Scheduling with a Limited Testing Budget

Tight Results for the Offline and Oblivious Settings*

Christoph Damerius
Department of Informatics, Universität Hamburg, Germany

Peter Kling
Department of Informatics, Universität Hamburg, Germany

Minming Li
Department of Computer Science, City University of Hong Kong, China

Chenyang Xu
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China

Ruilong Zhang[1]
Department of Computer Science and Engineering, University at Buffalo, NY, USA

Abstract

Scheduling with testing falls under the umbrella of the research on optimization with explorable uncertainty. In this model, each job has an upper limit on its processing time that can be decreased to a lower limit (possibly unknown) by some preliminary action (testing). Recently, Dürr et al. [10] has studied a setting where testing a job takes a unit time, and the goal is to minimize total completion time or makespan on a single machine. In this paper, we extend their problem to the budget setting in which each test consumes a job-specific cost, and we require that the total testing cost cannot exceed a given budget. We consider the offline variant (the lower processing time is known) and the oblivious variant (the lower processing time is unknown) and aim to minimize the total completion time or makespan on a single machine.

For the total completion time objective, we show NP-hardness and derive a PTAS for the offline variant based on a novel LP rounding scheme. We give a $(4+\epsilon)$-competitive algorithm for the oblivious variant based on a framework inspired by the worst-case lower-bound instance. For the makespan objective, we give an FPTAS for the offline variant and a $(2+\epsilon)$-competitive algorithm for the oblivious variant. Our algorithms for the oblivious variants under both objectives run in time $\mathcal{O}(poly(n/\epsilon))$. Lastly, we show that our results are essentially optimal by providing matching lower bounds.

2012 ACM Subject Classification Theory of computation → Scheduling algorithms

Keywords and phrases scheduling, total completion time, makespan, LP rounding, competitive analysis, approximation algorithm, NP hardness, PTAS

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.38

Related Version *Full Version*: `https://arxiv.org/abs/2306.15597`

Acknowledgements We thank the anonymous reviewers for their many insightful comments and suggestions. Chenyang Xu was supported in part by Science and Technology Innovation 2030 –"The Next Generation of Artificial Intelligence" Major Project No.2018AAA0100900, and the Dean's Fund of Shanghai Key Laboratory of Trustworthy Computing, East China Normal University. Ruilong Zhang was supported by NSF grant CCF-1844890.

[1] This work was partially done when the author was a student at City University of Hong Kong.
* All authors (ordered alphabetically) have equal contributions and are corresponding authors.

1 Introduction

With increased interest in applying scheduling algorithms to solve real-life problems, many models and methods have been addressing the uncertainty in the scheduling community. Several elegant models that capture uncertainty have been studied in the past two decades, most of which fall under the umbrella of the research on robust optimization [24, 7, 23, 29] or stochastic optimization [16, 18, 15, 14]. In those settings, the uncertainty is usually described by the input. In robust optimization, the input consists of several scenarios, while the input is sampled from a known distribution in stochastic optimization. In some practical cases, we can gain additional information about the input by paying extra costs, e.g., money, time, energy, memory, etc. This model is also known as *explorable uncertainty*, which aims to study the trade-offs between the exploration cost and the quality of a solution.

An intriguing scheduling model for explorable uncertainty was proposed by Dürr et al. [10] under the name of *scheduling with testing*. In their model, before executing a job, one can invest some time to *test* that job, potentially reducing its processing time. A practical use case is code optimization, where we could either simply run programs/codes (jobs) as they are or preprocess them through a code optimizer to hopefully improve their execution times.

Their model considers the test cost as the time spent by the machine, which is certainly important and captures many applications stated in [10]. However, it may fail to describe some scenarios. For example, in the code optimization problem, the code optimizer may be an expert who might need to be employed by other companies. This situation is usually faced by cloud computing companies [25, 6], which accept some tasks and want to assign them to servers. They can employ experts to optimize some time-intensive tasks to speed up the execution. In this way, the server can finish more tasks, thus creating more profit for the company. After optimizing, the experts return the optimized tasks to the cloud computing company, and the company can start to assign tasks to servers. Thus, optimizing does not use servers' time. Different tasks may require a different amount of effort from the expert to optimize and therefore needs different cost. The company has a fixed budget and aims to select some tasks to optimize (test) such that the total processing time of tasks is minimized.

Informally, we consider a natural variant of the model proposed in [10], in which we are given a set of n jobs and a total budget B for testing. Each job j has an upper limit on the processing time $p_j^\wedge$ and testing cost c_j. After testing, the processing time of job j decreases to a lower limit $p_j^\vee$, which is possibly hidden for the algorithms. We refer to the model as the *offline* version if $p_j^\vee$ is known by the algorithm for all jobs j; otherwise, it is called *oblivious* version. The paper considers two objectives: the total completion time objective and the makespan objective, which are two well-studied objectives for scheduling problems in the literature [19, 5, 22, 9]. The formal definition of our problem is stated in Section 2.

Note that the offline version of the model in Dürr et al. [10] is easy, even if testing a job j requires a job-specific amount of time t_j. Testing a job is then beneficial if $p_j^\wedge - p_j^\vee > t_j$. In contrast, we show that the offline version of our budgeted variant of the problem is NP-hard, assuming each job takes a job-specific amount of budget to be tested. We study both the offline and the oblivious settings. Further, we differentiate between the uniform cost variant, where each job takes one unit of budget to be tested, and a non-uniform variant, where the testing cost is job-specific.

1.1 Our Contributions

The paper studies the problem of Scheduling with a Limited Testing Budget (SLTB) under both the total completion time minimization objective (SLTB_{TC}) and the makespan minimization objective (SLTB_{M}). For both objectives, we further distinguish the offline and oblivious settings.

