Load Balancing: The Long Road from Theory to Practice

Sebastian Berndt
University of Lübeck, Lübeck, Germany

Max A. Deppert
Kiel University, Kiel, Germany

Klaus Jansen
Kiel University, Kiel, Germany

Lars Rohwedder
EPFL, Lausanne, Switzerland

Abstract

There is a long history of approximation schemes for the problem of scheduling jobs on identical machines to minimize the makespan. Such a scheme grants a \((1 + \varepsilon)\)-approximation solution for every \(\varepsilon > 0\), but the running time grows exponentially in \(1/\varepsilon\). For a long time, these schemes seemed like a purely theoretical concept. Even solving instances for moderate values of \(\varepsilon\) seemed completely illusional. In an effort to bridge theory and practice, we refine recent ILP techniques to develop the fastest known approximation scheme for this problem. An implementation of this algorithm reaches values of \(\varepsilon\) lower than \(2/11 \approx 18.2\%\) within a reasonable timespan. This is the approximation guarantee of \textsc{Multifit}, which, to the best of our knowledge, has the best proven guarantee of any non-scheme algorithm.

2012 ACM Subject Classification Theory of computation → Scheduling algorithms

Keywords and phrases approximation scheme, makespan scheduling, parameterized algorithm, implementation

Funding Max A. Deppert: Research supported by German Research Foundation (DFG) project JA 612/20-1
Klaus Jansen: Research supported by German Research Foundation (DFG) project JA 612/20-1
1 Introduction

Makespan minimization on identical parallel machines (often denoted by $P||C_{\text{max}}$) asks for a distribution of a set $J$ of $n = |J|$ jobs to $m \leq n$ machines. Each job $j \in J$ has a processing time $p_j$ and the objective is to minimize the makespan, i.e., the maximum sum of processing times of jobs assigned to a single machine. More formally, a schedule $\sigma: J \rightarrow \{1, \ldots, m\}$ assigns jobs to machines. The load $\ell_{\sigma,i}$ of machine $i$ in schedule $\sigma$ is defined as $\sum_{j \in \sigma^{-1}(i)} p_j$ and the makespan $\mu(\sigma) = \max_i \{\ell_{\sigma,i}\}$ is the maximal load. The goal is to find a schedule $\sigma$ minimizing $\mu(\sigma)$. This is a widely studied problem both in operations research and in combinatorial optimization and has led to many new algorithmic techniques. For example, it has led to one of the earliest examples of an approximation scheme and the use of the dual approximation technique [13]. The problem is known to be strongly NP-hard and thus we cannot expect to find an exact solution in polynomial time. Many approximation algorithms that run in polynomial time and give a non-optimal solution have been proposed for this problem. From a theory point of view, the strongest approximation result is a polynomial time approximation scheme (PTAS) which gives a $(1 + \varepsilon)$-approximation, where the precision $\varepsilon > 0$ can be chosen arbitrarily small and is given to the algorithm as input. This goes back to a seminal work by Hochbaum and Shmoys [13]. The running time of such schemes for $P||C_{\text{max}}$ were drastically improved over time [1, 14, 24] and the best known running time is $2^{O(1/\varepsilon \log^2 (1/\varepsilon))} \log(n) + O(n)$ due to Jansen and Rohwedder [16], which is subsequently called the JR-algorithm. The JR-algorithm is in fact an algorithm for integer programming, but gives this running time when applied to a natural formulation of $P||C_{\text{max}}$. A PTAS with a running time of $f(1/\varepsilon) \cdot n^{O(1)}$ like in the JR-algorithm is called an efficient polynomial time approximation scheme (EPTAS). It follows from the strong NP-hardness that no fully polynomial time approximation scheme (FPTAS), an approximation scheme polynomial in both $n$ and $1/\varepsilon$, exists unless $P = NP$.

PTAS’s are often believed to be impractical. They tend to yield extremely high (though polynomial) running time bounds even for moderate precisions $\varepsilon$, see Marx [25]. By some, the research on PTAS’s has even been considered damaging for the large gap between theory and practice that it creates [27]. Although EPTAS’s (when FPTAS’s are not available) are sometimes proposed as a potential solution for this situation [25], we are not aware of a practical implementation of an EPTAS. For example, an approximation scheme for euclidean TSP was implemented by Rodeker et al., but the algorithm was merely inspired by an EPTAS and it does not retain the theoretical guarantee [26]. Although this is an interesting research direction as well, it remains an intriguing question whether one can obtain a practically relevant EPTAS implementation with actual theoretical guarantees. On the one hand, we believe that this is an important question to ask concerning the relevance of such a major field of research. On the other hand, such a PTAS implementation has great advantages in itself, since it exhibits a clean and generic design that is not specific to any concrete precision, as well as a (theoretically) unlimited potential of the precision.

Our Results.

As a major milestone we obtain a generic PTAS implementation that achieves in reasonable time a precision which beats the best known guarantee of a polynomial time non-PTAS algorithm. This precision to the best of our knowledge is $2/11 \approx 18.2\%$, which is guaranteed by the MULTIFIT algorithm. The claim might appear vague, since the running time depends not only on $\varepsilon$, but also on the instance. We believe that it is plausible nevertheless: The algorithm we use, which is based on the JR-algorithm, reduces the problem to performing
Several Fourier transformations (FFTs) where the size of the FFT input depends only on $\varepsilon$ and not the instance itself. Hence, the running time for all instances (using the same precision) is very stable and predictable. This is in the spirit of an EPTAS running time. We successfully run experiments of our implementation for a precision of $\varepsilon < 2/11$ and thus make the claim that this precision is practically feasible in general. This is also the main message of our paper. For completeness, we provide comparisons of the solution quality obtained empirically. While the theoretical guarantee of the PTAS is better, the difference to non-PTAS algorithms is marginal at this state and it is not yet evident in the experiments. The execution of the PTAS is computationally expensive and the considered precision is on the edge of what is realistic for our implementation. However, we believe that further optimization or more computational resources can lead to also empirically superior results. Nevertheless, the successful execution with a low precision value forms a proof of concept for practical PTAS’s.

Towards obtaining such an implementation we need to fine-tune the JR-algorithm significantly. In particular, it requires non-trivial theoretical work and novel algorithmic ideas. In fact, our variant has a slightly better dependence on the precision, namely $2^{O(1/\varepsilon \log(1/\varepsilon) \log \log(1/\varepsilon))}$, giving the best known running time for this problem. Our approach also greatly reduces the constants hidden by the $O$-notation. We first construct an integer program (IP) — the well-known configuration IP — that implies a $(1+\varepsilon)$-approximation by rounding the processing times. This IP has properties that allow sophisticated algorithms to solve it efficiently. We present several reduction steps to simplify and compress the IP massively. As extensions of this configuration IP are widely used, we believe this to be of interest in itself. For the makespan minimization problem, we obtain an IP where the columns of the constraint matrix have $\ell_{\infty}$-norms bounded by 2 and $\ell_1$-norms bounded by $O(\log(1/\varepsilon))$. In contrast, in the classical configuration integer program used in many of the previous PTAS’s both of these norms are bounded by $O(1/\varepsilon)$. This allows us to greatly reduce the size of the FFT instances in the JR-algorithm without losing the theoretical guarantee. For example, for $\varepsilon \approx 17.29\%$, our reduced IP lowers the instance sizes for FFT from $49 \times 12$ words for the configuration IP to $5 \times 12$ words.

Another important aspect in the algorithm is the rounding of the processing times. In general, one needs to consider only $O(1/\varepsilon \log(1/\varepsilon))$ different rounded processing times (to guarantee a precision of $\varepsilon$). This number has great impact on the size of the FFT instances. For concrete $\varepsilon$ the general rounding scheme might not give the optimal number of rounded processing times. We present a mixed integer linear program that can be used to generically optimize the rounding scheme for guaranteeing a fixed precision $\varepsilon$ (or equivalently, for a fixed number of rounded processing times).

**Related Work.**

The running time $f(1/\varepsilon) \cdot n^{O(1)}$ is a fixed-parameter running time, if $1/\varepsilon$ is treated as a parameter. In recent years, the study of practically usable parameterized algorithms has been a growing field of research. This need for practically usable parameterized algorithms has led to the Parameterized Algorithms and Computational Experiments (PACE) challenge [4, 6, 7]. This challenge has brought up surprisingly fast algorithms for important problems such as treewidth. Note that the fastest known such algorithm due to Tamaki [28] is based on an algorithm by Bouchitté and Todinca [5], which was widely believed to be purely theoretic. Our work can thus be viewed as an extension of these works to the field of approximation algorithms.

Many approximation algorithms for $P||C_{\text{max}}$ were developed over time. The first such
algorithm was the longest processing time first (LPT) algorithm by Graham, that achieved approximation ratio $4/3$ [10]. In [17], Coffman et al. presented the MULTIFIT algorithm that achieved a better approximation ratio of $13/11$. It was later shown by Yue that this analysis is tight, i.e. there are instances where MULTIFIT generates a solution with value $13/11 \cdot \text{opt}$ [29]. Kuruvilla and Palette combined the LPT algorithm and the MULTIFIT algorithm in an iterative way to obtain the Different Job and Machine Sets (DJMS) algorithm [22].

2 Algorithm

The general idea of our algorithm follows a typical approach for approximation schemes. We follow the dual approximation technique by performing a binary search on the optimal makespan. Here it suffices to construct an algorithm that for a given value $T$ either finds a schedule of makespan at most $(1 + \varepsilon)T$ or determines that $T$ is smaller than opt. Then we simplify the instance such that there are no jobs of very small processing time ($\leq \varepsilon T$) and jobs of very large processing time ($\geq (1 - 2\varepsilon)T$). The former is standard, whereas the latter reduction step is novel. This already reduces the range of processing times significantly for moderate values of $\varepsilon$. The remaining jobs are rounded via a novel rounding to $O(1/\varepsilon \log(1/\varepsilon))$ different processing times. We can then formulate the problems as an integer program and solve it via the algorithm of Jansen and Rohwedder [16]. Interestingly, our new rounding scheme allows us to compress the well-known configuration integer program quite significantly to obtain a better running time.

