Approximability of Scheduling with Fixed Jobs *

Mark Scharbrodt † Angelika Steger ‡ Horst Weisser †

September 2, 1999

Abstract

Scheduling problems of minimizing the makespan on identical parallel machines are among the most well studied problems — especially in the field of approximation. In modern industrial software however, it has become standard to work on a variant of this problem, where some of the jobs are already fixed in the schedule. The remaining jobs are to be assigned to the machines in such a way that they do not overlap with fixed jobs. This problem variant is the root of many real world scheduling problems where pre–assignments on the machines are considered, such as cleaning times or jobs that have already started. In our paper we investigate the approximability of the scheduling problem with fixed jobs. We present a polynomial time approximation scheme ($PTAS$) for the case that the number m of machines is constant. For our $PTAS$ we propose a new technique by partitioning an underlying packing problem into a reasonable unrelated family of restricted bin packing problems. We also generalize the $PTAS$ to the case that the machines are independent and run at different speeds. Moreover, we will demonstrate that, assuming $P \neq NP$, there is no arbitrarily close approximation in the general case when the number of machines is part of the input. This will be extended by showing that there is no asymptotic $PTAS$ in the general machine case. We finally show that there exists no $FPTAS$ in the constant machine case, unless $P = NP$. These results contrast to the classical problem of minimizing the makespan where the existence of a $PTAS$ resp. of an $FPTAS$ for the variable resp. the constant machine case has been proven.

key words: polynomial approximation scheme, scheduling, makespan

1 Introduction

In the standard scheduling problem of minimizing the makespan one is given a set of n independent jobs with integral processing times (or *sizes*) p_j to be scheduled on m identical machines. The optimization problem is to assign all

*An abstract of this paper appeared in the *Proceedings of the Tenth Annual ACM–SIAM Symposium on Discrete Algorithms, Baltimore, 1999, pp. 961-962.*

†Lehrstuhl für Brauereianlagen und Lebensmittel–Verpackungstechnik, Technische Universität München in Weihenstephan, D–85350 Freising–Weihenstephan, Germany, {scharbro, weisser}@blv.blm.tu–muenchen.de

‡Institut für Informatik, Technische Universität München, D–80290 München, Germany, steger@informatik.tu-muenchen.de

jobs to the machines, minimizing the latest completion time (the makespan) C_{max}. Even though this problem is NP–hard, it is usually considered as a relatively easy problem, in the sense that quite good and efficient approximation algorithms exist.
The investigation of such approximation algorithms with good (worst case) performance guarantee (the ratio of the makespan achieved by the algorithm over the makespan of the best schedule, maximized over all possible inputs) was pioneered by Graham. He showed that a simple list schedule achieves a performance ratio of $2 - \frac{1}{m}$. In subsequent papers this has been improved to a (offline) 1.2–approximation [Gr69], [CGJ78] and [Fr84]. In [HS87], Hochbaum and Shmoys proposed a linear time approximation scheme introducing the technique of *dual approximation*. Note that, as the minimum makespan problem is strongly NP–hard (see [GJ79]), no fully polynomial time approximation scheme can exist for this problem, unless $P = NP$. If the number of machines is not part of the input but fixed in advance (the so-called constant machine case), the situation is slightly better. Here already Graham [Gr69] had provided a polynomial approximation scheme. In 1976 Sahni [S76] then presented a fully polynomial time approximation scheme for the minimum makespan problem.
Unfortunately, these good news of the approximability for the standard scheduling problem is confronted by the fact that most real world scheduling problems come with additional side-constraints violating the simple setup of the standard scheduling problem considered above. Constraints like release dates, classes of precedence constraints or set up problems give an example, but in fact, the magnitude of variants and special models for scheduling is almost Babylonian. We refer to [LLRS93] and [Pi95] for a survey and additional pointers to the literature.
One aspect common to most variants of the scheduling problems considered in the literature is that they assume that the machines are freely available until all jobs are processed. This, however, again conflicts with many real world applications. Consider for example applications in production planning where one typically deals with pre–assigned jobs. Examples are jobs that are currently processed (usually at the beginning of the schedule) or pre-assignments such as cleaning jobs and/or preventive maintenance. A second example is software for industrial scheduling which usually provides functions for interactive scheduling. Such a system allows to fix some jobs manually while the remaining jobs are scheduled in an automated way. The idea behind this is that users can incorporate their own knowledge or preferences into the construction of the schedule.
Modeling such kind of scheduling problems is easily achieved by considering some of the jobs, say k of the n jobs, as *fixed*. That is, they are a priori assigned to be performed on a given machine at a given time. The objective is to schedule the remaining $n - k$ jobs in such a way that C_{max} is minimized.
Using a straightforward reduction from 3–*PARTITION* one immediately checks that the scheduling problem with fixed jobs is already strongly NP–hard in the single machine case (see also Theorem 6.1). Since the completion time of the last fixed job is a lower bound for the optimal makespan C_{opt}, a $(2 + \epsilon)$–approximation can be achieved even if the available periods between subsequent

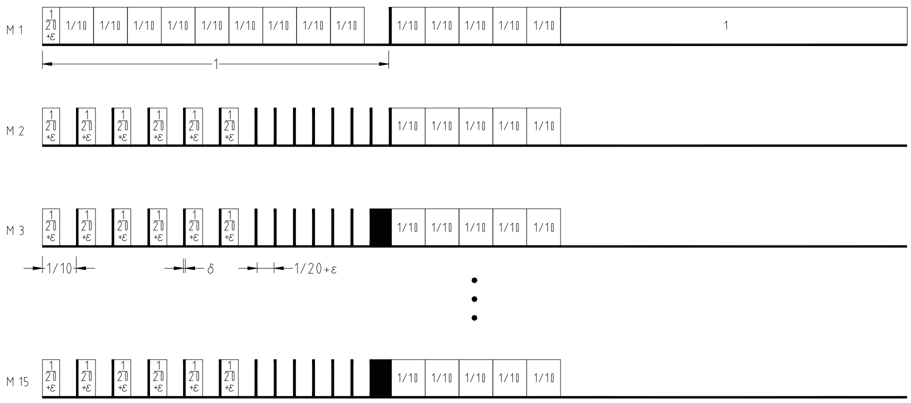

Figure 1: Lower bound example for fixed job variant of Graham's LIST–algorithm. The 15 machine instance consists of 1 job with processing time 1, 84 jobs with processing time $\frac{1}{10}$ and 85 jobs with processing time $\frac{1}{20} + \epsilon$ for a suitable small value $\epsilon > 0$. The instance has also a series of fixed jobs depicted as black squares (resp. lines for those of an infinitesimal length δ). The above schedule with makespan $2.5 + \delta$ is obtained for a suitable order of the job list. The optimal makespan is $1 + \delta$.

fixed jobs are ignored. Namely, the $n - k$ free jobs can be scheduled using the dual approximation scheme by Hochbaum and Shmoys ([HS87]) and then the corresponding partial schedule on these jobs is started after the last fixed job. As for the online case the variant of Graham's LIST–algorithm that also assigns the jobs only after the last fixed job achieves a worst case ratio of $3 - \frac{1}{m}$. A more sophisticated variant can additionally use the gaps between two subsequent fixed jobs, i. e. each job is scheduled as early as possible. Figure 1, however, gives an example where the latter version of Graham's LIST–algorithm achieves only a performance ratio slightly smaller than 2.5. We omit further details and will focus in this paper on the approximability status of our problem. We will show that the scheduling problem with fixed jobs has, apart from its practical relevance, some appealing properties from a purely complexity theoretic point of view. We present a polynomial time approximation scheme for the constant machine case. The approximation scheme will be further generalized to the case that the machines process the jobs at different speeds. This result is essentially best possible, as we also show that on the one hand in the case that the number of machines is specified as part of the input a polynomial time approximation scheme does not exist, unless $P = NP$, and that on the other hand in the constant machine case a fully polynomial time approximation scheme does not exist, unless $P = NP$.
The remainder of the paper is organized as follows: In Section 2, we outline the crucial steps of the polynomial time approximation scheme for the scheduling problem with fixed jobs. Section 3 then presents the final algorithm. This is followed by a detailed analysis of the algorithm in Section 4. In Section 5 we deal with the uniform parallel machine case and Section 6 finally presents non–approximability results for the scheduling problem with fixed jobs.

