(Incremental) Priority algorithms

Allan Borodin* Morten N. Nielsen† Charles Rackoff*

June 4, 2003

Abstract

We study the question of which optimization problems can be optimally or approximately solved by "greedy-like" algorithms. For definiteness, we will limit the present discussion to some well-studied scheduling problems although the underlying issues apply in a much more general setting. Of course, the main benefit of greedy algorithms lies in both their conceptual simplicity and their computational efficiency. Based on the experience from online competitive analysis, it seems plausible that we should be able to derive approximation bounds for "greedy-like" algorithms exploiting only the conceptual simplicity of these algorithms. To this end, we need (and will provide) a precise definition of what we mean by greedy and greedy-like.

Corresponding author:
Allan Borodin,
Department of Computer Science,
Email: bor@cs.toronto.edu.
Phone : (416)978-6416. Fax: (416)-978-1931.

Keywords: Priority algorithms, greedy algorithms, scheduling.

1 Introduction

1.1 The problems

Given its conceptual simplicity and computational efficiency, one often first tries to solve a combinatorial optimization problem by a greedy algorithm. It is also well accepted that many optimization

*Department of Computer Science, University of Toronto. Email: {bor, rackoff}@cs.toronto.edu.
†Department of Mathematics and Computer Science, University of Southern Denmark, Odense. Email: nyhave@imada.sdu.dk. Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

problems require more sophisticated approaches in order to obtain a good (worst case) approximation algorithm. But clearly in order to *prove* that a greedy or greedy-like approach will not yield a good approximation we need a precise definition for the intuitive concept of a greedy algorithm. It is also our belief that precise definitions and lower bound studies can help in the development of new algorithms. Our formalization is based on the order in which inputs are considered and hence it is important to first specify what are the inputs for a given problem. In this paper we consider various scheduling problems where there is a rather obvious and natural notion of an input. However, we also argue that the study of greedy approximation algorithms initiated here can be extended to other problem domains.

We consider the problem of scheduling jobs on parallel machines for various objective functions. Let m denote the number of (identical) machines available. For the basic maximization problem considered here, a job is characterized by a tuple of non-negative numbers $J_j = (r_j, d_j, p_j, w_j)$, where r_j denotes the release time, d_j the deadline, p_j the processing time, and w_j the weight (or profit) of the job. Given a sequence of jobs, the algorithm is allowed to reject jobs, but will only benefit from accepted jobs. Letting A denote the set of accepted jobs, the value of the objective function is $W(A) = \sum_{J_j \in A} w_j$. The algorithm must return a feasible schedule, which is an assignment of each accepted job J_j to a start time s_j on one of the m machines, such that no two jobs assigned to the same machine will overlap, i.e., $[s_i, s_i + p_i) \cap [s_j, s_j + p_j) = \emptyset$ for two different jobs J_i and J_j assigned to the same machine. (If one job starts at the time when another job finishes, it is not considered an overlap.) For every job, the assigned start time s_j must satisfy $r_j \leq s_j$ and $s_j + p_j \leq d_j$. We do not consider preemptive schedules, so a job must be run to completion from the assigned start time. Using the notation of [25, 30], the *job scheduling problem* for m identical machines is expressed as $P|r_j|\sum w_j \bar{U}_j$ when the number of machines is also a variable parameter of the input. We will devote much of our attention to the special case of *interval scheduling* for which $p_j = d_j - r_j$. For both of the interval and job scheduling problems, one can consider the following standard weight functions:

- Unit profit: $w_j = 1$ for all jobs J_j
- Proportional profit: $w_j = p_j$ for all jobs J_j
- Arbitrary profits: w_j arbitrary

For a minimization problem, we consider the classical *makespan problem*. Namely, a job J_j has a size p_j, and every job must be scheduled. The *load* on a machine is the sum of the sizes of jobs scheduled on that machine. The objective is to minimize the maximum load on any machine. Again using the standard scheduling notation, the makespan problem is expressed as $P| |C_{\max}$.

1.2 The class of greedy and other priority algorithms

For the scheduling problems of concern in this paper, we wish to abstract the main properties that constitute deterministic greedy-like algorithms [1]. First, greedy algorithms satisfy an "incremental

[1] We will concentrate on deterministic algorithms in this paper. All of our algorithm classes can be extended to permit randomized algorithms but (with one exception) that is beyond the scope of this paper. In Section 5.3, we do mention one randomized algorithm, introduced only for the purpose of discussing greedy vs non-greedy scheduling.

(by) priority" property in that the schedule is constructed incrementally with each input being considered once. To determine in which order the input jobs should be considered, the algorithm assigns a total ordering to the set of all possible jobs.

Sometimes we will describe the ordering of inputs by a priority function π mapping the set of all possible jobs to the reals, where the input job with the highest priority must be handled by the algorithm first [2]. One class of algorithms, Fixed Priority, decides this total (unchangeable) ordering before any job is scheduled and the ordering cannot be changed. For S an input sequence of jobs, the structure of a Fixed Priority algorithm is then as follows:

Fixed Priority
Ordering: Determine, without looking at S, a total ordering of all possible jobs
while not empty(S)
    $next$:= index of job in S that comes first in the ordering
    Decision: Decide if and how to schedule job J_{next}, and remove J_{next} from S

We emphasize that the (non-preemptive scheduling) decision made in each iteration of the algorithm is *irrevocable* [3]. It should also be understood that the algorithm has an internal state on which the decision is based. At one extreme, the state can record just the *configuration* (of the machines) for the scheduled jobs, disregarding jobs not scheduled. In the terminology of online algorithms, these might be called *memoryless* algorithms. (In an adaptive memoryless algorithm, the ordering or priority function used in each iteration is also just a function of the configuration and not the entire history.) At the other extreme the state can record all jobs seen thus far whether or not the job was scheduled.

We now define a more general class of algorithms, Adaptive Priority, where it is possible to specify a new ordering (on the set of all possible jobs) after each job is processed; this ordering can thus depend on jobs already seen, but not on future jobs. We again emphasize that the decisions made by the algorithm are irrevocable, and are based on an internal state. An Adaptive Priority algorithm then has the following structure:

Adaptive Priority
while not empty(S)
    Ordering: Determine (without looking at S) a total ordering of all possible jobs
    $next$:= index of job in S that comes first in the ordering
    Decision: Decide if and how to schedule job J_{next}, and remove J_{next} from S

Clearly, the fixed priority algorithms are a special case of adaptive priority algorithms and we will hereafter **identify the concept of greedy-like algorithms by the class of (adaptive)**

[2] We assume that when ordering the jobs, each job includes an integer "identifier". This allows us to have multiple copies of jobs which would otherwise be identical. It also allows us to discuss the online setting as follows: we only consider adversaries that present jobs whose identifiers are strictly increasing, and we insist that algorithms prioritize the jobs according to the identifier. Note that we are *not* insisting in general that adversaries, when presenting a set of n jobs, must choose jobs with exactly the identifiers $\{1, \cdots, n\}$. It is not clear to what extent results still hold with this weakened adversary.

[3] In the concluding section, we briefly discuss how we might define priority algorithms in the context of scheduling problems that allow preemption.

priority algorithms. We argue that "greedy algorithms" satisfy an additional property. Namely, we claim that a greedy algorithm is not only fixed or adaptive priority but moreover, **a greedy algorithm makes its irrevocable decision so that the objective function is optimized as if the input currently being considered is the final input**. For the case of the job scheduling profit problem, since the value of the objective function is completely determined by the subset of scheduled jobs, a greedy algorithm must accept and somehow schedule the current job if it can be scheduled. For the makespan problem, the greedy criteria forces the algorithm to schedule the job so as to minimize the current makespan after scheduling the job. For identical machines, one particular greedy decision is to schedule the job on a currently least loaded machine. In general, there can be many possible greedy choices.

Within the priority algorithm framework, it is natural to ask whether or not the greedy restriction is really a restriction. In the special case of online algorithms, there are many problems where greedy algorithms provably yield bad approximations (i.e. competitive ratios much worse than what is obtained by the best known online algorithms). It seems plausible then that greediness should also be restrictive for priority algorithms, or at least for the case of fixed priority algorithms. We give some evidence in Corollary 4 and Theorem 9 that the greedy condition is indeed a restriction for the class of fixed priority algorithms.

For the job scheduling profit problem considered in this paper, one might conjecture that it is always possible to convert an adaptive algorithm to be greedy by simply changing the ordering so that a job which would not be accepted is given low priority. However, we have only been able to prove such a result for memoryless algorithms. On the other hand, we do not have any example where a non greedy adaptive algorithm provably (or even apparently) yields a better approximation ratio than what can be obtained by a greedy algorithm. We state the following result in terms of the job scheduling problem but note that it holds for any problem where the subset of accepted jobs determines the value of the objective function and the objective function is "monotone" in the solution[4].

Lemma 1 For the job scheduling profit problem, any memoryless adaptive priority algorithm can be converted to a (memoryless adaptive) greedy priority algorithm which achieves at least the same profit.

Proof Intuitively, the priority function can be modified to give the lowest priority to any job that would be rejected by the decision criteria. These lowest priority jobs can then be scheduled at the end (if still possible) and this will only increase the profit. We need to show how to simulate (in each iteration) any non-greedy memoryless algorithm $\mathcal{A}$ by a greedy memoryless algorithm $\mathcal{A}'$.

Rather than "consider" a job that would not be scheduled, algorithm $\mathcal{A}'$ will give this job a low priority (i.e. a priority lower than any job that would be scheduled now). Moreover, algorithm $\mathcal{A}'$ must also guarantee that a job that would have indeed been considered but not scheduled will continue to receive low priority in subsequent iterations. The idea is as follows. When we give a priority (based on the current configuration) to a job J, we can determine (from the configuration) whether it would now be scheduled and if not we give job J low priority. But how do we ensure

[4]That is, for a profit (respectively, cost) optimization problem and for any two subsets S and S' with $S \subset S'$, $profit(S) \leq profit(S')$ (respectively, $cost(S') \leq cost(S)$).

that if J was considered (and rejected now by $\mathcal{A}$) that we wouldn't schedule this job later (given a new configuration)? Since we are assuming a memoryless algorithm, as we start a new iteration (say having scheduled r jobs), we can reconstruct the order in which we have scheduled jobs thus far during this stage, say in the order $J_1, \ldots, J_r$. We consider each of the configurations $\emptyset, \{J_1\}, \{J_1, J_2\}, \ldots, \{J_1, \ldots J_{r-1}\}$. Then in determining the priority of any job J, we consider its priority in any of these configurations, say $\{J_1, \ldots, J_i\}$ with $i < r$, and if it would not be scheduled in this configuration and the priority of J is higher than J_{i+1}, then we give J low priority.

□

We do not know if the same observation holds for adaptive algorithms without the memoryless constraint. Nor would this observation necessarily hold for other problems (like the makespan problem) where different ways of scheduling a job clearly changes the value of the objective function.

2 Related results and justification of the model

To the best of our knowledge, a precise definition of greedy approximation algorithms has not appeared in the literature although many excellent undergraduate texts (for example, see [11] and [16]) discuss greedy algorithms in a clear manner using illustrative examples. There is an elegant abstract setting in which greedy algorithms can be formulated, namely that of matroid embeddings (see, for example, [16] and [33]). Within this framework, it is the choice of the next job to be considered (i.e. the ordering function) that characterizes the greedy nature of the algorithm. That is, the algorithm (adaptively) chooses its next job so as to optimize the objective function (as if this were the last input). Clearly this kind of greedy algorithm is a special case of our adaptive algorithms. More specifically, we regard our 2-approximation adaptive algorithm Chain (for maximizing the processing time of intervals scheduled on two machines; see Theorem 5) as a greedy algorithm whereas a more restrictive view would require the algorithm to use a LPT (longest processing time) priority rule which only achieves a 3-approximation. Similarly, for the makespan problem, the more restrictive view would necessitate a SPT (shortest processing time) rule whereas LPT provides a provably better approximation ratio. Since the emphasis of our work is to establish that interesting bounds can be established for priority algorithms, our lower bounds are stronger by allowing any priority function.