We defer some of the proofs in this section to the appendix, since they require some lengthy, but straight-forward calculations. We write $\log = \log_2$ to denote the logarithm to base 2.

2.1 Rounding scheme

It is well known that all jobs $j$ with $p_j \leq \varepsilon T$ can be discarded and added greedily after solving the remaining instance. Let $J_{\text{small}} = \{j \in J \mid p_j \leq \varepsilon T\}$ and $J_{\text{large}} = J \setminus J_{\text{small}}$. The feasibility of this approach follows from the following lemma:

Lemma 1. Let $\delta_{\text{large}}$ be a schedule of $J_{\text{large}}$ with makespan $\mu(\delta_{\text{large}})$. Adding the jobs from $J_{\text{small}}$ greedily gives a schedule $\delta$ with makespan

$$\mu(\delta) \leq \max\{\mu(\delta_{\text{large}}), \text{opt}(J) + \varepsilon T\}.$$ 

The procedure can be implemented to run in time $O((|J_{\text{small}}| + m) \cdot \log(|J_{\text{small}}| + m))$.

Furthermore, we can also get rid of huge jobs $J_{\text{huge}}$ with processing times at least $(1 - 2\varepsilon)T$, as each such job can only be paired with at most one other job from $J_{\text{large}}$ without violating the guess $T$. It is easy to see that we can pair a huge job with the largest possible large job without losing optimality.

Lemma 2 (informal). There is an optimal schedule $\delta_{\text{large}}$ of $J_{\text{large}}$ where each huge job is paired with the largest possible large job (or not paired at all).

As we now know how to place all of the jobs in $J_{\text{huge}}$ optimally, we can ignore them and their paired jobs in the following. After removing all of these jobs, we are left with the remaining jobs $J_{\text{rem}}$ that we still need to schedule. For all $j \in J_{\text{rem}}$, we now know that we have $p_j \in (\varepsilon T, (1 - 2\varepsilon)T)$. We will now round these remaining item sizes in order to reduce the number of different processing times in our instance. In order to do this, we first split
the interval \((\varepsilon T, (1 - 2\varepsilon)T)\) into \(\log(1/\varepsilon)\) growing intervals of size \(2^i\varepsilon T\) (starting with \(i = 0\)). Each of these intervals is then split into \(1/\varepsilon\) smaller intervals of the same size.

For example, for \(\varepsilon = 1/6\) and \(T = 1\), the growing intervals \((1/6, 1/3]\) and \((1/3, 2/3]\) are split into smaller intervals with the following boundaries:

\[
\begin{align*}
\frac{1}{6} &+ \frac{1}{36} &+ \frac{1}{6} &+ \frac{2}{36} &+ \frac{1}{6} &+ \frac{3}{36} &+ \frac{1}{6} &+ \frac{4}{36} &+ \frac{5}{36}, \\
\frac{1}{3} &+ \frac{1}{18} &+ \frac{1}{18} &+ \frac{2}{18} &+ \frac{1}{18} &+ \frac{3}{18} &+ \frac{1}{18} &+ \frac{4}{18} &+ \frac{5}{18}.
\end{align*}
\]

More formally, for \(i \in \mathbb{Z}_{\geq 0}\), let \(I_i = (2^i\varepsilon T, 2^{i+1}\varepsilon T]\). In the example above, we thus have \(I_0 = (1/6, 1/3]\) and \(I_1 = (1/3, 2/3]\). We further partition an interval \(I_i\) into \([1/\varepsilon]\) subintervals \(I_{i,k} = (b_{i,k}, b_{i,k+1}] \cap I_i\), with \(b_{i,k} = 2^i\varepsilon T + k\varepsilon 2^i T\) for \(k \in \{0, \ldots, [1/\varepsilon - 1]\}\). Hence, the above exemplary boundaries are exactly the values \(b_{i,k}\) for \(i \in \{0, 1\}\) and \(k \in \{0, \ldots, 5\}\). The processing time of any remaining job \(j \in J_{\text{rem}}\) is rounded down to the next lower boundary. We denote this rounded processing time of \(j\) by \(\hat{p}_j\).

**Lemma 3** (informal). There are \(O(1/\varepsilon \log(1/\varepsilon))\) rounded processing times \(\hat{p}_j\) and a schedule \(\hat{\sigma}\) of the rounded processing times implies a schedule \(\sigma\) of the original processing time with \(\mu(\sigma) \leq (1 + \varepsilon)\mu(\hat{\sigma})\). Furthermore, the sum of two boundaries \(b_{i,k}\) and \(b_{i,k'}\), where \(k\) and \(k'\) have the same parity, is equal to some boundary \(b_{i+1,k''}\).

The last property of the lemma that every two boundaries \(b_{i,k}\) and \(b_{i,k'}\) of the same interval (with the same parity of \(k\) and \(k'\)) sum up to a boundary \(b_{i+1,k''}\) in the next interval will be heavily used next. Intuitively, this property implies that whenever a job with rounded processing time \(b_{i,k}\) and another job with rounded processing time \(b_{i,k'}\) are scheduled on the same machine, we can treat them as a single job with rounded processing time \(b_{i+1,k''}\). This allows us to characterize the possible ways to assign rounded jobs to machines in a more compact way, which in turn allows us to solve the corresponding integer program much faster.

### 2.2 A new integer program

Integer programs are widely used to design approximation algorithm and approximation schemes. The classical result of Lenstra and Kannan [18, 19] shows that an integer program with \(n\) variables can be solved in time \(n^{O(n)} \cdot |I|^{O(1)}\), where \(|I|\) is the encoding length of the integer program (i.e. the binary encoding of all numbers in the objective function, the right-hand side, and the constraint matrix). This result was heavily used in the past to design approximation schemes. In fact, using Lemma 3 with this algorithm already yields an algorithm with running time double exponential in \(1/\varepsilon\). In the past years, other parameters besides the number of variables were studied, including the number of constraints, the largest entry in the constraint matrix or the properties of the graph corresponding to the constraints (see e.g. [8, 15, 21]). We will make use of the recent results that use the number of constraints (i.e. the number of rows of the constraint matrix) and the largest entry of the constraint matrix.

A basic concept of many algorithms for \(P|\|C|\max\) is the configuration integer program, which we will also use. Roughly speaking, for each possible way \(c\) to put jobs on a machine (called a configuration), this integer program has a variable \(x_c\) indicating how often this configuration is used. Then the integer program expresses that all jobs should be scheduled and that the number of configurations used should not exceed \(m\) via suitable constraints.

More formally, the integer program is constructed in the following way. Let \(d\) denote the number of rounded item sizes. We will index a vector \(x \in \mathbb{Z}^d\) by pairs \((i,k)\), corresponding
Load Balancing: The Long Road from Theory to Practice

to the values used in the rounded item sizes $b_{i,k}$ and denote its corresponding entry by $x[i,k]$. A vector $c \in \mathbb{Z}_+^n$ thus describes a possible way to schedule jobs on a machine, where the value $c[i,k]$ describes how many jobs with rounded processing time $b_{i,k}$ are put on a machine. We call such a vector $c$ a configuration, if the resulting load of the machine does not exceed $T$, i.e. $\sum_{i,k} c[i,k] \cdot b_{i,k} \leq T$. Let $C$ be the set of all configurations. For each $c \in C$, we have a variable $x_c$ that describes how often configuration $c$ is used, i.e. $x_c$ machines are scheduled according to $c$. As we only have $m$ machines available, we are only allowed to use at most $m$ configurations. Hence $\sum_{c \in C} x_c \leq m$. To guarantee that all jobs are scheduled, let $n_{i,k}$ be the number of items with rounded processing time $b_{i,k}$. Now, summing over all chosen configurations, we want that they contain at least $n_{i,k}$ jobs of rounded processing time $b_{i,k}$. Hence, $\sum_{c \in C} x_c \cdot c[i,k] \geq n_{i,k}$. Combining these with the natural requirement that $x_c \in \mathbb{Z}_{\geq 0}$, we obtain the following integer program called the configuration $IP$: $\sum_{c \in C} x_c \leq m$ $\sum_{c \in C} x_c \cdot c[i,k] \geq n_{i,k} \ \forall (i,k)$ $x_c \in \mathbb{Z}_{\geq 0} \ \forall c \in C$ $\text{(confIP)}$

As described above, the important parameters in the algorithm that we want to use are the number of rows of the constraint matrix ($d$ in our case) and the largest entry in the constraint matrix ($\max_{c \in C, (i,k)} |c[i,k]|$). Now, the first property of Lemma 3 already shows that the number of rows of the configuration $IP$ is bounded, i.e. $d \leq O(1/\varepsilon \cdot \log(1/\varepsilon))$. As every boundary $b_{i,k}$ is at least $\varepsilon T$ and we aim for a maximal load of $T$, we can easily see that the largest entry of a configuration and thus of the constraint matrix is at most $1/\varepsilon$. A closer look reveals that we actually have the slightly stronger bound of $\|c\|_1 \leq 1/\varepsilon$ for all $c \in C$. Without jumping too far ahead, the algorithm of Jansen and Rohwedder [16] discussed in Section 2.3 will thus yield a running time $2^{O(1/\varepsilon \log(1/\varepsilon))} + O(n)$, which is slightly too high to be usable in practice for our desired approximation guarantee of $\varepsilon < 2/11$. To decrease this running time, we will make use of the last property of Lemma 3 which will give an improved bound of $\|c\|_1 \leq O(\log(1/\varepsilon))$ and thus improve the running time to $2^{O(1/\varepsilon \log(1/\varepsilon)^{\log(\log(1/\varepsilon))})} + O(n)$. Moreover, the hidden constants are significantly lower. This is a sufficient improvement for the algorithm to run in reasonable time for $\varepsilon < 2/11$.