2 The polynomial time approximation scheme

In what follows we will present the polynomial time approximation scheme for the scheduling problem with fixed jobs. We attack the scheduling problem as a bin packing problem with bins of variable sizes. Given a candidate C for the makespan a corresponding bin packing problem is obtained by defining the gaps between two subsequent fixed jobs on the same machine and before the first fixed job on every machine to be "closed" bins which must not be overloaded and the remaining extra bin on every machine that starts after the last fixed job and ends at time C to be an "open" bin which might be overloaded. The items to be packed are the $n-k$ jobs that are not fixed. We call this packing problem shortly VBP. Summarizing, our algorithm has the following framework:

- *Propose a candidate C for the makespan within a binary search procedure: Check whether the underlying packing problem VBP has a feasible solution that does not overload the closed bins, but may overload the open bins "slightly".*
- *Output the schedule corresponding to the packing found for the smallest C.*

For the special case of the standard scheduling problem, the problem VBP agrees with the standard bin packing problem: all bins have the same capacity and are "open" bins. The polynomial time approximation scheme proposed by Hochbaum and Shmoys [HS87] approximates the *feasibility* of the bin packing problem in that each bin is allowed to be overloaded by up to a factor of $1+\epsilon$. This yields — speaking in terms of the underlying scheduling problem — a $(1+\epsilon)$–approximation of the makespan. The resulting relaxed bin packing problem is fairly easy to solve as an integer linear program on a constant number of variables. To achieve this, items are rounded in their sizes and small items are temporarily disregarded and later inserted by a first–fit strategy.
For the scheduling problem with fixed jobs the situation is considerably harder. In particular, two problems appear. Firstly, the rounding of item sizes could yield infeasible schedules caused by overlappings with fixed jobs. Secondly, even the small items turn out to be hard to pack. We clear these hurdles that come along with fixed jobs by building our algorithm out of three major parts. The first building block is a partition of the problem VBP into a family of (reasonably) unrelated restricted packing problems. We then develop a procedure for solving a single instance of such a packing subproblem as an integer linear program (ILP). The final part consists of a series of greedy packings that packs the items/jobs left over by the solutions of the packing subproblems.
In the next sections we will consider these ingredients of our algorithm in turn and show how they can be tied together to build up the entire algorithm for approximately solving the VBP. We start with some definitions:

Definition 2.1 *For any set I, consisting of bins or items, let $SIZE(I)$ denote the total size of all elements in I. We further define* $\max(I)$ *and* $\min(I)$ *to be the maximum resp. minimum size of an element in I. Sizes of items i will be denoted by p_i and sizes of bins j by s_j.*

As a notational convenience we will further assume that all bins and items in instances of VBP are already scaled by $\frac{1}{C}$. Their sizes are therefore bounded by one. We also assume that $\epsilon > 0$ is an arbitrary but fixed constant.

2.1 Building block 1: The partition procedure

We will create a family of packing problems by partitioning the bin set according to bin sizes. The partition is based on identifying classes of bins which will have no significant impact on the packing and can therefore be neglected. This will then allow us to partition the packing problem VBP into separate instances of packing problems so that in each the minimum and the maximum bin capacity will not differ "too much".

Lemma 2.2 *Given an instance I for the packing subproblem VBP defined via C where all bins and items are scaled by $\frac{1}{C}$ and where the bins are sorted according to non–increasing sizes we can, for $t = \lceil \frac{4}{\epsilon} \rceil + 3$ and $\epsilon < 1$, identify in linear time a partition $B = \hat{B} \cup B_1 \cup \ldots \cup B_r$ of the bin set such that*

$$\max(\hat{B}) \leq \epsilon \cdot \max(B), \tag{1}$$

$$SIZE(\hat{B}) \leq \epsilon \cdot SIZE(B), \tag{2}$$

and such that for all $i = 1, \ldots, r$:

$$\begin{array}{rcll} \min(B_i) & \geq & \epsilon^t \max(B_i), & \textit{and} \\ \max(B_{i+1}) & < & \epsilon \min(B_i). & \end{array} \tag{3}$$

Proof. We first partition the bin set B in linear time into a number $u \leq |B|$ of *buckets* $\bar{B}_i$, $1 \leq i \leq u$. We do that by placing the bins successively into buckets, opening a new one as soon as the size of the bin currently under consideration is smaller than ϵ times the size of the maximum bin in the current bucket. As we assumed that the bins are sorted by non–increasing sizes this can easily be done in linear time. The buckets $\bar{B}_i$ therefore have the following properties:

$$\min(\bar{B}_i) \geq \epsilon \max(\bar{B}_i) \qquad \text{and} \qquad \min(\bar{B}_i) > \max(\bar{B}_{i+1}) \qquad \text{for all } i \geq 1.$$

The following procedure will then identify the desired partition in linear time.

procedure partition

- $s \leftarrow 1$; $j \leftarrow 2$; $\hat{B} \leftarrow \emptyset$; $i \leftarrow 1$; $B_1 \leftarrow \emptyset$;
- **while** $j \leq u$ **do begin**
  - **if** $\max(\bar{B}_j) \geq \epsilon \cdot \min(\bar{B}_{j-1})$ **then do begin**
    - **if** $j - s = \lfloor \frac{t}{4} \rfloor - 1$ **then do begin**
      - Choose $s \leq i^* \leq j$ such that $SIZE(\bar{B}_{i^*}) = \min_{s \leq i \leq j} SIZE(\bar{B}_i)$;
      - **if** $i^* > s$ **then** $B_i \leftarrow B_i \cup \bar{B}_s \cup \ldots \cup \bar{B}_{i^*-1}$;
      - **if** $B_i \neq \emptyset$ **then** $i++$; $B_i \leftarrow \emptyset$;
      - **if** $i^* = 1$ **then** $B_i \leftarrow \bar{B}_{i^*}$ **else** $\hat{B} \leftarrow \hat{B} \cup \bar{B}_{i^*}$;
      - **if** $i^* < j$ **then** $B_i \leftarrow B_i \cup \bar{B}_{i^*+1} \cup \ldots \cup \bar{B}_j$;
      - $s \leftarrow j + 1$;
      - **end**

end

else do begin

if $s < j$ **then** $B_i \leftarrow B_i \cup \bar{B}_s \cup \ldots \cup \bar{B}_{j-1}$;

if $B_i \neq \emptyset$ **then** $i++$; $B_i \leftarrow \emptyset$;