The reader might ask why our priority algorithm definitions require the algorithm to order all possible jobs rather than just the set of actual input jobs. Obviously, an algorithm which can arbitrarily order a fixed set of jobs can choose an ordering relative to which there is an optimal (and greedy) scheduling of the jobs. Hence some restriction on how the jobs can be ordered is necessary. One possible restriction is to allow only certain ordering algorithms, for example, where the ordering is determined by a binary comparison tree. Informally, we claim that such restrictions on the ordering can be modeled within our framework.

Our priority algorithm framework can be viewed as a generalization of online algorithms [5]. In the context of online algorithms, the question as to what is greedy or "greedy-like" has been (at least

[5]Here we speak of online algorithms in the sense of what Sgall [37] calls "one by one" online algorithms which is the usual meaning within the area of competitive analysis. See also Footnote 2.

implicitly) considered. For example, consider the following quote in the survey by Karlin and Irani (p. 528, Chapter 13 in [27]): "In a somewhat vacuous sense, all online algorithms are greedy, since an online algorithm is defined by a function f of the current request, the current state, and the history." However, they recognize that most people would view many on-line algorithms as being at best greedy-type but not greedy. And thus they go on to say: "Nonetheless, there are some natural greedy-type algorithms that work well in certain situations." Although we would not completely agree with their assessment of what constitutes a "natural greedy-type" algorithm, their comments are very much in the spirit of the definitions we are advocating. We take a very restrictive view of the meaning of greedy but our definition of priority algorithms (applied in the online setting) seems to capture what they mean by "greedy-type". More specifically, we would argue that the work function algorithm (WFA) is *not* a greedy algorithm. In fact, for the K-server problem we would say that the only greedy algorithms are those that move a closest server to serve a request. Similarly, we do not view the $2 - \varepsilon$ competitive algorithms for the makespan (also called load balancing) problem on identical machines (nor the more competitive algorithms for non-identical machines) as being greedy. Our work is motivated to some extent by lower bounds in competitive analysis which are generally derived without any complexity considerations [6].

Our framework should be compared with Charikar *et al.* [14]; they consider online "incremental clustering" algorithms that must make irrevocable decisions (whether to insert a new input into an existing cluster or to merge two existing clusters) on a sequence of inputs [7], but they do not allow the algorithm to determine the ordering of the inputs. Note that our notion of "greedy" is more constrained than that of [14]. The online lower bounds of Charikar *et al.* for minimizing the maximum diameter can be contrasted with the approximation ratio of Gonzalez's [22] algorithm which can be viewed as an example of an (offline) priority algorithm in the context of hierarchical clustering. The Gonzalez [22] clustering algorithm is greedy when the objective function is to minimize the maximum radius.

Have we captured the full power of greedy and greedy-like algorithms by these priority algorithms? Our priority algorithm definition is based on a "one input per iteration" framework. An alternative greedy or greedy-like framework is possible for various "hierarchical problems". In this alternative framework, an algorithm initially constructs a trivial partial solution and then on each iteration an irrevocable refinement is made. For example, in incremental clustering (into say k clusters) we can start with each point as its own cluster and then in each iteration two (or more) clusters are merged and the algorithm continues until there are only k clusters. Similarly, in Huffman coding we start with each symbol as a single node prefix tree and in each iteration we merge two prefix trees and continue until there is only one prefix tree. In order to put this alternative framework into our priority framework, we can extend the definition of priority algorithms so that the decision function can look at (say) the next two highest priority inputs and make an irrevocable decision about the highest priority input (or the two highest priority inputs). In the case of Huffman coding, it suffices to consider a fixed priority algorithm with this limited lookahead. We claim that some of our lower bounds can be modified to give (weaker) bounds in this extended model and we provide one such result in Corollary 2. Returning to our "one input at a time" priority algorithm framework, it is not difficult to consider more complex kinds of priority functions depending on additional information rather than just the individual inputs. For example, we could allow the priority and/or scheduling functions to depend on n, the number of inputs (or possibly depend on simply computed values

[6]The main exception are a few results concerning memoryless algorithms. [10]

[7]Some greedy clustering algorithms only assume that an input is specified by its location in (say) Euclidean space while other clustering algorithms assume that each input specifies the vector of distances to each of the other inputs.

such as $\sum p_j, \max p_j, \frac{\max_j p_j}{\min_j p_j}, \sum w_j$, etc.). (See Section 4.) There is, of course, no limit to the kinds of additional information that might be provided to the algorithm. It is perhaps impossible to find one tractable definition that would subsume anything greedy-like. However, we claim that most of the common algorithms we call greedy seem to satisfy our priority algorithm definition and this basic definition provides a natural starting point to begin our development.

3 Specific results

3.1 Notation and basic definitions

For an algorithm A we use $A(S)$ to denote the performance of A on an input sequence S. The optimum value on the sequence S is denoted by $\mathrm{OPT}(S)$. For a maximization problem, A is said to be a *ρ-approximation* for the problem, if $\rho \cdot A(S) \geq \mathrm{OPT}(S)$ for all input sequences S. For a minimization problem, A is said to be a *ρ-approximation* for the problem, if $A(S) \leq \rho \cdot \mathrm{OPT}(S)$. Note that in both cases $\rho \geq 1$. For a set of jobs S, we let $W(S)$ denote the sum of the profits of jobs in the set. When discussing performance ratios, we often refer to OPT as an optimal algorithm. For the arbitrary profits case, we let $\delta_i = \frac{w_i}{p_i}$, i.e., the profit per unit size, $\Delta = \frac{\max_i \delta_i}{\min_i \delta_i}$, and $\delta = \min_i \delta_i$.

3.1.1 Interval Scheduling

Even for arbitrary profits this problem can be optimally solved in polynomial time using network flows [3]. We will provide approximation lower bounds for priority algorithms, thereby showing that network flows cannot be seen as priority algorithms. In contrast, for the simpler unit profit version there is an optimal priority algorithm. (See [13] and [19].)

Unit profit	FIXED PRIORITY	ADAPTIVE PRIORITY
GREEDY	all m $\rho = 1$	
(not necessarily greedy)		

For proportional profit we cannot achieve optimal profit by priority algorithms. In this case, the best fixed priority algorithm is LPT ("longest processing time" first) which is a 3-approximation algorithm. We summarize our results for proportional profit as follows.

Proportional profit	FIXED PRIORITY	ADAPTIVE PRIORITY
GREEDY	all m $\rho = 3$	m even $\rho \leq 2$ $m = 2$ $1.56 \leq \rho$
(not necessarily greedy)	$m = 2$ $2 \leq \rho$	$m = 1$ $\rho = 3$

For arbitrary profits, we show that not even the strongest adaptive class contains an algorithm with a constant approximation ratio.

Arbitrary profits	Fixed Priority	Adaptive Priority
Greedy	all m $\rho \geq \Delta$	
(not necessarily greedy)	$m \geq \log(\Delta)$ $\rho \in O(\log \Delta)$	all m $\rho \in \omega(1)$

3.1.2 Job Scheduling

The more general job (not just intervals) scheduling profit problem is strongly NP-hard even for the case of one processor and unit profit [21]. Bar-Noy, Guha, Naor and Schieber [7] analyze an adaptive greedy interval scheduling algorithm that achieves a $\frac{(1+1/m)^m}{(1+1/m)^m-1}$-approximation for unit profit job scheduling on m identical machines. The algorithm schedules one machine at a time. Whenever a job completes, the next job to be scheduled is that job (if any) which can be completed soonest [8]. Note the adaptive nature of this algorithm since (except for the first job which is chosen so as to minimize $r_j + p_j$), the next job to be considered depends on the completion time of the previous job scheduled. We show that for one machine, no (adaptive) priority algorithm can achieve an approximation ratio better than 2, thus matching the Bar-Noy *et al.* upper bound. We also show that the simple fixed priority algorithm SPT ("shortest processing time" first) is within a factor of 3 from the optimal algorithm.

For proportional profit and arbitrary profits, all of the interval scheduling lower bounds clearly still hold. It is also the case that LPT becomes a 4-approximation algorithm for proportional profit.

Unit profit	Fixed Priority	Adaptive Priority
Greedy	all m $\rho \leq 3$	$m = 1$ $\rho \leq 2$ m general $\rho \leq \frac{(1+1/m)^m}{(1+1/m)^m-1}$ $m \to \infty$ $\rho \leq \frac{e}{e-1} \approx 1.58$
(not necessarily greedy)		$m = 1$ $2 \leq \rho$

3.1.3 Makespan

The makespan problem $P|\,|C_{\max}$ is strongly NP-hard [21]. Using the concept of a "dual approximation algorithm", Hochbaum and Shmoys [26] provide a polynomial time approximation scheme

[8] It is easy to view the Bar-Noy *et al* algorithm as an adaptive greedy priority algorithm. In each iteration, the ECT (earliest possible completion time) algorithm will give higher priority to jobs which can be scheduled on the current machine. When applied to interval scheduling on one machine, the ECT algorithm becomes the optimal EDD (earliest due date) algorithm and the Bar-Noy *et al.* algorithm for identical machines becomes EDD using a first fit rule when there is more than one machine that can accept the interval being considered. For one machine, the ECT greedy algorithm was previously analyzed to be a 2-approximation algorithm in closely related settings by Adler *et al* [1] and Spieksma [38]. More specifically, Spieksma considers the JISP problem where a job is a set of intervals in which a job can be scheduled. In its most general version, the intervals associated with each job can have different processing times and profits. Bar-Noy *et al* [7] and then Erlebach and Spieksma [18] consider a "one-pass" algorithm for the weighted job scheduling (respectively, the weighted JISP) problems and derive constant approximation bounds. This one-pass algorithm and the two-pass algorithms of Bar-Noy *et al* [6] and Berman and DasGupta [9] are not priority algorithms as intervals can be removed (i.e. the decision to schedule an interval is not irrevocable). Hence they do not contradict the non-constant approximation lower bounds we derive for arbitrary profits.

(PTAS). Using dynamic programming, Sahni [35] exhibits a fully polynomial time approximation scheme (FPTAS) for any fixed m. In this paper, we will be particularly interested in the performance of LPT (Longest Processing Time first) which belongs to the class of FIXED PRIORITY, GREEDY algorithms. In his classical paper, Graham [23] showed that the online greedy list scheduling algorithm (LS) has an approximation ratio of $2 - \frac{1}{m}$. The input sequence used to show that this bound is tight motivates the natural choice of the LPT priority rule. In [24], Graham shows that LPT has an approximation ratio of $\frac{4m-1}{3m}$. Seiden, Sgall and Woeginger [36] show that LPT is optimal for 2 machines when the jobs must be considered by non-increasing size, which seems to be the most natural choice for the class FIXED PRIORITY. For uniformly related machines, LPT is also a constant approximation algorithm[9], whereas for uniformly related machines, the greedy on-line algorithm (LS) has a $\theta(\log m)$ approximation ratio [15, 4].

Returning to identical machines, we generalize the result in [36] by showing that for 2 machines LPT is optimal with respect to all priority algorithms. We also show that LPT is optimal in the class FIXED PRIORITY, GREEDY for 4 machines.