To improve the bound on $\|c\|_1$, we will add new columns $\hat{c}$ to the configuration $IP$. Remember that Lemma 3 states that all for all boundaries $b_{i,k}$ and $b_{i,k'}$ with $k \mod 2 = k' \mod 2$, there is $b_{i+1,k'} = b_{i,k} + b_{i,k'}$. The main idea behind these new columns $\hat{c}$ is that whenever we use a job with processing time $b_{i,k}$ and a job with processing time $b_{i,k'}$ on the same machine, we can treat this as a single job with processing time $b_{i+1,k'}$. Each new column $\hat{c}(i,k,k')$ will do this exact replacement. The final observation that we need is that in all configurations $c \in C$ with $\|c\|_1 > 2 \log(1/\varepsilon)$, we can do such a replacement: There are only $\log(1/\varepsilon)$ growing large intervals $i$ in our rounding and in each interval we can choose at most two boundaries of different parity $k \mod 2 \neq k' \mod 2$. Hence, if $\|c\|_1 > 2 \log(1/\varepsilon)$, configuration $c$ uses two jobs with processing times $b_{i,k}$ and $b_{i,k'}$ with $k \mod 2 = k' \mod 2$ and we can thus reduce this configuration via $\hat{c}(i,k,k')$.

By adding the columns $\hat{c}$ to the configuration integer program, we can remove all configurations $c$ except those in $C_{\text{red}} = \{c \in C : \|c\|_1 \leq 2 \log(1/\varepsilon)\}$. Let us denote this $IP$ by $IP_{C_{\text{red}}}$. Our discussion above thus implies the following lemma.

**Lemma 4 (informal).** For all solutions to the integer program $IP_{C_{\text{red}}} \hat{c}$, we can compute in linear time a solution to (confIP) and vice versa.
The advantage the system \( \text{IP}_{C_{\text{rot}}} \) gives us is that all columns have an \( \ell_1 \)-norm bounded by \( O(\log(1/\varepsilon)) \). The running time of the JR-algorithm directly depends on the discrepancy of the underlying constraint matrix. This improved bound on the \( \ell_1 \)-norm then allows us to bound this discrepancy leading to a faster running time (see Sec. 2.3 for a more thorough discussion). Furthermore, the \( \ell_\infty \)-norm of each column is at most 2 (due to the columns in \( \hat{C} \)). Already for relatively large values of \( \varepsilon \), this reduces the number of columns significantly. For example, for \( \varepsilon = 1/6 \), the number of columns is reduced from 409 down to 213.

2.3 Applying the JR-algorithm

Jansen and Rohwedder [16] described an algorithm for integer programming and applied it to the configuration \( \text{IP} \) for \( P||C_{\max} \). This algorithm reduces the task of solving the integer program to a small number of fast Fourier transformations (FFTs). The size of the FFT input depends on the number of rows of the constraint matrix as well as its discrepancy. Using the properties of our new integer program we are able to derive much better bounds on the discrepancy of the constraint matrix. Intuitively, the discrepancy of a matrix \( A \) measures how well the value \( A \cdot (1/2, 1/2, \ldots, 1/2)^T \) can be approximated by the term \( Az^T \), where \( z \) is some binary vector.

\[ \textbf{Definition 5 (Discrepancy)} \] For a matrix \( A \in \mathbb{R}^{m \times n} \) the discrepancy of \( A \) is given as

\[ \text{disc}(A) = \min_{z \in \{0,1\}^n} \left\| A \left( z = \left( \frac{1}{2}, \ldots, \frac{1}{2} \right)^T \right) \right\|_{\infty}. \]

Moreover, the hereditary discrepancy of \( A \) is then defined as

\[ \text{herdisc}(A) = \max_{I \subseteq \{1, \ldots, n\}} \text{disc}(A_I) \]

where \( A_I \) denotes the matrix \( A \) restricted to the columns \( I \).

We sketch the main ideas of the JR-algorithm and refer to [16] for details. The algorithm is based on the idea of splitting the solution to an \( \text{IP} \) \( \{Ax = b, x \in \mathbb{Z}_{\geq 0}\} \) into two parts \( x' + x'' = x \) where \( Ax' \) and \( Ax'' \) are almost the same. Hence, computing all solutions of the \( \text{IP} \) with \( b' \in \mathbb{H}(b/2) \) we can derive a solution with \( b \). Here \( \mathbb{H}(b/2) \) is an axis-parallel hypercube with sufficiently large side length surrounding \( b/2 \). The algorithm then iterates this idea. Indeed, the running time of the algorithm greatly depends on the bound of how evenly a solution can be split, that is, how large the hypercube needs to be. For this, discrepancy is a natural measure. A closer inspection of the JR-algorithm shows that it suffices for their algorithm to take a hypercube of side length \( 4 \text{herdisc}(A) - 1 \). Thus, the total number of elements in \( \mathbb{H}(b/2^i) \) is at most \((4 \text{herdisc}(A))^m\) for all \( i \). The central subprocedure in the algorithm is then to try to combine any two solutions for \( b', b'' \in \mathbb{H}(b/2^i) \) to a solution for \( b' + b'' \in \mathbb{H}(b/2^{i-1}) \). Instead of the naive algorithm that takes quadratic time (in the number of elements of \( \mathbb{H}(b/2^i) \)) it can be implemented more efficiently as multivariate polynomial multiplication where the input polynomials have \( m \) variables and maximum degree \( 4 \text{herdisc}(A) - 1 \). This in turn can be computed efficiently using FFT on inputs of size \( O((4 \text{herdisc}(A))^m) \), see Appendix [C] for details.

Evaluating the discrepancy.

The only remaining task now is to bound the discrepancy of our compressed integer program \( \text{IP}_{C_{\text{rot}}} \) presented in Section 2.2. Since its columns have small \( \ell_1 \)-norm, the classical Beck-Fiala theorem allows us to give a very strong bound on its discrepancy.
**Theorem 6 (Beck, Fiala [2]).** For every matrix $A \in \mathbb{R}^{m \times n}$ where the $\ell_1$-norm of each column is at most $t$, it holds that $\text{herdisc}(A) < t$.

Moreover, Bednarchak and Helm [3] observed that for $t \geq 3$ the bound can be improved to $\text{herdisc}(A) \leq t - 3/2$. This can be applied directly to our bounds on the $\ell_1$-norm of the configurations of $\text{IP}_{\text{red, c}}$. Let $A_{\text{red, c}}$ denote the corresponding constraint matrix. Since for each column $c$ of $A_{\text{red, c}}$, we have $\|c\|_1 \leq 2 \log(1/\varepsilon)$, we get

$$\text{herdisc}(A_{\text{red, c}}) \leq O(\log(1/\varepsilon)).$$

Moreover, the number of rows $m$ is one plus the number of rounded processing times, that is, $O(1/\varepsilon \log(1/\varepsilon))$. The algorithm needs to perform $O(\log(n))$ many FFTs on input of size $(\log(1/\varepsilon))^{O(1/\varepsilon \log(1/\varepsilon))}$. Hence, the total running time thus becomes

$$2^{O(1/\varepsilon \log(1/\varepsilon) \log \log(1/\varepsilon))} \log(n) + O(n).$$

Here $O(n)$ is necessary for the preprocessing. For a concrete precision $\varepsilon \approx 17.29\%$ using the optimized rounding scheme (see next section) gives us a bound of 3 on the $\ell_1$-norm and therefore a very moderate bound of 5 on the side length of each hypercube.

**The complete algorithm**

To sum up the description of our algorithm, we present all steps of the algorithm together in pseudocode in Fig. [1]. Note that this pseudocode is not optimized (in contrast to our implementation). For example, the computation of the sets $J_{\text{small}}$, $J_{\text{large}}$, and $J_{\text{huge}}$ can be done in one sweep and the values $b_{i,k}$ and the rounded processing times $\tilde{p}_j$ can also be computed concurrently. The binary search is performed until lower and upper bound differ by a factor less than $1 + \varepsilon'$. This parameter $\varepsilon'$ can be taken negligibly small (in contrast to $\varepsilon$), since the running time grows only logarithmically in $\varepsilon'$.

**Optimizing the rounding scheme**

Taking a closer look at the proof of Lemma 3 reveals that we only used the last property of Lemma 3 to obtain the simplified integer program. Now, we want to find the best rounding: Either minimize the number of rounded processing times $d$ for a given $\varepsilon$ or minimize the precision $\varepsilon$ for a given $d$. Fortunately, the task to decide whether such a rounding for given $d$ and $\varepsilon$ exists can be formulated as a mixed integer program. This allows us to obtain the best rounding in a generic preprocessing step. In the following, $x_i$ will denote the $i$-th rounded processing time with $x_0 \geq x_1 \geq \cdots \geq x_{d-1}$. Without loss of generality, we assume that our current guess $T$ is equal to 1 here. As our remaining item sizes are in the interval $(\varepsilon, 1 - 2\varepsilon)$ and we want to obtain a rounding that only produces an error of $1 + \varepsilon$, we need to guarantee that

$$(1 + \varepsilon)x_0 \geq (1 - 2\varepsilon),$$
$$x_0 \leq (1 - 2\varepsilon),$$
$$x_{d-1} \leq \varepsilon(1 + \varepsilon).$$

Furthermore, we need to make sure that $x_i$ and $x_{i+1}$ are relatively close, i.e. $(1 + \varepsilon)x_{i+1} \geq x_i$. 
while \( (1 + \varepsilon')L < R \) do: // binary search for \( \text{OPT} \)
  \( T = (R + L)/2 \) // guess makespan
  \( J_{\text{small}} = \{ j \in \{1, \ldots, n\} \mid p_j \leq \varepsilon T \} \)
  \( J_{\text{large}} = \{1, \ldots, n\} \setminus J_{\text{small}} \)
  \( J_{\text{huge}} = \{ j \in J_{\text{large}} \mid p_j \geq (1 - 2\varepsilon)T \} \)
  \( J_{\text{large}} = J_{\text{large}} \setminus \{ \{j\} \} \)
  for \( i = 0, \ldots, \lfloor \log((1 - 2\varepsilon)/\varepsilon) \rfloor \) :
    for \( k = 0, \ldots, [1/\varepsilon - 1] \) :
      \( b_{i,k} = 2^i \varepsilon T + k\varepsilon^2 2^i T; n_{i,k} = 0 \)
  endfor
endwhile
return the schedule produced for \( L \)

Figure 1 The non-optimized pseudocode of our algorithm. Here, \textbf{break} means that the current \textbf{while}-iteration is aborted and the next iteration is started.