$s \leftarrow j$;

end

$j++$;

end

$B_i \leftarrow B_i \cup \bar{B}_s \cup \ldots \cup \bar{B}_u$;

Informally speaking, the algorithm considers the buckets from $\bar{B}_1$ to $\bar{B}_u$. Whenever it detects a sequence of $\lfloor \frac{t}{4} \rfloor$ successive buckets it chooses the smallest of those buckets for $\hat{B}$. To see that the computed partition $\hat{B} \cup B_1 \cup \ldots \cup B_r$ has the desired properties, observe first that a class B_j consists of at most $2\lfloor \frac{t}{4} \rfloor \leq \frac{t}{2}$ successive buckets $\bar{B}_i$ and that, by construction, the minimum bin sizes of any two of these successive buckets are within a factor of ϵ^2. This implies (3). Property (1) follows from the special treatment of the case $i^* = 1$. To see that also (2) holds observe that whenever an index i^* is chosen it clearly satisfies the following property:

$$SIZE(\bar{B}_{i^*}) \leq \frac{1}{j-s+1} \sum_{i=s}^{j} SIZE(\bar{B}_i) = \frac{1}{\lfloor \frac{t}{4} \rfloor} \sum_{i=s}^{j} SIZE(\bar{B}_i).$$

Taking the sum over all buckets $\bar{B}_{i^*}$, we obtain (observe that the buckets involved on the right hand side are different for different buckets $\bar{B}_{i^*}$)

$$\sum_{j \in \hat{B}} s_j = \sum_{i=1}^{r} SIZE(\bar{B}_{i^*}) \leq \frac{1}{\lfloor \frac{t}{4} \rfloor} SIZE(B) \leq \epsilon \cdot SIZE(B).$$

□

Lemma 2.2 induces the following family of restricted variable sized bin packing problems which we will call $RVBP_i$, $i = 1, \ldots, r$.
Problem $RVBP_i$ consists of

all bins contained in B_i,

job set $J_i := \{j \in J \mid \epsilon \min(B_i) \leq p_j \leq \max(B_i)\}$.

The aim is to minimize the total size of items that remain unpacked for each $RVBP_i$.
It is an important observation that all packing problems $RVBP_i$ have the property that the size of the minimum item resp. bin and that of the maximum item resp. bin differ by at most a factor of ϵ^{t+1} resp. ϵ^t. And, as we will see in the the next section, such packing problems can be approximately solved as ILPs. We may solve them *separately* and later merge the separate solutions of the $RVBP_i$s using a suitable greedy procedure (Section 2.3). The greedy procedure packs the items left over by the $RVBP_i$s. Hereby, an overload of the bins will be generated. We will show that the overload for bins B_i is bounded by an ϵ fraction of the bin size. For bins $\hat{B}$, obtained from Lemma 2.2, we can only guarantee that the overload is bounded by the size of each such bin. Inequality

(2) of Lemma 2.2, however, implies that one can afford such a strong overload for $\hat{B}$ since $SIZE(\hat{B})$ is only a small fraction of the total bin size. As we will see later, the corresponding overload can therefore be rescheduled without affecting the makespan too much.

2.2 Building block 2: Restricted variable sized bin packing

In this section we present two versions of a restricted packing problem ($RVBP$). The first version of $RVBP$ is formulated in the form $RVBP[\delta, \rho, u, v]$, meaning that all item resp. bin sizes have to belong to the interval $[\delta, 1]$ resp. the interval $[\rho, 1]$ and u and v determine the maximum number of *distinct* item and bin sizes that are allowed. The second version, written as $RVBP[\delta, \rho]$, relaxes the constraints on the input instances and allows the bins and items to take on arbitrarily many distinct sizes. Their minimal sizes however remain bounded by δ and ρ. Thus, all item sizes are in the interval $[\delta, 1]$ and all bin sizes are in the interval $[\rho, 1]$. We consider the problems $RVBP$ as *optimization problems.* That is, we aim at finding a solution which minimizes the sum of the sizes of all items that remain unpacked.
In the next two subsections we show that $RVBP[\delta, \rho]$ can be (approximately) solved in linear time by solving a related $RVBP[\delta, \rho, u, v]$ problem exactly.
Before we do that we note that the problems $RVBP_i$ which were introduced at the end of the last section can all be viewed as instances of $RVBP[\epsilon^{t+1}, \epsilon^t]$ by simply scaling all bin and item sizes by $1/\max(B_i)$.

2.2.1 $RVBP[\delta, \rho, u, v]$

The input instance $I = I(J, B)$ for $RVBP[\delta, \rho, u, v]$ can be written by two multi sets $J = \{n_1 : p_1, n_2 : p_2, \ldots, n_u : p_u\}$ and $B = \{k_1 : s_1, k_2 : s_2, \ldots, k_v : s_v\}$, such that $1 \geq s_1 > s_2 > \cdots > s_v \geq \rho$, $1 \geq s_1 \geq p_1 > p_2 > \cdots > p_u \geq \delta$, $n = \sum_{i=1}^{u} n_i$ and $k = \sum_{i=1}^{v} k_v$, where n is the total number of items, k the total number of bins and where n_i and k_i give the number of items resp. bins of size p_i and s_i. Define a *configuration* to be a packing of a bin of a specific size with a specific set of items such that the sum of item sizes does not exceed the bin size. A configuration for a bin can be denoted by a u–vector $(l_1, \ldots, l_u)$ of non–negative integers, such that l_i is the number of items of size p_i that are packed into that bin. Since we are only packing items that are greater or equal than δ, we know that at most $\lfloor \frac{1}{\delta} \rfloor$ items can fit into a bin, thereby, limiting the number of possible configurations to a constant $q = q(\delta, u) \leq (u+1)^{\lfloor \frac{1}{\delta} \rfloor}$ for each of the v possible bin sizes.
Consider now a feasible solution x to an instance I of $RVBP[\delta, \rho, u, v]$. Clearly, x can be specified by a vector $x = (x_1, \ldots, x_{q \cdot v})$ where x_j denotes the number of bins that are packed according to configuration j.
Define now

- $\mathcal{T}_i$ the set of legal configurations for bins of size s_i
- a_{lj} as the number of items of size p_l that are used in configuration j

$RVBP[\delta, \rho, u, v]$ can then be formulated as the following integer program:

$$
\begin{array}{ll}
\textit{minimize} & \sum_{l=1}^{u} (n_l - \sum_j a_{lj} x_j) p_l \\
\textit{subject to} & \\
\forall 1 \le l \le u & \sum_j a_{lj} x_j \le n_l \\
\forall 1 \le i \le v & \sum_{j \in \mathcal{T}_i} x_j \le k_i \\
 & x_j \ \text{ integral}
\end{array}
\tag{4}
$$

We call $OPT(I)$ the optimal value for the packing problem, i. e., the minimum possible total size of items that remain unpacked. Since we assumed that δ, ρ, and u are constants, we can obtain the ILP in time linear in v. Lenstra [Le83] has shown that the ILP can be solved in time polynomial in in the size of the linear program provided u and v are fixed, i.e., in time $Poly(\log n, \log k, \frac{1}{\delta}, \frac{1}{\rho})$, where δ and ρ are constants.