Finally, we depart from the identical machines model and briefly consider the makespan problem in the context of the "subset model" (also called, the restricted machines model). In this model, a job can only be scheduled on some (allowable) subset of the machines. The processing time p_j for any job is the same on any of the allowable machines. This model can be viewed as a special case of the unrelated machines model with the processing time $p_j(i)$ of the j^{th} job on machine i restricted to the set $\{p_j, \infty\}$. In this model, we give some evidence that FIXED PRIORITY greedy algorithms cannot improve upon the approximation ratio obtainable by the simple online greedy algorithm.

	FIXED PRIORITY	ADAPTIVE PRIORITY
GREEDY	$m = 4$ LPT optimal	$m = 2$ LPT optimal
(not necessarily greedy)		all m $\rho \geq 7/6$

4 Adversaries and lower bounds

4.1 The role of the adversary

Clearly, a lower bound on the approximation ratio is "simply" a matter of constructing an appropriate nemesis input set for each possible algorithm (within the class of algorithms being considered). It is often useful to view this nemesis set construction as a game between an adversary and an algorithm. In the more specialized framework of online competitive analysis, against a deterministic algorithm the adversary can be viewed as constructing the next input after observing the algorithm's behavior on the previous inputs. In our setting the role of the adversary is not quite so powerful.

Let us first consider our basic framework where the priority algorithm "does not know the number

[9] More precisely, for uniformly related machines, the best known bounds on LPT give an approximation ratio between 1.52 and 1.583. (See [17] for the lower bound and [20] for the upper bound.)

n (or any other additional information such as $\sum p_j$) of input jobs". That is, neither the priority function nor the scheduling function depend on n. Consider the class Fixed Priority. We now view the interaction between the adversary and the algorithm as follows: the adversary presents a possibly large but finite set S of potential jobs. The algorithm determines a total ordering on these jobs. The adversary selects a subset $S' \subset S$ as the actual input set. Although there will be a bound n on the size of S', the actual size of S' depends on the algorithm. That is, the adversary is free to remove jobs not yet considered by (the scheduling function of) the algorithm. Furthermore, for the fixed priority class, we can allow (without loss of generality) the adversary to substitute any job in the total ordering by any number of copies of the job. (See Lemma 2 for details.) The situation is not much different for the class Adaptive Priority. Even though the algorithm can change the priority function in each iteration, we can still think of the adversary as adaptively removing jobs from S so as to derive the actual input set S'. We note that the different views of the adversary are just for conceptual convenience; since the algorithms are deterministic, the adversary can determine S' as soon as the algorithm definition is revealed. In other words, a job not yet considered has no influence on the priorities given, and therefore we allow the adversary to remove jobs not yet handled by the algorithm.

Now let us assume that the algorithm knows the exact number n of inputs in the nemesis input set. In general, we do not know how to convert a lower bound within the basic framework above to this more powerful class of algorithms (although all of our lower bounds may still hold). However, we can show that most of our profit maximization lower bounds can be extended to this class of algorithms. For example, in Corollary 1 we show how to extend the lower bound in Theorem 3 for interval scheduling on one machine with proportional profit. (Note that it is not sufficient to simply use very small "dummy" jobs as the algorithm can give such jobs highest priority and then use knowledge of the number of non-dummy jobs to possibly defeat the adversary.) In addition, most of these profit maximization lower bounds can be extended to the analogous "ρ-approximation decision problem".

4.2 A useful combinatorial lemma

Below we prove a property which will be useful in deriving lower bounds with respect to fixed priority algorithms. More specifically, we will use this lemma for the makespan problem and in generalizing lower bounds for interval scheduling from one machine to multiple identical machines. We will say that according to this lemma the adversary is allowed to *copy* jobs.

Lemma 2 Let $S = \langle s_1, s_2, \ldots, s_{nk} \rangle$ be a sequence containing at least k copies of n different job types. For a given integer m, and $k = m \cdot n$, a subsequence S' can be extracted with the following properties

- All jobs of the same type are adjacent in S'
- The sequence S' contains exactly m copies of each of the n job types

Proof The subsequence S' is constructed in n phases. In phase 1, we let P_1 denote the minimal prefix of S such that one job type, which we call J_1, is repeated exactly m times. Let $S_1 \subseteq S \setminus P_1$

denote the subsequence of the remaining sequence where all occurrences of type J_1 are removed. In phase i job J_i is found by constructing the sets P_i and S_i from S_{i-1} in a similar way. Let T_i denote the m jobs of type J_i from P_i. Then, S' is the concatenation of T_i for all i. The same job type cannot be picked in two different phases, since it is deleted from the remaining sequence when it is first chosen. To ensure that every job type will be found in some phase, we claim that it is sufficient to have mn copies. To see this, consider an arbitrary job type J; in the worst case, this job type is picked in the last phase. At most $m-1$ copies are removed of job type J in each of the $n-1$ preceding phases and m copies must be left for the last phase. Therefore $k = (m-1)(n-1) + m \leq mn$ copies of job type J in the original sequence suffices to find every job in exactly one phase. □

When using this lemma, instead of removing jobs, we actually substitute with jobs of sufficiently small profit (most often small jobs). In this way the indices of jobs staying in the sequence are the same. This will only be done in cases where the addition of these sufficiently small jobs does not change the objective function considerably.

5 Interval Scheduling

As we shall see in this section, the results for Interval Scheduling are quite different for the various profit models. In particular, it is well known that there is an optimal algorithm for the unit profit model in the weakest class of priority algorithms. However, for arbitrary profits, the picture changes dramatically. In this profit model, there does not exist a priority algorithm with a constant approximation ratio. The proportional profit model provides a nice intermediary case where we can obtain non-trivial constant approximation factors.

5.1 Unit profit

When each interval has profit 1, the objective function is the number of intervals accepted. The algorithm considered below sorts the intervals according to increasing deadline. Looking at the machine on which a job is scheduled, all other intervals already in the schedule occur earlier. This creates a *gap*, which is the unused space just before the start time of the interval just scheduled. When assigning an interval to a machine, the Best-Fit scheduling rule chooses the machine on which the least gap is created. (Ties are broken arbitrarily.) The following theorem was independently shown in [13] and [19].

Theorem 1 The FIXED PRIORITY algorithm considering the intervals by increasing deadline and using a Best-Fit scheduling rule is optimal for interval scheduling with unit profit.

5.2 Proportional profit

We now consider proportional profit, where the profit of an interval is equal to the processing time and the objective is to maximize the sum of profits of intervals accepted. Let S be an input sequence. For notational simplicity, we let S be the indices of the jobs (= intervals). However, for readability we still refer to the i^{th} job as J_i. We call a partial function $\sigma : S \to \{1, 2, \ldots, m\} \cup \{nil\}$ a *partial schedule*, if for all distinct $i, j \in S$ such that $\sigma(i)$ and $\sigma(j)$ are defined and $\sigma(i) = \sigma(j) \neq nil$, there is no overlap between the two intervals J_i and J_j. Jobs mapping to a number are called accepted jobs, while jobs mapping to the special value nil are called rejected jobs. The set of jobs accepted by σ is denoted by $A(\sigma)$, and the profit $W(\sigma)$ of a partial schedule σ is the sum of the profits of the accepted intervals, $W(A(\sigma))$. A (total) schedule σ is a partial schedule that is defined on all of the input S. We say that a schedule σ' extends a partial schedule σ, if σ' agrees with σ wherever σ is defined.

Below we analyze the LPT ("longest processing time") algorithm [10]. It uses $\pi_i = w_i = p_i$, i.e., sorts the jobs by non-increasing size. When a job can be scheduled this is done on an arbitrary machine. Having sorted the jobs, we can assume that $p_1 \geq p_2 \ldots \geq p_n$.

Theorem 2 The fixed priority, greedy algorithm LPT which schedules intervals by non-increasing size is a 3-approximation (for all m) for interval scheduling with proportional profit. (That is, for the problem $P|p_j = d_j - r_j| \sum p_j \bar{U}_j$.)

Proof Let S be an input sequence. We prove the following claim by induction on the number of jobs scheduled:

CLAIM: After considering k jobs and obtaining a partial schedule σ_k, there is a (total) schedule σ' extending σ_k, such that if $X' = A(\sigma') \setminus A(\sigma_k)$ denotes the set of jobs accepted by σ' but not by σ_k, then $3W(\sigma_k) + W(X') \geq OPT(S)$. (Note that since σ' extends σ_k, the jobs in X' have not yet been considered by LPT.)

To see that an approximation factor of 3 is obtained from this claim, note that when the last job has been considered, there is no proper extension (i.e. $X' = \emptyset$) and the current profit is therefore within a factor of 3 of the optimal profit on the entire sequence.

Before any job has been considered, the claim holds trivially. Assume that after considering k jobs obtaining the partial schedule σ_k there is an extension σ' of σ_k such that $3W(\sigma_k) + W(X') \geq OPT(S)$ for $X' = A(\sigma') \setminus A(\sigma_k)$. Let $k+1$ be the next job (index) to be considered by LPT. If J_{k+1} is rejected, then the claim is immediate. So assume J_{k+1} is scheduled. Let σ_{k+1} denote *LPT*'s schedule after J_{k+1} is scheduled, and let m denote the machine where J_{k+1} was scheduled by LPT. If $k+1$ is contained in $A(\sigma')$, we let $m' = \sigma'(k+1)$ denote the machine on which it was scheduled in σ'. There are three cases to consider. In all cases, we specify an extension σ'' of σ_{k+1} and prove that for $X'' = A(\sigma'') \setminus A(\sigma_{k+1})$ we have $W(X'') \geq W(X') - 3w_{k+1}$. This is sufficient since $3(W(\sigma_k) + w_{k+1}) + W(X'') \geq \mathrm{OPT}(S)$ follows.

[10] Here we abuse the terminology and use LPT to denote the priority rule as well as any greedy algorithm that uses this rule. The same rule (and same abuse of terminology) is used in other contexts, such as the makespan problem which will be discussed in Section 7.1.

- Case 1: $k+1$ is contained in X' and $m = m'$.
- Case 2: $k+1$ is not contained in X'.
- Case 3: $k+1$ is contained in X' and $m \neq m'$.

If Case 1 occurs, we let $\sigma'' = \sigma'$. In this case, $W(X'') = W(X') - w_{k+1} \geq W(X') - 3w_{k+1}$.

For the next two cases, let O be the set of jobs scheduled by σ' on machine m that overlap J_{k+1}. Let $C \subseteq O$ denote those intervals that are completely covered by J_{k+1}. (An interval J_j is completely covered by J_{k+1}, if $[r_j, d_j) \subseteq [r_{k+1}, d_{k+1})$.) We have $W(C) \leq w_{k+1}$. Note also that $O \setminus C$ will contain at most two intervals, and each of these have a profit of at most w_{k+1} since the intervals are considered by decreasing sizes. Since σ'' must be an extension of σ_{k+1}, we specify σ'' such that $\sigma''(k+1) = m$.

If Case 2 occurs, we define σ'' identical to σ', except that $\sigma''(k+1) = m$ and $\sigma''(j) = nil$ for all $j \in O$. Since $W(O) \leq 3w_{k+1}$ by the above remarks, we have $W(X'') \geq W(X') - 3w_{k+1}$.

If Case 3 occurs, we define σ'' identical to σ', except that $\sigma''(k+1) = m$ and $\sigma''(j) = m'$ for all $j \in C$ and $\sigma''(j) = nil$ for all $j \in O \setminus C$. Since $X'' = X' \setminus (\{k+1\} \cup (O \setminus C))$, we have $W(X'') \geq W(X') - 3w_{k+1}$. □

For the purpose of studying interval scheduling with arbitrary profits (see Section 5.3), we now want to consider how well the LPT algorithm performs on m machines when compared against the optimal way of scheduling the same intervals on $m' \geq m$ machines.