Our main results are summarized in Table 1. For the objective of total completion time minimization, in the offline setting, we show that the problem is NP-hard even when all the lower processing times are 0 by a reduction from the PARTITION problem, and then give a PTAS. The PTAS is derived based on a novel LP rounding scheme. Further, we find that there exists an FPTAS if all the jobs share the same lower processing time. For the oblivious setting, we give a $(4+\epsilon)$-competitive deterministic algorithm for any ϵ (we use the concept of the competitive ratio following the previous work [10]). The ratio is almost tight since we prove that no deterministic algorithm has a competitive ratio strictly better than 4. For the objective of makespan minimization, the main results are derived based on a connection between our problem and the classical 0-1 knapsack problem. We prove that the offline setting is NP-hard and admits an FPTAS, while for the oblivious setting, an almost tight competitive ratio of $2+\epsilon$ can be obtained.

Table 1 The summary of our results. The vector $\mathbf{p}^\vee := (p_1^\vee, \dots, p_n^\vee)$ is the lower processing time vector, and $\mathbf{p}^\vee \in \mathbb{R}_{\geq 0} \cdot \mathbf{1}$ means that all the entries of the vector share the same value. ϵ is an arbitrary positive parameter.

		UB (SLTB_{TC})	LB (SLTB_{TC})	UB (SLTB_{M})	LB (SLTB_{M})
Offline	$\mathbf{p}^\vee \in \mathbb{R}^n_{\geq 0}$	PTAS (Thm. 4)	NP-C (Thm. 1)	FPTAS	NP-C
	$\mathbf{p}^\vee \in \mathbb{R}_{\geq 0} \cdot \mathbf{1}$	FPTAS			
Oblivious	—	$4+\epsilon$	4	$2+\epsilon$	2

Paper Organization. We first state some useful notation in Section 2, and then give an overview of our techniques in Section 3. In the remaining part of the main body (Section 4), we describe a PTAS for the offline SLTB with the total completion time objective, the most interesting and technical part of our work. Due to space limitations, the proofs are omitted in this version. All proofs and our other results can be found in the full version [8].

1.2 Related Work

Explorable Uncertainty. Scheduling with testing falls under the umbrella of the research on optimization with explorable uncertainty, where some additional information can be obtained through queries. The model under the stochastic setting can be traced back to Weitzman's Pandora's Box problem [28] and it remains an active research area up to the present [17, 11]. The model under the adversarial setting was first coined by Kahan [21] to study the number of queries necessary to obtain an element set's median value. So far, many optimization problems have been considered in this setting, e.g. caching [27], geometric tasks [4], minimum spanning tree [20, 26], knapsack [12] and so on.

Scheduling with Testing. The problem of scheduling with testing was first coined by Dürr et al. [10]. They consider a model where each testing operation requires one unit of time and mainly investigate non-preemptive schedules on a single machine to minimize the total completion time or makespan. Since the offline version of the problem (algorithms know the lower processing time of each job) is trivial, they mainly consider the online version. They present a 2-competitive deterministic algorithm for total completion time minimization while the deterministic lower bound is 1.8546. They also gave a 1.7453-competitive randomized algorithm while the randomized lower bound is 1.6257. For makespan minimization, they give a 1.618-competitive deterministic algorithm and show that it is optimal for the deterministic setting. They also present a 4/3-competitive randomized algorithm and show that it is optimal.

Later, Albers and Eckl [2] consider the non-uniform testing case where the testing time depends on the job. They investigate the single-machine preemptive and non-preemptive scheduling to minimize the total completion time or makespan. The offline version of this problem is still trivial, so they mainly consider the oblivious version. They present a 4-competitive deterministic algorithm for total completion time minimization and a 3.3794-competitive randomized algorithm. If preemption is allowed, the deterministic ratio can be further improved to 3.2361. All lower bounds are the same as in the uniform testing case. For makespan minimization, they extend the algorithm proposed in [10] and show that the approximation can be preserved in the non-uniform testing case.

Scheduling with testing on identical machines is also considered in the literature [3]. The authors mainly consider the makespan minimization in both non-preemptive and preemptive settings. They look into the non-uniform testing case. For the preemptive setting, they present a 2 competitive algorithm which is essentially optimal. For the non-preemptive setting, they give a 3.1016-competitive algorithm for the general testing case, and the ratio can be improved to 3 when each test requires one unit of time. Later, Gong et al. [13] improved the non-preemptive ratios to 2.9513 and 2.8081 for non-uniform and uniform testing cases, respectively.

2 Preliminaries

An *instance* to Scheduling with a Limited Testing Budget (SLTB) is a 5-tuple $\mathcal{I} = (J, \mathbf{p}^\wedge, \mathbf{p}^\vee, \mathbf{c}, B)$. $J = [n]$ denotes a *set of n jobs*. Each job j has an *upper limit on the processing time* $p_j^\wedge \in \mathbb{R}_{\geq 0}$, a *lower processing time* $p_j^\vee \in [0, p_j^\wedge]$ and a *testing cost* $c_j \in \mathbb{R}_{\geq 0}$. These parameters are collected in the *lower and upper limit processing time vectors* $\mathbf{p}^\vee$ and $\mathbf{p}^\wedge$, respectively, and a *vector of testing costs* $\mathbf{c}$. Additionally, a total amount of *budget* $B \in \mathbb{R}_{\geq 0}$ is given.

Each job j can be executed either in a tested or untested state. When job j is tested, j will take $p_j^\vee$ time to process; otherwise, it requires $p_j^\wedge$ time. If a job is tested, it consumes c_j budget; otherwise, no budget is consumed.