To guarantee the last property of Lemma 8 we need to ensure that every configuration with more than \( 2\log(1/\varepsilon) \) processing times can be reduced. Hence, we construct for every subset \( X' \subseteq \{0, \ldots, d - 1\} \) of size \( L + 1 \) an indicator variable \( y_{X'} \) that is 1 iff there is a single configuration containing all of these processing times, i.e. \( \sum_{i \in X'} x_i \leq 1 \). We also use another indicator variable \( z_{i_1,i_2,i} \) that is 1 iff \( x_{i_1} + x_{i_2} = x_i \). Now, we need to guarantee that all configurations containing \( L + 1 \) item sizes can be reduced to \( L \) item sizes. Hence, if \( y_{X'} = 1 \) this implies that one can get rid of one of the item sizes, i.e. \( z_{i_1,i_2,i} = 1 \) for some \( i_1, i_2 \in X' \) and \( i \in \{0, \ldots, d - 1\} \). All of these indicator variables and implications can be easily introduced via the big \( M \) method or can be directly formulated for the mixed integer program solver. Finally, the \( x_i \) are fractional variables and the indicator variables are integral. The complete, formal formulation of the mixed integer program \( \text{MIP}_{\varepsilon,d} \) can be found in Section B in the appendix. Now, we can perform a binary search to optimize either \( \varepsilon \) or \( d \). More formally, for a given \( d \), we can find the minimal \( \varepsilon(d) \), or for a given \( \varepsilon \), we can find the minimal \( d(\varepsilon) \). See Table 1 for the precision achievable with certain values of \( d \).

### 2.4 Non-PTAS algorithms

Currently, heuristic algorithms like LPT, the MULTIFIT algorithm [17] and its derivative DJMS [22] give some of the best algorithms to solve instances of \( P|C_{\text{max}} \) in practice. For the sake of completeness here we describe briefly how they work.

---

\[ p_{\text{max}} = \max_{j=1,\ldots,n} \{ p_j \} \]

\[ \text{LB}(I) = \max \{ p_{\text{max}}, \sum_{j=1}^{n} p_j/m \} \quad \text{// compute bounds} \]

\[ L = \text{LB}(I); R = 2\text{LB}(I) \]

\[ \text{while } (1 + \varepsilon')L < R \text{ do} : \quad \text{// binary search for } \text{OPT} \]

\[ T = (R + L)/2 \quad \text{// guess makespan} \]

\[ J_{\text{small}} = \{ j \in \{1, \ldots, n\} \mid p_j \leq \varepsilon T \} \]

\[ J_{\text{large}} = \{1, \ldots, n\} \setminus J_{\text{small}} \]

\[ J_{\text{huge}} = \{ j \in J_{\text{large}} \mid p_j \geq (1 - 2\varepsilon)T \} \]

\[ J_{\text{large}} = J_{\text{large}} \setminus \{ \{j\} \} \]

\[ \text{for } j \in J_{\text{huge}} : \]

\[ \text{find } j' \in J_{\text{large}} \text{ with } p_{j'} \text{ minimal and } p_j + p_{j'} \leq T \]

\[ J_{\text{large}} = J_{\text{large}} \setminus \{ \{j'\} \} \]

\[ \text{endfor} \]

\[ \text{for } i = 0, \ldots, \lfloor \log((1 - 2\varepsilon)/\varepsilon) \rfloor : \]

\[ \text{for } k = 0, \ldots, [1/\varepsilon - 1] : \]

\[ b_{i,k} = 2^i \varepsilon T + k\varepsilon^2 2^i T; n_{i,k} = 0 \]

\[ \text{endfor} \]

\[ \text{endfor} \]
Table 1 Optimal rounding scheme for some numbers of rounded processing times $d$.

| $d$ | optimal precision $\varepsilon(d)$ |
|-----|------------------------------------|
| 9   | 0.172874755859                     |
| 10  | 0.160867004395                     |
| 11  | 0.15059387207                      |

The MULTIFIT algorithm was presented by Coffman et al. [17] and its exact bound of $13/11$ was proved by Yue [29]. It is based on iteratively applying the First-Fit-Decreasing algorithm, which, for a given guess on the makespan $T$, sorts the jobs by their processing times in non-increasing order and the machines in an arbitrary order. Then, each job $j$ is packed onto the first machine where it fits i.e. where the load of $T$ is not exceeded. To approximate an optimum solution of an instance $I$, the MULTIFIT algorithm takes $I$ and a maximum number of rounds $t$ as the input and starts with computing a lower bound $\ell = \max\{\frac{1}{m} \sum_{j} p_j, p_1, p_n + p_{n+1}\}$ (where $p_1 \geq \cdots \geq p_n$) and an upper bound $u = \text{LPT}(I)$ to bound the intended makespan. Then MULTIFIT aims to compute the smallest feasible makespan $T^* \in [\ell, u]$, which admits a First-Fit-Decreasing packing of all jobs to $m$ bins, via binary search in at most $t$ rounds. Obviously, that maximum number of rounds $t$ may be dropped if all numbers in the input are integral. However, it can happen that $t$ rounds are over or even $T^* = u$ does not admit a feasible First-Fit-Decreasing packing. Then the algorithm returns the solution obtained by LPT.

The DJMS algorithm of Kuruvilla and Paletta [22] combines the LPT approach with the MULTIFIT approach by splitting machines and jobs into active and closed ones. At the beginning of the algorithm, all jobs and machines are active. Now, in each iteration, the LPT algorithm is applied to the active jobs and active machines to compute an upper bound on the makespan $T_{\text{LPT}}$. Afterwards, MULTIFIT with the upper bound of $T_{\text{LPT}}$ is applied to the active jobs and machines. In the solution produced by MULTIFIT, we search for the least loaded active machine $i^*$ whose load exceeds the lower bound $\ell$ (as defined for MULTIFIT). All machines with the same load as $i^*$ and the jobs on them are declared close. These steps are repeated until all machines are closed. In their computational experiments, DJMS gave the best makespan compared with LISTFIT (an algorithm by Gupta and Ruiz-Torres [12]), LPT, and MULTIFIT [22].

3 Implementation

The algorithm was implemented in the C++ programming language. To compute fast Fourier transformations we used the C library FFTW3 [9] in version 3.3.8 and we applied OpenMP (https://www.openmp.org/) in version 5.0 for parallelization. We plan to upload a cleaned up version of our implementation on github (https://www.github.com). The experiments were computed in the HPC Linux Cluster of Kiel university using 16 cpu cores and 100GB of memory per instance.

3.1 Computational results

In the following we refer to our algorithm as LRTP. To compare our implementation with the non-PTAS algorithms LPT, MULTIFIT, and DJMS we compute solutions to a set of instances first considered by Kedia [20] in 1971. Since then it has been used by various authors to
S. Berndt, M.A. Deppert, K. Jansen, and L. Rohwedder investigate the quality of scheduling algorithms on identical machines (cf. [11, 22, 23]). The instances are grouped into four families E1, E2, E3, E4 (see Table 2 for an overview). Each family consists of classes of 100 instances and the instances of each class were generated respecting three common parameters; namely, a number of machines \( m \), a number of jobs \( n \), and a universe interval \( U \), used to uniformly select the integer processing times of the jobs. Overall there are 90 classes \((m, n, U)\) considered, i.e. 9000 instances in total. These instances do not exceed a machine number of \( m = 10 \), so we generated a family BIG of additional classes where \( m \in \{25, 50, 75, 100\} \), \( n = 4m \), and \( U = [1,1000] \) (see Table 7) to give evidence to the fact that LRTP solves larger instances in reasonable time too. See Figure 2 for a makespan comparison of LRTP with the non-\(\text{ptas}\) algorithms for class \((m = 4, n = 8, U = [1,20])\) of family E1 where the 100 instances of the class are presented from left to right while the makespan grows from bottom to top.

