with a fixed value $LB$ of a lower bound for the makespan of the instance defined by $(A, \mathbf{b})$. Here, the scheduling problem is given in canonical form [136] wherein $A$ is the constraint matrix and $\mathbf{b}$ is the right-hand side vector. With each iteration the algorithm truncates the polyhedron $\mathcal{P}(LB)$ defined by

$$\begin{array}{rcll}\sum_{i\in[n]} p_{ij}x_{ij} & \le & LB, & \forall\, j\in[m]\\ \sum_{j\in[m]} p_{ij}x_{ij} & \le & LB, & \forall\, i\in[n]: p_i^{\max} > \frac{1}{m}\sum_{k\in[n]} p_k^{\min}\\ \sum_{j\in[m]} x_{ij} & = & 1, & \forall\, i\in[n]\\ x_{ij} & \in & [0,1], & \forall\, i\in[n], j\in[m]\end{array}$$

**Algorithm 19** CUTTING-PLANE-SCHEDULING($A, \mathbf{b}$)

```
Input: constraint matrix $A$
       vector $\mathbf{b}$
Output: integer solution $\bar{\mathbf{x}}$
 1: $LB$ := COMPUTE-LOWER-BOUND($A, \mathbf{b}$) $- 1$;
 2: $UB$ := COMPUTE-UPPER-BOUND($A, \mathbf{b}$);
 3: let $\mathbf{x}$ be the integer solution corresponding to $UB$;
 4: $LB := LB + 1$;
 5: $\mathcal{I} := \emptyset$,
 6: if $LB = UB$ then
 7:     return $(\mathbf{x}, LB)$;
 8: end if
 9: loop
10:     solve LP($LB, \mathcal{I}$) defined by $LB$ and $\mathcal{I}$;
11:     if LP($LB, \mathcal{I}$) has no solution $\mathbf{x}$ then
12:         go to step (4);
13:     end if
14:     if $\mathbf{x} \in \{0, 1\}^{mn}$ then
15:         return $(\mathbf{x}, LB)$;
16:     end if
17:     $I$ := FIND-NEW-CUT($LB, \mathcal{I}, \mathbf{x}$);
18:     if $I = -1$ then
19:         $\bar{\mathbf{x}}$ := BRANCH-AND-BOUND($A, \mathbf{b}, LB$));
20:         return $\bar{\mathbf{x}}$;
21:     end if
22:     $\mathcal{I} := \mathcal{I} \cup \{I\}$;
23: end loop
```

using cuts defined by Theorem 5.1.1. These cuts that belong to the set $\mathcal{I}$ are assumed to be computed by a function FIND-NEW-CUT($LB, \mathcal{I}, \mathbf{x}$). It returns $-1$ if no cut has been found. Otherwise, it generates a new valid cut $I$ which is stored in $\mathcal{I}$. In the former case, a BRANCH-AND-BOUND algorithm is used with the current value of $LB$ as its initial lower bound. The following linear program LP($LB, \mathcal{I}$) is used to decide whether a truncated by $\mathcal{I}$ polyhedron $\mathcal{P}(LB)$ is empty, or to provide a feasible solution $\mathbf{x}$:

$$\begin{array}{rcll}
\sum\limits_{i\in[n]} p_{ij}x_{ij} & \le & LB, & \forall\ j \in [m] \\
\sum\limits_{j\in[m]} p_{ij}x_{ij} & \le & LB, & \forall\ i \in [n] : p_i^{\max} > \frac{1}{m} \sum\limits_{k\in[n]} p_k^{\min} \\
\sum\limits_{i\in J_j} p_{ij}x_{ij} & \le & LB - \sum\limits_{i\in J_j} (p_{ij} - \Delta_j)^+(1 - x_{ij}), & \forall\ j \in [m] : j \text{ active and } J'_j \neq \emptyset \\
\sum\limits_{j\in[m]} x_{ij} & = & 1, & \forall\ i \in [n] \\
x_{ij} & \in & [0, 1], & \forall\ i \in [n], j \in [m].
\end{array}$$

The external loop, which starts in step (4) and ends in step (12) of Algorithm 19, increases by one the current value of $LB$. If $LB = UB$, then the solution given by the approximation algorithm of Davis and Jaffe is optimal and the main algorithm stops. Otherwise, a new polyhedron $\mathcal{P}(LB)$ is ready to be investigated by the internal loop (cf. lines (9)-(23)).

## 5.2 Branch-and-Price Method

A branch-and-price technique, which is similar to the branch-and-cut method, is a powerful framework for solving hard optimization problems. It offers an interesting alternative for the general purpose mixed integer programming based on decomposition or cutting planes. Branch-and-price combines column generation with a branch-and-bound scheme. The first one usually produces at the root node of the exploration tree tight lower bounds that are further improved while branching. Branching also helps to generate integer solutions. However, as we show it in the next chapter, branch-and-bound can be quite weak at computing good integer solutions rapidly since the solution for the relaxed master problem (see Section 2.2) rarely takes on integer values.

In this section we present the general ideas of the branch-and-price algorithm. We begin with a theoretical introduction and simple applications. Then we show how it can be used to compute exact solutions to the $R||C_{\max}$ problem. Afterward, we discuss various branching strategies that allow for column generation at any node in the branch-and-bound tree. Finally, we propose a general cooperation scheme between branch-and-price and local search techniques to help branch-and-price in finding good integer solutions earlier.

### 5.2.1 Preliminaries

Column generation is an efficient exact method for solving large-scale (fractional) optimization problems (see Section 2.2). It works with a restricted master problem, RMP, that consists of a linear problem defined on the current, usually much smaller than in the original problem, set of columns, and a pricing subproblem, PSP, that iteratively generates improving columns. When the master problem contains integral constraints on some of its variables, then usually column generation and branch-and-bound are combined. The resulting method is called branch-and-price.

The philosophy of branch-and-price is similar to that of branch-and-cut except that the procedure focuses on column generation rather than on row generation. In fact, pricing and cutting are complementary procedures for tightening an LP relaxation.

In branch-and-price sets of columns are left out of the LP relaxation because there are too many of them to be handled efficiently and most of them would have their associated variables equal to zero in an optimal solution anyway. Then to check the optimality of the current RMP, a pricing subproblem PSP is solved to try to identify columns to enter the basis of the RMP. If such columns have been found, then the RMP is re-optimized. Branching occurs when no columns price out to enter the RMP but the optimal LP solution does not satisfy the integrality conditions. Branch-and-price, which is also a generalization of branch-and-bound technique with LP relaxations, allows column generation to be applied throughout the exploration of the branch-and-bound tree what we see later.

At the first glance it may seem that branch-and-price involves nothing more than combining well-known ideas for solving linear programs by column generation with branch-and-bound. However, as Appelgren [7] observed almost 30 years ago, it is not that simple. There are several fundamental difficulties in applying column generation techniques, which were originally designed for linear programming, to integer programming methods [92]. These include, among others, the following drawbacks:

- conventional integer programming branching on variables may not be effective because fixing variables can destroy the structure of the pricing subproblem, and
- solving these linear programs to optimality may not be efficient, in which case different rules need to be applied to manage the branch-and-price tree.

**Related work.** Recently several specialized branch-and-price algorithms have been proposed in the literature for different integer problems. Routing and scheduling has been a particularly fruitful application area of branch-and-price, see, e.g., Desrochers, Desrosiers, and Solomon [29] for vehicle routing problems; Desrochers and Soumis [31] and Anbil, Johnson ,and Tanga [6] for crew scheduling problems; Barnhart, Johnson, Nemhauser, and Vance [13] for cutting stock problems; or Savelsbergh [150] for generalized assignment problem. For a good survey on the branch-and-price technique and its applications we refer the reader to [12, 30].

### 5.2.2 Generic-Branch-and-Price Scheduling Algorithm

We are ready now to present the generic structure of the branch-and-price algorithm (see Algorithm 20) which we have designed for scheduling unrelated parallel machines. The overall structure of the algorithm is determined by the search procedure in the branch-and-bound tree and corresponds to the `while`-loop in Algorithm 20.

We begin with the computation of the initial integer solution to the scheduling problem defined by constraint matrix $A$, and vectors $\mathbf{b}$ and $\mathbf{c}$ as defined in (2.8). To this end we use a simple heuristic which we present later in Section 6.2.2 while discussing the implementation details of the algorithm. The makespan of the initial solution $\bar{\mathbf{x}}$ defines us the initial upper bound $UB$ for the optimal solution. The (fractional) lower bound, $LB$, we set to zero. Next, we create the first subproblem $sp$ (it is just the relaxed original problem given in (2.8)) corresponding to the *root-node* of the branch-and-bound tree, and add it together with the actual lower bound $LB$ into the list of subproblems $SP$.

We proceed with the exploration of the branch-and-bound tree unless there is no subproblem in $SP$ (i.e., the search tree is empty). In every step of the `while`-loop, we first remove one subproblem $sp$ from $SP$ and compute an optimal solution $\mathbf{x}$ to $sp$ using column generation. In the generic implementation of the algorithm we choose for subproblem $sp$ the one from $SP$ with the currently smallest lower bound $LB$. We call it a *best-first* rule. There exists other rules which we discuss later in Section 5.2.5. The optimal solution $\mathbf{x}$ defines a new $LB$. If $\mathbf{x}$ represents an integer solution and $LB < UB$, then a new best integer solution has been found and the upper bound $UB$ is updated accordingly. Otherwise, we check if branching is needed, i.e., if $LP \leq UB$. Note that now $\mathbf{x}$ is considered to be

**Algorithm 20** GENERIC-BRANCH-AND-PRICE-SCHEDULING($A, \mathbf{b}, \mathbf{c}$)

```
Input: constraint matrix $A$
       vectors $\mathbf{b}$ and $\mathbf{c}$
Output: integer solution $\bar{\mathbf{x}}$
 1: compute integer solution $\bar{\mathbf{x}}$ to the scheduling problem defined by $(A, \mathbf{b}, \mathbf{c})$;
 2: $UB :=$ makespan($\bar{\mathbf{x}}$);
 3: $LB := 0$;
 4: $SP := \emptyset$;
 5: create root-node problem $sp(LB)$ and add it to $SP$;
 6: while $SP \neq \emptyset$ do
 7:     take subproblem $sp$ from $SP$ with the currently smallest $LB$;
 8:     compute optimal solution $\mathbf{x}$ to $sp$ by column generation;
 9:     $LB :=$ makespan($\mathbf{x}$);
10:     if $\mathbf{x}$ is an integer solution then
11:         if $LB < UB$ then
12:             $UB := LB$;
13:             $\bar{\mathbf{x}} := \mathbf{x}$;
14:         end if
15:     else
16:         if $LB \leq UB$ then
17:             choose fractional variable $x_{ij}$ from $\mathbf{x}$;
18:             create two subproblems $sp(x_{ij} = 0, LB)$ and $sp(x_{ij} = 1, LB)$;
19:             add new subproblems to $SP$;
20:         end if
21:     end if
22: end while
23: return integral solution $\bar{\mathbf{x}}$;
```

a fractional solution to $sp$. To generate two new subproblems, we first choose some fractional variable $x_{ij}$ on which we perform the branch. We describe in the next section which branching strategy we choose. Afterward the variable has been chosen, two new subproblems corresponding to $x_{ij} = 0$ and $x_{ij} = 1$ are added into $SP$. Together with each new subproblem we save the actual value of $LB$. The algorithm ends when *all* nodes from the branch-and-bound tree have been traversed. It may take much time until some good integer solution is found. In Section 5.2.5 we present a hybrid computation scheme which we use to help branch-and-price to find good integer solutions earlier.

## 5.2.3 Branching Strategies

An LP relaxation solved by column generation in Algorithm 20 is not necessarily integral, and thus applying a standard branch-and-bound procedure to the restricted master problem with its current columns does not guarantee an optimal solution. After branching it may be the case that there exists a column that would price out favorably but it is not present in the master problem. Therefore, to find an optimal solution, we must generate columns after branching. However, suppose that we use the conventional branching rule based on the variable dichotomy (see the next paragraph), we branch on a fractional variable $x_{ij}$, and we are in the branch in which $x_{ij}$ is fixed to zero. Then in the column generation phase it is possible (and quite likely what the tests show) that the optimal solution to the pricing subproblem is the same assignment represented by $x_{ij}$. In that case, it becomes necessary to generate the column with the second smallest reduced cost. At depth $d$ in the branch-and-bound tree we may need to find the column with $d^{th}$ smallest reduced cost.

In order to prevent columns that have been branched on from being regenerated, we must choose a branching rule that is compatible with the pricing subproblem. By compatibility we mean that we must be able to modify the column generation subproblem so that columns that are infeasible due to the branching constraints will not be generated and the column generation subproblem will remain tractable.

The challenge in formulating a branching strategy is to find one that excludes the current solution, validly partitions the solution space of the problem, and provides a pricing subproblem that is still tractable.

**Types of branching strategies for MIP.** In general, branching strategies for $0-1$ linear programs are based on fixing variables. When we fix a single variable, then we have a *variable dichotomy* strategy, and when a set of variables is fixed, then we obtain a *generalized upper bound dichotomy* or GUB dichotomy, for short. The simplest, variable dichotomy branching method is to divide the subproblem $sp$ into two subproblems $sp_0$ and $sp_1$ defined by fixing a fractional variable $x_{ij}$ of the optimal solution to $sp$, to 0 and 1, respectively. Its main disadvantage is the *uneven* division of the solution space of $sp$. In most cases, one of the resulting subproblems (usually $sp^0$) has an almost identical set of solutions to that of $sp$ (weak-side branch), whereas the other is much more constrained (strong-side branch). The depth of such unbalanced search trees is rather large what is caused by the long paths of weak-side branches needed to resolve some subproblems. When applied to the $R||C_{\max}$ problem, the depth of weak-side branches corresponding to one variable is at most $m-1$. Thus, the maximum depth of the complete branch-and-bound tree is $n(m-1)$.

The above disadvantage has led to the development of other branching strategies exploiting the set-partitioning structure of the problem (see, e.g., the branching rule of Ryan and Foster [148]). One such a strategy is the already mentioned GUB dichotomy branching method [130]. We explain its main features when applied to the $R||C_{\max}$ problem. Here, each constraint of the form $\sum_{j\in[m]} x_{ij} = 1$, for $i \in [n]$, is called a GUB constraint. Note that the MIP formulation of the scheduling problem given in (1.2) contains $n$ GUB constraints. Let $S$ represent the set of variables appearing on the left-hand side of a particular equality. Exactly one variable in $S$ will be set to 1 in any feasible $0-1$ solution vector $\mathbf{x}$ to (1.2). It follows that, for $S_1 \subset S$, the single variable of $S$ set to 1 will be either in $S_1$ or in $S \setminus S_1$. Equivalently, the problem can be partitioned into two subproblems, each defined by setting the variables of either $S_1$ or $S \setminus S_1$ to 0. For the resulting subproblems in order to have almost equally large sets of feasible solutions, $S_1$ and $S \setminus S_1$ should be of approximately the same cardinality.