We consider offline and oblivious versions. For the offline version, the algorithm knows the complete instance $\mathcal{I}$. For the oblivious version, the lower processing time vector $\mathbf{p}^\vee$ is hidden from the algorithm, and the remaining information of the instance is known a priori.

In this work, we only consider non-preemptive and, w.l.o.g., gapless schedules on a single machine. Once a job starts executing, other jobs cannot be processed until the current job is finished. Thus, a schedule corresponds to a specific ordering of jobs. We define $I \coloneqq [n]$ to be the *set of positions*. The job in position $i \in I$ will be the i^{th} job executed in the schedule.

A *schedule* $S = (\sigma, J_\vee)$ for an instance $\mathcal{I} = (J, \mathbf{p}^\wedge, \mathbf{p}^\vee, \mathbf{c}, B)$ is defined by a *job order* σ and a *testing job set* $J_\vee \subseteq J$. The job order $\sigma : J \to I$ is a bijective function that describes the order in which the jobs are processed (i.e., job j is the $\sigma(j)$-th processed job in the non-preemptive schedule). The testing job set $J_\vee \subseteq J$ represents a set of jobs to test with $\sum_{j \in J_\vee} c_j \leq B$.

Given a schedule S, we can indicate whether a job is tested using a *set of types* $T \coloneqq \{\vee, \wedge\}$. We say that j is *of type* $\vee$,$\wedge$ if it is *tested*, *untested*, respectively. If S schedules a job j of type t into position i, we also say that *position i is of type t*. For a schedule $S = (\sigma, J_\vee)$ and a job j, let the *type* $t_S(j)$ *of* j *in* S be $\vee$ if $j \in J_\vee$ and $\wedge$ otherwise. Denote by $C_j \coloneqq \sum_{j' \in J, \sigma(j') \leq \sigma(j)} p_{j'}^{t_S(j')}$ the *completion time* of job j in schedule S. The total completion time is the sum of all completion times, i.e., $\sum_{j \in J} C_j$, and the *makespan* is the maximum completion time among all jobs, i.e., $\max_{j \in J}\{C_j\}$.

Given a testing job set $J_\vee$, the optimal ordering of the jobs is easy to determine. The ordering is relevant for the total completion time minimization but not for the makespan minimization. It is a well-known fact that the SPT rule (shortest processing time first) orders the jobs optimally for total completion time minimization. The processing times are in our case $p_j^\vee$ if job j is tested and $p_j^\wedge$ otherwise. Thus, an optimal schedule can be easily constructed from an optimal testing job set $J_\vee$.

3 Overview of Techniques

In this section, we focus on the total completion time minimization and give technical overviews for the offline model and the oblivious model.

3.1 Offline SLTB under Total Completion Time Minimization

For offline SLTB_{TC}, we mainly show the following theorem. The NP-hardness is proved via a reduction from the Partition problem. Due to space limitations, the proofs are omitted and can be found in the full version [8], we focus on introducing the high-level ideas of our PTAS, the most interesting and technical part of this paper.

▶ **Theorem 1.** *The offline $SLTB_{TC}$ problem is NP-hard even when the lower processing time of each job is* 0*, and admits a PTAS.*

Our algorithm is based on an integer linear programming (ILP) formulation for offline SLTB_{TC}. The ILP contains variables $x_{j,i,t}$ that dictate whether job $j \in J$ should be scheduled in position $i \in I$ of type $t \in T$. (See Section 4.1 for the exact definition of this ILP.) The ILP is conceptually similar to the classical matching ILP on bipartite graphs [1], with jobs and positions representing the two disjoint independent sets of the bipartition. A matching would then describe an assignment of jobs to positions. However, there are two main differences. First, we have two variables per pair of job and position (distinguished by the type $t \in T$). This translates to each job-position pair having two edges that connect them in the (multi-)graph. Second, the total cost of jobs tested is restricted by some budget B. This causes a dependency when selecting edges in the graph.

Our approach combines a rounding scheme of the ILP with an exploitation of the cost structure of the problem. We relax the ILP to an LP by allowing the variables $x_{j,i,t}$ to take on fractional values between 0 and 1. We start with an optimal LP solution and then continue with our rounding scheme, which consists of two phases. In the first phase, we round the solution such that all fractional variables correspond to the edges of a single cycle in the graph mentioned above. These variables are hard to round directly without overusing the budget. Here we start the second rounding phase. We relax some of the constraints in the LP to be able to continue the rounding process. Specifically, we allow certain positions to schedule two jobs (we call these positions *crowded*). We end up with an integral (but invalid) solution that has some crowded positions. Then, we "decrowd" these positions by moving their jobs to nearby positions (shifting the position of some other jobs one up), and show that we can bound the cost of moving a job this way in terms of its current contribution to the overall cost. Observing that moving a job from position i to position i' (note that positions are counted from right to left) increases that job's contribution by a factor of i'/i, if a crowded position lies far to the right (i is small), we cannot afford to move one of its jobs too far away. For example, in the extreme case that the rightmost position is crowded (i.e., $i = 1$), even the smallest possible move of one of its jobs to the second-rightmost position (i.e., $i' = 2$) already doubles that job's contribution. Thus, our algorithm tries to avoid producing crowded positions that lie too far to the right (at small positions).