2.2.2 $RVBP[\delta, \rho]$

The second problem we consider is $RVBP[\delta, \rho]$ which is defined similar to $RVBP[\delta, \rho, u, v]$ except that the number of distinct bin and item sizes need no longer be constants. The bin and item sizes, however, are still restricted to the intervals $[\delta, 1]$ for items and $[\rho, 1]$ for the bins. Again, we assume that the items are sorted according to non–increasing item sizes.

We will show in the sequel how $RVBP[\delta, \rho, u, v]$ can be used for approximately solving $RVBP[\delta, \rho]$. The underlying idea of our reduction of an instance for $RVBP[\delta, \rho]$ to an instance for $RVBP[\delta, \rho, u, v]$ is the so called linear grouping approach introduced in [DL81]. Here we will apply the grouping technique simultaneously to items *and* bins.

An instance for the $RVBP[\delta, \rho, u, v]$ is obtained from an instance $I = (J, B)$ of $RVBP[\delta, \rho]$ by the following construction: First, we group the item set $J = \{1, \ldots, n\}$ into groups $G_j = \{(j-1) \cdot K_1 + 1, \ldots, j \cdot K_1\}$ for $j = 1, \ldots, u$ and $G_{u+1} = \{u \cdot K_1 + 1, \ldots, n\}$, each of which, except the last group, consists of K_1 items where K_1 is a non–negative integer to be specified later. We define two functions INCREASING and DECREASING. The first function rounds the items of each group to its largest element. DECREASING rounds the item sizes of group G_j down to the size of the largest item in group G_{j+1} (the items in the last group are rounded down to the size of the smallest item). We will call the item groups obtained by INCREASING by $H = H_1 H_2 \ldots H_{u+1}$ and the groups obtained by DECREASING by $F = F_1 F_2 \ldots F_{u+1}$. The point of this definition is that the lists F_j and H_j, $(j = 1, \ldots u+1)$ are almost the same, except for the first resp. last item group. More specifically, as $F_j = H_{j+1}$ for $j < u$ and $H_{u+1} \subset F_u$, a packing for $\bigcup_{j \le u} F_j$ induces a packing for $\bigcup_{j=2}^{u+1} H_j$. This suggests the following algorithm: Find a packing for the list F, except for group F_{u+1}, and obtain a solution that is as good as the optimal solution of the original input (recall that items are rounded down in size). Consider the solution as a packing for the list H except of a group H_1 of items that are not packed. Clearly, the total size of items that belong to H_1 is at most K_1. Construct finally a feasible packing for the original input instance of the $RVBP[\delta,\rho]$ from the packing of H by decreasing the item sizes down to their original sizes. It clearly holds:

$$OPT(I_H) \leq OPT(I_F) + K_1 \leq OPT(I) + K_1,$$

where I_H and I_F are the instances with item lists H resp. F.
Similarly, we will deal with the bins. We will group the bins into bin groups, each (except the last group) of which consisting of a number K_2 of bins, and define INCREASING to increase bins of the j^{th} bin group to the smallest bin in group $j-1$ and the bins of the first group to the largest bin. DECREASING on the other hand rounds all bins in one group down to the size of the smallest bin in the group. Solving the $RVBP$ on the modified input list where INCREASING is applied and rounding back bin sizes to their original sizes will again yield a feasible packing where the sum of the item sizes that are not packed is $OPT(I) + K_2$. Putting together the grouping of bins and the grouping of the items in a single step will yield the desired reduction for I to an instance for the $RVBP[\delta,\rho,u,v]$ where we set $u = \lfloor\frac{|J|}{K_1}\rfloor$ and $v = \lfloor\frac{|B|}{K_2}\rfloor$. If K_1 and K_2 are defined by

$$K_1 := \lceil |J|\mu \rceil \qquad \text{and} \qquad K_2 := \lceil |B|\mu \rceil$$

(for an arbitrary but fixed constant $1 \geq \mu > 0$) then u and v are constants, and the corresponding problem $RVBP[\delta,\rho,u,v]$ can thus be solved in linear time, as shown in the previous section. The following lemma is therefore evident.

Lemma 2.3 *Let $1 \geq \mu > 0$ be an arbitrary, but fixed constant and let $K_1 := \lceil |J|\mu \rceil$ and $K_2 := \lceil |B|\mu \rceil$. Then we can find for any instance $I = I(J,B)$ for $RVBP[\delta,\rho]$ in linear time a solution such that the total size of all items that remain unpacked is at most*

$$OPT(I) + K_1 + K_2 \leq OPT(I) + \mu(|J| + |B|)) + 2.$$

(Here $OPT(I)$ denotes the total size of all items that remain unpacked in an optimal solution of the instance $I = I(J,B)$.)

2.2.3 Solving the subproblems $RVBP_i$

As already outlined above, the problems $RVBP_i$ defined at the end of Section 2.1 can all be viewed as instances of $RVBP[\epsilon^{t+1},\epsilon^t]$ by simply scaling all bin and item sizes by $1/\max(B_i)$. Clearly, Lemma 2.3 also applies for these scaled instances. That is, the sum of the sizes of unpacked items for problem $RVBP_i$ can thus be bounded by $OPT(RVBP_i) + (\mu(|J_i| + |B_i|) + 2)\max(B_i)$. We will see later that an appropriate choice for μ will guarantee that the sum of sizes of unpacked items is small enough for an $(1+\epsilon)$–approximation of the makespan.
For the $RVBP_1$, we have to consider two cases. In the case that all bins are smaller than ϵ we solve the $RVBP_1$ approximately as for all other problems $RVBP_i$. Otherwise, we proceed in a slightly different way. (The reason for this

is that items that remain unpacked may be as large as C_{opt}, and contrary to all other subproblems $RVBP_i$ their effect on the final schedule is therefore so significant that they have to be packed more carefully.) For solving $RVBP_1$ in that case, we first observe that if the number of items with size at least ϵ^{t+2} exceeds $\frac{m}{\epsilon^{t+2}}$ there exists no feasible packing, as the total bin capacity of all bins in VBP is bounded by m (recall that we scaled all bins and items by $1/C$). For $RVBP_1$ we consider the bin set $\tilde{B}_1 := B_1 \cup \{b \in \hat{B} \mid size(b) \geq \min(J_1)\}$, that is we may view $RVBP_1$ directly as a problem of type $RVBP[\epsilon^{t+2}, \epsilon^{t+2}, \frac{m}{\epsilon^{t+2}}, |\tilde{B}_1|]$ which can be solved optimally in constant time, as $|\tilde{B}_1|$ is also, trivially, bounded by $|\tilde{B}_1| \leq \frac{m}{\epsilon^{t+2}}$. If some items remain unpacked we can safely infer that there exists no feasible packing for VBP.