By recursively partitioning sets $S_1$ and $S \setminus S_1$, eventually a subproblem will be left with a single variable of $S$ not set to 0, which is bound to be 1. In our case, if $|S_1| = |S \setminus S_1| = m/2$, a variable is set to 1 after at most $\lceil \log m \rceil$ partitions (levels of the branch-and-bound tree) since each equality GUB constraint involves $m$ variables. Whenever a variable is set to 1, branching proceeds by selecting another GUB constraint, i.e., another equality whose left-hand side has at least one variable still not set to 0. It is not difficult to prove that GUB dichotomy compared to variable dichotomy drastically reduces the depth of the search tree. More specifically, since complete branching on a single GUB constraint requires up to $\lceil \log m \rceil$ levels, the maximum tree depth is $n \lceil \log m \rceil$.

**Our branching strategy.** The idea of branching in our Branch-and-Price algorithm is to employ the standard formulation of the scheduling problem directly, i.e., to use the formulation given in (2.8) but with $x_{ij} \in \{0, 1\}$ (integrality constraints). We do not use the disaggregated formulation based on the Dantzig-Wolfe decomposition as it was indicated already in the previous subsection. In our case, fixing variables $x_{ij}$ to zero forbids job $i$ to be assigned to machine $j$, and fixing variable $x_{ij}$ to one requires job $i$ to be assigned to machine $j$. More specifically, to forbid a job $i$ to be assigned to machine $j$, we set $x_{ij} := 0$. To require a job $i$ to be assigned to machine $j$, we set $x_{ij} := 1$, and all variables for columns associated with job $i$ except $x_{ij}$, we set to zero, i.e., $x_{ik} := 0$ for all $k \in [m] \setminus \{j\}$.

Note that we use the variable dichotomy branching strategy described above. Its main drawback is the uneven division of the solution space (i.e., the search tree is heavily unbalanced). However, when compared to the GUB dichotomy strategy, our branching method is able to produce feasible integer solutions much earlier than the balanced approach. This is of special interest for an effective approximation algorithm based on Algorithm 20. More details are given in the next subsection.

### 5.2.4 Computational Issues

In the previous subsections we have presented the foundations of branch-and-price algorithms in general. Now, we want to discuss some important computational issues that need to be considered when implementing a branch-and-price algorithm.

**Initial restricted master problem.** To start the column generation scheme, an initial restricted master problem has to be provided. This initial RMP must have a feasible LP relaxation to ensure that proper dual information (dual variables, cf. Section 2.2) is passed to the pricing subproblem. Depending on the application, it is not always obvious how to construct such an initial RMP. However, if it does exist, it can *always* be found with a two-phase method similar to the method used by simplex algorithms to find an initial basic feasible solution. It is the so called phase-I of simplex using "big-M" penalty. Here, a set of artificial variables with large negative costs and associated columns that form an identity matrix is used. The artificial variables ensure that a feasible solution to the LP relaxation exists. Note that an initial RMP with a feasible LP relaxation has to be provided at *each* node of the branch-and-bound tree. Therefore, the artificial variables are usually kept at all nodes of the branch-and-bound tree what saves significantly the computation time.

To find an initial RMP with a feasible LP relaxation, heuristics can also be applied successfully. We present some of them while discussing the implementation details of the column generation in Section 6.2.2.

**Column management.** In the minimization process of a linear program, any column with negative reduced cost is a candidate to enter the basis (see Section 2.2). The pricing subproblem in column generation is to find a column with the smallest reduced cost. Therefore, if a column with negative reduced cost exists, the pricing subproblem will identify it. This guarantee that the optimal solution to the LP can be found.

However, it is not necessary to select the column with the smallest reduced cost [12]. Any column with a negative reduced cost will do the job. This observation can further improve the overall efficiency when the pricing subproblem is computationally intensive. Depending on the pricing subproblem, it may even be possible to generate more than one column with negative reduced cost into RMP per iteration without a large increase in computation time (cf. Section 6.2.2). It is evident that such a scheme increases the computation time per iteration of column generation since a larger RMP has to be solved, but it may decrease the number of iterations. Especially, in large linear programs containing thousands of columns, one need to consider this possibility and choose carefully the number of new columns entering the basis of RMP per iteration. The reader can see in the next chapter how it does influence the convergence, and thus the overall performance of the BRANCH-AND-PRICE scheduling algorithm.

Another observation coming from the practice shows that during the column generation process, RMP keeps growing. It may be advantageous to delete *non-basic* columns with large positive reduced costs from RMP in order to reduce the time per iteration.

Vanderbeck [170] discusses many issues related to the selection of a subset of *good* columns. On the one hand, we are trying to produce an integer solution to the master problem, on the other hand we are trying to solve LP relaxation of RMP. Limited computational experience seems to suggest that when the pricing subproblem can be solved to optimality efficiently, then adding only the most profitable column works best, and when the pricing subproblem is computationally intensive using approximations and adding multiple columns works best.

**LP termination.** The branch-and-bound framework has some inherit flexibility that can be exploited in branch-and-price algorithms like our. Note that branch-and-bound is essentially an enumeration scheme that is enhanced by fathoming based on bound comparisons. To control the size of the branch-and-bound tree, it is best to work with strong bounds. Clearly, there is a trade-off between the computational efforts associated with computation of good bounds but evaluation of small trees, and the computation of weaker bounds but evaluation of bigger trees. In the case of LP-based branch-and-bound algorithms in which the LPs are solved by column generation, there is a very natural way to explore this trade-off. Instead of solving RMP to optimality, i.e., generating new columns as long as profitable columns exist, we can choose to prematurely end the column generation process and work with the bounds of the final RMP value. Lasdon [108] and Farley [43] describe simple and easy to compute bounds on the final RMP value based on the LP value of the current (prematurely terminated) RMP and the corresponding reduced costs. This is especially important in view of the *tailing-off* effect that many column generation schemes exhibit, i.e., requiring a large number of iterations to prove the LP optimality.

**Dual solutions.** Recall that the objective function of the pricing subproblem depends on the dual variables of the LP relaxation of RMP. Consequently, if there are alternative dual solutions, we may pick any point on the face defined by these solutions. Simplex algorithms will give a vertex on this face, whereas interior point algorithms will give a point in the *center* of this face. A central point appears to have the advantage of giving a better representation of the face. Although, no extensive computational tests have been done to investigate the difference, it seems that using interior point methods works somewhat better [118] (see our computational results in Section 6.3.1).

**Approximate feasible solutions.** A branch-and-price algorithm can easily be turned into an effective approximation algorithm when a good feasible integer solution is the major concern and proving the optimality of lesser or no importance. This is accomplished by branching and searching the tree in a greedy fashion. If the goal is to prove the optimality, it always makes sense to choose a branching decision that divides the solution space evenly, i.e., we expect to be equally likely to find a good solution at either of the two nodes created. To this end we can use the GUB dichotomy strategy (cf. Section 5.2.3). If the goal is to find a good feasible solution, it makes sense to choose a branching decision that divides the solution space in such a way that we are more likely to find a good solution in one of the two nodes created and then choose this nodes for evaluation first. We then greedily search the tree always following the branch that is more likely to yield a good feasible solution. In [11] for crew paring problem, e.g., the LP relaxation is not completely re-optimized at each node of the branch-and-bound tree, instead a maximum number of column generation iterations per node is set and columns are generated only when the LP bound increased significantly above the value of the root LP.

A somewhat similar greedy search strategy for finding good feasible solutions has been developed based on the standard, variable dichotomy branching rule. The approach is effective when maintaining the ability to backtrack in the tree search is not important [12]. Each time a variable with a high fractional value is chosen and its value is permanently fixed to one. In [118] this strategy has successfully been applied to the crew pairing problem.

### 5.2.5 Heuristic Extensions of Branch-and-Price Algorithm

Now we present a hybrid computation scheme which we use to help branch-and-price to find good integer solutions earlier. Our method is based on the techniques presented in [30] which were originally developed for the vehicle routing problem with time windows. Similar to that method, our acceleration scheme is based on the cooperation between column generation and *local search*. Before we describe the cooperation scheme, we give some important facts about local search techniques.

Local search is a completely different optimization technique with opposite properties to just presented branch-and-bound methods. Local search algorithms use operators to define a *neighborhood* around a given solution or a set of solutions. The subregion of the search space is then explored iteratively to generate better solutions, whereas various strategies (metaheuristics), e.g., taboo search, simulated annealing, evolutionary algorithms, ant systems, and others are used to move from one neighborhood to the next one in order to escape *local minima*. Local search algorithms are effective at generating quickly excellent solutions. However, they do not provide the user with a lower bound on the objective. Hence, the difference between the solution obtained with a local search method and the optimal solution cannot be estimated, and thus the user does not know if more time should be invested to reach a better solution. For a good survey on the local search techniques used in combinatorial optimization we refer the reader to [1, 102, 119].

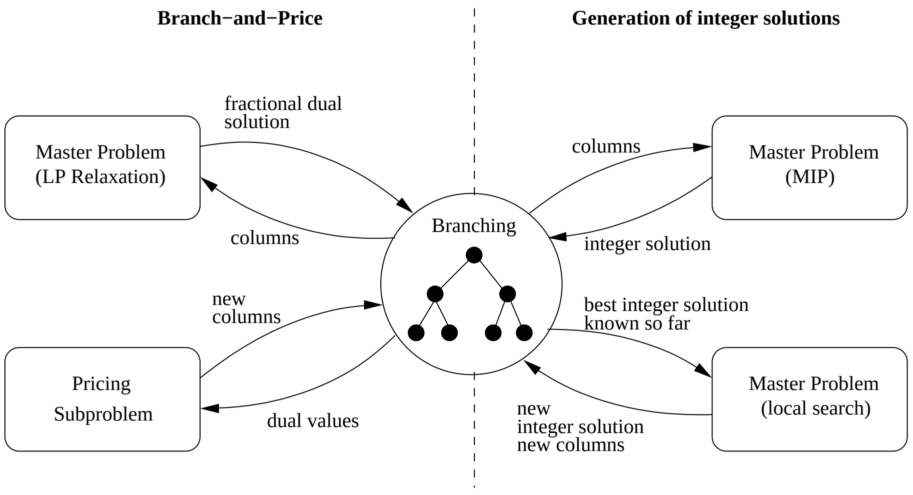

Branch−and−Price
Generation of integer solutions
Master Problem (LP Relaxation)
fractional dual solution
columns
Branching
new columns
dual values
Pricing Subproblem
columns
Master Problem (MIP)
integer solution
best integer solution known so far
Master Problem (local search)
new integer solution new columns

Figure 5.1: Cooperation scheme based on ideas from [30].

The cooperation scheme which we use is depicted in Figure 5.1. The left hand side of the figure shows the usual relaxed master problem and the pricing subproblem of the branch-and-price method. Note that the pricing subproblem can be solved by any optimization method (see Section 6.2.2). On the right hand side of the figure two components for obtaining integer solutions are specified. Firstly, a mixed integer programming (MIP) solver is called regularly (after termination of column generation, see Section 6.2.2) on the master problem with the set of columns of the current restricted master problem without relaxing the integrality constraints. If the MIP solver is called at the root node of the branch-and-price tree, then the best integer solution found so far is used as the first solution of the MIP problem. However, if the MIP solver is called at a node further down the branch-and-price tree, then the best integer solution found so far might not be valid for the branching decisions taken at that node. Hence, it cannot always be used as a first solution for the MIP problem. The effort spent on solving the MIP problem is controlled with a time or node limit. When the allowed time limit is reached, the exploration of the branch-and-price tree is resumed. Secondly, local search is also called regularly (see Section 6.2.2) to solve the master problem with initial solution being the best integer solution found so far. Unlike the MIP solver, local search is not restricted to combining existing columns of the current restricted master problem, i.e., local search may not only provide a better combination of existing columns, but it may also introduce new columns. Consequently, the columns generated by it are more diverse what is likely to accelerate pricing, e.g., because it has greater chances to overcome the degeneracy.

The strength of this hybrid scheme is diversification by means of using different algorithms for solving the same problem [30]. Branch-and-price obviously benefits from local search that is more effective at finding feasible solutions. But in turn, local search benefits from branch-and-price that provides it with diverse initial solutions. Indeed, the main drawback of local search algorithms is to escape local minima. To overcome this, the strategy of various metaheuristics is the attempt to control a series of moves that increase the value of the objective function in order to reach a different and more promising region of the solution space. In our cooperation scheme, the upper bound for the master problem and the MIP cutoff are always updated with the value of the best integer solution found so far. Thus, when the MIP solver finds a new integer solution or when the solution of the relaxed master problem is integer, it is by construction an improvement on the last local optimum found by local search. So, diversification is achieved and the lower bound for the objective function is improved at the same time.

Branch-and-price is an exact method. It will in the end find the optimal integer solution, and thus our cooperation scheme is not very useful when exploring the branch-and-price tree to optimality. However, the complete exploration may find good integer solutions only late in the computation. The cooperation scheme helps to find good integer solutions at an earlier stage of the computation what has numerous advantages. First, the user can stop the optimization as soon as satisfied with the quality of the integer solution found so far, and use truncated exploration of the branch-and-price tree as a powerful heuristic that also provides tight lower bounds. Next, good upper bounds are helpful to solve the pricing subproblem more effectively. A good upper bound may also reduce the number of iterations between master problem and subproblem at each node. Finally, knowing a good upper bound early might help to explore only a relatively small number of nodes in the branch-and-price tree. Given a fixed branching strategy, a *best-first* exploration strategy guarantees that only the children nodes of a node with a lower bound smaller than the optimal objective value have to be explored. In this sense, best-first search guarantees that a minimum number of nodes is explored. However, best-first search can fail to produce good integer solutions until at the very end of the tree exploration. This is why other exploration strategies such as *depth-first* search are often preferred, although they lead to higher number of explored nodes. The cooperation scheme which we use allows us to choose a tree exploration strategy such as best-first search, because this scheme does not rely only on branching to generate integer solutions.

# 6

# Experimental Study

In this chapter we present an experimental study on various algorithms for solving the $R||C_{\max}$ problem. We restrict our research only to the algorithms discussed in detail in the previous chapters. These algorithms, however, represent to our best knowledge the present state-of-the-art of solution methods for this scheduling problem. They use almost all types of computational frameworks developed for this problem so far. The algorithms range from purely combinatorial algorithms using generalized network flows or simple list scheduling, to more sophisticated methods based on linear programming and rounding. In the tests we consider both deterministic as well as randomized approaches. Furthermore, in our selection of algorithms there are techniques which compute exact solutions, like, e.g, those based on the branch-and-bound scheme, or approximate solutions with (e.g., Approximation-Unsplittable-Truemper and the two-step approaches) or without (e.g., the large neighborhood search heuristic) guarantee for the optimum. Some of them, e.g., list scheduling algorithms, go back to the late 1960s when they were for the first time formulated for solving simple optimization problems. We consider also very new techniques like, e.g., our new purely combinatorial Approximation-Unsplittable-Truemper algorithm.

In the following sections we discuss first the motivation of the experimental study. Then, we give the complete list of tested algorithms together with some, relevant to the experiments, implementation details. Here, we put much more attention to the improvement proposals and various implementation issues of the Approximation-Unsplittable-Truemper algorithm. Next, we present three different test models according to which test instances are generated. Here, we describe also shortly the experimental setup which we use in our simulations. Finally, we present the computational results followed by a careful analysis.