To this end, the rounding process in this phase is specifically tailored to control where crowded positions can appear in the integral solution. We look at the $f(\epsilon) = 2/\epsilon + 1$ smallest (rightmost) positions that appear on the current path (representing fractional variables), and select one of them (let's call it the cut-position) to cut the path into two halves. This is done such that each half contains $1/\epsilon$ many of the smallest positions on the current path. By shifting workload along each of these two halves, we can make one of them integral. This integral half gives us $1/\epsilon$ positions that are not crowded, while the cut-position might have become crowded (as might any future cut-position in the remaining fractional path). Because we cut somewhere in the $f(\epsilon)$ rightmost positions of the path, we can show that for each crowded position, there are many positions further to the right of the schedule that are not crowded (this is basically what our charging argument formalizes). In the end, this allows us to prove that no job is moved too far from its original position (relative to its original position), keeping the cost increase due to such moves small.

3.2 Oblivious SLTB under Total Completion Time Minimization

For the oblivious model where the lower processing time vector $\mathbf{p}^\vee$ is hidden, we show that $(4+\epsilon)$ approximation can be obtained, and further, prove that the ratio is the best possible.

▶ **Theorem 2.** *For oblivious SLTB$_{TC}$ and any $\epsilon > 0$, there exists a deterministic algorithm with a competitive ratio of $(4+\epsilon)$, while no deterministic algorithm can obtain a competitive ratio strictly smaller than 4.*

We start by considering the oblivious uniform $\mathrm{SLTB}_{\mathrm{TC}}$ problem to build some intuition. The uniform case limits the number of tested jobs, i.e., we can test at most k jobs. Clearly, for the worst-case analysis, we can assume that each job j tested by our algorithm has $p_j^\vee = p_j^\wedge$; that is, we exhaust the budget, but no job's processing time gets reduced. In contrast, for all the jobs tested by an optimal solution, their processing times can be reduced to 0. Thus, from this perspective, regardless of which jobs we test, our total completion time remains unchanged, but the optimum depends on our tested jobs because the adversary can only let the job j that is not tested by our algorithm have $p_j^\vee = 0$.

Then we find that the oblivious uniform $\mathrm{SLTB}_{\mathrm{TC}}$ problem is essentially equivalent to the following optimization problem: given a set of jobs J with $\mathbf{p}^\wedge$ and $\mathbf{p}^\vee = \mathbf{0}$, the goal is to select k jobs such that the minimum total completion time obtained by testing at most k unselected jobs is maximized. The selected jobs can be viewed as the jobs tested by our algorithm, while the minimum total completion time obtained by testing unselected jobs is the optimum of oblivious uniform $\mathrm{SLTB}_{\mathrm{TC}}$. When our objective value is fixed, a larger optimum implies a better competitive ratio. For this much easier problem, it is easy to see that the best strategy is selecting the k jobs with the largest upper processing time, which is the set of jobs that would be tested by an optimal solution of $\mathrm{SLTB}_{\mathrm{TC}}$ instance $\mathcal{I} = (J, \mathbf{p}^\wedge, \mathbf{p}^\vee = \mathbf{0}, \mathbf{c} = \mathbf{1}, k)$.

We build on the above argument to give the algorithm for the non-uniform case $\mathcal{I} = (J, \mathbf{p}^\wedge, \mathbf{p}^\vee, \mathbf{c}, B)$. The basic idea is constructing an auxiliary instance $\widetilde{\mathcal{I}} := (J, \mathbf{p}^\wedge, \widetilde{\mathbf{p}^\vee} = \mathbf{0}, \mathbf{c}, B)$, solving the instance optimally or approximately, and returning the obtained solution. Use $\mathrm{ALG}(\cdot)$ and $\mathrm{OPT}(\cdot)$ to denote the objective values obtained by our algorithm and an optimal solution of an input instance, respectively. By the theorem proved in the offline model, we have $\mathrm{ALG}(\widetilde{\mathcal{I}}) \le (1+\epsilon)\mathrm{OPT}(\widetilde{\mathcal{I}})$ for any $\epsilon > 0$. In the analysis, we show that our objective value can be split into two parts: $\mathrm{ALG}(\mathcal{I}) \le 2\mathrm{ALG}(\widetilde{\mathcal{I}}) + 2\mathrm{OPT}(\mathcal{I})$, and therefore, due to $\mathrm{OPT}(\widetilde{\mathcal{I}}) \le \mathrm{OPT}(\mathcal{I})$, a competitive ratio of $(4+2\epsilon)$ can be proved.

The lower bound is shown by a hard instance $\mathcal{I} = (J, \mathbf{p}^\wedge = \mathbf{1}, \mathbf{p}^\vee, \mathbf{c} = \mathbf{1}, B = \frac{n}{2})$, where the adversary always lets our tested jobs have lower processing time 1 and the processing time of any other job be 0. Apparently, any deterministic algorithm's objective value is $n(n+1)/2$, while an optimal solution can achieve a total completion time of $n(n+2)/8$, which implies a lower bound of 4. [2]

3.3 SLTB_M under Makespan Minimization

▶ **Theorem 3.** *The offline $SLTB_M$ problem is NP-hard and admits an FPTAS, while for oblivious $SLTB_M$, an almost tight competitive ratio of $2+\epsilon$ can be obtained (for any $\epsilon > 0$).*

The offline SLTB problem under makespan minimization is closely related to the classical 0-1 knapsack problem. The classical 0-1 knapsack problem aims to select a subset of items such that (i) the total weight of the selected items does not exceed a given capacity; (ii) the total value of the selected items is maximized. To see the connection, consider the testing cost of each job as the weight of each item and the profit of testing a job $(p_j^\wedge - p_j^\vee)$ as the value of an item. Then we build on the algorithmic idea of the knapsack dynamic programming and design an FPTAS for the offline setting.