Lemma 3 LPT is a ρ-approximation algorithm using m machines against OPT using m' machines, for $\rho = \frac{3m'}{m} + \max\{0, 1 - \frac{3}{m}\}$.

Proof For convenience, assume that the m machines used by LPT are disjoint from the m' machines used by OPT. Let Y be the set of jobs scheduled by both LPT and OPT; let the set of jobs scheduled by LPT be $X \cup Y$ (where $X \cap Y = \emptyset$), and let the set of jobs scheduled by OPT be $Z \cup Y$ (where $Z \cap Y = \emptyset$). We will get an upper bound on $W(Z)$ by dividing up the profit of each job in Z into parts and assigning each part to some job in $X \cup Y$.

Consider a job J in Z with profit w. Each machine in LPT's schedule will contain a larger job overlapping J, since otherwise J would have been scheduled by LPT. Since LPT has m machines, we assign a fraction $\frac{1}{m}$ of w to one such job on each of the m machines. Consider the profit assigned to a job J' in X of profit w'; J' will have at most $\frac{3w'}{m}$ profit assigned from each machine in OPT, since the jobs on this machine that overlap J' and whose profit is no bigger than w' will have total profit at most $3w'$. Therefore each job in X has been assigned at most $3\frac{m'}{m}$ times its own profit, and so the total profit assigned to X is at most $\frac{3W(X)m'}{m}$. Now consider the profit assigned to a job J' in Y of profit w'; J' will have no profit assigned to it from the machine of OPT on which J' is also scheduled, and J' will have at most $\frac{3w'}{m}$ profit assigned from each of the other $m'-1$ machines in OPT. Therefore each job in Y has been assigned at most $3\frac{m'-1}{m}$ times its own profit, and so the

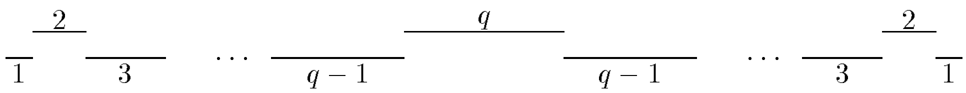

Figure 1: The "long jobs" from the worst case sequence for any priority algorithm for $m = 1$.

total profit assigned to Y is at most $\frac{3W(Y)(m'-1)}{m}$. So the total profit assigned to $X \cup Y$, which is $W(Z)$, is at most $\frac{3W(X)m'}{m} + \frac{3W(Y)(m'-1)}{m}$.

Note that the profit of OPT is $W(Y)+W(Z)$, and the profit of LPT is $W(X)+W(Y)$. From above, we have $W(Y)+W(Z) \leq W(Y)+\frac{3W(X)m'}{m}+\frac{3W(Y)(m'-1)}{m} = (W(X)+W(Y))\frac{3m'}{m}+W(Y)(1-\frac{3}{m}) \leq (W(X)+W(Y))(\frac{3m'}{m}+\max\{0, 1-\frac{3}{m}\})$. □

The next theorem and the following corollary show that LPT is best possible among FIXED PRIORITY, GREEDY algorithms for interval scheduling with proportional profit.

Theorem 3 For $m = 1$, no ADAPTIVE PRIORITY algorithm has an approximation ratio better than 3 for interval scheduling with proportional profit.

Proof Let ε and q be given. The adversary sequence consists of "long jobs" and "short jobs". The long jobs are depicted in Figure 1 and consist of $2q-1$ jobs, with two jobs for each of the sizes $1, 2, \ldots, q-1$, and one job of size q. Every long job overlaps two other long jobs by ε except for the two jobs at the end which only overlap one long job each (also by ε). Additionally, for each job J_i of size p_i the adversary gives 3 non intersecting short jobs of size $\frac{p_i-2\varepsilon}{3}$ all included within the interval of job J_i and not intersecting the adjacent long jobs. The short jobs relating to job J_i will clearly all fit together on one machine.

The algorithm assigns priorities to the jobs, and we claim that the job with the highest priority, say J_1, must be scheduled even though the algorithm is not greedy. If the first job is not scheduled the algorithm will not be competitive on the sequence consisting of J_1 alone. The adversary changes the sequence such that all jobs not intersecting J_1 are removed. The optimal algorithm will reject J_1 and schedule all remaining jobs. Depending on the size of job J_1 there are four cases to consider.

- Case 1: J_1 is a short job.
- Case 2: J_1 has size = profit 1.
- Case 3: J_1 has size j, for $1 < j < q$
- Case 4: J_1 has size q.

If Case 1 occurs and the algorithm accepts a small job of profit = size $\frac{p_i-2\varepsilon}{3}$, OPT will get a job of profit p_i, and the ratio is at least $\frac{p_i}{(p_i-2\varepsilon)/3}$.

If Case 2 occurs, OPT will get one long job of profit 2 and three small jobs with a total profit $1-\varepsilon$. The ratio is $\frac{2+(1-\varepsilon)}{1}$.

If Case 3 occurs, OPT will get two long jobs of profits $j-1$ and $j+1$ and three small jobs with a total profit $j-2\varepsilon$. The ratio is $\frac{(j-1)+(j+1)+(j-2\varepsilon)}{j}$.

If Case 4 occurs, OPT will get two jobs of profit $q-1$ and small jobs with a total profit $q-2\varepsilon$. The ratio is $\frac{2(q-1)+(q-2\varepsilon)}{q}$.

In all cases the ratio is arbitrarily close to 3 by making ε sufficiently small and q sufficiently large. $\square$

Corollary 1 The lower bound of Theorem 3 holds even if the algorithm knows n in advance. That is, the ordering and decision function of the algorithm can also depend on n.

Proof The simplest idea would be to augment the set S used in the basic argument to include sufficiently many copies of some small interval. This would essentially suffice for greedy algorithms since we need only consider what the algorithm does on S, safely ignoring the rather worthless small jobs. But for non-greedy algorithms, the algorithm can use the presence of these small jobs (giving them the highest priority) to infer the true size of the input set and adjust its behavior accordingly. We take the following approach. In the basic framework for Theorem 3, we have a 3-approximation lower bound using a set S of potential inputs and the adversary uses at most 6 jobs (intervals) in deriving this lower bound . In the lower bound for the new model, the adversary will always present exactly 7 inputs. The adversary makes 3 "independent" clones of the set S, where "independent" means that the intervals covered by these clones are located at disjoint parts of the real (time) line. We will then make 5 additional copies of each interval in each clone. We then observe the first 2 intervals considered by the algorithm. As long as the algorithm rejects an interval in some clone (i.e. does not schedule it), the adversary (i.e. OPT) accepts the interval and then eliminates the remaining jobs in the clone. If the algorithm rejected the first 2 intervals it considered, then the adversary selects 5 copies of a smallest interval in the last clone and accepts one copy (knowing the algorithm will accept at most one copy). Because the algorithm rejected 2 previous intervals, the desired ratio of 3 is obtained. If the algorithm accepts one of the first two intervals it considers in one of the clones, then the adversary resorts to the basic argument by putting up to 5 additional jobs in the input sequence, and filling it up (if necessary) with copies of the job accepted by the algorithm. $\square$

Motivated by the example of Huffman coding, we give some further evidence of the "robustness of the priority framework". Namely we consider an extension of the model and present a lower bound for this extended model.

Corollary 2 Consider an "extended adaptive greedy priority algorithm" which is allowed to look at the next two highest priority inputs and then make an irrevocable decision concerning the input of highest priority. We first consider adaptive greedy algorithms in this extended model which (if possible) must schedule the input of highest priority. Then the lower bound of Theorem 3 can be modified to show a 2-approximation lower bound. Essentially the same argument yields a

2-approximation lower bound for any extended fixed priority (not necessarily greedy) algorithm; that is, an algorithm which considers jobs in some fixed order and has the ability to see the $i+1^{st}$ highest priority job before deciding on whether or not to schedule the i^{th} highest priority job.

Proof For the input set, we need 2 long jobs (say of size q) which have an ε intersection and two sets of ε size non-intersecting small jobs. There are $(q-\varepsilon)/\varepsilon$ small jobs included within the interval of each of the long jobs. For a greedy adaptive algorithm, there are two cases to consider and in each case it is easily seen that the adversary can force the desired bound.

- Case 1: The algorithm first schedules a short job in which case the adversary removes all "unseen" (i.e. not amongst the two highest priority jobs) short jobs within the same interval as the scheduled job.
- Case 2: The algorithm first schedules a long job in which case the adversary removes all unseen short jobs within the interval spanned by the other long job.

The proof is similar for an extended fixed priority algorithm using the same input set. If the highest priority job is scheduled then the proof is as in the adaptive greedy case. If the highest priority job is a long job which is not scheduled, then the adversary removes all the short jobs within that interval (with the possible exception of the second highest priority job). If the highest priority job is a short job which is not scheduled and the second highest priority job is another short job, then the adversary removes all other jobs. Finally if the highest priority job is a short job which is not scheduled and the second highest priority job (say J) is a long job, we can ignore the highest priority job and then argue as before depending on whether or not J is scheduled. $\square$

Corollary 3 For any m, no FIXED PRIORITY, GREEDY algorithm has an approximation ratio which is better than 3 for interval scheduling with proportional profit.

Proof Let N denote the number of jobs in the sequence above. The adversary gives mN copies of each job. Since we consider the class FIXED PRIORITY, the priorities are calculated once before any job is considered. According to Lemma 2, the adversary is able to extract a sequence with m adjacent copies of jobs having the same specifications. Since we consider a greedy algorithm, all m copies of the job with the highest priority must be accepted giving identical configurations on all machines. $\square$

If we remove the greedy restriction, but still consider FIXED PRIORITY algorithms, an easy extension of the sequence above gives a lower bound of 2 on the approximation ratio for interval scheduling with proportional profit.

Theorem 4 For $m = 2$, no algorithm in the class FIXED PRIORITY (not necessarily greedy) is a ρ-approximation for $\rho < 2$ for interval scheduling with proportional profit.

Proof We use the same sequence as in the proof of Theorem 3, but with two copies of each job. Since we consider the class Fixed Priority, (using Lemma 2 again) we can assume that identical jobs are adjacent in the total ordering. If the two jobs with the highest priority are both scheduled, the adversary can remove all jobs not overlapping these jobs from the input sequence. The performance ratio will be 3 by the proof of Theorem 3. If the second job is rejected by the algorithm, the adversary can end the sequence and the ratio is 2, since OPT can schedule both of these jobs. □

We now show that adaption can help. The intuition behind the algorithm can be seen in Figure 2.

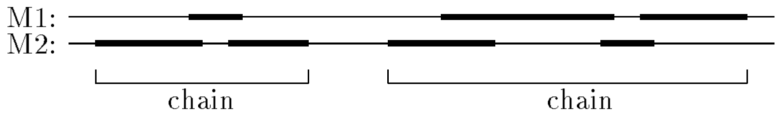

Figure 2: Chain-2 for $m = 2$. Every time unit covered by jobs from the sequence will be covered by the algorithm by extending the current schedule as much to the right as possible in each step.

At any point in time, the algorithm tries to extend the current schedule building a chain going back and forth between the machines. For $m = 2$, we describe a 2-approximation algorithm, Chain-2 from the class Adaptive Priority, Greedy. This algorithm will become a central part of a 2-approximation for an even number of machines. In hindsight, our Chain-2 algorithm turns out to be the exact analogue of a 2-approximation algorithm constructed by Baruah *et al.* [8] for the same proportional profit interval scheduling problem in the setting of "real-time online" preemptive scheduling [11]. Our analysis used to prove Theorem 5 follows the analysis used by Baruah *et al.*.