2.3 Building block 3: Greedy Packing

We will now come to the final ingredient of our algorithm, the greedy packing. It will be used to pack the two classes of items that have not been packed into any bin yet. The remaining items to be packed are given by the two sets

$\hat{J}_i \subset J_i$, and

$J^* = J - \bigcup_{i=1}^{r} J_i$,

where $\hat{J}_i$ denotes the set of items that remain unpacked in the solution of the problem $RVBP_i$. The second set J^* is the union of all items that have not been considered in any packing subproblem $RVBP_i$, as simply their sizes are "between" the item sizes of two subsequent packing problems. It is natural, to use the partition $J^* := \bigcup_{i=1}^{r} J_i^*$, where $J_i^* := \{j \in J \mid \epsilon \min(B_i) > p_j > \max(B_{i+1})\}$ (with the convention that $\max(B_{r+1}) = 0$).
In order to get an intuition on how the greedy packing proceeds we revisit Lemma 2.2. Note first, that the bin capacity may not be fully exploited under a greedy procedure, unless we allow all bins (even the "closed" bins) to be overloaded by one item each. Lemma 2.2 implies that one can afford to overload bins B_i^* up to their own size (cf. Section 4) while bins B_i may only be overloaded by an ϵ fraction. To ensure that no bin will be overloaded further we proceed as follows: After having solved a problem $RVBP_i$ $(i = 1, \ldots, r)$ we pack the items $p \in \hat{J}_i$ that remain unpacked in the solution of the $RVBP_i$ into the bins with size greater than $\max(B_i)$ (including bins from $\hat{B}$). For short, we will call those bins $B_i^<$, with $B_1^<$ being the empty set. Secondly, we greedily pack for each i the items in J_i^*. Again, we allow the bins to be overloaded by an item j – if its size is not larger than that of the bin. Then, for $i < r$, we continue with $RVBP_{i+1}$. Lemma 2.2 states that the bin sizes of problems $RVBP_i$ and $RVBP_{i+1}$ are separated at least by an ϵ fraction of $\min(B_i)$. It follows that each item $i \in J$ that is greedily packed considers only bins $j \in B - \hat{B}$ where $p_i \leq \epsilon \cdot s_j$. Moreover, the bins from $\hat{B}$ considered by i are at least as large as p_i. So, if i overloads any bin, it clearly respects the above bound on the overload for bins.
At the end of the algorithm we remove all items that overload a closed bin and pack them with a greedy algorithm into the open bins. This final step will

guarantee that the corresponding schedule will be feasible, as all overlappings with fixed jobs are cleared.

3 The complete algorithm

We are now able to present the complete algorithm. Recall that an instance I for the scheduling problem with fixed jobs is given by

- A set of n jobs $1, \ldots, n$ with integer processing times $p_1, \ldots, p_n$.
- A subset $i_1, \ldots, i_k$ of k fixed jobs. Each fixing consists of a starting time and an assigned machine.

The number m of machines and the desired approximation ratio are not part of the input. They are considered to be fixed. The completion time of the last fixed job is denoted by C_F. According to the context, jobs j are viewed as *items* with size p_j corresponding to their processing times.
As already outlined in the beginning of Section 2, the basic framework of our algorithm is a binary search for the optimum makespan C_{opt}. A useful lower bound for C_{opt} is $\max\{\sum_{j=1}^{n} \frac{p_j}{m}, \max_j p_j, C_F\}$. It can be easily shown that any list schedule has a makespan at most $3\max\{\sum_{j=1}^{n} \frac{p_j}{m}, \max_j p_j, C_F\}$, again by scheduling each job after the last fixed job as pointed out in the introduction.
One easily checks that there are a few simple criteria which indicate that a proposed candidate C for the makespan is smaller than C_{opt} resp. that the packing problem VBP corresponding to C has no suitable solution. We defer the precise arguments for these facts to the proof of Lemma 4.3. The algorithm can then be stated as follows:

[Algorithm A_ϵ for the makespan problem with fixed jobs]

Input: Instance I consisting of fixed and free jobs.
Output: A feasible schedule for I.

1. Sort jobs by non-increasing sizes.

2. $ub \leftarrow 3\max\{\sum_{j=1}^{n} \frac{p_j}{m}, \max_j p_j, C_F\}$;
$lb \leftarrow \max\{\sum_{j=1}^{n} \frac{p_j}{m}, \max_j p_j, C_F\}$;

while $(ub - lb \geq 1)$ **do begin**

$C \leftarrow lb + \frac{ub-lb}{2}$;
Obtain an instance of a packing problem VBP with bins B and jobs J by linearly traversing the machines and scaling all bins and items by $\frac{1}{C}$.
Call the function for solving VBP defined via the scaled bins and items. If a solution for the packing problem is obtained, set $ub \leftarrow C$, else (FAILURE) set $lb \leftarrow C$.

end

3. If no packing has been detected so far, set $C \leftarrow ub$ and again solve the packing problem VBP on the bins and items scaled by $\frac{1}{C}$.
4. Interpret the obtained packing as a schedule and return it.

We will now present the function for packing the items into the bins, which is the essential part of the algorithm.

[Function for solving VBP]

Input: Set J of items and set B of bins
Output: A packing of items into bins (which may slightly overload the open but not the closed bins) or FAILURE

1. Identify the bin sets $B_1, \ldots, B_r$ according to Lemma 2.2 and construct the problems $RVBP_i$ as indicated at the end of Section 2.1.

   For $i = 1, \ldots, r$ let

$$\begin{aligned} B_i^* &:= \{j \in B \mid \min(B_i) > s_j > \max(B_{i+1})\} \text{ and} \\ J_i^* &:= \{j \in J \mid \epsilon \min(B_i) > p_j > \max(B_{i+1})\} \end{aligned}$$

   (with the convention that $\max(B_{r+1}) = 0$);

2. **if** $\max(B_1) \geq \epsilon$ **do begin**

   if items $j \in J$ exist with $p_j > \max(B_1)$, exit with FAILURE.

   if $|J_1| \geq \frac{m}{\epsilon^{t+2}}$, exit (infeasibility detected) **else** solve $RVBP_1$ optimally (cf. Section 2.2.1), hereby we consider all bins from $B_1 \cup \{j \in \hat{B} \mid s_j \geq \min(J_1)\}$. Exit with FAILURE if some items remain unpacked;

   else

   Solve $RVBP_1$ approximately (cf. Section 2.2.1 resp. Section 2.2.3) with $\mu := \epsilon^{t+2}$.

   Let $\bar{J}_1$ denote the set of all items which remain unpacked.

   if $SIZE(\bar{J}_1) > (\mu \cdot (|J_1| + |B_1|) + 2) \max(B_1) + SIZE(B_1^*)$ **then** exit (infeasibility detected);

   end

   Pack all items $j \in J_1^*$ with a greedy algorithm into the bins $B_1 \cup B_1^*$. The greedy algorithm is allowed to put an item into any bin that is not yet completely filled (even if – after the item is inserted – the capacity of the bin is exceeded), provided the size of the item does not exceed the (total) size of the bin. Hereby, the bins should be filled up according to non–increasing sizes and the items should also be considered in sorted order according to non–increasing sizes. Exit with FAILURE if unpacked items remain.