We use the same framework as the total completion time minimization model for the oblivious setting and obtain a $(2+\epsilon)$-competitive algorithm. The ratio becomes better here since, for the makespan objective, we have $\text{ALG}(\mathcal{I}) \leq \text{ALG}(\widetilde{\mathcal{I}}) + \text{OPT}(\mathcal{I})$, saving a factor of 2. The lower bound proof is also based on the same hard instance $\mathcal{I} = (J, \mathbf{p}^\wedge = \mathbf{1}, \mathbf{p}^\vee, \mathbf{c} = \mathbf{1}, B = n/2)$. Any deterministic algorithm's makespan is n while the optimum is $n/2$, giving a lower bound of 2.

4 Offline Setting for SLTB under Total Completion Time Minimization

This section considers the Scheduling with a Limited Testing Budget problem under total completion time minimization (SLTB_TC) in the offline setting and aims to show the following theorem.

▶ **Theorem 4.** *There exists a PTAS for $SLTB_{TC}$.*

For convenience, we refer to a problem instance as a pair $\mathcal{I} = (J, B)$, dropping the processing time and cost vectors $\mathbf{p}^\vee$, $\mathbf{p}^\wedge$, and $\mathbf{c}$ (which we assume to be implicitly given). Moreover, in this section, we consider the job positions $I = [n]$ *in reverse order* to simplify the calculations. That is, a job j scheduled in position $i \in I$ is processed as the *i-th last* job.

4.1 ILP Formulation and Fixations

We start by introducing our ILP formulation of the SLTB_TC problem and defining the term *fixation* of a (relaxed) instance of our ILP. Such fixations allow us to formally fix the values of certain variables in the (relaxed) ILP when analyzing our algorithm.

[2] Since in the worst-case, the upper and lower processing times of jobs tested by the algorithm are equal, it does not help if the algorithm can be adaptive, i.e., change its testing strategy based on such an information.

ILP Formulation. Our ILP has indicator variables $x_{j,i,t}$ that are 1 if job j is scheduled at position i of type t and 0 otherwise. The *contribution* of such a job to the total completion time is[3] $i \cdot p_j^t$. We have constraints to ensure that each of the n positions schedules one job, that each job is scheduled once, and that the cost of tested jobs do not exceed the budget. The equivalence between ILP solutions and SLTB_{TC} schedules is formalized in Lemma 5.

Consider an instance $\mathcal{I} = (J, B)$ of the SLTB_{TC} problem. We define an ILP $ILP_{\mathcal{I}}$, with the variables $x_{j,i,t}$ for each job $j \in J$, position $i \in I$ and type $t \in T$.

$$\min \sum_{j \in J, i \in I, t \in T} i \cdot p_j^t \cdot x_{j,i,t}$$

$$\text{s.t.} \sum_{j \in J, t \in T} x_{j,i,t} = 1 \quad \forall i \in I \quad (1) \qquad \sum_{i \in I, t \in T} x_{j,i,t} = 1 \quad \forall j \in J \quad (2)$$

$$\sum_{j \in J, i \in I} c_j x_{j,i,\vee} \leq B \quad (3) \qquad x_{j,i,t} \in \{0, 1\} \ \forall j \in J, i \in I, t \in T \quad (4)$$

For $ILP_{\mathcal{I}}$ with variable set $X_{\mathcal{I}} \coloneqq \{ x_{j,i,t} \mid j \in J, i \in I, t \in T \}$, a *solution* $x : X_{\mathcal{I}} \to \mathbb{R}$ assigns each variable in $X_{\mathcal{I}}$ a value. Solution x is called *valid* if it satisfies the four constraints and *invalid* otherwise. For a (possibly invalid) solution x for $\mathcal{I}$ we define its *cost* as $C_{\mathcal{I}}(x) \coloneqq \sum_{j \in J, i \in I, t \in T} i \cdot p_j^t \cdot x_{j,i,t}$ and its *budget use* $B_{\mathcal{I}}(x) \coloneqq \sum_{j \in J, i \in I} c_j x_{j,i,\vee}$ (we omit $\mathcal{I}$ from $C_{\mathcal{I}}$ and B if it is clear from the context). We refer to the different constraints as (1) *position constraints*, (2) *job constraints*, (3) *budget constraint*, and (4) *integrality constraints*.

▶ **Lemma 5.** *Let $\mathcal{I} = (J, B)$ be an instance for $SLTB_{TC}$. For each valid solution x to $ILP_{\mathcal{I}}$ there exists a schedule S for $\mathcal{I}$ with $C(S) = C(x)$ and vice versa. Each can be computed from the other in polynomial time.*

Relaxation and Fixations. Our algorithm and analysis use relaxed variants of $ILP_{\mathcal{I}}$ that *fix* certain ILP variables (indicating that, e.g., certain jobs must be tested). It also keeps track of *crowded* positions, in which our algorithm may (temporarily) schedule two jobs (violating the position constraints). We introduce the notion of a *fixation* $\mathcal{F}$ to formally define these relaxed variants $LP_{\mathcal{I},\mathcal{F}}$ of $ILP_{\mathcal{I}}$. [4]

▶ **Definition 6.** A *fixation* $\mathcal{F} = (J(\mathcal{F}), X(\mathcal{F}), I^C(\mathcal{F}))$ of an SLTB_{TC} instance $\mathcal{I}$ consists of:

1. a set of *tested* jobs $J(\mathcal{F}) \subseteq J$,
2. a set of *fully-fixed* variables $X(\mathcal{F}) \subseteq X_{\mathcal{I}}$ where $j \notin J(\mathcal{F})$ for all $x_{j,i,t} \in X(\mathcal{F})$, and
3. a set of *crowded positions* $I^C(\mathcal{F}) \subseteq I$.