Let $m = 2$ and let $S = \langle J_1, J_2, \dots, J_n \rangle$ be an input sequence. Let $d_j = r_j + p_j$ be the finishing time of job j. The algorithm will alternate between machines when scheduling jobs. Initially, the algorithm picks a job (=interval) with the earliest release time (ties can be broken arbitrarily but for definiteness say we break ties in favor of the longest processing time). We will say that a job J_k *properly extends* a job J_j if $r_j \leq r_k \leq d_j$ and $d_j < d_k$). The algorithm proceeds in *phases* where during any phase we consider (i.e. give highest priority to) jobs which properly extend the most recently scheduled job, call it $J_{n(i)}$. If there is a job properly extending $J_{n(i)}$, we choose (i.e. give the highest priority to) one having the latest completion time. This job is then denoted $J_{n(i+1)}$. If there is no job properly extending $J_{n(i)}$, then we start a new phase by choosing (as done initially) a job with the earliest release time larger than $d_{n(i)}$. If we cannot start a new phase (i.e. there are no jobs with release time larger than $d_{n(i)}$), then the algorithm terminates. (Formally, the analysis can assume no other jobs are scheduled but in order to view this as a greedy adaptive algorithm, the algorithm can greedily schedule the remaining jobs in any order.)

We first show that the algorithm produces a feasible schedule.

Lemma 4 If $J_{n(i+1)}$ properly extends $J_{n(i)}$, then it can be scheduled having already scheduled $J_{n(1)}, \dots, J_{n(i)}$.

[11] In contrast to the concept of online algorithms as used in competitive analysis, the more classical scheduling literature uses online algorithms to denote those algorithms that make decisions at any time t based on knowledge of all jobs that have been released by time t. To avoid ambiguity, we refer to such algorithms as being real-time online.

Proof By a simple induction on i, it is easy to see that $d_{n(1)} < d_{n(2)} < \ldots < d_{n(i+1)}$. If $J_{n(i)}$ started a new phase then the lemma clearly holds. So assume $J_{n(i+1)}$ properly extends $J_{n(i)}$. We need only show that $r_{n(i+1)} > d_{n(i-1)}$. But if $r_{n(i+1)} \leq d_{n(i-1)}$, and $J_{n(i+1)}$ properly extends $J_{n(i)}$, then $J_{n(i+1)}$ properly extends $J_{n(i-1)}$ and since $d_{n(i-1)} < d_{n(i)} < d_{n(i+1)}$, $J_{n(i)}$ would not have been the furthest extension of $J_{n(i-1)}$ contradicting the definition of the algorithm. □

For convenience we consider all release times and deadlines to be integers.[12] Before we prove that CHAIN-2 is a 2-approximation for $m = 2$, we prove a lemma saying that the algorithm will schedule jobs in all time units covered by the input sequence.

We use the term *density* for the number of jobs covering a time unit, and let $d(S,t) = |\{J_i \mid [t, t+1) \subseteq [r_i, d_i)\}|$. Similarly, we use $C_2(S,t)$ to denote the number of jobs CHAIN-2 has scheduled covering time $[t, t+1)$. (We will refer to this time unit as *slot t*.)

Lemma 5 For $m = 2$, $\forall t$: if $d(S,t) \geq 1$, then $C_2(S,t) \geq 1$.

Proof Suppose some job J_k covers a time slot $[t, t+1)$. If CHAIN-2 schedules this job then this slot is clearly covered. If J_k is not scheduled by CHAIN-2, we prove by cases that CHAIN-2 covers this slot.

1. For some $i \geq 1$, $r_{n(i)} \leq r_k \leq d_k \leq d_{n(i)}$. In this case, $J_{n(i)}$ covers slot t.
2. For some $i \geq 1$, J_k properly extends $J_{n(i)}$. If $t \leq d_{n(i)}$, slot t is clearly covered by CHAIN-2. Otherwise, by the way CHAIN-2 chooses $J_{n(i+1)}$ in a phase, $d_k \leq d_{n(i+1)}$ and again slot t is covered.
3. For some $i \geq 0$, $d_{n(i)} < r_k < r_{n(i+1)}$. But since $d_{n(i)} < r_{n(i+1)}$, a new phase has started and $r_k < r_{n(i+1)}$ contradicts the way a job is chosen to start a new phase. (For $i = 0$, we assume $d_{n(i)} = -\infty$.)
4. CHAIN-2 has scheduled a total of (say) q jobs and $d_{n(q)} < r_k$. But then J_k (or some other job) would have been scheduled so that there would have been more than q jobs scheduled.

□

Theorem 5 The algorithm CHAIN-2 is a 2-approximation for $m = 2$ for interval scheduling with proportional profit.

Proof Similar to the above definitions, we let $\mathrm{OPT}(S,t)$ denote the number of jobs scheduled by OPT covering the time unit t. It is easy to see that $d(S,t) \geq \mathrm{OPT}(S,t)$. From Lemma 5 we get $2C_2(S,t) \geq \mathrm{OPT}(S,t)$, for all t. Since $\mathrm{OPT}(S) = \sum_t \mathrm{OPT}(S,t)$, we get $2C_2(S) \geq \mathrm{OPT}(S)$. □

[12] To be precise, we could instead look at points in time. All sums in the proof would instead be integrals.

We now define CHAIN which is a 2-approximation for an even number of machines. CHAIN also belongs to the class of greedy adaptive algorithms. CHAIN uses CHAIN-2 as a subroutine on pairs of machines. First, CHAIN-2 is run on the input sequence using machines 1 and 2. In this process some jobs are scheduled and removed from the sequence. Let S_1 denote the remaining sequence. Then, CHAIN-2 is run on S_1 using machines 3 and 4, returning the sequence S_2, and so on. In total CHAIN-2 is run $\frac{m}{2}$ times, each time using a new pair of machines. Clearly, CHAIN is an adaptive priority algorithm and its greediness is insured by Lemma 5.

Theorem 6 The algorithm CHAIN is a 2-approximation for m even for interval scheduling with proportional profit.

Proof Let t be arbitrary. We use $C(S,t)$ for the number of jobs scheduled by CHAIN covering $[t, t+1)$. In each run CHAIN-2 uses two machines and the decrease in the density of remaining jobs in time unit $[t, t+1)$ is at most two. Write the density as $d(S,t) = 2j$ or $d(S,t) = 2j - 1$ for some integer j. Since CHAIN-2 will schedule at least one job in $[t, t+1)$ if $d(S,t) \geq 1$ by Lemma 5, we get $C(S,t) \geq j$. The theorem is proved by observing that $d(S,t) \geq \mathrm{OPT}(S,t)$ for all t. □

One reason that greedy algorithms are desirable is due to their efficiency. An algorithm using a fixed priority scheme will most likely base this priority on (some kind of) sorting and each step tends to be very efficient (i.e. $O(1)$ or $O(\log n)$). Typically then, the total complexity bound is $O(n \log n)$. This is not necessarily true for adaptive algorithms, since the priorities must be recomputed at every state. However, the above algorithm can be implemented in time $O(n \log n)$ and space $O(n)$ using an augmented balanced binary search tree in a standard way [13].

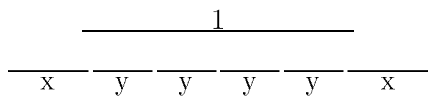

Figure 3: Lower bound for an adaptive, greedy algorithm for $m = 2$.

Theorem 7 For $m = 2$, no ADAPTIVE PRIORITY GREEDY algorithm is a ρ-approximation algorithm for interval scheduling with proportional profit, if $\rho < 1 + \frac{\sqrt{17}-3}{2} \approx 1.56$.

Proof Let $\varepsilon > 0$ be given. We use three job sizes: $1, x = \frac{\sqrt{17}-3}{4} \approx 0.28$, and $y = \frac{1-2\varepsilon}{4}$. Note that $y < x$. Two copies of each of the jobs shown in Figure 3 are presented to the algorithm. The job of size 1 overlaps each of the jobs of size x by ε. Consider the decision tree with two decisions in Figure 4. If a job of size x or y is picked as the first or second job chosen by the algorithm, we end up in Case 2, otherwise we end up in Case 1. If Case 1 occurs, OPT will reject the jobs of size 1 and the ratio is $\frac{2((1-2\varepsilon)+2x)}{2} = \frac{1+2x-2\varepsilon}{1}$. If Case 2 occurs, we instead consider the sequence consisting of the job(s) picked by the algorithm and the jobs of size 1. In this case, OPT accepts two jobs of length 1, while the algorithm accepts at most $1 + x$, since $y < x$. The ratio is at least

[13]Without loss of generality, we assume $m \leq n$.

$\frac{2}{1+x}$. Equating these two expressions, we find that the maximum value is obtained for $x = \frac{\sqrt{17}-3}{4}$. Inserting this value of x into the ratios above, we get the claimed lower bound, since ε can be arbitrarily small. $\square$

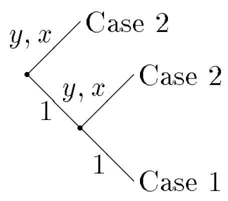

Figure 4: Decision tree with two decisions for an adaptive, greedy algorithm for $m = 2$.

5.3 Arbitrary profits

Recall, that for each job J_i we define $\delta_i = \frac{w_i}{p_i}$, i.e., the profit per unit size, $\Delta = \frac{\max_i \delta_i}{\min_i \delta_i}$, and $\delta = \min_i \delta_i$. Note that the case of proportional profits is precisely the special case of $\Delta = 1$. It follows that any ρ-approximation algorithm for proportional profits is also a $\rho \cdot \Delta$-approximation algorithm for arbitrary profits.

Theorem 8 For $m = 1$, no ADAPTIVE PRIORITY algorithm is a ρ-approximation for $\rho < \Delta$ for interval scheduling with arbitrary profits. This lower bound holds for the case that the algorithm knows Δ.

Proof We first assume that the algorithm acts greedily and schedules the job to which it gives highest priority. For any constant c, the adversary gives one job of size $p_1 = 1$ and profit $w_1 = 1$ and c^2 non-intersecting jobs of size $p_i = \frac{1}{c^2}$ and profit $w_i = \frac{1}{c}$ all within the unit interval of the first job. All the small jobs can be scheduled on one machine. Note that $\Delta = c$ for this input set. If the large job has the highest priority, OPT will reject that job and schedule all other jobs. The ratio is $\frac{c^2 \frac{1}{c}}{1} = c$. If a small job, J_i, has highest priority, we instead consider the sequence consisting of $\langle J_1, J_i \rangle$. In this case OPT will reject the small job and accept J_1. The ratio is $\frac{1}{\frac{1}{c}} = c$.

Now we consider the case that the algorithm does not act greedily and can reject jobs which do not conflict. If the job having highest priority is the large job and it is rejected, then the adversary schedules this job and removes all but one small job so that $\Delta = c$ is preserved. The situation when a small job has highest priority and is rejected requires a little more care. To handle this case, we need to redefine the initial set of inputs so that it now contains c^2 non overlapping subsets of jobs (i.e. the jobs in the j^{th} subset are all contained in the unit interval [j, j+1]), each subset defined as before to have one large job and c^2 small jobs. As long as the algorithm continues to select a small job from some subset to have highest priority and then reject it, the adversary schedules this small job and deletes all other jobs in that subset. This continues until one of the following happens:

1. The highest priority job (in some subset) is scheduled in which case the adversary proceeds as in the greedy case and also removes all the jobs from the remaining subsets.

2. A large job in some subset is selected by the algorithm to have highest priority and rejected in which case, the adversary acts as above by scheduling the large job and one small job and removing all other remaining jobs.