3. for $i = 2$ **to** r **do begin**

Solve $RVBP_i$ approximately (cf. Section 2.2.3) with $\mu := \epsilon^{t+2}$. Let $\hat{J}_i$ denote the set of items which remain unpacked and let

$$B_i^< := B_1 \cup B_1^* \cup \ldots \cup B_{i-1} \cup B_{i-1}^*.$$

Pack items $j \in \hat{J}_i$ with a greedy algorithm into the bins of $B_i^<$. We allow that an item is put into a bin which is not yet completely filled even if the capacity of the bin is exceeded. Let $\bar{J}_i$ denote the set of all items which remain unpacked.

if $SIZE(\bar{J}_i) > (\mu(|J_i| + |B_i|) + 2)\max(B_i) + SIZE(B_i^*)$, exit with FAILURE;

Pack all items $p \in J_i^*$ with a greedy algorithm into the bins of $B_i^< \cup B_i \cup B_i^*$. The greedy algorithm is allowed to put an item into any bin that is not yet completely filled (even if – after the item is inserted – the capacity of the bin is exceeded), provided the size of the item does not exceed the (total) size of the bin. Hereby, the bins should be filled up according to non–increasing sizes and the items should also be considered in sorted order according to non–increasing sizes. Exit with FAILURE if unpacked items remain.

end

4. For every overloaded bin remove the *last* item from this bin. Let J_R denote the set of all removed items.

Let $\bar{J} := \bar{J}_1 \cup \cdots \cup \bar{J}_r \cup J_R$.

Pack all items in $\bar{J}$ with a greedy algorithm into the open bins, putting each item into that bin which has currently the least overload.

end

4 Analysis

We are now ready to tie the results of the previous sections together in order to obtain the following theorem:

Theorem 4.1 *There exists an $\mathcal{O}(n \log n)$ variant of Algorithm A_ϵ that finds a feasible schedule with performance ratio $1 + 10\epsilon$ for an instance I of the scheduling problem with fixed jobs and a constant number of machines.*

We prove the theorem in two steps. We first show that Algorithm A_ϵ will always construct a feasible schedule with the desired performance ratio. Then we show how to adapt A_ϵ in order to achieve the $\mathcal{O}(n \log n)$ bound on the computation time.

Lemma 4.2 *For all $C \geq C_{opt}$ the function for solving VBP constructs a packing that does not overload the closed bins.*

Proof. We have to show that for $C \geq C_{opt}$ the algorithm will never exit without a packing. It suffices to consider the i^{th} iteration of the **for**-loop in Step 3, for an arbitrary i. The function for solving VBP will only return FAILURE if one of the item sets $\hat{J}_i$ or J_i^* cannot be completely packed by the greedy algorithm. Consider first the item set $\hat{J}_i$, i.e., the items that remain unpacked in the solution of $RVBP_i$. We fix an arbitrary optimal schedule σ_{opt} with makespan C_{opt}. Let $\hat{J}_i^\sigma$ denote the set of items $j \in J_i$ that are not packed into bins B_i in the optimal schedule σ_{opt}. Within the solution of $RVBP_i$ items in J_i are packed in the bins B_i by solving $RVBP_i$ (almost) optimally (resp. optimally, if $i = 1$ and $\max(B_1) \geq \epsilon$). The corresponding error is bounded by $(\mu(|J_i| + |B_i|) + 2) \cdot \max(B_i)$ (Lemma 2.3). Since the value of the optimal solution for $RVBP_i$ is at most $SIZE(\hat{J}_i^\sigma)$ we obtain:

$$SIZE(J_i - \hat{J}_i) \geq SIZE(J_i - \hat{J}_i^\sigma) - (\mu(|J_i| + |B_i|) + 2) \cdot \max(B_i). \quad (5)$$

Now items J_i fit only into bins $B_i^< \cup B_i \cup B_i^*$ but not into any smaller bin. So, all bins $\bigcup_{j=i+1}^{r}(B_j \cup B_j^*)$ need not be considered for items in J_i, even in an optimal schedule. Since $C \geq C_{opt}$ we therefore have

$$SIZE(\hat{J}_i^\sigma) \leq fcap_i + SIZE(B_i^*), \quad (6)$$

where $fcap_i$ denotes the remaining total free capacity of all bins in $B_i^<$ before the start of the greedy algorithm of A_ϵ ($B_i^<$ contains at this time only items that must be scheduled into these bins in σ_{opt} as well). Thus,

$$SIZE(J_i) \leq fcap_i + SIZE(B_i^*) + SIZE(J_i - \hat{J}_i^\sigma). \quad (7)$$

Inequalities (5) and (7) together yield

$$\begin{aligned} SIZE(\hat{J}_i) &= SIZE(J_i) - SIZE(J_i - \hat{J}_i) \\ &\leq fcap_i + SIZE(B_i^*) + (\mu(|J_i| + |B_i|) + 2) \cdot \max(B_i). \end{aligned} \quad (8)$$

As the greedy algorithm may overload the bins, it will only leave items unpacked iff all bins in $B_i^<$ are completely full. That is, the total size of all items that remain unpacked will be at most $SIZE(B_i^*) + (\mu(|J_i| + |B_i|) + 2) \cdot \max(B_i)$. This guarantees that the algorithm does not exit at the corresponding **if**-condition whenever $C \geq C_{opt}$.
Now consider the items in J_i^*. Observe that they, as well, fit only into bins from $B_i^< \cup B_i \cup B_i^*$. Again it holds that all items that are packed into $B_i^< \cup B_i \cup B_i^*$ before applying the greedy algorithm to J_i^*, are also packed into these bins in the optimal solution. Furthermore, as we do pack the items in sorted order and do fill the bins according to non–increasing size, a similar property holds throughout the greedy packing for all bins from B_i^*: whenever we consider an item i and the largest bin that is not yet completely filled is j then all bins with size at least s_j contain only items of size at least p_i. That is, either i can be packed into j (if $p_i \leq s_j$) or we have detected that there is no feasible packing which cannot be the case for $C \geq C_{opt}$. That is, the greedy algorithm has to succeed with packing all items from J_i^*.

This shows that for all $C \geq C_{opt}$ the algorithm will never exit without a packing. As one also easily checks that the algorithm does indeed pack all items and, by construction, no closed bin is overloaded, this completes the proof of the lemma.

□

Lemma 4.3 *For $C \geq C_{opt}$ the total size of all elements in J_R is bounded by*

$$SIZE(J_R) \leq 2\epsilon \cdot SIZE(B). \tag{9}$$

Proof. The set J_R consists of all elements which had to be removed from an over-full bin. By construction the maximum size of an item which created an overload in a bin from B_i is $\epsilon \min(B_i)$ and the maximum size of an item j that created an overload in a bin k from B_i^* is s_k. Moreover, the way B_i^* is defined, we have $\sum_{k \in \bigcup_{i=1}^r B_i^*} s_k = SIZE(\hat{B})$ with $\hat{B}$ being defined according to Lemma 2.2. Using Lemma 2.2, we obtain

$$\begin{aligned} \sum_{j \in J_R} p_j &\leq \sum_{i=1}^{r} \epsilon \cdot min(B_i)|B_i| + \sum_{i=1}^{r} \sum_{k \in B_i^*} s_k \\ &\leq \sum_{i=1}^{r} (\epsilon \cdot SIZE(B_i)) + SIZE(\hat{B}) \\ &\leq 2\epsilon \cdot SIZE(B). \end{aligned}$$

□

Together with the trivial bounds,

$$\begin{aligned} SIZE(J) &\leq m, \text{ and} \\ SIZE(B) &\leq m \end{aligned}$$

(which hold as we assumed that all items and bins were scaled by $1/C$) we obtain the following corollary:

Corollary 4.4 *For $C_{opt} \leq C$ and $\epsilon \leq \frac{1}{2}$ the total size of all elements in $\bar{J}$ is bounded by*

$$SIZE(\bar{J}) \leq \epsilon \cdot (5m + 4). \tag{10}$$

Proof. The minimum size of an item resp. bin given in an instance of problem $RVBP_i$ is not smaller than $\max(B_i)\epsilon^{t+1}$. This tells us that $|B_i| \leq \frac{SIZE(B_i)}{\epsilon^{t+1}\max(B_i)}$ and that $|J_i| \leq \frac{SIZE(J_i)}{\epsilon^{t+1}\max(B_i)}$. Thus we have

$$\sum_{i=1}^{r} |B_i| \max(B_i) \leq \sum_{i=1}^{r} \frac{SIZE(B_i)}{\epsilon^{t+1}} \leq \frac{m}{\epsilon^{t+1}} \tag{11}$$

and

$$\sum_{i=1}^{r} |J_i| \max(B_i) \leq \sum_{i=1}^{r} \frac{SIZE(J_i)}{\epsilon^{t+1}} \leq \frac{m}{\epsilon^{t+1}}. \tag{12}$$

Observe that $\bar{J}_1$ is empty whenever we solved $RVBP_1$ optimally, that is, whenever $\max(B_1) \geq \epsilon$. That is, using the notation $i_0 := 1$ if $\max(B_1) < \epsilon$ and $i_0 := 2$ otherwise, we obtain (recall that we chose $\mu := \epsilon^{t+2}$)

$$\begin{aligned}
\sum_{j \in \bar{J}} p_j &= \sum_{i=i_0}^{r} \sum_{j \in \bar{J}_i} p_j + \sum_{j \in J_R} p_j \\
&\leq \sum_{i=i_0}^{r} [(\mu(|B_i| + |J_i|) + 2) \cdot \max(B_i) + SIZE(B_i^*)] + 2\epsilon \cdot SIZE(B) \\
&\leq 2\epsilon m + 2\epsilon \sum_{i=0}^{\infty} \epsilon^i + SIZE(\hat{B}) + 2\epsilon \cdot SIZE(B) \\
&\leq \epsilon \cdot (2m + 4 + 3m),
\end{aligned}$$

again using that $SIZE(\hat{B}) \leq \epsilon \cdot SIZE(B)$ and $SIZE(B) \leq m$. □

Corollary 4.4 immediately allows us to bound the makespan computed by algorithm A_ϵ.

Corollary 4.5 *For $C \geq C_{opt}$ and $\epsilon \leq \frac{1}{2}$ the algorithm finds a feasible schedule with makespan C_{A_ϵ} such that*

$$C_{A_\epsilon} \leq (1 + 9\epsilon) \cdot C.$$

Proof. First observe that all jobs in $J \setminus \bar{J}$ are packed in such a way that the resulting makespan is at most C. We thus only have to consider the items in $\bar{J}$. Clearly, if we pack them into m (initially empty) bins in a greedy fashion choosing for each item the bin which has currently the least load, the maximum load of the m bins will be, for $m > 1$, bounded by

$$\frac{1}{m} \sum_{j \in \bar{J}} p_j + \max_{j \in \bar{J}} p_j \leq 5\epsilon + 4\frac{1}{m} \cdot \epsilon + \epsilon \leq 8\epsilon,$$

where the first inequality follows from Corollary 4.4 and the fact that – by construction – all items in $\bar{J}$ have size at most ϵ. Similarly, for $m = 1$ we can directly apply Corollary 4.4 and obtain a load of the bin bounded by

$$\sum_{j \in \bar{J}} p_j \leq 9\epsilon.$$

Clearly, by also using up the remaining free capacity of the open bins the resulting load can only decrease.
That is, the makespan of the resulting schedule satisfies

$$C_{A_\epsilon} \leq C + 9\epsilon \cdot C.$$

(Note that the additional factor of C is due to the fact that within the packing problem all item and bin sizes where scaled by $1/C$.) □

In what follows, we will proof the $\mathcal{O}(n \log n)$ computation time of algorithm A_ϵ. Clearly, items can be sorted by non–increasing sizes in time $\mathcal{O}(n \log n)$. Consider now the computation time needed for solving a single instance of VBP:

- We need time $\mathcal{O}(n \log n)$ for sorting bins by non–increasing size and then for each packing problem VBP. Computing the classes B_i can then be done in linear time according to Lemma 2.2.
- For each packing problem $RVBP_i$, $1 \le i \le r$ we need linear time for constructing the ILP and polylogarithmic time for solving the ILP. Thus the computation time for each problem $RVBP_i$ is linear. In the case that problem $RVBP_1$ is solved by complete enumeration, then again linear time is needed (cf. Section 2.2.1 resp. 2.2.3).

  Packing items with the greedy algorithm consumes linear time, since we simply fill up the bins successively.
- Identifying the set J_R and packing all items in $\bar{J}$ can again be carried out in linear time.

This shows that we can solve a VBP in time $\mathcal{O}(n \log n)$. It remains to count the number of iterations of VBP that are carried out within the binary search frame. In the original scheduling problem all processing times are assumed to be integers, so the binary search can be terminated, when $ub - lb < 1$. This yields already a polynomial time bound for our algorithm. We will use the following lemma, though, to improve the computation time.

Lemma 4.6 *If algorithm A_ϵ is executed with k iterations of binary search, the resulting schedule has a makespan of at most $(1+9\epsilon)(1+2^{-(k-1)})C_{opt}$.*

Thus, only $\mathcal{O}(\log(\frac{1}{\epsilon}))$ iterations are required to obtain a $(1+10\epsilon)$-approximation. The lemma is easily proven by observing that we have $ub - lb = 2^{-(k-2)} \max\{\sum_{j=1}^{n} \frac{size(p_j)}{m}, \max_j size(p_j), C_F\}$ in iteration k and that $C_{opt} \ge lb \ge \max\{\sum_{j=1}^{n} \frac{size(p_j)}{m}, \max_j size(p_j), C_F\}$.

5 The uniform parallel machine case

In the uniform parallel machine case we assume that the machines run at different speeds $r_1, \cdots, r_m$. More precisely, if job j is executed on machine i it takes p_j/r_i time units to be completed. Again the goal is to assign the jobs to the machines so that the last job finishes as early as possible. In [HS88], Hochbaum and Shmoys had presented a $PTAS$ for the standard version of the scheduling problem where no jobs are fixed. For the $PTAS$ they generalized their approach of dual approximation to the case where bins have different sizes. The computation time of their algorithm is a high polynomial in n.
The $PTAS$ presented in this paper for the scheduling problem with fixed jobs can also be generalized to the case where machines run at different speeds and the number of machines is a constant m. For simplicity we normalize the speed of the fastest machine to 1 and so we have $r_i \le 1,\ (i = 1, \cdots, n)$. Let us further assume a binary search procedure proposes a candidate C for the makespan. Since every machine i can process r_i units of processing time in one time unit, it can process a total of $C \cdot r_i$ of processing time before the deadline C.