For a set operator $\circ \in \{\cup, \cap, \setminus\}$ and a set of positions $\bar{I} \subseteq I$, we use the notation $\mathcal{F} \circ \bar{I} \coloneqq (J(\mathcal{F}), X(\mathcal{F}), I^C(\mathcal{F}) \circ \bar{I})$ to express the change to the crowded positions of $\mathcal{F}$.

Given a fixation $\mathcal{F}$, we define the following relaxed variant $LP_{\mathcal{I},\mathcal{F}}$ of $ILP_{\mathcal{I}}$:

1. For each $x_{j,i,t} \in X_{\mathcal{I}}$ we relax the integrality constraint to $0 \leq x_{j,i,t} \leq 1$ (*unit constraints*).
2. For each $x_{j,i,t} \in X(\mathcal{F})$, we add the constraint $x_{j,i,t} = 1$ (*fully-fixed constraints*).
3. For each $j \in J(\mathcal{F})$, we add the constraint $\sum_{i \in I} x_{j,i,\vee} = 1$ (*tested job constraints*).
4. For each $i \in I^C(\mathcal{F})$, we relax the position constraint to $\sum_{j \in J, t \in T} x_{j,i,t} \in \{0, 1, 2\}$.

The resulting LP is omitted in this version and can be found in the full version [8].

[3] Remember that we consider the position in *reverse* order. Thus, the job at position i is the i-th last job.

[4] Our PTAS will enumerate through a polynomial number of fixations, and solve the problem for each one of them. The approximation guarantee is then derived for the fixation that is consistent with the optimal solution.

4.2 Graph-theoretic Perspective & Paths

Consider an SLTB_{TC} instance $\mathcal{I} = (J, B)$ with fixation $\mathcal{F}$ and a (fractional) solution x to $LP_{\mathcal{I},\mathcal{F}}$. The main building block of our algorithm is a rounding scheme based on the following graph-interpretation of $\mathcal{I}$ and corresponding paths based on the current solution x:

▶ **Definition 7.** The *instance graph* $G_{\mathcal{I}} \coloneqq (J \cup I, E)$ is a bipartite multi-graph between the jobs J and positions I with exactly two edges between any pair $j \in J$ and $i \in I$. We identify the edge set E with the variable set $X_{\mathcal{I}}$ and refer to a variable $x_{j,i,t} \in E = X_{\mathcal{I}}$ also as an *edge of type* $t \in \{\vee, \wedge\}$ *between* j *and* i.

▶ **Definition 8.** A *path* P *in solution* x is a weighted path from $i_s \in I$ (*start position*) to $i_e \in I$ (*end position*) in $G_{\mathcal{I}}$, where the weight of an edge $x_{j,i,t} \in P$ is its value in x. P is called *integral* if all its weights are integral and *fractional* if they are all (strictly) fractional.

Nodes and edges in P must be pairwise distinct, except for possibly equal start and end positions $i_s = i_e$, in which case we refer to P also as a *cycle*. We define $J(P)$ as the path's set of jobs, $I(P)$ as its set of positions, and $K(P) \coloneqq I(P) \setminus \{i_s, i_e\}$. Moreover, $X(P)$ is the sequence of edges/variables from start to end position in P. We say the i-th edge in $X(P)$ is even/odd if i is even/odd, such that P reaches $j \in J(P)$ via an odd edge x_j^O and leaves j via an even edge x_j^E. We similarly use t_j^O and t_j^E to denote the type of x_j^O and x_j^E, respectively.

Next, we define *shift operations*, which move workload along paths by increasing the volume of one job at any position $i \in I(P)$ while decreasing the volume of another job at i.

▶ **Definition 9.** A δ*-shift* of a path P in x decreases the value of all odd edges (variables) of P by δ and increases the value of all even edges (variables) of P by δ.

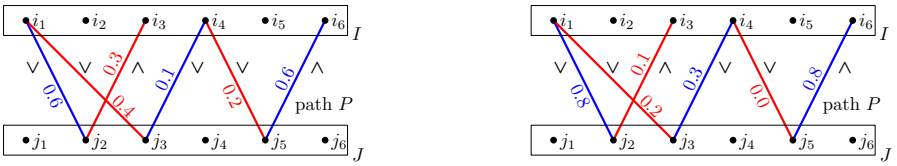
Shift operations (see Figure 1) change the budget use $B(x)$ at a path-dependent (positive or negative) *budget rate* (defined below) and might create crowded positions. Our algorithm's first two phases (Sections 4.3 and 4.4) carefully pair shift operations such that performing paired shifts does not increase the budget and does not create too many crowded positions.

Define the *budget rate* of a path P to be $\Delta(P) \coloneqq \sum_{j \in J(P)} c_j \cdot (\mathbb{1}_{|t_j^E = \vee} - \mathbb{1}_{|t_j^O = \vee})$. Let P be a path in a solution x for $LP_{\mathcal{I},\mathcal{F}}$ without crowded positions (i.e., $I^C(\mathcal{F}) = \varnothing$). P is called y*-alternating* (or simply *alternating*) if all odd edges have weight y and all even edges have weight $1 - y$. Lemma 10 below formalizes the effect of a δ-shift in terms of the path's budget rate. Since our analysis can be restricted to paths with very specific, alternating edge values, we also formalize such *alternating paths* and show how they are affected by δ-shifts (see also Figure 2).