## 6.1 Motivation

The computational experiments which results we present in the last section of this chapter are motivated, firstly, by the importance of the unrelated scheduling problem in general both for the theoretical and the practical research, and secondly, by the wish to evaluate in practice new algorithms presented in this thesis which we have designed for solving the $R||C_{\max}$. Here, we pay special attention to the Approximation-Unsplittable-Truemper algorithm presented in Chapter 3. As already indicated there, this new, purely combinatorial approach computes 2-approximate solutions within the best *worst-case* running time known so far. Therefore, the primary goal of our research here is to evaluate its efficiency in *practice*. In the following we present two implementations of the algorithm (with and without heuristic improvements), and compare them with algorithms delivering both approximate and exact solutions which have been developed for this problem over the past decades. We are especially interested in those based on the two-step approach what is motivated by their simplicity and common usage in practice. As measures of interest, the total computation time, the value of makespan, and the usage of operational memory have been chosen.

## 6.2 Algorithms and Experimental Setup

### 6.2.1 List of Tested Algorithms

In order to evaluate our new algorithms and especially the Approximation-Unsplittable-Truemper algorithm, eighteen deterministic and randomized algorithms and heuristics [27, 71, 85, 112, 125, 126, 160, 168] for the $R||C_{\max}$ problem have been implemented and tested. In the following, for the sake of completeness of the presentation, we give a short list of all of them. Additionally, in several cases, implementation details are presented which, from our point of view, are important to the experiments. In order to simplify the presentation and later the analysis we introduce here for each algorithm a short version of its name which can be found, together with a short description and references, in the following tables.

**2–approximation algorithms:**

| *Name* | *Short description* | *Sections* |
|---|---|---|
| TS-Simplex | implementation of the algorithm by Lenstra et al. (see Algorithm 1); it solves LP relaxation with *simplex* algorithm and uses generic rounding; | 2.1.2, 2.1.3, 6.2.2 |
| TS-Barrier | implementation of the algorithm by Lenstra et al. (see Algorithm 1); it solves LP relaxation with *barrier* algorithm and uses direct rounding; | 2.1.2, 2.1.4, 6.2.2 |
| TS-Column | implementation of the improved two-step approach (see Algorithm 2); it solves LP relaxation with *column generation* algorithm and uses generic rounding; | 2.2, 6.2.2 |
| TS-Randomized | implementation of randomized two-step approach (see Algorithm 3); it solves LP relaxation with *column generation* algorithm and uses *randomized* rounding; | 2.3 |
| UTA-generic | generic implementation of the Approximation-Unsplittable-Truemper algorithm (see Algorithm 6); | 3, 6.2.2 |
| UTA-improved | improved implementation of the Approximation-Unsplittable-Truemper algorithm (see Algorithm 6); | 3, 6.2.2 |

**Heuristic approaches:**

| *Name* | *Short description* | *Sections* |
|---|---|---|
| Ibarra-Kim-{A,B,C,D} | implementations of the four simple list scheduling heuristics given by Ibarra and Kim; | 4.1.1 |
| Davis-Jaffe | implementation of the list scheduling algorithm by Davis and Jaffe (see Algorithm 7); | 4.1.2 |
| LPT-ver{1,2,3} | implementations of the three adaptations of the LPT algorithm by Graham (see Algorithm 8, 9 and 10); | 4.1.3 |
| Partial-Enumeration | implementation of the second heuristic using partial enumeration designed by Mokotoff and Jimeno (see Algorithm 12); it solves the MIP subproblem with the *branch-and-cut* algorithm from the ILOG CPLEX 9.0 package; | 4.2 |
| Branch-and-Greed | implementation of the large neighborhood search heuristic by Sourd based on a branch-and-greed scheme (see Algorithm 15 and 16); | 4.3 |

**Exact solution methods:**

| *Name* | *Short description* | *Sections* |
|---|---|---|
| Cutting-Plane | implementation of the exact cutting plane algorithm given by Mokotoff and Chrétienne (see Algorithm 19); | 5.1 |
| Branch-and-Price | implementation of the exact branch-and-price algorithm (see Algorithm 20); | 5.2, 6.2.2 |

### 6.2.2 Selected Implementation Details

Now, as it was indicated in the previous section, we give some selected implementation details for several algorithms listed above. Here, we pay special attention to the implementation details of the Approximation-Unsplittable-Truemper algorithm. More specifically, we describe very carefully the three most crucial improvements of the generic Unsplittable-Truemper algorithm which lead to a significant increase of the overall performance of the algorithm.

**TS-Simplex:** The LP relaxation is solved with the *sifting* algorithm from the ILOG CPLEX 9.0 package [87]. In the second step, we round the fractional solution by applying the generic matching algorithm by Lenstra et al. from [112].

**TS-Barrier:** The LP relaxation is solved with the *barrier* method from the ILOG CPLEX 9.0 package. In the second step, we round the non-basic (fractional) solution computed with the interior point algorithm into an integer solution by applying the direct rounding algorithm by Shmoys and Tardos [157] as described in Section 2.1.4. In the case we are solving the $P||C_{\max}$ problem, this relative complicated method can be repleaced with a much simpler rounding procedure proposed by Plotkin et al. [141] (cf. Section 2.1.3).

**TS-Column:** In the first step, in order to find a feasible initial set of columns for the restricted master problem RMP, a hybrid method is used. It needs $O(nm \log n)$ running time and is composed of three subroutines. Most of the initial columns are generated with the algorithm by Jaffe and Davis [27]. Additional columns are computed with two simple heuristics. They compute columns which correspond to two different feasible solutions for the original scheduling problem. Both heuristics assign the jobs sequentially and differ from each other in the order the jobs are processed. Interestingly, only when all three algorithms are used, the initialization produces very effective (i.e., significantly speeding up the convergence of the column generation process) set of columns. Resulting RMPs are solved with the *primal simplex* algorithm from the ILOG CPLEX 9.0 package. (During the implementation process, we have tested a number of different algorithms for solving RMPs. But only the primal simplex algorithm delivered a satisfactory performance. That can be explained by the primal structure of RMPs.) Because of the sparse structure of the constraint matrix $A$, the reduced costs $\bar{c}_k$, and thus all pricing subproblems PSP can be computed very fast in time $O(nm)$ by enumeration. With each iteration of the column generation process, we add up to 20 new columns to the constraint matrix $\mathbf{a}$ of RMPs provided at least one column with a negative reduced cost exists. In the second, rounding phase of the algorithm, we try first to find an integral optimal solution to the MIP defined by the constraint matrix $\mathbf{a}$ of the last optimal RMP. To solve this MIP, we use the *branch-and-cut* algorithm from the ILOG CPLEX 9.0 package. Note that the integer solution to this MIP is not necessarily optimal for the original scheduling problem. If the optimal integral solution cannot be found within an allowed time limit, the rounding procedure by Lenstra et al. from [112] is called for the fractional optimal solution found with the column generation algorithm and a 2-approximate solution to the original scheduling problem is computed.

**UTA-generic:** We have developed two versions of Approximation-Unsplittable-Truemper algorithm. The first one, UTA-generic, is a generic implementation of the approximation algorithm. We give now two main improvements which we have carried out in the structure of the Unsplittable-Truemper algorithm in order to make it more effective.

The first improvement regards the definition of the machine partitions given in Section 3.3 by Definition 3.3.2. In order to improve the quality of solution produced by the algorithm for practical instances, we redefine this definition in the following way:

**Definition 6.2.1** *Let $T \in \mathbb{N}$ and $\alpha$ be a $T$-feasible assignment. We partition the set of machines $M$ into three subsets:*

$$\begin{aligned} M^{-}(\alpha) &= \{j \,|\, \delta_j(\mathbf{P}, \alpha) \leq T\} \\ M^{0}(\alpha) &= \{j \,|\, T + 1 \leq \delta_j(\mathbf{P}, \alpha) \leq T + a_j\} \\ M^{+}(\alpha) &= \{j \,|\, \delta_j(\mathbf{P}, \alpha) \geq T + a_j + 1\} \end{aligned}$$

*where* $a_j = \min\{T; \max_{i \in J}\{p_{ij} < \infty\}\}$.

Note that by this new definition the upper bounds for loads on machines in $M^+$ and $M^0$ become smaller. Since now the maximal load on machines in $M^0$ may be less than $2T$ (as forced by Definition 3.3.2), the makespan of the resulting schedule may also be smaller then $2T$. This new definition has no influence on the correctness of the theoretical results presented in Section 3.4 and Section 3.5.2. Nevertheless, for the sake of completeness, we give in the following the complete proof of a new version of Lemma 3.4.1. It is sufficient to prove Lemma 6.2.1 to show that the new definition of the machine partitions implies no changes in the structure of the Unsplittable-Blocking-Flow algorithm to ensure its convergence. Lemma 6.2.2 which is an adaptation of Lemma 3.4.2 to Definition 6.2.1 summarizes this observation.

**Lemma 6.2.1** *Let $v_0$ be a helpful machine of minimum height. Let $M^+$, $M^0$, and $M^-$ be the machine partitions as given in Definition 6.2.1. Then there exists a sequence $v_0, \dots, v_r$ where $s(v_i) = v_{i+1}$, for all $0 \le i \le r-1$, $v_{2i} \in M$, for all $0 \le i \le r/2$ and $v_{2i+1} \in J$, for all $0 \le i < r/2$ with the following properties:*

*(a)* $(v_i, v_{i+1}) \in E^0_\alpha$ *and* $h(v_i) = h(v_{i+1}) + 1$

*(b)* $\delta_{v_0} \ge T + 1 + p_{s(v_0),v_0}$

*(c)* $T + 1 \le \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} - p_{s(v_{2i}),v_{2i}} \le T + a_{v_{2i}}, \forall 0 < i < r/2$

*(d)* $\delta_{v_r} + p_{s(v_{r-2}),v_r} \le T + a_{v_r}$

*where* $a_{v_{2i}} = \min\{T; \max_{k \in J}\{p_{k,v_{2i}} < \infty\}\}$, *for all* $0 \le i \le r/2$.

**Proof:** By Definition 3.4.1, $0 < h(v_0) < \infty$ and thus there exists a path from $v_0$ to a machine in $M^-$ that defines the height of $v_0$. On this path (a) must hold. Furthermore, condition (b) follows directly from the definition of a helpful machine.

Note that a machine $j \in M^+$ is helpful if $h(j) < \infty$. So if we start a path with a helpful machine of minimum height, then all machine nodes $v_2, v_4, \dots, v_{r-2}$ belong to $M^0$ and $v_r$ may belong to $M^0$ or $M^-$. Therefore

$$\delta_{v_{2i}} \le T + a_{v_{2i}}, \forall 0 < i \le r/2.$$

Furthermore, none of these nodes is helpful which implies that

$$\delta_{v_{2i}} < T + 1 + p_{s(v_{2i}),v_{2i}}, \forall 0 < i \le r/2.$$

There are two cases to consider now. If $\delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} \le T + a_{v_{2i}}$, then $r = 2i$ and condition (d) holds. On the other hand, since $a_{v_{2i}} \ge p_{s(v_{2i}),v_{2i}}$

$$\begin{aligned} \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} &\ge T + a_{v_{2i}} + 1 \\ \Rightarrow \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} - p_{s(v_{2i}),v_{2i}} &\ge T + a_{v_{2i}} + 1 - p_{s(v_{2i}),v_{2i}} \\ \Rightarrow \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} - p_{s(v_{2i}),v_{2i}} &\ge T + 1, \end{aligned}$$

what proves the lower bound given in (c). To proof the upper bound, note that $v_{2i}$ is not helpful $\forall 0 < i \leq r/2$, and $a_{v_{2i}} \geq p_{s(v_{2i-2}),v_{2i}}$. It follows that

$$\begin{array}{rcl} \delta_{v_{2i}} & < & T + 1 + p_{s(v_{2i}),v_{2i}} \\ \Rightarrow \quad \delta_{v_{2i}} - p_{s(v_{2i}),v_{2i}} + p_{s(v_{2i-2}),v_{2i}} & \leq & T + p_{s(v_{2i-2}),v_{2i}} \\ \Rightarrow \quad \delta_{v_{2i}} + p_{s(v_{2i-2}),v_{2i}} - p_{s(v_{2i}),v_{2i}} & \leq & T + a_{v_{2i}}, \end{array}$$

proving the upper bound given by (c). This completes the proof of the lemma. ∎

**Lemma 6.2.2** *Let $\beta$ be the assignment computed by the call to* UNSPLITTABLE-BLOCKING-- FLOW*$(\alpha, G^0_\alpha(T), \mathbf{P}, T)$. Then*

*(a)* $j \in M^-(\alpha) \Rightarrow \delta_j(\mathbf{P},\beta) \geq \delta_j(\mathbf{P},\alpha)$

*(b)* $j \in M^0(\alpha) \Rightarrow T + 1 \leq \delta_j(\mathbf{P},\beta) \leq T + a_j$

*(c)* $j \in M^+(\alpha) \Rightarrow \delta_j(\mathbf{P},\beta) \leq \delta_j(\mathbf{P},\alpha)$.

*where* $a_j = \min\{T; \max_{i \in J}\{p_{ij} < \infty\}\}$.

The second improvement concerns the introduction of two different data structures which are used to represent the bipartite graph given by Definition 3.3.1. We have a *job-machine* oriented structure which is used globally in binary search by the approximation algorithm and locally by the UNSPLITTABLE-BLOCKING-FLOW algorithm to represent admissible graphs (see Section 3.4). This data structure consists of machine and job nodes. The information about edges adjacent to a given node is saved in this node in two double-linked lists (one for in-coming edges, one for out-going edges). The second, *machine* oriented structure is

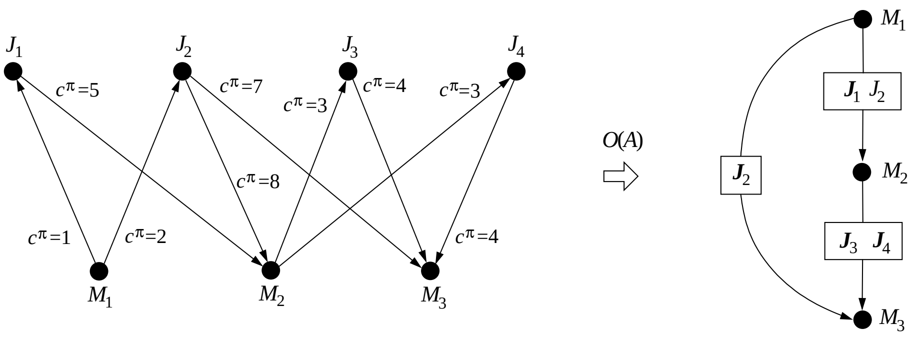


Figure 6.1: Contraction of the bipartite graph into the machine oriented representation.

used to speed up the computations of admissible graphs. It consists only of machine nodes. Information about adjacent job nodes and edges is saved in machine nodes. To this end, for each machine node $j$, we use a double-linked list $L$ to save each machine node $k$ which is connected with machine node $j$ by a two-edge in-coming path, i.e., for which there exist two edges $(k, i) \in E^1_\alpha$ and $(i, j) \in E^2_\alpha$ in graph $G_\alpha(T)$ (see Definition 3.3.1) defining a two-edge path from machine node $k$ to machine node $j$ via job node $i$. Note that there can exist more than one two-edge path connecting machine $k$ with machine $j$. To this end, each job node $i$ through which these paths are passing is saved together with node $k$ in the list $L$. Furthermore, we compute for each such two-edge path its reduced cost, $c^{\pi}_{ki} + c^{\pi}_{ij}$, and mark a job node which lies on a path with a smallest reduced cost as *active*. More specifically, job node $i$ is active if $i = \arg\min_{l \in J}\{c^{\pi}_{kl} + c^{\pi}_{lj} \mid (k,l) \in E^1_{\alpha} \wedge (l,j) \in E^2_{\alpha}\}$. For the case there are more than one two-edge path with a smallest reduced cost, all jobs lying on them are marked as active.