On the other hand, C defines a packing problem with open and closed bins of variable sizes. We scale those bins and all the jobs by $\frac{1}{C}$ and multiply the sizes of bins that correspond to machine i with r_i. That is, an item j that is placed into any bin on machine i occupies $\frac{p_j}{r_i} \cdot C$ time units in the schedule on machine i. Again, the task is to decide whether there exists a packing of the scaled item set J into the resulting set of bins B. For our $PTAS$ we refine algorithm A_ϵ in that we place all items that are assigned to the set $\bar{J}$ now to the fastest machine. As $SIZE(J) \leq m$ and $SIZE(B) \leq m$, Corollary 4.4 still applies and so, for $C \geq C_{opt}$ and $\epsilon \leq \frac{1}{2}$, we obtain a total makespan of at most $C + \frac{(5m+4)\epsilon}{\max_i(s_i)} \cdot C = C + (5m+4)\epsilon \cdot C$. Calling this refined algorithm $A_\epsilon^{'}$ we obtain the following theorem:

Theorem 5.1 *There exists an $\mathcal{O}(n \log n)$ variant of Algorithm $A_\epsilon^{'}$ that finds a feasible schedule with performance ratio $1 + (5m+5) \cdot \epsilon$ for an instance I of the scheduling problem with fixed jobs and a constant number m of machines which run at different speeds.*

6 Non-approximability results

The following two theorems indicate the limits in the approximability of the scheduling problem with fixed jobs.

Theorem 6.1 *For a constant number m of machines, there is no fully polynomial time approximation scheme for the scheduling problem with fixed jobs, unless $P = NP$.*

Proof. It is well known (*cf.* [GJ79]) that no strongly NP-complete problem can be approximated by a fully polynomial time approximation scheme, unless $P = NP$. It thus suffices to reduce some strongly NP-complete problem to the scheduling problem with fixed jobs and a constant number, say 1, of machines. We will reduce from

3-$PARTITION$

INPUT: A finite set A of $3n$ elements of sizes $s_1, \ldots, s_{3n} \in \mathbb{Z}^+$ and a bound $B \in \mathbb{Z}^+$ such that $\frac{B}{4} < s_i < \frac{B}{2}$ for all $1 \leq i \leq 3n$ and that $\sum_{i=1}^{3n} s_i = n \cdot B$.

QUESTION: Does there exist a partition of A into n disjoint sets $A_1, \ldots, A_n$ such that for $1 \leq i \leq n$, $\sum_{s_j \in A_i} s_j = B$?

This problem is known to be strongly NP-complete, see [GJ79]. Given an instance of 3-$PARTITION$ we construct an instance of the 1-machine scheduling problem as follows. For every item we introduce a job whose processing time corresponds to the size of the item. Furthermore, we add a fixed job of size 1 at times $k \cdot B + (k-1)$ for all $1 \leq k \leq n$. Clearly, there exists a schedule with makespan $n \cdot (B+1)$ if and only if there exists the desired partition of the items into n sets. $\square$

Theorem 6.2 *If the number m of the machines is part of the input, there exists no polynomial time algorithm that solves the scheduling problem with fixed jobs within a performance guarantee of $\frac{3}{2} - \epsilon$ for all $\epsilon > 0$, unless $P = NP$.*

Proof. This time we reduce from the following variant of 3-$PARTITION$, which is also known to be NP-complete, see [GJ79]:

INPUT: Disjoint sets A and B containing n resp. $2n$ elements of sizes $a_1, \ldots, a_n \in \mathbb{Z}^+$ resp. $b_1, \ldots, b_{2n} \in \mathbb{Z}^+$ and a bound $L \in \mathbb{Z}^+$ such that $\sum a_i + \sum b_j = nL$.

QUESTION: Does there exist a permutation $\pi \in \mathcal{S}_{2n}$ such that for all $1 \leq i \leq n$: $a_i + b_{\pi(2i-1)} + b_{\pi(2i)} = L$?

This time we provide n machines, choose an integer number K such that $K \geq (\frac{1}{2} - \epsilon)L/(2\epsilon)$, and fix for every machine M_i, $1 \leq i \leq n$, a job with processing time a_i which ends at time $2K + L$. Furthermore, we have $2n$ non-fixed jobs with processing times $K + b_i$, $1 \leq i \leq 2n$. Clearly, there exists a schedule with makespan $2K + L$ if and only if there exists the desired permutation π. In addition, if such a permutation does not exist every schedule will have a makespan of at least $3K + L$. As $(3K + L)/(2K + L) \geq 3/2 - \epsilon$ by choice of K, the claim of the theorem follows. □

The proof of Theorem 6.2 indicates that the lower bound $\frac{3}{2}$ for the approximability of the makespan is also valid for large C_{opt}. As a consequence, there exists no *asymptotic* $PTAS$ for the scheduling problem with fixed jobs, if the number m of machines is specified as part of the input, unless $P = NP$.

7 Conclusion

We presented a polynomial time approximation scheme for the scheduling problem with fixed jobs, provided the number of machines is fixed. Our $PTAS$ is based on a new reduction to a series of restricted variable size bin packing problems ($RVBP$s) which are obtained by eliminating bins that are not important for the optimal solution. The algorithm then solves the $RVBP$s with the help of an integer linear programming approach. Items that cannot be packed are subsequently considered by a greedy algorithm. A final list scheduling algorithm packs only remaining items. The computation time is $\mathcal{O}(n \log n)$ for every fixed desired performance guarantee of $1 + \epsilon$. Also, the $PTAS$ can be extended to the case where machines run at different speeds. Our algorithm is best possible in that we also showed that neither a $PTAS$ for the scheduling problem with fixed jobs and an arbitrary number of machines nor an $FPTAS$ for the constant machine case exists, unless $P = NP$.

Acknowledgment

We are grateful to Gerhard Woeginger for suggesting the current version of the proof of Theorem 6.2, which improved our original bound of 5/4 to 3/2.

References

[CGJ78] E. G. Coffmann, Jr., M. R. Garey and D. S. Johnson. An application of bin–packing to multiprocessor scheduling, *SIAM J. Computing* 7: 1–17, 1978.

[CGJ84] E. G. Coffmann, Jr., M. R. Garey and D. S. Johnson. Approximation algorithms for bin packing problems: An updated survey. In *Analysis and Design of Algorithms in Combinatorial Optimization*, Ausiello and Luverfini (eds.), Springer Verlag, New York, 49–106, 1984.

[DL81] W. Fernandez de la Vega and G. S. Luecker, Bin packing can be solved within $1+\epsilon$ in linear time, Combinatorica 1: 349–355, 1981.

[Fr84] D. K. Friesen. Tighter bounds for the multifit processor scheduling algorithm. *SIAM J. Computing* 13: 170–181, 1984.

[Gr66] R. L. Graham. Bounds for certain multiprocessing anomalies, *Bell System Tech. J.* 45: 1563–1581, 1966.

[Gr69] R. L. Graham. Bounds on multiprocessing timing anomalies, *SIAM J. Applied Mathematics* 17: 263–269, 1969.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to theory of NP–Completeness.* W. H. Freeman, San Francisco, 1979.

[HS87] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: practical and theoretical results. *Journal of ACM* 34(1): 144–162, 1987.

[HS88] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *Siam J. Computing* 17(3): 539–551, 1988.

[Le83] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research* 8: 538–548, 1983.

[LLRS93] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys. Sequencing and scheduling: algorithms and complexity. In *Handbooks in Operations Research and Management Science, Vol. 4*, North Holland, 445–522, 1993.

[Pi95] M. Pinedo. *Scheduling: Theory, Algorithms and Systems.* Prentice–Hall, Engelwood Cliffs, NJ, 1995.

[S76] S. Sahni. Algorithms for scheduling independent tasks. *J. Assoc. Comput. Mach.* 23: 116–127, 1976.