▶ **Lemma 10.** *Let P be a path in a solution x for $LP_{\mathcal{I},\mathcal{F}}$ without crowded positions (i.e., $I^C(\mathcal{F}) = \varnothing$). Shifting P in x by δ yields a (possibly invalid) solution $\tilde{x}$ with $B(\tilde{x}) = B(x) - \delta \cdot \Delta(P)$. If P is y-alternating in x, then it is $(y - \delta)$-alternating in $\tilde{x}$.*

4.3 First Phase: Eliminating all but one cycle

Consider an optimal valid solution x to $LP_{\mathcal{I},\mathcal{F}}$ without crowded positions (i.e., $I^C(\mathcal{F}) = \varnothing$). Lemma 11 below is our main tool for rounding fractional variables in x. Consider a set of variables that form a fractional path in x. Essentially, we want to use a shift operation from Definition 9 on such a path to make some of its variables integral. If such a shift increases the budget use $B(x)$ (rendering the solution invalid), we can suitably shift a second path (possibly using a negative δ) in parallel to ensure that the budget use $B(x)$ does not increase.



(a) Path P with start and end positions i_3 and i_6. **(b)** Same path P after the shift.

Figure 1 A path P in a solution x before and after a shift by $\delta = 0.2$. Edges are labeled with their type ($\vee$ or $\wedge$) and weight from a given solution x. Odd edges are red, and even edges are blue. The budget rate computes as $\Delta(P) = c_{j_2}(1-0) + c_{j_3}(1-1) + c_{j_5}(0-1) = c_{j_2} - c_{j_5}$.

(a) Alternating paths P (solid) and P' (dashed). **(b)** Same paths by $\delta = 0.3$ and $\delta' = 0.6$, respectively.

Figure 2 Illustration of Lemma 11 for alternating paths $P = (i_3, \dots, i_4)$ (solid) and $P' = (i_4, \dots, i_6)$ (dashed) with budget rates $\Delta(P) = c_{j_3} \coloneqq 2$ and $\Delta(P') = c_{j_5} \coloneqq 1$. Shifting P by $\delta = 0.3$ and P' by $\delta' = 0.6$ keeps the budget use constant. P and P' stay alternating, P' becomes integral, and i_3, i_4, i_6 (the start/end positions) violate the position constraints after these shifts.

Such shifts might also cause the violation of the position constraints at the path's start and end positions. We keep track of such violations by adding those positions to the crowded position set $I^C(\mathcal{F})$ of the fixation $\mathcal{F}$. Lemma 11 formalizes this approach (see also Figure 2).

▶ **Lemma 11.** *Consider x a valid solution for $LP_{\mathcal{I},\mathcal{F}}$. Let P be a fractional path in x with $\Delta(P) = 0$ or P, P' be two fractional paths in x with $X(P) \neq X(P')$ and $\Delta(P), \Delta(P') \neq 0$. We can efficiently shift P (and P', if existing) in x to yield a valid solution $\tilde{x}$ for $LP_{\mathcal{I},\tilde{\mathcal{F}}}$ with:*

1. *$C(\tilde{x}) \leq C(x)$ and $B(\tilde{x}) = B(x)$*
2. *$\tilde{\mathcal{F}} = \mathcal{F} \cup I'$, where I' is the set of all start and end positions of non-cyclic paths involved.*
3. *$\tilde{x}$ contains more integral variables than x.*

Lemma 11 allows us to shift along general paths (instead of cycles) at the cost of creating crowded positions. We rely on this in Section 4.4 and deal with the crowded positions in Section 4.5. However, to keep the number of crowded positions small and reduce their impact on the final solution, we apply Lemma 11 on cycles for as long as possible. This avoids the creation of crowded positions since shifts along cycles cannot change the net workload at any position. Indeed, note that given any node in a path P that is incident to a fractional edge must have a second fractional edge, or it would violate its job/position constraint. This allows us to complete any fractional path to a cycle. Thus, we can keep applying Lemma 11 to *cycles* (not creating crowded positions) until there is at most one cycle with a non-zero budget rate left (a *blocking cycle*). Let x be a solution to $LP_{\mathcal{I},\mathcal{F}}$. A path P of x is called *critical* if $X(P) = \{\, x_{j,i,t} \in X_{\mathcal{I}} \mid x_{j,i,t} \in (0,1) \,\}$. A *blocking cycle* of x is a critical cycle P with $\Delta(P) \neq 0$. Lemma 12 formalizes the idea above.

▶ **Lemma 12.** *Let $\mathcal{I}$ be an instance, $\mathcal{F}$ be a fixation with $I^C(\mathcal{F}) = \varnothing$, and x be a valid optimal solution to $LP_{\mathcal{I},\mathcal{F}}$. Then we can compute in polynomial time a valid optimal solution $\tilde{x}$ to $LP_{\mathcal{I},\mathcal{F}}$ such that all variables in $\tilde{x}$ are integral, or we find a blocking cycle of $\tilde{x}$. Further, if P is a blocking cycle of x, then P is alternating.*

4.4 Second Phase: Rounding the blocking cycle

Assume that we used Lemma 12 to compute a blocking cycle P for a solution x to $LP_{\mathcal{I},\mathcal{F}}$ with $I^C(\mathcal{F}) = \varnothing$. Because P is critical, all fractional variables are in $X(P)$. Also, since $\Delta(P) \neq 0$, applying Lemma 11 directly is impossible. Instead, we cut up P repeatedly into two paths P_1, P_2, and then use Lemma 11 on these paths (see Algorithm 1). Cutting is done by selecting any position $i \in K(P)$, and separating the path at i: P_1 will be the path starting at the start position of P and end at i. P_2 will be the path starting at i and ending at the end position of P. We abbreviate this operation by $P_1, P_2 \leftarrow \textsc{Cut}(P, i)$. The drawback of this approach is that Lemma 11 does not guarantee that the resulting solutions still fulfill the position constraints of the start/end positions of P_1, P_2, respectively. That is why we add them to $I^C(\mathcal{F})$ in the process.