The second structure is thus a contracted version of the first structure and can be computed in time $O(A)$ where $A$ is the number of edges in $G_{\alpha}(T)$. The usage of two different data structures is motivated by huge differences in size between the graphs used globally in the binary search and the admissible graphs used by Unsplittable-Blocking-Flow. The introduction of the contracted structure improves significantly the computation times of admissible graphs. Figure 6.1 shows these two equivalent structures with a simple example. With bold characters are marked the active job nodes. Note that the second graph *includes* already information about the shortest direct connection between two given machine nodes.

**UTA-improved:** In the second version of the algorithm we use, additionally to the changes discussed in the previous paragraph for the implementation of UTA-generic, a better interphase initialization of the approximation algorithm. It takes place between two consecutive binary search steps of the algorithm. By using it, not all results computed in the previous phase get lost with the beginning of the next phase, i.e., computed assignment and node potentials can be *reused*. Since our algorithm uses the Primal-Dual approach (see Section 3.3), an initial assignment must maintain the reduced cost optimality condition that is, for each edge $(i, j)$ in the bipartite graph, the reduced cost $c^{\pi}_{ij} \geq 0$ [3]. To initialize a new binary search step with the previous solution $\beta$ (with whole or just a part of it), we need to check if this condition is fulfilled.

There are two cases to consider when a binary search step terminates (cf. Algorithm 5). In the first case, the Unsplittable-Truemper terminates with $M^+ \neq \emptyset$, and the next binary search step is initialized with increased value of $T$. When $T$ increases, new edges can be added to the bipartite graph. For each such edge, we check if its reduced cost is nonnegative. Here, we use node potentials $\pi$ from the previous solution $\beta$. If only one edge does not fulfill the reduced cost optimality condition, we initialize the next step like in UTA-generic, i.e., each job is assigned to a machine where its processing time is minimum.

In the second case, Unsplittable-Truemper terminates with $M^+ = \emptyset$, and the next binary search step is initialized with $T$ of smaller value. When $T$ decreases, some edges must be deleted from the bipartite graph because their processing times are greater than $T$. In particular, some jobs may become unassigned. For each such job node $i$ which becomes unassigned, we choose machine node $j$ on which it has the minimal processing time. Afterward, using node potential $\pi$ from the previous solution $\beta$, we check for each edge $(j, i)$ if the reduced cost optimality condition is fulfilled. If this is the case, we use the previous solution $\beta$ as initialization. Otherwise, we initialize the next step as in UTA-generic. We show in Section 6.3 that the refined initialization process significantly improves the convergence of the computations and thus the overall efficiency of the algorithm, especially for difficult instances.

**Branch-and-Price:** To compute the initial assignment $\alpha$, we use the same hybrid method like in the TS-Column algorithm. Each subproblem *sp* is solved with the column generation method. Here, we use the same parameters like described for the TS-Column algorithm. After a subproblem *sp* is solved, we check its integrality by enumeration in time $O(nm)$. Simultaneously, we search for a fractional variable to branch on. Having found a fractional variable, we generate two new subproblems as described in Section 5.2.3, and insert them into a heap-organized list of subproblems $SP$. The subproblem with the currently smallest lower bound $LB$ in $SP$ is on the top of the heap.

Now we describe several implementation details concerning the cooperation scheme which we use to accelerate the convergence of the algorithm. For solving the MIP problems we employ the *branch-and-cut* algorithm with time limit from the ILOG CPLEX 9.0 package. To perform the local search we use a *large neighborhood search* scheme [155]. It is based upon a process of continual partial assignment and re-optimization. In particular, it proceeds by iteratively fixing some assignment variables of the problem to their values in the currently best integer solution $\bar{\mathbf{x}}$ (obtained with column generation or the MIP solver) and solving (re-optimizing) a smaller MIP subproblem on the rest of the variables (they define a neighborhood $\mathcal{V}(\bar{\mathbf{x}})$ of $\bar{\mathbf{x}}$) with the *branch-and-cut* algorithm from the ILOG CPLEX 9.0 package. If a better integer solution cannot be found during a given number of iterations, then the subproblem is either enlarged, i.e., more jobs are fixed to their values in the currently best integer solution, or interrupted when the allowed time limit elapsed. Otherwise, $\bar{\mathbf{x}}$ is replaced by a new, better solution and the next MIP subproblem defined by a new neighborhood $\mathcal{V}(\bar{\mathbf{x}})$ is re-optimized. One iteration of partial relaxation and re-optimization is considered as the examination of a powerful neighborhood move. The procedure terminates when a satisfactory improvement in the best integer solution has been reached or when a specified time limit has elapsed.

As it can be expected, a fair amount of tuning time may be required in order to know when and for how long the MIP solver and local search should be called. Solving completely the MIP formulation of the master problem is time consuming, therefore we use a time limit for the MIP solver. Moreover, the MIP solver is called only when we know that it has a good chance to find an improved integer solution. To decide on this, we observe the *integrality gap* between the best known integer solution and the value of the current fractional solution to the master problem obtained with column generation. When it is *relatively* high, then we call the MIP solver. Another reason to start the MIP solver may be a small number of integer-infeasible variables in the fractional solution to the master problem.

Local search is called for post-optimization each time a new integer solution is found by the MIP solver or when the solution to the relaxed master problem is integer. Finally, a simple adaptive scheme can be used which decreases or increases the frequency and the computation time allocated to the MIP solver and local search according to their respective success rates [30]. Here, appropriate statistics have to be collected and analyzed.

### 6.2.3 Test Models

To evaluate the algorithms we use artificially generated test instances. Each instance belongs to one of the three different test models. The models are defined as follows:

**Model A:** There is no correlation between the machines and the jobs. Processing times $p_{ij}$ are generated from uniform distribution [1..100].

**Model B:** The jobs are correlated. Each processing time $p_{ij} = b_i + d_{ij}$ where $b_i$ and $d_{ij}$ are two randomly generated integers from uniform distributions on [1..100] and [1..20], respectively. Intuitively, $b_i$ can be regarded as a mean value of processing time of job $i$, and $d_{ij}$ as its deviation.

**Model C:** The machines are correlated. Each processing time $p_{ij} = a_j + d_{ij}$ where $a_j$ and $d_{ij}$ are two randomly generated integers from uniform distributions on [1..100] and [1..20], respectively. Here, similar to Model B, $a_j$ can be seen as a mean value of machine performance for machine $j$, and $d_{ij}$ as its deviation.

Note that the two last models put some restrictions on the processing times. They are also more difficult to solve, in particular, for the combinatorial algorithms. Details will be given in Section 6.3. Similar models were used previously in the literature, e.g, in [58, 126, 160]. To our best knowledge, there exists no generally accepted set of difficult test instances (benchmarks) for the $R||C_{\max}$ problem.

### 6.2.4 Experimental Setup

All tested algorithms, including both versions of Approximation-Unsplittable-Truemper, are implemented in C/C++ using standard libraries and compiled with a GNU compiler gcc 3.2.3. The *simplex*, *sifting*, *barrier*, and *branch-and-cut* procedures, all from the ILOG CPLEX 9.0 Callable Library [87], are used together with the ILOG Concert Technology 2.0 [86] in the implementation of TS-Simplex, TS-Barrier, TS-Column, TS-Randomized, Branch-and-Price, and Partial-Enumeration algorithms. All techniques are computationally tested on two machines. The first one, a Sun Fire 3800 machine equipped with eight 900MHz ultraSPARC III processors and 8GB RAM working under Solaris 9 operating system, is used in particular to test the ability of the algorithms to solve large-scale problem instances. Here, we are able to observe how much operational memory the algorithms need to solve a very large problem. The second machine, a desktop computer with a single 2.4GHz Pentium IV processor and 1GB RAM working under Linux 2.4.21 operating system. We use it to carry out efficiency tests, i.e., tests where the quality of solution together with the computation time are measured. In the analysis of the results, we do not point out unless it is of crucial importance, on which machine given results have been obtained.

## 6.3 Computational Results

In the following we present various computational results obtained from the experiments on the algorithms listed in Section 6.2.1. The goal of our practical simulations is to evaluate the Approximation-Unsplittable-Truemper algorithm and other new algorithms, which we propose in this work, and compare them with various techniques broadly used in practice to solve the $R||C_{\max}$ scheduling problem. The results are divided into three separate sections. In the first part, we discuss the results delivered by the algorithms computing solutions with approximation factor of 2, i.e., by all two-step algorithms plus the two implementations of the Approximation-Unsplittable-Truemper algorithm. In the second part, we concentrate more carefully on the quality of solutions delivered by the Approximation-Unsplittable-Truemper algorithm. To this end, we compare it with the two branch-and-bound methods, Branch-and-Price and Cutting-Plane both used for computing exact solutions to the scheduling problem. Since the importance of the heuristic methods solving very efficiently large-scale practical problems cannot here be neglected, we dedicate the last section to them and compare the efficiency and quality of solutions produced by methods from Chapter 4 with those computed by our combinatorial algorithms.

### 6.3.1 Evaluation of $2$-approximation Algorithms

As it was already indicated, we evaluate first the computational efficiency and the makespan quality of the 2-approximation algorithms presented in this thesis. We test them on a large number of problem instances generated according to the three models defined in Section 6.2.3. To analyze the results, which include computation times and makespans, we use two types of diagrams. In the first paragraph, we make use of the *performance diagrams* to present the *collective* behavior (i.e., across all tested instances) of the algorithms. We use them both for the computation time and the makespan. They show, similar to the histograms, for each tested method the percentage of tested instances for which this method produced the best result. In the second paragraph of this section we use traditional *time graphs* showing the total average computation time in dependence of the problem size. The *size* of instance is given as a tuple $(n, m)$. The growth of the size is then defined as an increase in $n$, or $m$, or in both. Finally, in the last paragraph we show the computation time in function of the number of machines, while the number of jobs stays fixed.

**Performance diagrams.** For each model we choose the following sizes of instances: $m \in \{10, 25, 50, 100, 200, 350, 500, 750, 1000\}$ and $n = 10m$. The factor of 10 is motivated by the outcome from the first tests with Model C. Those tests have shown that the Approximation-Unsplittable-Truemper algorithm needs the most time when the number of jobs $n$ is about tenfold greater than the number of machines $m$. For the other models we have not observed similar relation between $n$ and $m$. In the main tests, for each problem size $(n, m)$, 10 different instances are generated and solved. To our best knowledge, it is the first time when instances of such large sizes are considered in an experimental research on the $R||C_{\max}$ problem. The largest instances considered so far contained up to 1000 jobs and 50 machines [57, 125, 126, 167]. Obtained results are used then to construct performance diagrams, both for the total computation time and the makespan. The value of performance ratio is given always on the x-axis. It shows, for each result of a given instance, the ratio between this result and the best result among all methods for this instance. The y-axis shows, for each method, the percentage of all tested instances for which the performance ratio is not greater than a given value of performance ratio. Note that the more upper left is a given graph located, the better is the result which it describes. Moreover note that for

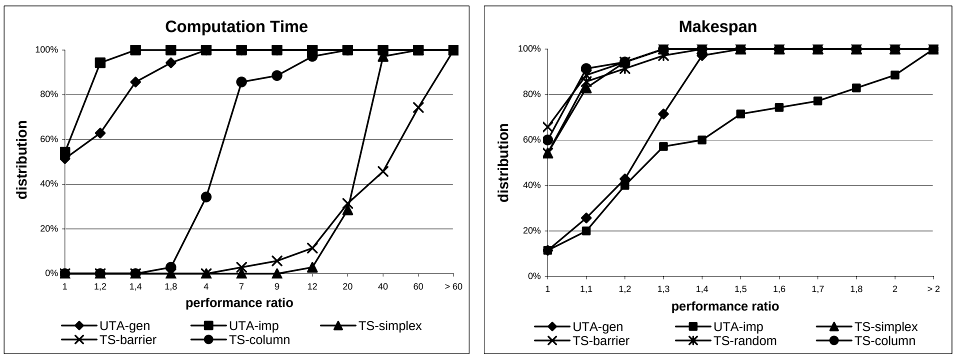


Figure 6.2: Performance diagrams for Model A.

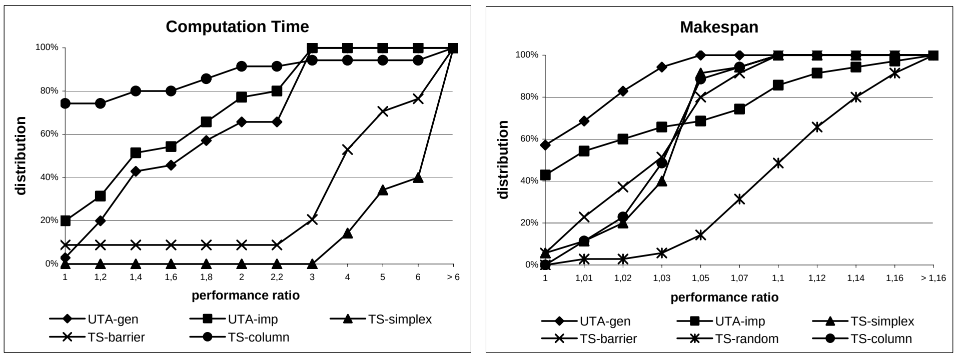


Figure 6.3: Performance diagrams for Model B.

performance ratio equal to 1, the sum of distribution values (indicated on the y-axis) computed over all methods is often greater than 100%. This can be explained by the fact that for some test instances several methods return exactly the same results (i.e., the same value of makespan or computation time). In particular, in the measurements of computation time we observe this only for a few test instances, and thus their influence on the main results is negligible. A reason for this is the precision of measurement which is limited to 0.01 sec.