3. The algorithm has rejected c^2 small jobs, one from each subset, to which it gave highest priority. In this case, the adversary removes all other small jobs from this last subset and schedules the large job from this subset.

In all cases, the ratio $\Delta = c$ is preserved and the ratio is at least c.

□

Corollary 4 For all m, no algorithm from Fixed Priority, Greedy is a ρ-approximation for $\rho < \Delta$ for interval scheduling with arbitrary profits.

Proof The proof is similar to the proof of Corollary 3. □

In the following, we show that non-greedy choices can help. We use a "Classify and randomly select" (CRS) algorithm as our decision algorithm. If m is "large", the algorithm can be derandomized. The performance of a randomized algorithm A is measured according to expectation over the random choices made by the algorithm, and in this context we say that the algorithm is a *ρ-approximation*, if $\rho \cdot E_A[S] \geq \mathrm{OPT}(S)$, where $E_A[S]$ denotes the expected profit obtained by A on input sequence S. Especially in the online world this is the common way of measuring the performance of a randomized algorithm. Unfortunately, there is no guarantee that the expected value is obtained with high probability, since the variation from the mean can be very large. (See [31] for a discussion of this subject.)

Theorem 9 If Δ and δ are known in advance, there is a randomized $O(\log \Delta)$ approximation algorithm, CRS, in the class Fixed Priority (not necessarily greedy) for interval scheduling with arbitrary profits. If $m = \Omega(\log \Delta)$, then the algorithm is deterministic and δ need not be known in advance.

Proof We first consider the deterministic algorithm for "large m". We assume $\Delta = 2^k$ for some integer k, and $m \geq k$. (It is easy to see how to modify the argument for $\Delta = b^k$ for any base b.) Jobs are divided into k classes depending on δ_i. Class C_1 contains jobs with $\delta_i \in [\delta, 2\delta]$. Class C_i for $1 < i \leq k$ contains jobs with $\delta_i \in (\delta 2^{i-1}, \delta 2^i]$. We divide the machines into k groups $M_1, M_2, \ldots, M_k$ of size at least $s = \lfloor \frac{m}{k} \rfloor$. Let $s' = m - sk$, then the first s' groups contain $s+1$ machines and the other groups contain s machines. Jobs are ordered according to the LPT rule. When a job arrives, the job is classified into a class C_i and scheduled on an arbitrary machine in the group M_i, if possible. Otherwise the job is rejected.

It suffices to show that $(13 \log \Delta) CRS(C_i) \geq \text{OPT}(C_i)$, since then $\text{OPT}(S) = \sum_i \text{OPT}(C_i) \leq (13 \log \Delta) \sum_i CRS(C_i) = (13 \log \Delta) CRS(S)$.

Let C_i be one of the $\log \Delta$ classes. Let V_i denote the total length of jobs scheduled from this class by OPT, and let W_i denote the total length scheduled by CRS from C_i. By Lemma 3 we know that $(3 * 2k + 1) W_i \geq V_i$, since OPT has m machines and CRS uses at least s machines for this class. Note that $\frac{m}{s} \leq 2k$. Since we consider class C_i, we know that $CRS(C_i) \geq \delta 2^{i-1} W_i$ and $\text{OPT}(C_i) \leq \delta 2^i V_i$. The performance ratio is $\frac{\text{OPT}(C_i)}{CRS(C_i)} \leq \frac{2V_i}{W_i}$ from which the bound follows by the above relation between V_i and W_i.

For all m, we can use simple randomization to obtain the same asymptotic bound. This also requires δ to be known by the algorithm. Jobs are classified as before, and for every class C_i, we have a deterministic algorithm A_i scheduling jobs from class C_i only, using all the machines. The randomized algorithm is a mixed strategy invoking one of the $\log \Delta$ deterministic algorithms with equal probability. Given an input sequence S, let $\text{OPT}(C_i)$ denote the performance of the optimal algorithm on class C_i. The expected performance of the algorithm is $E_A[S] = \sum_i p_i A_i(S)$, where p_i denotes the probability of invoking algorithm A_i. Here we have $p_i = \frac{1}{\log \Delta}$ for all i. Since A_i only schedules jobs from class C_i, we can write $E_A[S] = \frac{1}{\log \Delta} \sum_i A_i(C_i)$. We now show that $\text{OPT}(C_i) \leq 6 A_i(C_i)$, which is sufficient, since $\text{OPT}(S) = \sum_i \text{OPT}(C_i)$. Let C_i denote an arbitrary class, let V_i denote the total length of jobs scheduled from this class by OPT, and let W_i denote the length scheduled by A_i. We know from Theorem 2 that $V_i \leq 3W_i$. Since we consider class C_i, we have $\frac{\text{OPT}(C_i)}{A_i(C_i)} \leq \frac{\delta 2^i V_i}{\delta 2^{i-1} W_i} \leq \frac{2V_i}{W_i} \leq 6$. □

We now describe how to get eliminate the assumption that δ should be known in advance for the deterministic version. The algorithm can use an estimate δ' of δ. When the first job J_1 is considered, we let $\delta' = \delta_1$. Every time a job J_i arrives with $\delta_i < \delta'$, we continue to halve δ' until $\delta' \leq \delta_i$. When the entire sequence is considered, $\frac{\delta}{2} \leq \delta' \leq \delta$. In the worst case this will require one more class than when δ is known in advance.

Theorem 10 For all m, no Adaptive Priority algorithm has a constant approximation ratio for interval scheduling with arbitrary profits.

Proof As a natural generalization of the two-level sequence from Theorem 8, we here use a sequence consisting of $m+1$ levels. On each level a total processing time of $c(m+1)$ is given all using the same time slot, say $[0, c(m+1)]$. Jobs on level j have size P_j and profit W_j and they can all be scheduled on the same machine. When going from level j to level $j+1$ the profit per unit size increases. Each level is divided into a number of groups G_j. A job J on level j defines a group on level $j+1$, consisting of the jobs overlapping J. More specifically, level 1 consists of $c(m+1)$ jobs of size $P_1 = 1$ and profit $W_1 = \frac{1}{m+1}$. On level 1, there is one group, i.e., $G_1 = 1$. On level j, the number of groups is equal to the number of jobs on the preceding level, i.e., $G_j = \frac{c(m+1)}{P_{j-1}}$. To calculate W_j and P_j we use the following equations

$$W_j \cdot G_j = \frac{1}{m+1} \tag{1}$$

$$(m+1) W_j = P_j \cdot G_j \tag{2}$$

Note that we can calculate G_j, W_j, and P_j (in this order) for every level. During the run of the algorithm, the adversary changes the sequence according to the following rule:

Adversary rule If the algorithm schedules a job from group $\mathcal{G}$, all other jobs from $\mathcal{G}$ are removed.

An upper bound on the algorithm's performance is 1. To see this, note that the algorithm will get at most one job from each group according to the adversary rule. On level j, there are G_j groups, and according to (1), $W_j \cdot G_j = \frac{1}{m+1}$. Hence the profit from one of the $m+1$ levels is at most $\frac{1}{m+1}$.

According to (2), the total profit in every group is c, since the number of jobs in a group on level j is $\frac{1}{G_j}\frac{(m+1)c}{P_j}$. Assume the algorithm rejects all jobs from a group. According to the adversary rule, the jobs originally in that group remain in the sequence. If OPT schedules these job, it has a performance of at least c, and the ratio is $\frac{c}{1} = c$.

We now show that the algorithm is forced to reject all jobs from at least one group, which proves the theorem. Looking at the algorithm's schedule, there must be at least one job, J_1, scheduled from the group on level 1. On level 2, consider the group of jobs overlapping J_1. Again we must be able to find a job, J_2, scheduled from this group. For each level $j > 1$, we consider only one group corresponding to the job J_{j-1} which was scheduled on the previous level. Note that $j-1$ machines must be completely covered by jobs from previous levels in the time slot used by that group. This process can only continue through the first m levels. $\square$

We now find Δ for the sequence above. Combining Equations 1 and 2 we obtain $P_j = \frac{1}{G_j^2}$. By inserting the expression for G_j, we obtain $P_j = \frac{(P_{j-1})^2}{(c(m+1))^2}$. Knowing that $P_1 = 1$, we can get the following solution to the recurrence

$$P_j = \frac{1}{(c(m+1))^{2^j-2}}.$$

By combining Equation 2 and the expression for G_j, we can find a relation between W_j, P_{j-1}, and P_j: $W_j = c\frac{P_j}{P_{j-1}}$. Looking at the ratio $\frac{W_j}{P_j}$ and inserting the two equations just found, we obtain: $\frac{W_j}{P_j} = \frac{c}{P_{j-1}} = c(c(m+1))^{2^{j-1}-2}$. Now Δ can be found, $\Delta = \frac{\delta_{m+1}}{\delta_1} = \frac{c(c(m+1))^{2^m-2}}{1/(m+1)} = (c(m+1))^{2^m-1}$. Given Δ, we let $a = 2^m - 1$ and obtain a lower bound of $c = \frac{\sqrt[a]{\Delta}}{m+1}$.

6 Job Scheduling

Obviously, all lower bounds for interval scheduling apply to the more general job scheduling problem (for any of the profit models).

6.1 Unit profit

In the theorem below we show that the "shortest processing time" (SPT) rule that considers jobs by non-decreasing p_i, is a 3-approximation algorithm. Note that this priority rule does not take into account either the release time or the deadline of a job.

Theorem 11 For all m, the FIXED PRIORITY GREEDY algorithm SPT is a 3-approximation algorithm for job scheduling with unit profit.

Proof We use "extension" as defined in Section 5.2 and the proof technique from Theorem 2. The same claim can be proved, but this time there is only one case in the inductive step:

CLAIM: After considering k jobs obtaining a schedule σ_k, there is an extension σ', such that if $X' = A(\sigma') \setminus A(\sigma_k)$ denotes the set of not yet considered jobs scheduled in the extension σ', then $3W(\sigma_k) + W(X') \geq OPT(S)$.

Assume S is the input sequence. By the induction hypothesis there is an extension, σ', satisfying $3W(\sigma_k)+W(X') \geq \mathrm{OPT}(S)$, where $X' = A(\sigma') \setminus A(\sigma_k)$ contains jobs not yet seen by the algorithm but scheduled in the extension σ'. Let J_i with processing time p_i denote the next job to be considered by SPT. If this job is not scheduled, the claim is immediate. Otherwise, we let m_i denote the machine on which J_i will be scheduled. Let σ_{k+1} denote the schedule after this job has been scheduled, then $W(\sigma_{k+1}) = W(\sigma_k) + 1$. Note that every job in X' has processing time at least as large as p_i. Hence at most two jobs from X' to be scheduled on m_i by σ' will overlap J_i. We define the new extension σ'' to be σ' but with σ'' mapping to the value *nil* on the job J_i and the jobs overlapping J_i. Let $X'' = A(\sigma'') \setminus A(\sigma_{k+1})$. Then $w(X'') \geq w(X') - 3$ and σ'' is the new extension satisfying $3(\sigma_{k+1}) + w(X'') \geq \mathrm{OPT}(S)$. □

Theorem 12 No ADAPTIVE PRIORITY algorithm is a ρ-approximation for $\rho < 2$ for $m = 1$ for job scheduling, unit profit.

Proof In this proof, we use the following shorthand notation for a job J_i: $[r_i, d_i; p_i]$. The adversary gives the following sequence

- One copy of $[0, 3; 3]$ and $[1, 4; 3]$
- Four copies of $[0, 4; 1]$

We can assume that the algorithm schedules the highest priority job or else the adversary simply removes all other jobs and forces an infinite approximation ratio. If the algorithm gives highest priority to a job of processing time 3, the adversary removes the other job of processing time 3 and the algorithm will only be able to schedule one of the jobs of processing time 1. OPT will reject the job of processing time 3 and schedule all the jobs of processing time 1. The ratio is $\frac{4}{2} = 2$.