Table 2: An overview on the instance families

|   |   |   |
|---|---|---|
| **m** | **n** | **U** |
| E1 | 3, 4, 5 | 2m, 3m, 5m |
| E2 | 2, 3 | 10, 30, 50, 100 |
|    | 4, 6, 8, 10 | 30, 50, 100 |
| E3 | 3, 5, 8, 10 | 3m + 1, 3m + 2, 4m + 1, 4m + 2, 5m + 1, 5m + 2 |
| E4 | 2 | 10 |
|    | 3 | 9 |

In Tables 3, 4, and 5–7 we give the computation results which are prepared as follows. Note that the results for families E3, E4, and BIG can be found in the appendix in Section D. Each line summarizes the results for the 100 instances of class \((m, n, U)\). Column better is the number of instances where the makespan computed by LRTP is lower than the best makespan of the non-\(\text{ptas}\) algorithms while equal counts the instances where these are equal. Column \text{avg}_\text{quot} = \frac{\sum_{i=1}^{100} \text{LRTP}(i)}{\min\{\sum_{i=1}^{100} \text{LPT}(i), \sum_{i=1}^{100} \text{MF}(i), \sum_{i=1}^{100} \text{DJMS}(i)\}}\) compares the sums of the computed makespans (and is rounded to two decimal places) and column \text{avg}_\text{time} states the rounded-up average running time of LRTP in minutes. There were no instance classes, where the maximal running time exceeded \(2 \cdot \text{avg}_\text{time}\).

For each of the classes where \text{avg}_\text{time} = 1 the short running time is explained by the rather large quotient \(n/m\) causing trivial instances where nearly all jobs are either small (\(\leq \varepsilon\)) or huge (\(\geq 1 - 2\varepsilon\)).

Figure 2: Makespan comparison for 100 instances of class \((3, 8, [1, 20])\) (x: instances, y: makespan)
Table 3: Computational results for the classes of family E1

| family | m  | n   | U     | better | equal | avg_quot | avg_time |
|--------|----|-----|-------|--------|-------|---------|---------|
| E1     | 3  | 6   | [1,20] | 0     | 69    | 1.02    | 24      |
| E1     | 3  | 9   | [20,50]| 2     | 35    | 1.03    | 35      |
| E1     | 3  | 9   | [1,20] | 3     | 76    | 1.05    | 57      |
| E1     | 3  | 15  | [1,20] | 2     | 42    | 1.02    | 42      |
| E1     | 3  | 15  | [20,50]| 2     | 2     | 1.04    | 49      |
| E1     | 4  | 8   | [1,20] | 0     | 48    | 1.03    | 29      |
| E1     | 4  | 8   | [20,50]| 0     | 11    | 1.05    | 49      |
| E1     | 4  | 12  | [1,20] | 1     | 25    | 1.04    | 38      |
| E1     | 4  | 12  | [20,50]| 0     | 1     | 1.06    | 60      |
| E1     | 4  | 20  | [1,20] | 1     | 23    | 1.05    | 46      |
| E1     | 4  | 20  | [20,50]| 0     | 1     | 1.06    | 65      |
| E1     | 5  | 10  | [1,20] | 0     | 50    | 1.03    | 38      |
| E1     | 5  | 10  | [20,50]| 0     | 5     | 1.06    | 60      |
| E1     | 5  | 15  | [1,20] | 0     | 8     | 1.06    | 47      |
| E1     | 5  | 15  | [20,50]| 0     | 1     | 1.07    | 69      |
| E1     | 5  | 25  | [1,20] | 0     | 7     | 1.06    | 51      |
| E1     | 5  | 25  | [20,50]| 0     | 0     | 1.07    | 72      |

Table 4: Computational results for the classes of family E2

| family | m  | n   | U     | better | equal | avg_quot | avg_time |
|--------|----|-----|-------|--------|-------|---------|---------|
| E2     | 2  | 10  | [100,800]| 16    | 18    | 1.01    | 99      |
| E2     | 2  | 30  | [100,800]| 0     | 58    | 1       | 1       |
| E2     | 2  | 50  | [100,800]| 0     | 56    | 1       | 1       |
| E2     | 2  | 100 | [100,800]| 0     | 66    | 1       | 1       |
| E2     | 3  | 10  | [100,800]| 10    | 7     | 1.03    | 125     |
| E2     | 3  | 30  | [100,800]| 2     | 29    | 1       | 1       |
| E2     | 3  | 50  | [100,800]| 0     | 0     | 1       | 1       |
| E2     | 3  | 100 | [100,800]| 0     | 0     | 1       | 1       |
| E2     | 4  | 30  | [100,800]| 7     | 0     | 1.03    | 130     |
| E2     | 4  | 50  | [100,800]| 0     | 0     | 1.01    | 1       |
| E2     | 4  | 100 | [100,800]| 0     | 38    | 1       | 1       |
| E2     | 6  | 30  | [100,800]| 1     | 0     | 1.06    | 145     |
| E2     | 6  | 50  | [100,800]| 6     | 0     | 1.02    | 136     |
| E2     | 6  | 100 | [100,800]| 0     | 0     | 1       | 1       |
| E2     | 8  | 30  | [100,800]| 0     | 0     | 1.08    | 167     |
| E2     | 8  | 50  | [100,800]| 0     | 0     | 1.07    | 162     |
| E2     | 8  | 100 | [100,800]| 0     | 0     | 1.01    | 1       |
| E2     | 10 | 30  | [100,800]| 0     | 0     | 1.08    | 171     |
| E2     | 10 | 50  | [100,800]| 0     | 0     | 1.09    | 164     |
| E2     | 10 | 100 | [100,800]| 0     | 7     | 1.01    | 25      |

By their simple nature the non-PTAS algorithms compute solutions even to large instances within a few seconds. While the running time of the PTAS is generally on a high level, the increase in the running time as the number of machines and jobs grows is rather small due to the parameterized running time of our algorithm. We emphasize that the running time did not cross the border of 5.6 hours for any instance computed. We therefore see these experiments as a valid proof of concept for practically feasible PTAS’s. The empirical solution quality, although not our main focus, is not superior to the non-PTAS algorithms at the considered precision, though the difference is only small. This may be due to the random instances which do not necessarily exhibit a worst-case structure and that the difference in theoretical approximation guarantee is at this state fairly small. We believe that with additional computational resources or further optimizations an even lower precision can be reached, which will lead to a superior solution quality also in experiments.
References

1. Noga Alon, Yossi Azar, Gerhard J. Woeginger, and Tal Yadid. Approximation schemes for scheduling. In Michael E. Saks, editor, *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 5-7 January 1997, New Orleans, Louisiana, USA, pages 493–500. ACM/SIAM, 1997. URL: [http://dl.acm.org/citation.cfm?id=314161.314371](http://dl.acm.org/citation.cfm?id=314161.314371).

2. József Beck and Tibor Fiala. “Integer-making” theorems. *Discrete Applied Mathematics*, 3(1):1–8, 1981.

3. Debe Bednarchak and Martin Helm. A note on the beck-fiala theorem. *Combinatorica*, 17(1):147–149, 1997.

4. Édouard Bonnet and Florian Sikora. The PACE 2018 parameterized algorithms and computational experiments challenge: The third iteration. In *IPEC*, volume 115 of LIPIcs, pages 26:1–26:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

5. Vincent Bouchitté and Ioan Todinca. Listing all potential maximal cliques of a graph. *Theor. Comput. Sci.*, 276(1-2):17–32, 2002.

6. Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *IPEC*, volume 89 of LIPIcs, pages 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

7. M. Ayaz Dzulfikar, Johannes Klaus Fichte, and Markus Hecher. The PACE 2019 parameterized algorithms and computational experiments challenge: The fourth iteration (invited paper). In *IPEC*, volume 148 of LIPIcs, pages 25:1–25:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

8. Friedrich Eisenbrand and Robert Weismantel. Proximity results and faster algorithms for integer programming using the steinitz lemma. *ACM Trans. Algorithms*, 16(1):5:1–5:14, 2020.

9. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

10. Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

11. J. N. D. Gupta and A. J. Ruiz-Torres. A listfit heuristic for minimizing makespan on identical parallel machines. *Production Planning & Control*, 12(1):28–36, 2001.

12. Jatinder ND Gupta and Alex J Ruiz-Torres. A listfit heuristic for minimizing makespan on identical parallel machines. *Production Planning & Control*, 12(1):28–36, 2001.

13. Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

14. Klaus Jansen, Kim-Manuel Klein, and José Verschae. Closing the gap for makespan scheduling via sparsification techniques. *Math. Oper. Res.*, 45(4):1371–1392, 2020. [doi:10.1287/moor.2019.1036](https://doi.org/10.1287/moor.2019.1036).

15. Klaus Jansen, Alexandra Lassota, and Lars Rohwedder. Near-linear time algorithm for n-fold ilps via color coding. In *Proc. ICALP 2019*, pages 75:1–75:13, 2019.

16. Klaus Jansen and Lars Rohwedder. On integer programming and convolution. In *Proc. ITCS 2019*, pages 43:1–43:17, 2019.

17. Edward G. Coffman Jr., M. R. Garey, and David S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.*, 7(1):1–17, 1978.

18. Hendrik W. Lenstra Jr. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8(4):538–548, 1983.

19. Ravi Kannan. Minkowski’s convex body theorem and integer programming. *Math. Oper. Res.*, 12(3):415–440, 1987.

20. SK Kedia. A job scheduling problem with parallel processors. *Unpublished report, Department of Industrial and Operations Engineering. University of Michigan, Ann Arbor, MI*, 1971.

21. Martin Koutecký, Asaf Levin, and Shmuel Onn. A parameterized strongly polynomial algorithm for block structured integer programs. In *ICALP*, volume 107 of LIPIcs, pages 85:1–85:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
Lemma 1. Let $\delta_{\text{large}}$ be a schedule of $J_{\text{large}}$ with makespan $\mu(\delta_{\text{large}})$. Adding the jobs from $J_{\text{small}}$ greedily gives a schedule $\delta$ with makespan

$$\mu(\delta) \leq \max\{\mu(\delta_{\text{large}}), \text{OPT}(J) + \varepsilon T\}.$$  

The procedure can be implemented to run in time $O((|J_{\text{small}}| + m) \cdot \log(|J_{\text{small}}| + m))$.