The graphs in Figure 6.2, 6.3, and 6.4 show the performance diagrams for Model A, B, and C, respectively. Note that in the performance diagrams for computation time we do not consider the TS-Randomized method. It uses to solve the LP relaxations the column generation scheme, and thus its computation times are like those of the TS-Column method. In Model A, UTA-improved and UTA-generic perform much better (each method in about 60% of all instances) than the other methods. Clear to see is not only their advantage over the TS-Simplex method, but also over the TS-Column algorithm. Here, the barrier method delivers similar results like the simplex-based two-step approach. The schedules with the best makespans in Model A, however, are computed with the two-step approaches. Here, TS-Barrier produces the best makespans among all instances followed by TS-Column and TS-Simplex. In Model B with correlated jobs, unfortunately, our combinatorial algorithms are not so efficient as in the previous case. The UTA-generic performs better than the improved implementation and delivers best solutions only for 20% of all instances. The fastest method here is TS-Column which is the best for almost 75% of all

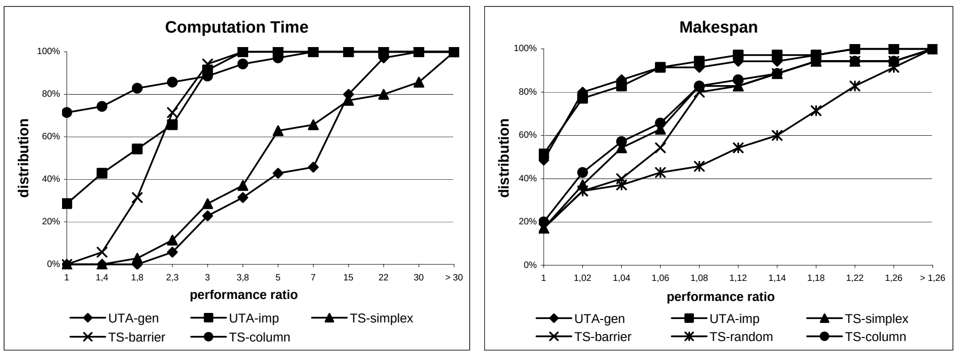


Figure 6.4: Performance diagrams for Model C.

instances. This method results also with the best computation times in Model C. Only for small instances ($n \leq 500, m \leq 50$), TS-Column is outperformed by other methods. TS-Simplex and TS-Barrier achieve in Model B the worst processing times. In Model B and C, UTA-generic performs better than TS-Simplex. In Model C, UTA-improved is much better than UTA-generic. This fact can be explained by the usage of the improved interphase initialization (see Section 6.2.2). A detailed explanation of this phenomenon is given in the last paragraph of this section. In Model B and C, both Unsplittable-Truemper algorithms produce schedules with the best makespans. Their advantage over other methods is clearly to observe in Model B where only in about 8% of all instances they return worse makespans.

**Computation time in dependence of the problem size.** Figure 6.5 shows for each test model the average computation time (of five different instances of the same size) in function of the problem size $(n, m)$. The number of jobs is set to $n = 10m$. The motivation for this is the same like that given in the previous paragraph. The diagrams show on x-axis the number of edges, $A$, in the bipartite graph of the scheduling problem (see Section 3.3). Note that the number of edges of not restricted scheduling problem is equal to $nm$. Therefore, an increase of $A$ can be interpreted as a growth of the scheduling problem. All presented in Figure 6.5 and 6.6 graphs are enriched with appropriate trend lines (of polynomial or exponential character) to make the analysis easier.

In the case when the instances grow, both versions of the Approximation-Unsplittable-Truemper algorithm perform in Model A and B much better than the TS-Simplex and TS-Barrier techniques (they have exponential character). Here, they also are the best solution methods (in Model A even much better than TS-Column). Moreover, in Model A, the sizes of instances for TS-Simplex and TS-Barrier are radically limited. Because of the heavy usage of operational memory, both *simplex* and *barrier* procedures are not able to solve instances larger than (5000, 500) on the second system with 1GB of RAM (cf. Section 6.2.4). Also in cases where it is still possible to deliver solutions, the computation times which they need are very long (more than 1000 sec.) in comparison with those of other methods. This is the reason why we do not depicted them on the graphs. The bad behavior of these methods again can be explained by the increased usage of many time-consuming memory operations. In Model C, unfortunately, the UTA-generic algorithm is the worst technique among all tested methods, whereas TS-Column is the best method. In Model

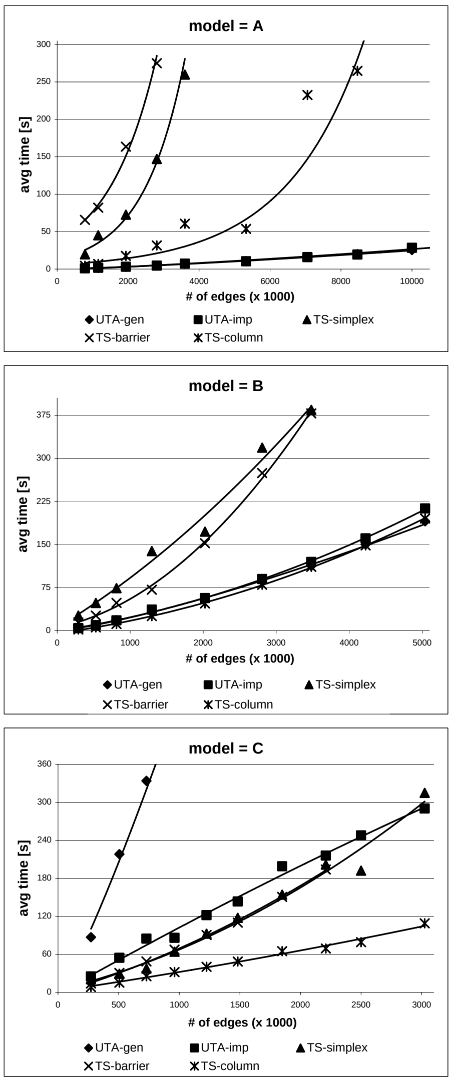


Figure 6.5: Computation time with variable number of edges.

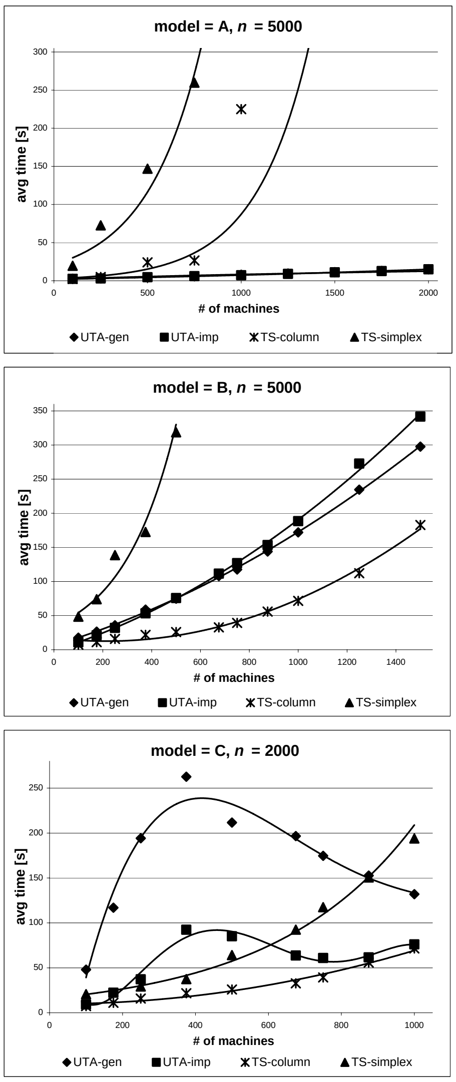


Figure 6.6: Computation time with constant number of jobs.

C, the performance of UTA-IMPROVED is due to improved interphase initialization comparable with that of TS-SIMPLEX or TS-BARRIER. Nevertheless, for more difficult instances, like those from Model B and C, both UNSPLITTABLE-TRUEMPER implementations need far less memory, and thus perform more stable and are easier to handle. For example in Model C the TS-SIMPLEX method needs approximately 10 times more operational memory than each implementation of the APPROXIMATION-UNSPLITTABLE-TRUEMPER algorithm.

**Computation time of instances with fixed number of jobs.** Figure 6.6 shows for each test model the average computation time (of five different instances of the same size) in function of the number of machines while the number of jobs stays constant. This corresponds to a situation when the scheduling system evolves and the jobs become more possibilities to be processed.

In Model A, both versions of the Approximation-Unsplittable-Truemper algorithm perform very similar and are much better than TS-Column and TS-Simplex methods. In Model B, our combinatorial algorithms are outperformed by TS-Column. The results for Model C indicate an huge influence of the interphase initialization on the computation time of the Approximation-Unsplittable-Truemper. Here, UTA-generic is much worse than UTA-improved. The bad performance of UTA-generic can be explained by the enormous number of inner `while`-loops caused by the unbalanced initial assignments. This again can be explained by the special property which the initial assignment has to fulfill (i.e., the reduced cost optimality condition must be maintained, see Section 3.5.1) and the character of Model C (i.e., machines with small $a_j$ allow for very low processing times $p_{ij}$, see Section 6.2.3), the jobs can be assigned initially only to a few machines, leaving other machines empty. This in turn results in a highly unbalanced initial assignment with a large makespan. Furthermore, the same initialization is used throughout all computations at the beginning of every binary search step. Consequently, much larger workload, in comparison with other models, is given to UTA-generic.

In Figure 6.6 we do not consider TS-Barrier, since the computation times for this method, but also for TS-Simplex, have been much longer than the processing times of the other methods. This can again be explained by a much larger number of time-consuming memory operations. More specifically, the experiments show that both methods need in average 10 times more operational memory in comparison with other methods what significantly reduces their usability.

Another interesting observation is the *concave* shape of the graphs for both versions of Approximation-Unsplittable-Truemper in Model C. It indicates a highly desired *natural* behavior of this method, i.e, the decrease of computation time when, roughly speaking, the scheduling problem becomes more resources to allocate. Note that such a situation does take place in Model C where by an increase in the number of machines increases simultaneously the number of assignment possibilities for the jobs.

### 6.3.2 Quality of Approximation-Unsplittable-Truemper's Solutions

We test now other LP-based algorithms, like the Branch-and-Price (B&P) and Cutting-Plane (CP) techniques, which we use for computation of exact optimal solutions for the $R||C_{\max}$ problem. Here, we present the experimental results for these methods, and, using them, compare the quality of solutions produced by them with those produced by our combinatorial algorithms, UTA-generic and UTA-improved, and by the best two-step approach, TS-Column. In this part of simulations we are also interested in the evaluation of performance of the cooperation scheme which we use in the implementation of the Branch-and-Price method. To this end, we compare it especially carefully with the Cutting-Plane method.

The algorithms are tested on a broad range of instances from models A, B and C, with $m$ and $n$ varying gradually from 3 to 20, and from 30 to 200, respectively. For each combination of $(n,m)$, we generate 10 different instances. In Table 6.1, 6.2 and 6.3, one for each test model, we compare five of the above mentioned algorithms with each other. For

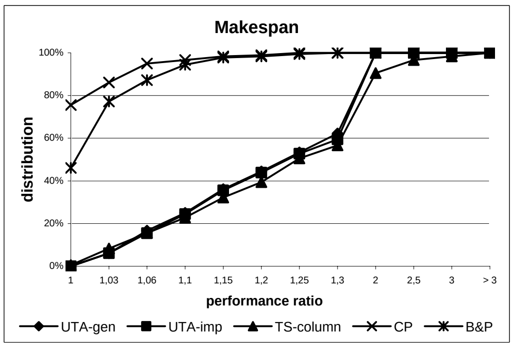


Figure 6.7: Performance diagram for makespan in Model A.

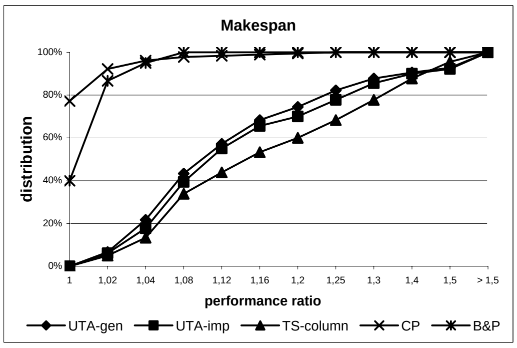


Figure 6.8: Performance diagram for makespan in Model B.

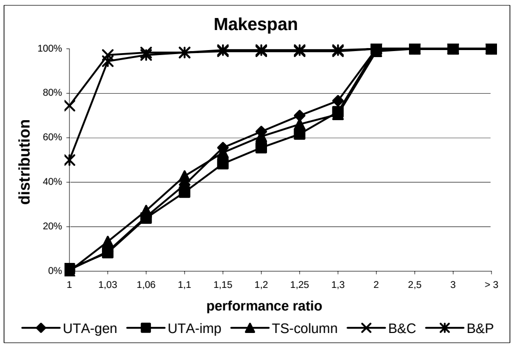


Figure 6.9: Performance diagram for makespan in Model C.

methods Branch-and-Price and Cutting-Plane, we set the time limits at 120 sec. When the optimal solution cannot be computed within that time limit, we take the best integer solution computed so far. For different sizes $(n, m)$ of test instances, the entries in each table give the average values over 10 test instances of the deviation with respect to the optimal makespan (obtained with Cutting-Plane algorithm with no time limit), and the computation time required for each of the five evaluated algorithms. The average percentage deviation for an algorithm $A$ has been computed as follows:

$$Error^A = \frac{1}{10} \sum_{t \in [10]} \frac{T_t^A - T_t^*}{T_t^*} \cdot 100\,[\%],$$

where $T_t^*$ and $T_t^A$ are the optimal and approximate makespan values of each test instance $t$, respectively. Because all computation times of UTA-generic, UTA-improved, and TS-Column have been at most 0.01 sec., we do not include them into the tables. Furthermore, in order to analyze the collective behavior of the algorithms across all tested problem sizes (see Section 6.3.1), we constructed for each test model the performance diagrams for the makespan. The corresponding graphs are given in Figure 6.7, 6.8 and 6.9.