If the algorithm gives the highest priority to a job, J_1, of processing time 1, at least one of the jobs (say J_2) of processing time 3 will overlap the time interval where this job has been scheduled. On the sequence $\langle J_1, J_2 \rangle$, OPT will be able to schedule both jobs, and the ratio is $\frac{2}{1} = 2$. $\square$

For interval scheduling, lower bounds can be extended from $m = 1$ to all m for the class Fixed Priority, Greedy. This is not necessarily the case for job scheduling, since two identical jobs can be scheduled on the same machine for this problem.

6.2 Proportional profit and Arbitrary Profit

It is easy to see how to modify the proof of Theorem 2 to show that LPT is a 4-approximation algorithm for job scheduling with proportional profit. Informally, for any job J_k scheduled by LPT with profit p_j, there is at most $3 \cdot p_k$ profit that is lost by LPT because of conflicts with jobs that will be considered later. And it is also easy to construct an example showing that the approximation ratio for LPT is no better than 4. Simply let the longest processing time job have a large window in which it can be scheduled. Wherever LPT schedules the job, the next three jobs can be intervals which will conflict with the one job scheduled and have total profit almost equal to $3 \cdot p_k$.

For arbitrary profit, we can again argue that $O(\Delta)$ is an upper bound (using the proportional profit case) and the non-constant lower bounds (e.g. Δ for $m = 1$) for interval scheduling hold immediately for the more general job scheduling.

7 Minimizing makespan

7.1 Identical machines

Theorem 13 For arbitrary $m \geq 2$, Graham [24] showed that LPT, a fixed priority greedy algorithm, has an (exact) approximation ratio of $\frac{4}{3} - \frac{1}{3m}$. We show that for all $m \geq 2$, no Adaptive Priority algorithm has an approximation ratio better than $\frac{7}{6}$ for makespan on identical machines. For $m = 2$ this bound is tight (matched by LPT).

Proof Let $k = \lfloor \frac{m}{2} \rfloor$. Let S denote the following generalization of the LPT nemesis sequence used by Graham [24] for $m = 2$.

- $2k$ jobs of processing time 3.
- $3(m - k)$ jobs of processing time 2.

An optimal algorithm can obtain a makespan of 6 on this sequence by scheduling 2 jobs of size 3 on k machines and 3 jobs of size 2 on $m - k$ machines. Let A denote an adaptive algorithm. If A ever schedules jobs of size 2 and 3 on the same machine (say machine 1), it will have a makespan of at least 7 on the entire sequence. This follows since after removing one size 2 job and one size 3 job, the total amount of remaining processing time is

$$(2k-1)\cdot 3 + (3(m-k)-1)\cdot 2 = 6m-5 = 6(m-1)+1$$

so that either an additional job is scheduled on machine 1 or the average load on the remaining machines is greater than 6. In the following we therefore assume that the algorithm always has jobs of size 2 and jobs of size 3 on separate machines.

Consider the shortest input subsequence, determined by the way in which the algorithm determines the priorities, for which the algorithm has makespan at least 6. We assume without loss of generality that the algorithm uses all of its m machines (or else the argument is even simpler). Call this subsequence S'. After scheduling S' say that the algorithm has exactly ℓ machines containing only size 3 jobs and $m-\ell$ machines containing only size 2 jobs. Clearly $1 \leq \ell \leq m-1$ or else the algorithm will be forced to have a "mixed" machine (containing both a size 3 and a size 2 job) in order to schedule all of S.

We now show that an optimal algorithm can obtain a makespan of 5 on S'. If the last job in S' has size 2, there are at most

- ℓ jobs of size 3.
- $2(m-\ell)+1$ jobs of size 2.

In this case OPT can schedule one of the jobs of size 2 with one of the jobs of size 3, and otherwise keep the job sizes separated; i.e., $\ell-1$ machines with one job of size 3 and at most $m-\ell$ machines with at most 2 jobs of size 2.

If the last job in S' has size 3, there are at most

- $\ell+1$ jobs of size 3.
- $2(m-\ell)$ jobs of size 2.

This time OPT will have two mixed machines and $\ell-1$ machines with one job of size 3 and (at most) $m-\ell-1$ machines with (at most) two jobs of size 2.

In both cases A will have a performance ratio of at least as large as $\frac{6}{5} > \frac{7}{6}$ on the sequence S'. $\square$

Theorem 14 For $m=4$ identical machines, no algorithm in the class FIXED PRIORITY, GREEDY can achieve an approximation ratio better than $\frac{5}{4}$ for the makespan problem. This is the ratio achieved by LPT.

Proof Recall that for 4 machines a worst case sequence against LPT is $S = \langle 7,7,6,6,5,5,4,4,4\rangle$ [24]. On this sequence LPT has makespan 15, whereas an optimal algorithm has makespan 12. For the class considered, the adversary is allowed to "copy" jobs (See Lemma 2). The adversary gives enough of the following job types [7,6,5,4] to be able to extract the sequences used below. The algorithm decides a total ordering of the four job types and based on this, we divide the argument into two cases. Let $i \to j$ denote the fact that job type i is before job type j in the total ordering.

- Case 1: $4 \to 6$ or $4 \to 7$ or $5 \to 7$.
- Case 2: otherwise.

If Case 1 occurs, we let s denote the small job (4 or 5) and l denote the long job (6 or 7) for which $s \to l$ holds in the total ordering.

The adversary changes the input set to the following

- 4 jobs of size s
- 5 jobs of size l.

We consider a greedy algorithm, and therefore the performance will be $s + 2l$. OPT can schedule 3 jobs of size s on the first machine and have a performance of $\max\{3s, 2l\}$. If 4 is before 6, we have $\frac{4+2\cdot 6}{3\cdot 4} = \frac{16}{12} > \frac{5}{4}$. If 4 is before 7, we have $\frac{4+2\cdot 7}{2\cdot 7} = \frac{18}{14} > \frac{5}{4}$. If 5 is before 7, we have $\frac{5+2\cdot 7}{3\cdot 5} = \frac{19}{15} > \frac{5}{4}$.

If Case 2 occurs, the adversary uses the "LPT input", namely, $S = \langle 7, 7, 6, 6, 5, 5, 4, 4, 4 \rangle$. We are left with the following possible permutations : $[7, 6, 5, 4], [7, 6, 4, 5], [7, 5, 6, 4], [6, 7, 5, 4], [6, 7, 4, 5]$. It is easy to see that for each of these permutations, the greedy algorithm will have makespan 15. This proves the theorem, since this is exactly LPT's performance on this input set.

□

7.2 Subset model

We consider the makespan problem for a model in which the machines are not identical. The unrelated machines model is the most general case of the parallel machine models. The subset model (also called the restricted machines model) is a special case of the unrelated machines model for which every job has processing times $p_j(i) \in \{p_j, \infty\}$. That is, a job can only be scheduled on some subset of the machines and for each "allowable" machine the processing time p_j depends only on the job and not the machine. Aspnes *et al.* [4] show that the online (i.e. the fixed priority is determined by the input sequence) greedy algorithm has the (very poor) approximation ratio m for the makespan on m unrelated machines. In contrast, for the subset model, Azar, Naor and Rom [5] show that the online greedy algorithm has an approximation ratio $\lceil \log m \rceil + 1$ which is essentially optimal for online algorithms even in the special case of unit processing times ($p_j = 1$ for all j). The adversary in the Azar *et al.* online lower bound constructs an input sequence in which every job has exactly two allowable machines. For an online lower bound, it is the adversary that determines the ordering of the inputs. However, the online lower bound argument of Azar *et al.* can be easily modified (see the next theorem) to show that any fixed or adaptive priority algorithm which first orders jobs by the size of the allowable set of machines (but letting the algorithm arbitrarily determine the ordering amongst jobs having the same number of allowable machines) will afford at best an $\Omega(\log m)$ approximation. Perhaps the most natural priority rule in this context is to give priority to jobs having the smallest set of allowable machines. To simplify the discussion we assume this "natural priority" but the proof is easily seen to apply to any ordering determined by the number of allowable machines (e.g. give priority to jobs having the largest number of allowable machines).

Theorem 15 For the subset model with unit processing times, the approximation factor ρ for any Adaptive Priority algorithm (whether or not the scheduling rule is greedy) which in every iteration gives highest priority to jobs having the least number of allowable machines (*no matter how the algorithm breaks ties*) is $\Omega(\log m)$ for the makespan problem.

Proof For completeness we sketch the Azar *et al.* proof. The analysis proceeds in phases. At the start of the i^{th} phase there is an "active" set of machines V_i of cardinality $m/(2^{i-1})$. The algorithm has load $i-1$ on each active machine while the adversary has no load on these active machines and has load one on every inactive machine. The i^{th} phase consists of $m/2^i$ jobs each having a distinct pair of active machines as its allowable subset. For each job in the i^{th} phase, the algorithm schedules the job on one machine and the adversary then schedules the job on the machine not chosen by the algorithm. The phase ends (and the new phase begins) with V_{i+1} being the subset of V_i defined by the machines chosen by the algorithm.

An inspection of this argument shows that the algorithm can arbitrarily order the jobs within a phase but it is the adversary which orders the phases. If the algorithm chooses to always give priority to those jobs having the fewest number of allowable machines, then we modify the Azar *et al.* argument as follows. We now have $m' = m + \log m$ machines where the last $\log m$ machines (call them "extra" machines) are used to determine the size of the allowable subset. Using these extra machines we can then simulate the phases of the Azar *et al.* argument. Namely, if a job in the i^{th} phase has an allowable subset $\{u, v\}$ then we construct a job in the i^{th} "batch" of inputs with allowable subset $\{u, v\} \cup \{m', \ldots m' - i + 1\}$. That is, every job in the i^{th} batch has an allowable subset of size $i + 2$. Initially all m non-extra machines are active. By the constraint on how the algorithm gives priority to jobs, all jobs in the i^{th} batch will be considered before jobs in the $(i+1)^{st}$ batch. When considering a job in the i^{th} batch, say with allowable subset $\{u, v\} \cup \{m', \ldots m' - i + 1\}$, the adversary always chooses a machine in $\{u, v\}$ not chosen by the algorithm. As in Azar *et al.* proof we maintain the invariant that at the end of considering batch i, the adversary has no load on any active machine or extra machine and load one on every inactive machine. If the algorithm chooses any extra machine more than $\log m$ times then we are done. Otherwise, in determining the new set of active machines we make inactive any machine in $\{u, v\}$ not chosen by the algorithm. Clearly, if there are $|V_i|$ active machines at the start of the i^{th} batch then there will be at least $(|V_i|/2) - i \log m$ active machines after considering the i^{th} batch. We can continue this process (simulating the i^{th} phase) for $\Omega(\log m)$ phases and for each batch the algorithm will increase its load on each active machine. That is, we have the invariant that at the end of considering batch i, the algorithm will have load i on every active machine.

□

A more careful analysis of $|V_i|$ and the number of phases being simulated in the previous argument shows that the lower bound is $\log m - o(\log m)$. We also note that similar modifications of the Azar *et al.* argument can defeat other simple orderings (i.e. force an $\Omega(\log m)$ lower bound). For example, consider the case when every job has exactly three allowable machines. Although all machines are identical, an algorithm can still break the symmetry by (say) lexicographically ordering the allowable subsets. In this case, we simply use the first $\log m$ machines as extra machines to make the lexicographic ordering correspond to phases in the Azar *et al.* argument.