Proof. There are two possible situations to consider: If the makespan does not increase, we have $\mu(\delta) = \mu(\delta_{\text{large}})$. If the makespan increases, consider the loads $\ell_1, \ldots, \ell_m$ of the machines. As the makespan increased, the differences between the loads is bounded by $\varepsilon T$, i.e. $|\ell_i - \ell_i'| \leq \varepsilon T$ for all $i, i'$. We thus have $\mu(\delta) - \ell_i \leq \varepsilon T$ for all $i$, as $\mu(\delta) = \max_i \ell_i$. Hence

$$\sum_j p_j = \sum_i \ell_i \geq \sum_i (\mu(\delta) - \varepsilon T) = m \cdot (\mu(\delta) - \varepsilon T)$$

and thus $(\sum_j p_j)/m \geq \mu(\delta) - \varepsilon T$. As $\text{OPT}(J) \geq (\sum_j p_j)/m$, we conclude $\mu(\delta) \leq \text{OPT}(J) + \varepsilon T$.

The time complexity of $O((|J_{\text{small}}| + m) \cdot \log(|J_{\text{small}}| + m))$ can be achieved by storing the loads of the machines in a min-heap.

Lemma 7 (Formal version of Lemma 2). Define $J_{\text{medium}} := \{j \in J \mid \varepsilon T < p_j \leq 2\varepsilon T\} \subseteq J_{\text{large}}$. Let $J_{\text{huge}} = \{j_1, \ldots, j_k\}$ with $p_{j_1} \geq p_{j_2} \geq \ldots p_{j_k}$ and $J_{\text{medium}} = \{j_1', \ldots, j_{k'}\}$ with $p_{j_1'} \geq p_{j_2'} \geq \ldots p_{j_{k'}}$. Define iteratively $\psi: \{1, \ldots, k\} \to \{1, \ldots, k', \infty\}$ with $\psi(j) = \min\{j' \leq k' \mid p_j + p_{j'} \leq 1 \land j' \notin \{\psi(j') \mid j < j'\}\}$, where $\min \emptyset = \infty$.

There is an optimal schedule $\delta_{\text{large}}$ of $J_{\text{large}}$ such that for each machine $i$ with $\delta_{\text{large}}^{-1}(i) \cap J_{\text{huge}} = \{j\}$, we have either $\delta_{\text{large}}^{-1}(i) = \{j\}$ if $\psi(j) = \infty$ or $\delta_{\text{large}}^{-1}(i) = \{j, \psi(j)\}$ if $\psi(j) < \infty$.

Proof. Intuitively, $\psi$ maps $j$ to the largest job $j'$ in $J_{\text{medium}}$ with which it can be put onto a machine and which is not already mapped to another huge job. If no such job $j'$ exists, $\psi(j) = \infty$.

Consider any optimal schedule $\delta_{\text{large}}$ of $J_{\text{large}}$ such that there is a machine $i$ with $\delta_{\text{large}}^{-1}(i) \cap J_{\text{huge}} = \{j\}$ with $\delta_{\text{large}}^{-1}(i) = \{j, j'\}$ where either $\psi(j) = \infty$ or $j' \neq \psi(j)$. If there are multiple
such machines, we consider the one with the job \( j = j_\ell \) with minimal index in the ordering \( j_1, \ldots, j_k \). As \( j_\ell \) has minimal index, if \( \psi(j) = \infty \), all fitting jobs from \( J_{\text{medium}} \) are packed with the jobs \( j_1, \ldots, j_{\ell-1} \). Hence, there is no job \( j' \) that can be packed with \( j_\ell \). This situation is thus not possible. If \( j' \neq \psi(j) \), consider the machine \( \delta_{\text{large}}(\psi(j)) = i' \). As \( p_{\psi(j)} + p_j \leq 1 \) and \( p_{\psi(j)} \geq p_{j'} \) (by definition of \( \psi(j) \)), we can exchange \( j' \) and \( \psi(j) \) and still keep a feasible optimal schedule.

Applying the above reason iteratively finally gives us an optimal schedule that always pairs the jobs \( j \) and \( \psi(j) \) if \( \psi(j) < \infty \).

Lemma 8 (Formal version of Lemma 3).
1. The number of different rounded processing times of \( J_{\text{ren}} \) is at most \([1/\varepsilon] \cdot \lceil \log((1-2\varepsilon)/\varepsilon) \rceil + 1\).

2. A schedule \( \hat{\sigma} \) of the rounded processing times implies a schedule \( \sigma \) of the original processing time with \( \mu(\sigma) \leq (1 + \varepsilon)\mu(\hat{\sigma}) \).

3. We have \( b_{i,k_1} + b_{i,k_2} = b_{i+1,(k_1+k_2)/2} \) for all \( i \) and all \( k_1, k_2 \) with \( k_1 \mod 2 = k_2 \mod 2 \).

Proof.

1. We have \( 2^{i+1}\varepsilon T = (1 - 2\varepsilon)T \) if \( i + 1 = \log((1 - 2\varepsilon)/\varepsilon) \). For \( i > \log((1 - 2\varepsilon)/\varepsilon) \), there is no job in any interval \( I_i \). Hence, there are at most \( \log((1 - 2\varepsilon)/\varepsilon) + 1 \) many values for \( i \) such that the processing times lie in \( I_i \). Every \( I_i \) is split into \([1/\varepsilon] + 1 \) intervals. Hence, there are at most \([1/\varepsilon] + 1 \cdot \lceil \log((1-2\varepsilon)/\varepsilon) \rceil + 1 \) many rounded processing times.

2. It is sufficient to show \( p_j \geq \bar{p}_j \geq (1 + \varepsilon)^{-1}p_j \). As we only round down, we have \( p_j \geq \bar{p}_j \) immediately. Let \( \bar{p}_j = b_{i,k} \) and hence \( p_j \in I_{i,k} \). Let \( (1 + \varepsilon)\bar{p}_j = (1 + \varepsilon)b_{i,k} = (1 + \varepsilon)[2^i\varepsilon T + k\varepsilon^2 2^i T] = 2^i\varepsilon T + k\varepsilon^2 2^i T + 2\varepsilon^2 2^i T \geq 2^i\varepsilon T + k\varepsilon^2 2^i T + 2\varepsilon^2 2^i T = 2^i\varepsilon T + (k + 1)\varepsilon^2 2^i T = b_{i,k+1} > p_j \).

3. We have \( b_{i,k_1} + b_{i,k_2} = 2^i\varepsilon T + k_1\varepsilon^2 2^i T + 2^i\varepsilon T + k_2\varepsilon^2 2^i T = 2^{i+1}\varepsilon T + (k_1 + k_2)\varepsilon^2 2^i T \).

As \( k_1 \mod 2 = k_2 \mod 2 \), we have \((k_1 + k_2)/2 \in \mathbb{Z}_{\geq 0} \) and thus \( 2^{i+1}\varepsilon T + (k_1 + k_2)\varepsilon^2 2^i T = 2^i\varepsilon T + [(k_1 + k_2)/2]\varepsilon^2 2^i T = b_{i+1,(k_1+k_2)/2} \). ▷

Proving Lemma 4

To prove Lemma 4, we first introduce some notations. For \( i*, k_1^*, k_2^* \) with \( k_2^* \mod 2 = k_2^* \mod 2 \), and we define the new column \( \hat{c} := \hat{c}(i^*, k_1^*, k_2^*) \) with (i) \( \hat{c}[i,k] = 2 \), if \( i = i^* \) and \( k = k_1^* = k_2^* \) or (ii) \( \hat{c}[i,k] = 1 \), if \( i = i^* \) and \( k \in \{k_1^*, k_2^*\} \) and \( k_1^* \neq k_2^* \) or (iii) \( \hat{c}[i,k] = -1 \), if \( i = i^* + 1 \) and \( k = (k_1^* + k_2^*)/2 \). In all other cases, we define \( \hat{c}[i,k] = 0 \).

For a subset \( C' \subseteq \hat{C} \), let \( \text{IP}_{C',\hat{C}} \) be the following integer program:

\[
\sum_{c \in \hat{C}} x_c \leq m; \quad \sum_{c \in C' \cup \hat{C}} x_c \cdot \hat{c}[i,k] \geq n_{i,k} \quad \forall (i,k); \quad x_c \in \mathbb{Z}_{\geq 0} \quad \forall c \in C' \cup \hat{C}.
\]

Let \( \text{IP}_C \) be the original configuration IP. We can now obtain the following switching lemma, which directly implies Lemma 4 as a feasible solution \( x^* \) of \( \text{IP}_C \) can be transformed into a feasible solution \( y^* \) of \( \text{IP}_{C',\hat{C}} \) and vice versa.
Lemma 9. Let \( x^* \) be a feasible solution of \( \text{IP}_{\mathcal{C}, \hat{c}} \).

1. If there is a \( c_1 \in \mathcal{C} \) with \( \|c_1\|_1 > 2\log((1-2\varepsilon)/\varepsilon) + 1 \) and \( x^*_c > 0 \), there are configurations \( c_2 \in \hat{\mathcal{C}} \) with \( \|c_2\|_1 < \|c_1\|_1 \) and \( c_3 \in \hat{\mathcal{C}} \), such that the vector \( y \) is a feasible solution of \( \text{IP}_{\mathcal{C}, \hat{c}} \) with \( y_{c_1} = x^*_c - 1 \), \( y_{c_2} = x^*_c + 1 \), \( y_{c_3} = 1 \) and \( y_c = x^*_c \) for all other \( c \).

2. If there is a configuration \( c_1 = \hat{c}(i^*, k_1^*, k_2^*) \in \hat{\mathcal{C}} \) with \( x^*_c > 0 \) and a configuration \( c_2 \in \mathcal{C} \) with \( c_2[i^* + 1, (k_1^* + k_2^*)] = 1 \) and \( x^*_c > 0 \), there is a configuration \( c_3 \in \mathcal{C} \), such that the vector \( y \) is a feasible solution of \( \text{IP}_{\mathcal{C}, \hat{c}} \) with \( y_{c_1} = x^*_c - 1 \), \( y_{c_2} = x^*_c - 1 \), \( y_{c_3} = x^*_c + 1 \) and \( y_c = x^*_c \) for all other \( c \).