The results given in Figure 6.7, 6.8 and 6.9 show that the quality of solutions obtained with both versions of our combinatorial approach are the same like those of TS-Column in Model A, or better then those of TS-Column in Model B and Model C. The three methods

Table 6.1: Average percentage makespan deviation and computation time for Model A.

| | | UTA-GEN | UTA-IMP | TS-COLUMN | CP | | B&P | |
|---|---|---|---|---|---|---|---|---|
| *m* | *n* | *Error* | *Error* | *Error* | *Error* | *Time* | *Error* | *Time* |
| 3 | 30 | 7.74 | 7.74 | 7.73 | 0.07 | 0.09 | 0.35 | 0.06 |
| | 50 | 5.33 | 5.33 | 10.07 | 0.04 | 0.15 | 0.14 | 0.10 |
| | 80 | 3.46 | 3.46 | 4.58 | 0.06 | 24.84 (2) | 0.25 | 0.21 |
| | 100 | 4.98 | 5.00 | 3.60 | 0.00 | 0.32 | 0.18 | 0.38 |
| | 200 | 1.77 | 1.47 | 2.52 | 0.00 | 1.10 | 0.00 | 0.68 |
| 5 | 30 | 17.93 | 17.78 | 31.72 | 0.38 | 21.38 | 3.28 | 0.08 |
| | 50 | 16.91 | 19.27 | 18.47 | 0.13 | 24.70 (2) | 0.71 | 0.17 |
| | 80 | 9.37 | 9.37 | 17.85 | 0.00 | 1.72 | 0.46 | 0.25 |
| | 100 | 8.80 | 8.80 | 11.47 | 0.05 | 25.88 (2) | 0.76 | 0.33 |
| | 200 | 6.29 | 6.29 | 5.32 | 0.00 | 10.20 | 0.09 | 2.19 |
| 10 | 30 | 40.75 | 42.85 | 71.05 | 0.85 | 35.58 (2) | 4.26 | 0.21 |
| | 50 | 29.00 | 28.70 | 42.07 | 0.59 | 42.31 (3) | 3.47 | 0.27 |
| | 80 | 20.14 | 20.14 | 38.39 | 0.25 | 37.39 (3) | 1.72 | 0.37 |
| | 100 | 27.66 | 31.92 | 28.61 | 0.00 | 25.10 | 1.38 | 0.48 |
| | 200 | 28.02 | 26.76 | 13.02 | 0.95 | 100.71 (6) | 0.63 | 30.1 |
| 15 | 30 | 27.00 | 27.00 | 75.09 | 0.77 | 38.90 (2) | 8.23 | 0.21 |
| | 50 | 34.69 | 40.69 | 95.40 | 0.67 | 49.49 (3) | 4.85 | 0.40 |
| | 80 | 29.62 | 30.21 | 51.10 | 0.53 | 55.82 (3) | 4.85 | 0.39 |
| | 100 | 29.09 | 36.74 | 39.37 | 1.60 | 113.51 (6) | 2.78 | 0.78 |
| | 200 | 40.75 | 45.63 | 22.09 | 1.97 | 120.54 (10) | 2.00 | 57.28 (3) |
| 20 | 30 | 48.71 | 48.71 | 165.97 | 4.05 | 64.85 (5) | 5.26 | 0.22 |
| | 50 | 48.38 | 48.38 | 85.03 | 8.98 | 96.78 (8) | 3.64 | 0.40 |
| | 80 | 60.58 | 60.58 | 105.43 | 7.25 | 103.61 (9) | 10.16 | 0.74 |
| | 100 | 34.86 | 36.78 | 73.11 | 2.46 | 101.79 (9) | 2.67 | 0.94 |
| | 200 | 40.88 | 41.62 | 32.20 | 5.70 | 120.81 (10) | 4.60 | 3.80 |

*Note*. When the time limit was reached, the number of such cases is shown in brackets.

are, unfortunately, outperformed by the exact algorithms. Here, the CUTTING-PLANE method produces more (for about 75% of all instances) better solutions than the BRANCH-AND-PRICE approach (for about 45% of all instances).

When we look more carefully at the results summarized in Table 6.1, 6.2 and 6.3, and additionally on the results from Section 6.3.1, then we can conclude that the exact methods based on branch-and-bound approach are efficient only for very small problem instances from Model A. The good quality of the solution is in vast majority of test instances obtained due to very long computation times, often exceeding the allowed time limits. Because of these long computation times, we do not even test the exact algorithms for larger problem sizes like those from the previous section. For large instances, we have to wait much longer for good solutions, which quality would be comparable with the quality obtained in much shorter time by APPROXIMATION-UNSPLITTABLE-TRUEMPER. Furthermore, observe that the average makespan deviation of solutions obtained with the UTA-GENERIC and UTA-IMPROVED algorithms becomes smaller when the number of machines $m$ increases while the number of jobs $n$ stays fixed. In other words, the algorithms compute better solutions, when the number of assignment possibilities for the jobs increases. This again indicate here their

Table 6.2: Average percentage makespan deviation and computation time for Model B.

| | | UTA-gen | UTA-imp | TS-Column | CP | | B&P | |
|---|---|---|---|---|---|---|---|---|
| *m* | *n* | *Error* | *Error* | *Error* | *Error* | *Time* | *Error* | *Time* |
| 3 | 30 | 6.73 | 6.73 | 8.68 | 0.18 | 48.71 (4) | 0.26 | 0.07 |
| | 50 | 3.20 | 3.20 | 5.50 | 0.21 | 24.59 (2) | 0.08 | 0.20 |
| | 80 | 2.28 | 2.84 | 4.40 | 0.00 | 1.06 | 0.04 | 1.37 |
| | 100 | 1.95 | 1.95 | 2.35 | 0.00 | 1.19 | 0.01 | 3.36 |
| | 200 | 1.10 | 1.23 | 1.20 | 0.00 | 7.70 | 0.00 | 5.94 |
| 5 | 30 | 11.76 | 11.75 | 16.50 | 0.17 | 65.09 (4) | 0.79 | 0.56 |
| | 50 | 7.09 | 8.45 | 10.74 | 0.00 | 58.25 (1) | 0.50 | 11.89 |
| | 80 | 5.51 | 5.22 | 7.21 | 0.49 | 79.26 (6) | 0.04 | 32.67 |
| | 100 | 4.39 | 3.84 | 5.43 | 0.22 | 120.98 (10) | 0.05 | 90.56 (7) |
| | 200 | 2.73 | 3.52 | 2.82 | 1.11 | 120.52 (10) | 0.31 | 118.35 (8) |
| 10 | 30 | 26.69 | 30.92 | 29.80 | 1.09 | 120.96 (10) | 4.96 | 2.52 |
| | 50 | 17.02 | 18.85 | 23.47 | 0.95 | 120.95 (10) | 1.31 | 40.09 (2) |
| | 80 | 10.83 | 9.44 | 16.08 | 1.02 | 120.96 (10) | 0.60 | 121.48 (10) |
| | 100 | 7.74 | 10.51 | 12.29 | 0.26 | 120.75 (10) | 0.70 | 121.40 (10) |
| | 200 | 4.95 | 6.65 | 6.40 | 3.55 | 120.50 (10) | 0.51 | 121.45 (10) |
| 15 | 30 | 59.34 | 59.34 | 52.73 | 1.19 | 97.33 (3) | 3.19 | 1.35 |
| | 50 | 28.95 | 27.69 | 37.15 | 1.39 | 121.28 (10) | 3.07 | 57.71 (4) |
| | 80 | 17.72 | 22.42 | 26.57 | 0.79 | 121.15 (10) | 1.62 | 121.21 (10) |
| | 100 | 13.84 | 15.41 | 17.52 | 1.53 | 120.75 (10) | 1.35 | 121.18 (10) |
| | 200 | 7.31 | 6.59 | 9.15 | 0.80 | 120.97 (10) | 1.19 | 121.24 (10) |
| 20 | 30 | 85.31 | 85.31 | 47.75 | 5.43 | 63.50 (4) | 1.35 | 6.87 |
| | 50 | 32.29 | 33.98 | 42.54 | 1.11 | 121.28 (10) | 3.67 | 44.08 (2) |
| | 80 | 23.36 | 28.67 | 32.89 | 0.82 | 121.10 (10) | 1.79 | 120.90 (10) |
| | 100 | 20.01 | 21.69 | 24.19 | 2.84 | 120.73 (10) | 2.19 | 121.01 (10) |
| | 200 | 10.60 | 12.02 | 14.24 | 1.64 | 120.82 (10) | 1.00 | 121.11 (10) |

*Note*. When the time limit was reached, the number of such cases is shown in brackets.

natural behavior (cf. the discussion in the last paragraph of Section 6.3.1).

Finally, we want to point out a very good performance of the Branch-and-Price method. Its performance in all models is comparable to that of the Cutting-Plane algorithm which is so far the best exact method proposed for the $R||C_{\max}$ problem. This surprisingly good behavior can be explained by the usage of the acceleration scheme using both mixed integer programming and local search.

### 6.3.3 Comparison with Heuristics

In the last section of this chapter we want to present the comparison of results computed by the Approximation-Unsplittable-Truemper algorithms with those obtained with various heuristics developed for the $R||C_{\max}$ problem in the course of the last decades. As it was already indicated in Section 6.2.1, here we take into consideration ten different heuristics, i.e., the four heuristics by Ibarra and Kim, one heuristic by Davis and Jaffe, the partial enumeration method by Mokotoff and Jimeno, and the branch-and-greed heuristic by Sourd. Additionally to them, we have implemented three adapted for $R||C_{\max}$ versions of the LPT

Table 6.3: Average percentage makespan deviation and computation time for Model C.

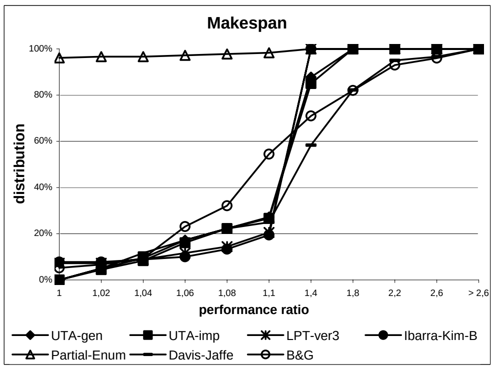
| | | UTA-GEN | UTA-IMP | TS-COLUMN | CP | | B&P | |
|---|---|---|---|---|---|---|---|---|
| *m* | *n* | *Error* | *Error* | *Error* | *Error* | *Time* | *Error* | *Time* |
| 3 | 30 | 14.22 | 17.20 | 6.79 | 0.06 | 26.37 (2) | 0.41 | 1.27 |
| | 50 | 19.20 | 19.20 | 3.34 | 0.12 | 73.87 (6) | 0.36 | 73.05 (6) |
| | 80 | 2.02 | 3.53 | 3.56 | 0.02 | 32.06 (2) | 0.10 | 29.60 (2) |
| | 100 | 11.32 | 15.35 | 3.56 | 0.03 | 35.21 (2) | 0.15 | 38.01 (2) |
| | 200 | 24.86 | 32.02 | 2.07 | 0.00 | 47.88 | 0.02 | 26.09 (2) |
| 5 | 30 | 13.78 | 13.70 | 12.31 | 0.48 | 75.13 (6) | 0.17 | 51.16 (4) |
| | 50 | 9.41 | 9.13 | 16.14 | 0.13 | 51.53 (4) | 0.17 | 51.16 (4) |
| | 80 | 6.75 | 6.22 | 4.40 | 0.46 | 120.93 (10) | 0.66 | 99.43 (8) |
| | 100 | 7.54 | 6.40 | 8.50 | 0.54 | 120.66 (10) | 0.51 | 100.46 (8) |
| | 200 | 9.48 | 21.75 | 4.83 | 0.46 | 110.56 (4) | 0.43 | 121.42 (10) |
| 10 | 30 | 31.07 | 33.04 | 29.82 | 1.80 | 73.01 (6) | 1.82 | 49.08 (4) |
| | 50 | 27.21 | 27.06 | 34.62 | 2.25 | 120.56 (10) | 2.33 | 108.27 (6) |
| | 80 | 14.92 | 14.73 | 16.95 | 1.90 | 120.67 (10) | 2.82 | 121.64 (10) |
| | 100 | 16.13 | 13.51 | 23.50 | 1.59 | 120.71 (10) | 2.41 | 121.40 (8) |
| | 200 | 13.26 | 14.50 | 10.02 | 0.97 | 120.70 (10) | 0.90 | 121.46 (10) |
| 15 | 30 | 31.46 | 37.04 | 59.74 | 3.01 | 74.88 (6) | 4.39 | 49.19 (4) |
| | 50 | 44.99 | 45.05 | 39.75 | 5.83 | 121.29 (10) | 4.88 | 121.63 (10) |
| | 80 | 26.66 | 38.63 | 39.84 | 10.31 | 121.12 (10) | 4.21 | 121.20 (10) |
| | 100 | 22.98 | 30.20 | 36.20 | 1.81 | 121.12 (10) | 2.14 | 121.44 (10) |
| | 200 | 11.04 | 12.23 | 11.11 | 1.24 | 120.88 (10) | 1.08 | 121.23 (10) |
| 20 | 30 | 37.44 | 44.44 | 80.59 | 1.81 | 84.40 (6) | 4.34 | 22.46 |
| | 50 | 57.76 | 133.42 | 59.73 | 6.08 | 121.30 (10) | 6.56 | 121.26 (10) |
| | 80 | 55.90 | 59.92 | 53.01 | 4.30 | 121.35 (10) | 6.92 | 121.21 (10) |
| | 100 | 29.83 | 34.60 | 30.56 | 2.90 | 121.08 (10) | 4.96 | 121.23 (10) |
| | 200 | 26.78 | 23.81 | 26.15 | 4.27 | 121.00 (10) | 3.66 | 121.22 (10) |

*Note*. When the time limit was reached, the number of such cases is shown in brackets.

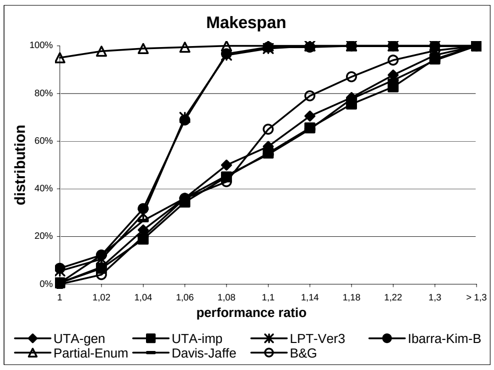
algorithm by Graham. This algorithm is known to perform well for the scheduling problem on related parallel machines (cf. Section 1.3.3). Because of the similarity of Model C to that scheduling problem, we add these adaptations into the experimental setup.

The algorithms are tested on a broad range of small test instances from Model A, B, and C with $m$ and $n$ varying from 3 to 20, and from 30 to 200, respectively. For each combination of $(n, m)$, we generate 10 instances. In Figure 6.10 we present for each model the makespan performance diagrams. In order to make the analyze more visible, the performance diagrams include, in each model, only one graph for the best Ibarra-Kim algorithm and one for the best LPT adaptation.

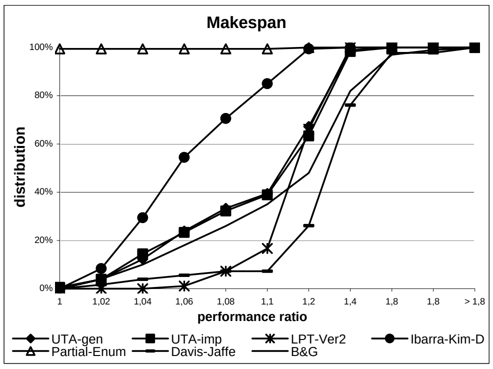
In Figure 6.10 we can see that the best results in all models are computed with the heuristic using partial enumeration by Mokotoff and Jimeno. In the implementation of this method, we sort the list of non-basic assignment variables for the MIP subproblem in non-increasing order of the simplex reduced costs (see Section 4.2). The time limit for the *branch-and-cut* algorithm solving the MIP subproblem in the PARTIAL-ENUMERATION method is set at 120 sec., and the value of parameter $r$ is set to 0.5 (this value has been found during a preliminary series of tests with different values of $r$). Although the PARTIAL-



Model A



Model B



Model C

Figure 6.10: Performance diagrams for heuristics.