Motivated by a previous version of this paper, Regev [34] has recently shown the following lower bound for *all* fixed priority algorithms.

Theorem 16 For the makespan problem on the subset model with unit processing times and exactly two allowable machines per job, any fixed priority algorithm has approximation ratio $\Omega(\frac{\log m}{\log \log m})$.

8 Conclusion

As indicated in the introduction, our characterization of priority algorithms and, in particular, greedy algorithms applies to a much wider class of problems beyond the basic scheduling problems considered in this paper. For example, using our definitions, it is easily seen that Kruskal's minimum spanning tree (forest) algorithm is a well known example of a fixed priority greedy algorithm. (The minimum spanning tree problem is also a classical example for the matroid embedding abstraction.) Prim's minimum spanning tree algorithm is easily formulated as an adaptive greedy algorithm where the priority function depends on the configuration so that edges not leaving the current connected component are given the lowest priority. Our definitions also allow us to classify Dijkstra's shortest path algorithm as an adaptive greedy algorithm. Recent papers have extended the priority framework to the facility location and set cover problems [2] and to a variety of graph theoretic problems [28].

For various metric and graph-theoretic problems, the input specification is an important issue. For example, when considering clustering algorithms in Euclidean space (respectively, an arbitrary metric space), it is often more natural (respectively, necessary) to assume that each input is specified by a vector of distances to the other inputs (and not just a specification of the input location). For greedy algorithms in other settings (e.g. prefix codes, the shortest common superstring, clustering), it is possible (as indicated in the introduction) to extend our definition so that the decision function is based on (say) the next 2 inputs in the ordering. In each such setting we have to precisely define what we mean by an irrevocable decision. For example, in the context of clustering, an irrevocable decision might be to name the cluster to which an input belongs or, as in incremental clustering, irrevocable may just mean that once two inputs are placed in the same cluster, they cannot be separated.

It is also possible to extend our framework to consider scheduling problems allowing preemption. Although this goes beyond the scope of the current paper, we briefly indicate one possible definition. In a fixed priority algorithm, we can allow the scheduling decision to partition a job and then (irrevocably) assign the job parts so that they do not overlap in time. This is essentially how Mc-

Naughton's [32] (optimal) rule works for the preemptive makespan problem on identical machines; however, McNaughton's rule first computes $\sum p_j$ and $\max_j p_j$. For adaptive priority algorithms, we may want to relax the "irrevocable" nature of the scheduling decision. One possibility is to say that the current job being considered is "indefinitely scheduled" until the next "critical time" where the critical times are when a job completes or when a job is released. At any critical time, the algorithm orders all the remaining jobs (where any job that has been executing is modified to account for any processing it has completed).

We claim that modifying our basic definitions to apply to these other settings can all be done in some reasonable way but we admit that the concept of irrevocable decision and how an input is specified may be more natural in some domains (such as non-preemptive scheduling) than in other domains. Moreover, we do not claim that our framework will capture (in a meaningful way) all the settings for which there are important greedy and greedy-like algorithms. For example, by our definition, any online demand paging algorithm is greedy. Moreover, we tend to think of "longest furthest distance", LFD, the optimal offline paging algorithm which evicts the page in the cache being accessed furthest in the future, as a greedy algorithm and we do not see how to formulate LFD as a priority algorithm.

We do believe that the initial results in this paper show that it is possible to precisely characterize an important class of algorithms and thereby make sense of statements such as:

- "This problem is hard; in particular, there is no good greedy approximation algorithm."
- "Either this problem has an optimal greedy algorithm or it does not have a sequence of greedy algorithms resulting in a PTAS for the problem."

Of course, a more ambitious agenda will also attempt to formulate and exploit a precise definition for other important classes of algorithms such as "dynamic programming".

We conclude with a few of the many open problems concerning scheduling:

- For the makespan problem, what is the best approximation ratio achievable by a fixed priority or adaptive priority algorithm? Is there a better ratio than the $\frac{4}{3} - \frac{1}{3m}$ ratio achieved by LPT? Is the $2 - \frac{1}{m}$ bound (achieved by Graham's [23] List Scheduling algorithm) optimal for the makespan problem when the jobs must also satisfy some precedence conditions?
- For any natural scheduling problem, is the best approximation ratio achievable by adaptive priority algorithms always a greedy algorithm? That is, can we find a problem for which non-greedy decisions improve the approximation ratio? We note that Corollary 4 and Theorem 9 in Section 5.3 indicate that non-greedy decisions can help for fixed priority algorithms; however, we would like to obtain such a result for a problem where additional assumptions (e.g. the knowledge of Δ and the assumption that $m \geq \log \Delta$) on the input sequence are not needed.
- What is the best priority algorithm approximation ratio for the makespan problem on non-identical processors? In particular for the subset model (i.e. the $R|p_j(i) \in \{p_j, \infty\}|C_{\max}$ problem) or the unrelated machines model (i.e. the $R|\,|C_{\max}$ problem), can we show that a constant approximation is not possible for adaptive priority algorithms? In particular, is

a constant approximation possible for the very special case of the subset model with unit processing time and when every job has the same number (e.g. two) of allowable machines? Note that Regev's $\Omega(\log m/\log\log m)$ lower bound (see Theorem 16) essentially solves this specific problem (within an $O(\log\log m)$ factor) for fixed priority algorithms. (It is interesting to note that by using a random ordering of the inputs, Broder, *et al.* [12] show that the greedy algorithm achieves an expected approximation ratio of $\Theta(\log m/\log\log m)$ for this specific problem.)

- Can we close the gap between the upper and lower bounds for the interval scheduling problem with proportional profits on $m = 2$ machines? What are the exact approximation bounds for arbitrary $m \geq 2$?
- When does it help to know n, the number of inputs? In particular, can we extend the makespan lower bounds in Section 7.1 to the case of known n? It should be noted that for identical machines and $n \leq 4$, LPT is an optimal algorithm (as follows from Graham's [23] analysis of LPT). What is the best approximation for a priority algorithm for known $n \geq 5$?
- When can a ρ-approximation lower bound for an optimization problem be extended to a lower bound for the related ρ-approximation decision problem that is, when does it help to know the value of the optimum cost/profit?
- Can every α-approximation algorithm in the context of real-time online preemptive scheduling be converted to an α-approximation (offline, non-preemptive) priority algorithm? In addition to the Baruah *et al.* [8] analogue of our Theorem 5, Koren and Shasha [29] have an analogue of our Theorem 9, although in this case the statement and analysis of their algorithm do not appear to be as directly related to our result as for the case of Theorem 5.

Acknowledgments

We thank Yuval Rabani and students at the Technion and The University of Toronto for their patience and feedback concerning some initial attempts at these definitions. We thank Baruch Schieber, Jiri Sgall, David Shmoys, and Gerhard Woeginger for continuously providing explanations and references concerning scheduling problems. Finally, we thank Joan Boyar and Kim Larsen for convincing us again that the intuitive argument previously used in Lemma 1 could indeed be made rigorous.

References

[1] M. Adler, A.L. Rosenberg, R.K. Sitaraman, and W. Unger. Scheduling time-constrained communication in linear networks. In *Proceedings of the Tenth International ACM Symp. on Parallel Algorithms and Architecture*, pages 269–278, 1998.

[2] S. Angelopoulos and A. Borodin. On the power of priority algorithms for facility location and set cover. In *Proceedings of the Fifth International Workshop, Approx 2002*, pages 26–39, Rome, Italy, September 2002.

[3] E. M. Arkin and E. L. Silverberg. Scheduling jobs with fixed start and end times. *Disc. Appl. Math*, 18:1–8, 1987.

[4] James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *Journal of the ACM*, 44(3):486–504, May 1997.

[5] Y. Azar, J. Naor, and R. Rom. The competitiveness of online assignments. *Journal of Algorithms*, 18(2):221–237, 1995.

[6] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. *JACM*, 48(5):1069–1090, 2001.

[7] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Approximating the throughput of multiple machines under real-time scheduling. *SICOMP*, 31(2):331–352, 2001.

[8] S. Baruah, G. Koren, B. Mishra, A. Raghunathon, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *FOCS: IEEE Symposium of Foundations of Computing*, pages 100–110, 1991.

[9] P. Berman and B. Das Gupta. Improvements in throughput maximization for real-time scheduling. In *STOC: ACM Symposium on Theory of Computing*, 2000.

[10] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, April 1998.

[11] Gilles Brassard and Paul Bratley. *Introduction to Algorithms*, chapter 6. Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[12] A.Z. Broder, A. Frieze, C. Lund, S. Phillips, and N. Reingold. Balanced allocations for tree-like inputs. *Information Processing Letters*, 55(6):329–332, 1995.

[13] M.C. Carlisle and E.L. Llyod. On the k-coloring of intervals. *Lecture Notes in Computer Science*, 497:90–101, 1991.

[14] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *STOC: ACM Symposium on Theory of Computing*, pages 626–635, 1997.

[15] Yookun Cho and Sartaj Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9(1):91–103, 1980.

[16] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to Algorithms*, chapter 28. MIT Press, Cambridge, Mass., 1990.

[17] Gregory Dobson. Scheduling independent tasks on uniform processors. *SIAM Journal on Computing*, 13(4):705–716, 1984.

[18] T. Erlebach and F.C.R. Spieksma. Interval selection: Applications, algorithms, and lower bounds. *Technical Report 152, Computer Engineering and Networks Laboratory, ETH*, October 2002.

[19] U. Faigle and W.M. Nawijn. Greedy k-decomposition of interval orders. In *Proceedings of the Second Twente Workshop in Graphs and Combinatorial Optimization*, pages 53–56, University of Twente, 1991.

[20] Donald K. Friesen. Tighter bounds for LPT scheduling on uniform processors. *SIAM Journal on Computing*, 16(3):554–560, June 1987.

[21] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, New York City, New York., 1996.

[22] T.E. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[23] R. L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell Systems Technical Journal*, 45:1563–1581, 1966.

[24] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(1):416–429, March 1969.

[25] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Mathematics*, 5:287–326, 1979.

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

[27] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-hard problems.* PWS Publishing Company, 1997.

[28] R. Impagliazzo and S. Davis. Models of greedy algorithms for graph problems. *Unpublished manuscript*, November 2002.

[29] G. Koren and D. Shasha. Moca: a multiprocessor on-line competitive scheduling algorithm for real-time system scheduling. *Theoretical Computer Science*, 128:75–97, 1994.

[30] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S. C. Graves, A. H. G. Rinnooy Kan, and P. Zipkin, editors, *Handbooks in Operations Research and Management Science, Volume 4: Logistics of Production and Inventory.* North-Holland, Amsterdam, 1993.

[31] Stefano Leonardi, Alberto Marchetti-Spaccamela, Alessio Presciutti, and Adi Rosén. On-line randomized call control revisited. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 323–332, San Francisco, California, January 1998.

[32] R. McNaughton. An n-job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15:101–109, 1959.

[33] B.M.E. Moret and H.D. Shapiro. *Algorithms from P to NP*, chapter 5. Benjamin/Cummings, Redwood City, California, 1991.

[34] Oded Regev. Priority algorithms for makespan minimization in the subset model. *Information Processing Letters*, 84(3):153–157, Septmeber 2002.

[35] Sartaj K. Sahni. Algorithms for scheduling independent tasks. *Journal of the ACM*, 23(1):116–127, January 1976.

[36] Steve Seiden, Jiri Sgall, and Gerhard Woeginger. Semi-online scheduling with decreasing job sizes. *Operations Research Letters*, 27(5):215–221, 2000.

[37] J. Sgall. On-line scheduling. *Lecture Notes in Computer Science*, 1442:196–231, 1998.

[38] F.C.R. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, (2):215–227, 1999.