Proof. Let \( x^* \) be a feasible solution of \( \text{IP}_{\mathcal{C}, \hat{c}} \).

1. As \( \|c_1\|_1 > 2\log((1-2\varepsilon)/\varepsilon) + 1 \), the pigeonhole principle implies that there are indices \( i^*, k_1^*, k_2^* \) with \( k_1^* \mod 2 = 0 \mod 2 \), such that \( \min\{c_1[i^*, k_1^*], c_1[i^*, k_2^*]\} \geq 1 \), if \( k_1^* \neq k_2^* \) or \( c_1[i^*, k_1^*] \geq 1 \), if \( k_1^* = k_2^* \). Choose \( c_3 = \hat{c}(i^*, k_1^*, k_2^*) \) and \( c_2 = c_1 - c_3 \). By construction, we have \( c_2 \in \mathcal{C} \) and \( \|c_2\|_1 = \|c_1\|_1 + 1 - 2 = \|c_1\|_1 - 1 \). For \( i = i^* \) and \( k = k_1^* \), we have

\[
\sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} y_c \cdot c[i, k] = \sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} x^*_c \cdot c[i, k] \geq nm_{i,k}.
\]

For \( i = i^* + 1 \) and \( k = (k_1^* + k_2^*)/2 \), we have

\[
\sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} y_c \cdot c[i, k] = \sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} x^*_c \cdot c[i, k] \geq nm_{i,k}.
\]

Finally, for all other \( i \) and \( k \), we have

\[
\sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} y_c \cdot c[i, k] = \sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} x^*_c \cdot c[i, k] \geq nm_{i,k}.
\]

as nothing changed here.

As \( \sum_{c \in \mathcal{C}} y_c = \sum_{c \in \mathcal{C}} x^*_c \leq m \), we can conclude that \( y \) is a feasible solution of \( \text{IP}_{\mathcal{C}, \hat{c}} \).

2. Choose \( c_3 = c_2 - c_1 \). By construction, we have \( c_3 \in \mathcal{C} \).

For \( i = i^* \) and \( k = k_1^* = k_2^* \), we have

\[
\sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} y_c \cdot c[i, k] = \sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} x^*_c \cdot c[i, k] \geq nm_{i,k}.
\]

For \( i = i^* + 1 \) and \( k = (k_1^* + k_2^*)/2 \), we have

\[
\sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} y_c \cdot c[i, k] = \sum_{c \in \mathcal{C} \cup \hat{\mathcal{C}}} x^*_c \cdot c[i, k] \geq nm_{i,k}.
\]
Finally, for all other $i$ and $k$, we have

$$
\sum_{c \in C' \cup \hat{C}} y_c \cdot c[i,k] = \sum_{c \in C' \cup \hat{C}} x^*_c \cdot c[i,k] \geq n_{i,k},
$$

as nothing changed here.

As $\sum_{c \in C} y_c = \sum_{c \in C} x^*_c \leq m$, we can conclude that $y$ is a feasible solution of $\text{IP}_{C,\hat{C}}$.

\[\blacktriangle\]

B  The complete MILP to optimize the rounding scheme

The $y$-variables can be easily introduced via the big $M$ method and by minimizing the $y$ variables.

$$
- \sum_{i \in X'} x_i + M \cdot y_{X'} \leq -1 \quad \forall X' \subseteq \{0, \ldots, d - 1\} \land |X'| = L + 1
$$

$$(y_{X'} = 1 \iff \sum_{i \in X'} x_i \leq 1)$$

We also construct another indicator variable $z_{i_1,i_2,i}$ that is 1 iff $x_{i_1} + x_{i_2} = x_i$ holds.

$$
z_{i_1,i_2,i} = 1 \iff x_{i_1} + x_{i_2} = x_i \quad \forall i_1, i_2, i \in \{0, \ldots, d - 1\}
$$

Again, this implication can be formulated via the big $M$ method or directly by a mixed integer program solver.

Now, we need to guarantee that all configurations containing $L + 1$ item sizes can be reduced to $L$ item sizes. Hence, if $y_{X'} = 1$ this implies that one can get rid of one of the item sizes, i.e. $z_{i_1,i_2,i} = 1$ for some $i_1, i_2 \in X'$ and $i \in \{0, \ldots, d - 1\}$:

$$
y_{X'} = 1 \implies \exists i', i'' \in X' \exists i \in \{0, \ldots, r - 1\} : z_{i',i'',i} = 1 \quad \forall X' \subseteq \{0, \ldots, d - 1\} \land |X'| = L + 1
$$

Again, this implication can be formulated via the big $M$ method or directly by an mixed integer program solver. Finally, the $x_i$ are fractional variables and the indicator variables are integral:

$$
x_i \in \{0, 1\} \quad \forall i \in \{0, \ldots, d - 1\}
y_{X'} \in \{0, 1\} \quad \forall X' \subseteq \{0, \ldots, d - 1\} \land |X'| = L + 1
$$

$$
z_{i_1,i_2,i} \in \{0, 1\} \quad \forall i_1, i_2, i \in \{0, \ldots, d - 1\}$$
In total, we obtain the following mixed integer program:

\[
\min \sum_{x'} y_{x'} + \sum_{i_1, i_2, i} z_{i_1, i_2, i} \quad \text{s.t.} \\
(1 + \varepsilon)x_0 \geq (1 - 2\varepsilon) \\
x_0 \leq (1 - 2\varepsilon) \\
x_{d-1} \leq \varepsilon \cdot (1 + \varepsilon) \\
(1 + \varepsilon)x_{i+1} \geq x_i \\
y_{x'} = 1 \iff \sum_{i \in X'} x_i \leq 1 \\
z_{i_1, i_2, i} = 1 \iff x_{i_1} + x_{i_2} = x_i \\
y_{x'} = 1 \Rightarrow \exists i_1', i_2' \in X' \exists i \in \{1, \ldots, n\} : z_{i_1', i_2', i} = 1 \quad \forall i_1, i_2, i \in \{0, \ldots, d - 1\} \\
x_i \in [0, 1] \\
y_{x'} \in \{0, 1\} \\
z_{i_1, i_2, i} \in \{0, 1\} \\
\forall i_1, i_2, i \in \{0, \ldots, d - 1\}
\]

\[
\sum_{x'} y_{x'} = 1 \quad \forall i = 1, \ldots, d - 2
\]

\[
y_{x'} = 1 \quad \forall X' \subseteq \{0, \ldots, d - 1\} \wedge |X'| = L + 1
\]

\[
y_{x'} \in \{0, 1\} \\
z_{i_1, i_2, i} \in \{0, 1\} \\
\forall i_1, i_2, i \in \{0, \ldots, d - 1\}
\]

\[
\sum_{x'} y_{x'} = 1 \quad \forall X' \subseteq \{0, \ldots, d - 1\} \wedge |X'| = L + 1
\]

C Computing Multi-Dimensional Convolutions with FFT

The JR-algorithm depends on multiplying multi-variate polynomials. Here we describe how to do this using the well-known fast Fourier transformation (FFT). We start by introducing multi-dimensional polynomials as well as multi-dimensional discrete Fourier transformations (DFTs).

**Definition 10.** A multivariate or multi-dimensional polynomial \( p \) of \( d \) variables \( x_1, \ldots, x_d \) and coordinate degree \( n = (n_1, \ldots, n_d) \) is a linear combination of monomials, i.e.

\[
p(x) = \sum_{k \leq n-1} p_k x^k \quad \text{where} \quad x^k := \prod_{j=1}^{d} x_j^{k_j} \quad \forall k \in \{0, \ldots, n_1 - 1\} \times \cdots \times \{0, \ldots, n_d - 1\},
\]

\( n - 1 = (n_1 - 1, \ldots, n_d - 1) \), and \( p_k \in \mathbb{C} \) f.a. \( k \leq n - 1 \). So \( p \) is a (univariate) polynomial of degree \( n_j \) in each coordinate direction \( j \).

**Definition 11.** For two multivariate polynomials \( f(x) = \sum_{k \leq n-1} f_k x^k \), \( g(x) = \sum_{k \leq n-1} g_k x^k \) the multi-dimensional discrete convolution \( f \ast g \) is defined by

\[
c_k = \sum_{j \leq k} f_j \cdot g_{k-j}, \quad \text{i.e.} \quad c_{(k_1, \ldots, k_d)} = \sum_{j_1=0}^{k_1} \sum_{j_2=0}^{k_2} \cdots \sum_{j_d=0}^{k_d} f_{(j_1, \ldots, j_d)} \cdot g_{(k_1-j_1, \ldots, k_d-j_d)}.
\]

Hence, a truly primitive approach to compute \( f \ast g \) takes time \( \mathcal{O}(\sum_{k \leq 2n-2} \Pi_{j=1}^{d}(k_j + 1)) \leq \mathcal{O}(2^d N^2) \) where \( N = \prod_{j=1}^{d} n_j \) is the number of all input points.