ENUMERATION heuristic computes here the best solutions among other techniques, its computation times are much longer. Almost for all test instances, the specified time limit is reached (about 120 sec.), whereas the computation times used both by the other heuristics and by the combinatorial algorithms UTA-GENERIC and UTA-IMPROVED are much less than 0.1 sec. In other words, the PARTIAL-ENUMERATION method takes advantage of the longer computation time in order to deliver better solutions.

In Model A and Model B the makespans obtained with our new combinatorial algorithms are similar to those computed by the DAVIS-JAFFE and BRANCH-AND-GREED heuristics. In Model B, the IBARRA-KIM-B and LPT-VER3 heuristics compute almost the same results, and both are the second best heuristics for this model. In Model C, however, the IBARRA-KIM-D technique is the second best heuristic after the PARTIAL-ENUMERATION method. It delivers half as many good solutions as the PARTIAL-ENUMERATION heuristic. In the same model both APPROXIMATION-UNSPLITTABLE-TRUEMPER implementations compute better schedules than BRANCH-AND-GREED, LPT-VER2, and DAVIS-JAFFE heuristics.

When we take into account that the computation times of the best heuristic here, PARTIAL-ENUMERATION, are much longer than the computation times needed by the other algorithms, then the results obtained by the combinatorial algorithms, UTA-GENERIC and UTA-IMPROVED, show that they can be as efficient as the heuristic algorithms, and thus can successfully compete with them in order to solve difficult scheduling problems.

# 7

# Conclusions and Open Questions

The problem of scheduling unrelated parallel machines forms an important and ongoing field of research. This is shown by a multitude of theoretical results and practical applications going back to the beginning of the last century. In the current competitive environment with rapidly changing conditions, effective scheduling has become a necessity for survival in the marketplace. In most cases, the choice of a *good* schedule has an immense impact on the system performance and the invested costs.

This work aims to bring more light into the field of scheduling unrelated parallel machines. To this end, we investigate the most basic but on the other hand the most general deterministic scheduling problem with machines in parallel, i.e., the problem of non-preemptive scheduling independent jobs on unrelated parallel machines with the goal to minimize the makespan ($R||C_{\max}$). Our choice is motivated by the importance of this problem for the research both from the theoretical and the practical point of view. While the problem might seem rather simple, it covers quite a lot of common cases.

Our goal is to present and improve the current state-of-the-art of the research on this problem. In particular, we try both to expand the collection of old results with new ones, and to evaluate and compare already existing solution techniques in concrete and well defined real-life settings in order to get a better understanding of the scheduling problem. Since an exact list of our results can be found in Section 1.5, we refrain here from repeating them in detail. Instead, we try to provide the reader with new highlights brought by this work and to give an outlook for future research.

One of the most interesting contributions of this work is a new, purely combinatorial 2-approximation algorithm for the $R||C_{\max}$ problem. We formulate the unrelated scheduling as a generalized flow problem in a bipartite network. Our approximation algorithm is a generic minimum cost flow algorithm without any complex enhancements. The minimum cost flow algorithm has been tailored to handle unsplittable flow by exploiting the special bipartite structure of the underlying flow network.

Many real world applications can be modeled as flow problems in a bipartite network (see, e.g., [3, 18, 139]). For this reason, we conjecture that our approach might also be helpful for other applications. Identifying the connection to network flows might be the key idea for obtaining combinatorial (approximation) algorithms for problems for which solving the LP relaxation and rounding is currently the (only) alternative. Our techniques and results do not improve upon the approximation factor for the unrelated scheduling problem. However, we still expect improvements for other hard problems to which our technique is applicable.

An interesting open problem would be to apply our techniques to more general scheduling problems. As the first step toward this end, we have already tried to use our approach to solve the *general assignment problem* (GAP). Here, for each parallel machine $j$ an upper bound $T_j$ for its load is specified, and each job is characterized, additionally to the processing time $p_{ij}$, by a cost $c_{ij}$. The goal is to assign each job to some machine without preemption so that the load constraints for each machine are maintained and the overall costs induced by the assigned jobs are minimized. It is easy to see that this problem is a generalization of the $R||C_{\max}$ problem. Unfortunately, we have only succeed to develop an algorithm for checking the existence of a 2-approximation solution for a given instance of GAP. Thus, the general optimization problem still stays open.

Another interesting open question is how to extend our combinatorial algorithm so it can be applied to solve the unrelated scheduling problem with additional parameters, such as precedence constraints, setup times, or due dates. At this moment, however, it is hard to see any possibility for any such improvements.

Since there still exists a possibility for improving the approximation factor for the $R||C_{\max}$ problem (recall from Section 1.4.1 that there exists a gap between the lower bound of $\frac{3}{2}$ and the best, so far, approximation factor of $2 - \frac{1}{m}$), it would be challenging either to design a new polynomial-time approximation algorithm for $R||C_{\max}$ with worst-case performance of $2 - \frac{1}{m} - \Delta$, or to provide a $\frac{3}{2} + \Delta$ inapproximability result for the $R||C_{\max}$ problem. Here, $\Delta$ denotes some small positive real number that does not depend on the input.

In addition to the combinatorial deterministic algorithm, we introduce two other algorithms. The first one is a randomized version of the classical two-step approach for the $R||C_{\max}$ problem. In our opinion it would be interesting to prove its probabilistic guarantee. The result by Raghavan and Thompson [145] can be a good basis for such probabilistic analysis. An interesting result in this regard has also been presented by Schultz and Skutella [152]. Their more specific variant of randomized rounding leads to a $(2 + \varepsilon)$-approximation algorithm for the $R|r_i|\sum w_i C_i$ problem (case with release times $r_i$ and the minimization of the total weighted completion time as the objective), and a $(\frac{3}{2} + \varepsilon)$-approximation algorithm for the $R||\sum w_i C_i$ problem. We think that their analysis could be adapted for the $R||C_{\max}$ problem.

The second algorithm is a branch-and-price approach for $R||C_{\max}$. Although we use several heuristic improvements which help it to find good integer solutions earlier, there still exists enough room to improve it further, e.g., by applying another local search metaheuristic like, e.g., guided taboo search or simulated annealing, or by a better parameter setting. Moreover, our implementation of the branch-and-price algorithm uses the variable dichotomy branching strategy which results in highly unbalanced search trees (cf. Section 5.2.3). It would be of special interest to apply and evaluate another branching strategy like, e.g., the GUB dichotomy method (cf. Section 5.2.3). Finally, the algorithm in the present state with the variable dichotomy branching strategy and the best-first exploration strategy can relatively easy be turned into an effective approximation algorithm for $R||C_{\max}$. Some hints for this are already given in Section 5.2.4.

In the thesis we present also a comprehensive evaluation of eighteen different algorithms for the $R||C_{\max}$ problem. Our goals here are threefold. First of all, our efforts are concentrated on the performance and the quality of solutions obtained with the Approximation-Unsplittable-Truemper algorithm in comparison with the two-step approaches based on linear programming and used often in practice as the first-choice solution methods. During the tests, a large number of experiments is performed with very large instances. To our best knowledge, it is the first time when instances of such sizes are considered in experimental studies on algorithms for $R||C_{\max}$. Our results indicate a clear advantage of the improved version of our combinatorial algorithm over LP-based techniques. In particular, for large test instances, it delivers solutions of better quality, works in most cased much faster, and requires much less operational memory. The last property makes it more efficient and easier to handle, especially when the size of the problem grows. Only for model with correlated machines our technique shows some weakness. Even though the improved interphase initialization eliminates this unfavorable behavior apparently, we still expect further improvements in the initialization based on earlier interruption of computations which may lead to the performance comparable with or better than the performance of methods using column generation.

The second goal of the experimental study is to compare our algorithm with exact methods for the $R||C_{\max}$ problem. The experiments indicate that the combinatorial algorithm is outperformed by them only when the quality of solution is considered. However, a better look at the results let us conclude that the time performance of exact methods based on branch-and-bound approach is satisfactory only for very small problem instances. The good quality of solutions is obtained due to very long computation times. Hence, for large instances we need to wait much longer for solutions whose quality would be comparable with the quality obtained in much shorter time by Approximation-Unsplittable-Truemper. Furthermore, it is worth to point out a very good performance of our branch-and-price algorithm. Its performance in all models is similar to that of the cutting plane algorithm, which is so far the best exact method proposed for $R||C_{\max}$. In our opinion it should not be difficult but rather time-consuming to improve our exact algorithm in order to beat the cutting plane method.

Finally, the last goal of the simulations constitute the comparison tests with various heuristics developed for $R||C_{\max}$ in the past decades. They show, in contrary to our expectations, that the Approximation-Unsplittable-Truemper algorithm can perform as efficiently as the heuristic algorithms, and thus can successfully compete with them in solving difficult scheduling problems.

# Bibliography

[1] E. Aarts and J. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[2] R. K. Ahuja, O. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[4] N. Alon, J. H. Spencer, and P. Erdös. *The Probabilistic Method*. Wiley-Interscience Series, John Wiley & Sons, Inc., New York, 1992.

[5] R. Anbil, J. J. Forrest, and W. R. Pulleyblank. Column generation and the airline crew pairing problem. In *Proceedings of the International Congress of Mathematicians Berlin, Extra Volume ICM 1998 of Doc. Math. J. DMV*, pages III 677–686, 1998.

[6] R. Anbil, E. L. Johnson, and R. Tanga. A global approach to crew-pairing optimization. *IBM Systems Journal*, 31:71–78, 1992.

[7] L. H. Appelgren. A column generation algorithm for a ship scheduling problem. *Transportation Science*, 3:53–68, 1969.

[8] R. Armacost and A. Salem. Unrelated parallel-machines scheduling with setup times and machine eligibility restrictions. In *Proceedings of IERC Conference*, 1999.

[9] Y. Azar and O. Regev. Strongly polynomial algorithms for the unsplittable flow problem. In *Proceedings of the 8th Conference on Integer Programming and Combinatorial Optimization (IPCO'01)*, LNCS 2081, pages 15–29, 2001.

[10] E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993.

[11] C. Barnhart, E. L. Johnson, A. Krishna, D. Mahidara, G. L. Nemhauser, E. Rebello, A. Singh, and P. H. Vance. *Advanced techniques in crew scheduling*. Presented at INFORMS, Los Angeles, CA, USA, 1995.

[12] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for huge integer programs. *Operations Research*, 46:316–329, 1998.

[13] C. Barnhart, E. L. Johnson, G. L. Nemhauser, and P. H. Vance. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3:111–130, 1994.

[14] R. E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[15] R. E. Bixby, J. W. Gregory, I. J. Lustig, R. E. Marsten, and D. F. Shanno. Very large-scale linear programming: A case study in combining interior point and simplex methods. *Operations Research*, 40(5):885–897, 1992.

[16] E. A. Boyd. Fenchel cutting planes for integer programs. *Operations Research*, 42:53–64, 1994.

[17] O. Briant, C. Lemaréchel, Ph. Meurdesoif, S. Michel, N. Perrot, and F. Vanderbeck. *Comparison of bundle and classical column generation*. Rapport de recherche INRIA 5453, 2005.

[18] P. Brucker. *Scheduling Algorithms*. Springer Verlag, 1995.

[19] P. Brucker. *Complex Scheduling*. Springer Verlag, 2005.

[20] P. Chrétienne, E. G. Coffman, J. K. Lenstra, and Z. Liu. *Scheduling Theory and its Applications*. John Wiley & Sons, 1995.

[21] E. G. Coffman, M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7:1–17, 1978.

[22] R. Conway, W. Maxwell, and L. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.

[23] H. Crowder, E. L Johnson, and M. W. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[24] R. J. Dakin. A tree search algorithm for mixed integer programming problems. *Computer Journal*, 8:250–255, 1965.

[25] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, New York, 1963.

[26] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.

[27] E. Davis and J. M. Jaffe. Algorithms for scheduling tasks on unrelated parallel processors. *Journal of ACM*, 28:721–736, 1981.

[28] D. den Hertog. *Interior Point Approaches to Linear, Quadratic and Convex Programming*. Kluwer Academic Publishers, 1994.

[29] G. Desaulniers, J. Desrosiers, and M. M. Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40:342–354, 1992.

[30] G. Desaulniers, J. Desrosiers, and M. M. Solomon. *Column Generation*. Springer Verlag, 2005.

[31] M. Desrochers and F. Soumis. A column generation approach to the urban transit crew scheduling problem. *Transportation Science*, 23:1–13, 1989.

[32] J. Desrosiers, M. Desrochers, and F. Soumis. Routing with time windows by column generation. *Networks*, 14:545–565, 1984.

[33] C. Dillenberger, L. Escudero, A. Wollensak, and W. Zhang. On practical resource allocation for production planning and scheduling with period overlapping setups. *European Journal of Operational Research*, 75:275–286, 1994.

[34] Y. Dinitz, N. Garg, and M. X. Goemans. On the single-source unsplittable flow problem. *Combinatorica*, 19(1):17–41, 1999.

[35] G. Dobson. Scheduling independent tasks on uniform processors. *SIAM Journal on Computing*, 13(4):705–716, 1984.

[36] A. G. Doig and A. H. Land. An automatic method for solving discrete programming problems. *Econometrica*, 28:493–520, 1960.

[37] O. du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen. Stabilized column generation. *Discrete Mathematics*, 194:229–237, 1999.

[38] P. S. Efraimidis and P. G. Spirakis. Randomized approximation schemes for scheduling unrelated parallel machines. *Electronic Colloquium on Computational Complexity (ECCC)*, 7(7), 2000.

[39] J. Elam, F. Glover, and D. Klingman. A strongly convergent primal simplex algorithm for generalized networks. *Mathematics of Operations Research*, 4:39–59, 1979.

[40] S. Elhedhli and J.-L. Goffin. The integration of an interior-point cutting-plane method within a branch-and-bound algorithm. *Mathematical Programming*, 100(2):267–294, 2004.

[41] L. Epstein and L. M. Favrholdt. Optimal non-preemptive semi-online scheduling on two related machines. *ACM Journal of Algorithms*, 57(1):49–73, 2005.

[42] S. Even. *Graph Algorithms*. Computer Science Press, 1979.

[43] A. A. Farley. A note on bounding a class of linear programming problems, including cutting stock problems. *Operations Research*, 38:922–924, 1990.

[44] A. Fishkin, K. Jansen, and M. Mastrolilli. Grouping techniques for scheduling problems: Simpler and faster. In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA'01)*, pages 206–217, 2001.

[45] L. Fleischer and K. D. Wayne. Fast and simple approximation schemes for generalized flow. *Mathematical Programming*, 91(2):215–238, 2002.

[46] L. R. Ford and D. R. Fulkerson. A suggested computation for maximal multicommodity network flows. *Management Science*, 5:97–101, 1958.

[47] L. R. Ford and D. R. Fulkerson. *Network Flows*. Princeton University Press, Princeton, New York, 1962.

[48] L. R. Ford and D. R. Fulkerson. A primal-dual algorithm for the capacitated hitchcock problem. *Naval Research Logistics Quarterly*, 4:47–53, 1967.

[49] D. K. Friesen. Tighter bounds for multifit processor scheduling algorithm. *SIAM Journal on Computing*, 13:170–181, 1984.

[50] D. K. Friesen. Tighter bounds for LPT scheduling on uniform processors. *SIAM Journal on Computing*, 16(3):554–560, 1987.

[51] D. K. Friesen and M. A. Langston. Bounds for multifit scheduling on uniform machines. *SIAM Journal on Computing*, 12:60–70, 1983.

[52] M. Gairing, T. Lücking, M. Mavronicolas, and B. Monien. Computing nash equilibria for scheduling on restricted parallel links. In *Proceedings of the 36th Annual ACM Symposium on the Theory of Computing (STOC'04)*, pages 613–622, 2004.

[53] M. Gairing, B. Monien, and A. Woclaw. A faster combinatorial approximation algorithm for scheduling unrelated parallel machines. In *Proceedings of 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, pages 828–839, 2005.

[54] H. L. Gantt. *Work, Wages and Profit. Their influence on the cost of living*. The Engineering Magazine Co., New York, 1910.

[55] H. L. Gantt. *Organizing for Work*. Harcourt, Brace and Howe, New York, 1919.

[56] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[57] M. Ghirardi and C. N. Potts. Makespan minimization for scheduling unrelated parallel machines: a recovering beam search approach. *European Journal of Operational Research*, 165(2):457–467, 2005.

[58] P. Glass, C. Potts, and P. Shade. Unrelated parallel machine scheduling using local search. *Mathematical Computing Modeling*, 20(2):41–52, 1994.

[59] P. C. Glimore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9:849–859, 1961.

[60] P. C. Glimore and R. E. Gomory. A linear programming approach to the cutting-stock problem – part II. *Operations Research*, 11:863–888, 1963.

[61] F. Glover, J. Hultz, D. Klingman, and J. Stutz. Generalized networks: A fundamental computer-based planning tool. *Management Science*, 24:12, 1978.

[62] F. Glover and D. Klingman. On the equivalence of some generalized network flow problems to pure network problems. *Mathematical Programming*, 4:269–278, 1973.

[63] J.-L. Goffin, A. Haurie, J.-Ph. Vial, and D. L. Zhu. Using central prices in the decomposition of linear programs. *European Journal of Operational Research*, 64:393–409, 1993.

[64] A. V. Goldberg, S. A. Plotkin, and É. Tardos. Combinatorial algorithms for the generalized circulation problem. *Mathematics of Operations Research*, 16:351–379, 1991.

[65] D. Goldfarb, Z. Jin, and J. B. Orlin. Polynomial-time highest-gain augmenting path algorithms for the generalized circulation problem. *Mathematics of Operations Research*, 22:793–802, 1997.

[66] R. E. Gomory. Outline of an algorithm for integer solutions of linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[67] R. E. Gomory. Solving linear programming problems in integers. In *Proceedings of the Symposium on Applied Mathematics 10*, pages 211–215, 1960.

[68] R. E. Gomory. An algorithm for integer solutions to linear programs. *Recent Advances in Mathematical Programming, McGraw Hill, New York*, pages 269–302, 1963.

[69] M. Gondran and M. Minoux. *Graphs and Algorithms.* Wiley, 1984.

[70] T. Gonzalez, O. H. Ibarra, and S. Sahni. Bounds for LPT schedules on uniform processors. *SIAM Journal on Computing*, 6(1):155–166, 1977.

[71] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

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

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

[74] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32:1195–1220, 1984.

[75] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric algorithms and combinatorial optimization*. Springer Verlag, 1988.

[76] V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd, and M. Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 19–28, 1999.

[77] G. Hadley. *Linear Programming*. Addison-Wesley, 1965.

[78] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Conference on Artificial Intelligence (ICAI'95)*, pages 607–613, 1995.

[79] J. Herrmann, J.-M. Proth, and N. Sauer. Heuristics for unrelated machine scheduling with precedence constraints. *European Journal of Operational Research*, 102:528–537, 1997.

[80] D. S. Hochbaum. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., 1997.

[81] D. S. Hochbaum and D. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34(1):144–162, 1987.

[82] D. S. Hochbaum and D. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539–551, 1988.

[83] K. Hoffman and M. W. Padberg. Improving LP-representations of zero-one linear programs for branch and cut. *ORSA Journal on Computing*, 3:121–134, 1991.

[84] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the Association for Computing Machinery*, 23(2):317–327, 1976.

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

[86] ILOG. *ILOG Concert Technology 2.0 Reference Manual*. ILOG Inc., Gentilly, France, 2003.

[87] ILOG. *ILOG CPLEX 9.0 User's Manual*. ILOG Inc., Gentilly, France, 2003.

[88] J. R. Jackson. An extension of Johnson's results on job lot scheduling. *Naval Research Logistics Quarterly*, 3:201–203, 1956.

[89] K. Jansen and L. Porkolab. Linear time approximation schemes for scheduling malleable parallel tasks. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, pages 490–498, 1999.

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

[91] W. S. Jewell. Optimal flow through networks with gains. *Operations Research*, 10:476–499, 1962.

[92] E. L. Johnson. Modeling and strong linear programs for mixed integer programming. *Algorithms and Model Formulations in Mathematical Programming*, NATO ASI Series 51:1–41, 1989.

[93] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1:61–68, 1954.

[94] M. Jünger, G. Reinelt, and S. Thienel. Practical problem solving with cutting plane algorithms in combinatorial optimization. *Combinatorial Optimization, Dimacs*, 20:111–152, 1995.

[95] S. Kapoor and P. M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing (STOC'86)*, pages 147–159, 1986.

[96] N. Karmarkar. New polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[97] R. M. Karp. *Reducibility Among Combinatorial Problems*. Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, Editors, Plenum Press, New York, 1972.

[98] R. M. Karp and C. H. Papadimitriou. On linear characterizations of combinatorial optimization problems. *SIAM Journal on Computing*, 11:620–632, 1982.

[99] L. G. Khachian. A polynomial time algorithm for linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[100] J. Kleinberg. *Approximation algorithms for disjoint paths problems*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1996.

[101] J. Kleinberg. Single-source unsplittable flow. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, pages 68–77, 1996.

[102] J. Kleinberg and É. Tardos. *Algorithm Design*. Addison-Wesley, 2005.

[103] S. Kolliopoulos and C. Stein. Approximation algorithms for single-source unsplittable flow. *SIAM Journal on Computing*, 31:919–946, 2002.

[104] S. G. Kolliopoulos and C. Stein. Improved approximation algorithms for unsplittable flow problems. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS'97)*, pages 426–435, 1997.

[105] P. Kolman and Ch. Scheideler. Improved bounds for the unsplittable flow problem. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 184–193, 2002.

[106] G. Lancia. Scheduling jobs with release dates and tails on two unrelated parallel machines to minimize the makespan. *European Journal of Operational Research*, 120:277–288, 2000.

[107] M. A. Langston. *Processor scheduling with improved heuristic algorithms*. PhD thesis, Texas A&M University, College Station, Texas, 1981.

[108] L. S. Lasdon. *Optimization Theory for Large Systems*. Macmillan, London, 1970.

[109] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart and Winston, New York, 1976.

[110] E. L. Lawler and J. Labetoulle. On preemptive scheduling of unrelated parallel processors by linear programming. *Journal of the ACM*, 25:612–619, 1978.

[111] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of ACM*, 46(6):787–832, 1999.

[112] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[113] J. H. Lin and J. S. Vitter. $\epsilon$-approximations with minimum packing constraint violation. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC'92)*, pages 771–782, 1992.

[114] J. W. S. Liu and C. L. Liu. Bounds on scheduling algorithms for heterogeneous computing systems. *Information Processing, J. L. Rosenfeld, ed., North-Holland, Amsterdam*, 74:349–353, 1974.

[115] M. E. Lübbecke and J. Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023, 2005.

[116] R. Mansini and M. G. Speranza. Heuristic algorithms for the portfolio selection problem with minimum transaction lots. *European Journal of Operational Research*, 114:219–233, 1999.

[117] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[118] R. Marsten. *Crew Planning at Delta Airlines*. Presentation at Mathematical Programming Symposium XV, Ann Arbor, MI, USA, 1994.

[119] S. Martello, I. Osman, C. Roucairol, and S. Voß. *Metaheuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, 1999.

[120] S. Martello, F. Soumis, and P. Toth. Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete Applied Mathematics*, 75:169–188, 1997.

[121] R. Kirkeby Martinson and J. Tind. An interior point method in Dantzig-Wolfe decomposition. *Computers and Operations Research*, 26(12):1195–1216, 1999.

[122] P. Meseguer. Interleaved depth-first-search. In *Proceedings of the 15th International Conference on Artificial Intelligence (ICAI'97)*, pages 1382–1387, 1997.

[123] P. Miliotis. Integer programming approaches to the traveling salesman problem. *Mathematical Programming*, 10:367–378, 1976.

[124] P. Miliotis, G. Laporte, and Y. Nobert. Computational comparison of two methods for finding the shortest complete cycle or circuit in a graph. *Operations Research*, 15:233–239, 1980.

[125] E. Mokotoff and P. Chrétienne. A cutting plane algorithm for the unrelated parallel machine scheduling problem. *European Journal of Operational Research*, 141:515–525, 2002.

[126] E. Mokotoff and J. L. Jimeno. Heuristics based on partial enumeration for the unrelated parallel processor scheduling problem. *Annals of Operations Research*, 117:133–150, 2002.

[127] B. Monien and A. Woclaw. Scheduling unrelated parallel machines. Computational results. In *Proceedings of 5th International Workshop on Experimental Algorithms (WEA'06)*, pages 195–206, 2006.

[128] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995.

[129] J. Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

[130] G. L. Nemhauser and L. Wolsey. *Integer and combinatorial optimization*. John Wiley & Sons, 1988.

[131] J. D. Oldham. Combinatorial approximation algorithms for generalized flow problem. *ACM Journal of Algorithms*, 38:135–169, 2001.

[132] K. Onaga. Dynamic programming of optimum flows in lossy communication nets. *IEEE Transactions on Circuit Theory*, 13:308–327, 1966.

[133] K. Onaga. Optimal flows in general communication networks. *J. Franklin Institute*, 283:308–327, 1967.

[134] I. M. Ovacik and R. Uzsoy. Decomposition methods for scheduling semiconductor testing facilities. *International Journal of Flexible Manufacturing Systems*, 8:357–388, 1996.

[135] M. W. Padberg and G. Rinaldi. Optimization of a 532 city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6:1–7, 1987.

[136] Ch. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[137] W. L. Pearn, S. H. Chung, and M. H. Yang. Minimizing the total machine workload for the wafer probing scheduling problem. *IIE Transactions*, 34(2):211–220, 2002.

[138] N. Piersma and W. van Dijk. A local search heuristic for scheduling unrelated parallel machines with efficient neighborhood search. *Mathematical and Computational Modeling*, 24:11–19, 1996.

[139] M. Pinedo. *Scheduling. Theory, Algorithms, and Systems*. Prentice Hall, 1995.

[140] M. Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer Series in Operations Research and Financial Engineering, 2006.

[141] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20(2):257–301, 1995.

[142] C. N. Potts. Analysis of linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics*, 10:155–164, 1985.

[143] T. Radzik. Faster algorithms for the generalized network flow problem. *Mathematics of Operations Research*, 23:69–100, 1998.

[144] T. Radzik. Improving time bounds on maximum generalized flow computations by contracting the network. *Theoretical Computer Science*, 312(1):75–97, 2004.

[145] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[146] T. J. Van Roy and L. A. Wolsey. Valid inequalities for mixed $0-1$ programs. *Discrete Applied Mathematics*, 4:199–213, 1986.

[147] T. J. Van Roy and L. A. Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35:45–57, 1987.

[148] D. M. Ryan and B. A. Foster. An integer programming approach to scheduling. *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*, pages 269–280, 1981.

[149] S. Sahni. Algorithms for scheduling independent tasks. *Journal of the ACM*, 23:116–127, 1976.

[150] M. W. P. Savelsbergh. A branch-and-price algorithm for the generalized assignment problem. *Operations Research*, 45:831–841, 1997.

[151] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1994.

[152] A. S. Schultz and M. Skutella. Scheduling unrelated parallel machines by randomized rounding. *SIAM Journal on Discrete Mathematics*, 15(4):450–469, 2002.

[153] P. Schuurman and G. Woeginger. Polynomial time approximation algorithms for machine scheduling: Ten open problems. *Journal of Scheduling*, 2:203–213, 1999.

[154] S. Seiden, J. Sgall, and G. J. Woeginger. Semi-online scheduling with decreasing job sizes. *Operations Research Letters*, 27(5):215–221, 2000.

[155] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of 4th International Conference on Principles and Practice of Constraint Programming (CP'98)*, pages 417–431, 1998.

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

[157] D. B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.

[158] M. Skutella. Approximating the single source unsplittable min-cost flow problem. *Mathematical Programming, Ser. B*, 91(3):493–514, 2002.

[159] W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[160] F. Sourd. Scheduling tasks on unrelated machines: Large neighborhood improvement procedures. *Journal of Heuristics*, 7:519–531, 2001.

[161] B. Srivastava. An effective heuristic for minimizing makespan on unrelated parallel machines. *Journal of Operational Research Society*, 49:866–894, 1998.

[162] É. Tardos and K. D. Wayne. Simple generalized maximum flow algorithms. In *Proceedings of the 6th Integer Programming and Combinatorial Optimization Conference (IPCO'98)*, pages 310–324, 1998.

[163] M. A. Trick. Scheduling multiple variable-speed machines. In *Proceedings of the 1st Conference on Integer Programming and Combinatorial Optimization*, pages 485–494, 1990.

[164] K. Truemper. On max flows with gains and pure min-cost flows. *SIAM Journal on Applied Mathematics*, 32(2):450–456, 1977.

[165] E. Tsang and C. Voudouris. Fast local search and guided local search and their application to British Telecom's workforce scheduling problem. *Operations Research Letters*, 20:119–127, 1997.

[166] P. M. Vaidya. Speeding up linear programming using fast matrix multiplication. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS'89)*, pages 332–337, 1989.

[167] S. L. van de Velde. Duality-based algorithms for scheduling unrelated parallel machines. *ORSA Journal on Computing*, 5(2):182–205, 1993.

[168] J. M. van den Akker, J.A. Hoogeveen, and S. L. van de Velde. Parallel machine scheduling by column generation. *Operations Research*, 47(6):862–872, 1999.

[169] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*. Mathematics and its Applications. Reidel Publishing Company, 1987.

[170] F. Vanderbeck. *Decomposition and Column Generation for Integer Programs*. PhD thesis, Université Catholique de Louvain, Belgium, 1994.

[171] T. Walsh. Depth-bounded discrepancy search. In *Proceedings of the 15th International Conference on Artificial Intelligence (ICAI'97)*, pages 1382–1387, 1997.

[172] L. A. Wolsey. *Integer Programming*. John Wiley & Sons, 1998.

[173] L. Yu, H. Shih, M. Pfund, W. Carlyle, and J. Fowler. Scheduling of unrelated parallel machines: An application to PWB manufacturing. *IIE Transactions on Scheduling and Logistics*, 34(11):921–931, 2002.