However, the convolution theorem says that \( \mathcal{F}(f \ast g) = N \cdot \mathcal{F}(f) \odot \mathcal{F}(g) \) where \( \mathcal{F} \) denotes the Fourier transform operator and \( \odot \) denotes the point-wise multiplication, i.e. \((v \circ w)_k = v_k \cdot w_k\). Therefore, one can compute the convolution \( f \ast g \) by computing

\[
f \ast g = \mathcal{F}^{-1}\{N \cdot \mathcal{F}(f) \odot \mathcal{F}(g)\}.
\]

For our goals we only depend on the discrete Fourier transformation as follows.
Definition 12. The multi-dimensional discrete Fourier transformation (DFT) \( \hat{f} = \mathcal{F}\{f\} \) of \( f \) is defined by

\[
\hat{f}_k = \sum_{\ell \leq n-1} f_\ell \cdot \exp \left( -i2\pi \sum_{j=1}^{d} \frac{k_j \ell_j}{n_j} \right) \quad \forall k \leq n - 1
\]

whereas the inverse multi-dimensional DFT \( \mathcal{F}^{-1}\{\hat{f}\} \) of \( \hat{f} \) is given by

\[
f_k = \frac{1}{N} \sum_{\ell \leq n-1} \hat{f}_\ell \cdot \exp \left( i2\pi \sum_{j=1}^{d} \frac{k_j \ell_j}{n_j} \right) \quad \forall k \leq n - 1.
\]

To give more light to these definitions let us consider the 1-dimensional case, i.e. \( d = 1, n = n_1 \). Then Definition 12 simplifies to

\[
\hat{f}_k = \sum_{\ell=0}^{n-1} f_\ell \cdot e^{-i2\pi \ell k/n} \quad \text{and} \quad f_k = \frac{1}{n} \sum_{\ell=0}^{n-1} \hat{f}_\ell \cdot e^{i2\pi \ell k/n} \quad \forall k = 0, \ldots, n - 1.
\]

Obviously, since \( \hat{f}_k = f(e^{-i2\pi \ell k/n}) \) for \( k = 0, \ldots, n - 1 \) one can compute the DFT of \( f \) in time \( O(n^2) \) by simply evaluating \( f \) in \( n \) points. Fortunately, this running time can be reduced to \( O(n \log n) \) by algorithmically exploiting the fact, that the evaluation of

\[
f(x) = (f_0 x^0 + f_2 x^2 + \ldots) + x \cdot (f_1 x^0 + f_3 x^2 + \ldots) = p(x^2) + x \cdot q(x^2)
\]

can always be written by the evaluation of two polynomials \( p, q \) of half-sized degrees for any input point \( x \). This running time improvement is the reason to call a DFT an FFT. Also the inverse can be computed efficiently by using just the same trick, since \( f_k = \frac{1}{n} \hat{f}(e^{i2\pi \ell k/n}) \). See Fig. 3 for an illustration of the whole computation.

\[
\begin{array}{c|c|c|c|c}
\hline
f, g & \text{DFT!} & f \ast g & \text{FFT!} & f \ast g \\
\hline
& 2 \cdot O(n^2) & O(n^2) & 2 \cdot O(n \log n) & O(n \log n) \\
\hline
\mathcal{F}\{f\}, \mathcal{F}\{g\} & O(n) & \mathcal{F}\{f\} \circ \mathcal{F}\{g\} & \mathcal{F}\{f\}, \mathcal{F}\{g\} & \mathcal{F}\{f\} \circ \mathcal{F}\{g\} \\
\hline
\end{array}
\]

Figure 3 Computing a 1D-convolution with discrete/fast Fourier transformations

Finally, turning back to the general case \( n = (n_1, \ldots, n_d) \) we can compute the multi-dimensional FFT of \( f \) by computing 1-dimensional FFTs. In more detail, we may divide the whole computation into \( N/n_j \) 1-dimensional transformations of size \( n_j \) for each coordinate direction \( j \). This yields the usual running time of

\[
\sum_{j=1}^{d} \frac{N}{n_j} O(n_j \log n_j) = O(N \sum_{j=1}^{d} \log n_j) = O(N \log N).
\]

D Computational Results for E3, E4, and BIG
| family | m   | n   | $U$     | better | equal | avg_quot | avg_time |
|--------|-----|-----|---------|--------|-------|----------|----------|
| E3     | 10  | 31  | [100,200] | 0      | 0     | 1.07     | 182      |
| E3     | 10  | 31  | [1,100]  | 0      | 0     | 1.08     | 82       |
| E3     | 10  | 32  | [100,200] | 0      | 2     | 1.07     | 179      |
| E3     | 10  | 32  | [1,100]  | 0      | 0     | 1.09     | 81       |
| E3     | 10  | 41  | [100,200] | 0      | 0     | 1.07     | 220      |
| E3     | 10  | 41  | [1,100]  | 0      | 0     | 1.1      | 82       |
| E3     | 10  | 42  | [100,200] | 0      | 0     | 1.08     | 225      |
| E3     | 10  | 42  | [1,100]  | 0      | 0     | 1.1      | 80       |
| E3     | 3   | 10  | [100,200] | 4      | 15    | 1.03     | 114      |
| E3     | 3   | 11  | [1,100]  | 3      | 21    | 1.03     | 67       |
| E3     | 3   | 13  | [100,200] | 20     | 7     | 1.03     | 100      |
| E3     | 3   | 13  | [1,100]  | 4      | 15    | 1.03     | 60       |
| E3     | 3   | 14  | [100,200] | 20     | 1     | 1.03     | 146      |
| E3     | 3   | 14  | [1,100]  | 6      | 7     | 1.03     | 62       |
| E3     | 3   | 16  | [100,200] | 30     | 2     | 1.02     | 166      |
| E3     | 3   | 16  | [1,100]  | 9      | 13    | 1.03     | 67       |
| E3     | 3   | 17  | [100,200] | 7      | 1     | 1.03     | 146      |
| E3     | 3   | 17  | [1,100]  | 5      | 11    | 1.03     | 59       |
| E3     | 5   | 16  | [100,200] | 1      | 1     | 1.05     | 135      |
| E3     | 5   | 16  | [1,100]  | 0      | 3     | 1.06     | 66       |
| E3     | 5   | 17  | [100,200] | 8      | 2     | 1.03     | 149      |
| E3     | 5   | 17  | [1,100]  | 2      | 4     | 1.05     | 68       |
| E3     | 5   | 21  | [100,200] | 19     | 3     | 1.02     | 184      |
| E3     | 5   | 21  | [1,100]  | 1      | 0     | 1.06     | 65       |
| E3     | 5   | 22  | [100,200] | 4      | 2     | 1.04     | 178      |
| E3     | 5   | 22  | [1,100]  | 1      | 1     | 1.06     | 63       |
| E3     | 5   | 26  | [100,200] | 11     | 1     | 1.04     | 185      |
| E3     | 5   | 26  | [1,100]  | 2      | 2     | 1.06     | 67       |
| E3     | 5   | 27  | [100,200] | 1      | 0     | 1.05     | 166      |
| E3     | 5   | 27  | [1,100]  | 2      | 1     | 1.05     | 67       |
| E3     | 8   | 25  | [100,200] | 1      | 0     | 1.06     | 176      |
| E3     | 8   | 25  | [1,100]  | 0      | 0     | 1.07     | 74       |
| E3     | 8   | 26  | [100,200] | 0      | 0     | 1.06     | 177      |
| E3     | 8   | 26  | [1,100]  | 0      | 0     | 1.07     | 81       |
| E3     | 8   | 33  | [100,200] | 0      | 1     | 1.06     | 202      |
| E3     | 8   | 33  | [1,100]  | 0      | 0     | 1.09     | 72       |
| E3     | 8   | 34  | [100,200] | 1      | 0     | 1.06     | 195      |
| E3     | 8   | 34  | [1,100]  | 0      | 0     | 1.08     | 78       |
| E3     | 8   | 41  | [100,200] | 0      | 0     | 1.06     | 207      |
| E3     | 8   | 41  | [1,100]  | 0      | 0     | 1.08     | 76       |
| E3     | 8   | 42  | [100,200] | 0      | 0     | 1.06     | 211      |
| E3     | 8   | 42  | [1,100]  | 0      | 0     | 1.08     | 77       |
### Table 6  
Computational results for the classes of family E4

| family | m | n          | U                | better | equal | avg_quot | avg_time |
|--------|---|------------|------------------|--------|-------|---------|----------|
| E4     | 2 | 10         | [1, 20]          | 4      | 61    | 1.01    | 39       |
| E4     | 2 | 10         | [1, 100]         | 14     | 21    | 1.01    | 48       |
| E4     | 2 | 10         | [20, 50]         | 10     | 12    | 1.02    | 46       |
| E4     | 2 | 10         | [50, 100]        | 14     | 9     | 1.03    | 59       |
| E4     | 2 | 10         | [100, 200]       | 15     | 3     | 1.02    | 71       |
| E4     | 2 | 10         | [100, 800]       | 14     | 8     | 1.01    | 97       |
| E4     | 3 | 9          | [1, 20]          | 0      | 43    | 1.03    | 37       |
| E4     | 3 | 9          | [1, 100]         | 6      | 29    | 1.03    | 57       |
| E4     | 3 | 9          | [20, 50]         | 2      | 9     | 1.05    | 61       |
| E4     | 3 | 9          | [50, 100]        | 2      | 0     | 1.05    | 75       |
| E4     | 3 | 9          | [100, 200]       | 2      | 0     | 1.05    | 95       |
| E4     | 3 | 9          | [100, 800]       | 5      | 14    | 1.03    | 132      |

### Table 7  
Computational results for the classes of the additional family BIG

| family | m | n          | U                | better | equal | avg_quot | avg_time |
|--------|---|------------|------------------|--------|-------|---------|----------|
| BIG    | 25| 100        | [1, 1000]        | 0      | 0     | 1.13    | 171      |
| BIG    | 50| 200        | [1, 1000]        | 0      | 0     | 1.14    | 172      |
| BIG    | 75| 300        | [1, 1000]        | 0      | 0     | 1.14    | 179      |
| BIG    | 100| 400       | [1, 1000]        | 0      | 0     | 1.15    | 180      |