Algorithm RepeatedCut starts with a solution $\tilde{x}$ and a critical path $\tilde{P}$. It cuts $\tilde{P}$ at some position $i \in K(\tilde{P})$ that is selected by a procedure SelectCutPosition (which is described later in Algorithm 2 in the next subsection). The algorithm then applies Lemma 11 to the two resulting paths, making at least one of them integral (as guaranteed by Lemma 10). After that, $\tilde{x}$ and $\tilde{P}$ are updated accordingly. RepeatedCut finishes when $|K(\tilde{P})| = 0$ (and therefore $\tilde{P}$ cannot be cut into two paths anymore). In such a case, RepeatedCut will reschedule that job to obtain an integral solution. It is also possible that no path remains after the application of Lemma 11. For such a case, $\tilde{x}$ is already integral. Thus, in both cases, the resulting integral solution $\tilde{x}$ is returned.

Algorithm 1 RepeatedCut.

Input: A valid solution x for $LP_{\mathcal{I},\mathcal{F}}$ with $I^C(\mathcal{F}) = \varnothing$, a blocking cycle P of x.

$\tilde{P}, \tilde{x} \leftarrow P, x$
while $\Delta(\tilde{P}) \neq 0$ **do**
  if $K(\tilde{P}) = \varnothing$ **then**
    $j \leftarrow$ unique job in $J(\tilde{P})$; $i_1, i_2 \leftarrow$ remaining two positions in $I(\tilde{P})$
    In $\tilde{x}$, reschedule j into position $\min(i_1, i_2)$ of type $\vee$ if $t_j^O = t_j^E = \vee$ and $\wedge$ else
    return $\tilde{x}$
  $i \leftarrow \textsc{SelectCutPosition}(\tilde{P})$
  $P_1, P_2 \leftarrow \textsc{Cut}(P, i)$
  Apply Lemma 11 to P_1, P_2 in $\tilde{x}$, changing $\tilde{x}$ accordingly
  if both paths became integral **then return** $\tilde{x}$
  $\tilde{P} \leftarrow$ the remaining fractional path
Apply Lemma 11 to $\tilde{P}$ in $\tilde{x}$, changing $\tilde{x}$ accordingly
return $\tilde{x}$

In the following, we make statements about the state of the variables involved in the execution of RepeatedCut at the beginning of an iteration of its **while**-loop. Consider the state of RepeatedCut (called on path P) at the beginning of the l'th iteration of the **while**-loop ($l \geq 1$). We denote by I_l^C the start/end position of P together with all positions selected by SelectCutPosition so far, and I_*^C the start/end position of P together with all positions selected by SelectCutPosition throughout the algorithm. Similarly, denote by $\tilde{x}_l$,$\tilde{P}_l$ the values of $\tilde{x}$,$\tilde{P}$ at that point, respectively, and $\tilde{x}_*$ for the returned solution by RepeatedCut. Denote $\tilde{\mathcal{F}}_l := (J(\mathcal{F}), X(\mathcal{F}), I_l^C)$ and $\tilde{\mathcal{F}}_* := (J(\mathcal{F}), X(\mathcal{F}), I_*^C)$.

▶ **Lemma 13.** *The following is a loop invariant of RepeatedCut for iteration $l \geq 1$: $\tilde{x}_l$ is a valid solution for $LP_{\mathcal{I},\tilde{\mathcal{F}}_l}$ and $\tilde{P}_l$ is a critical fractional alternating path in $\tilde{x}_l$, of which the start and end positions are in I_l^C. Also, $\tilde{x}_*$ is an integral valid solution for $LP_{\mathcal{I},\tilde{\mathcal{F}}_*}$.*

Based on Lemma 13, we can analyze the objective obtained by RepeatedCut:

▶ **Lemma 14.** *Consider an application of* RepeatedCut *on solution x for $LP_{\mathcal{I},\mathcal{F}}$ and a blocking cycle P. It returns in polynomial time a solution $\tilde{x}$ with $C(\tilde{x}) \leq C(x) + Z$, where Z is either 0 or the contribution of job j rescheduled by* RepeatedCut *in line 5 and $j \notin J(\tilde{\mathcal{F}}_*)$.*

4.5 Third Phase: Dealing with crowded positions

Lemma 14 guarantees that applying the algorithm RepeatedCut will return us an integral solution. However, that solution is valid for $LP_{\mathcal{I},\mathcal{F}}$ where $I^C(\mathcal{F})$ still contains some positions. Some of these positions may schedule two jobs, which makes this schedule not valid for $ILP_{\mathcal{I}}$. Our general strategy in this subsection is to move the jobs such that the cost of the solution does not increase too much. In Observation 15, we move each job to a new position and bound the cost created by that operation.

▶ Observation 15. Let x be an integral solution to $LP_{\mathcal{I},\mathcal{F}}$ for some fixation $\mathcal{F}$. Consider a job $j \in J$ that is scheduled in position $i \in I$ of type $t \in T$. Then rescheduling j into position i', i.e., setting $x_{j,i,t} \leftarrow 0$ and $x_{j,i',t} \leftarrow 1$ produces a (possibly invalid) solution $\tilde{x}$, in which the contribution of j increases by a factor of i'/i compared to x.

As mentioned above, there are still some positions that schedule two jobs. To obtain an integral valid solution for $ILP_{\mathcal{I}}$, we have to move the jobs in the schedule to new positions, such that there is exactly one job per position scheduled. We want to use Observation 15 to bound the increase in contribution for each job moved this way. Generally, we move the jobs as follows: For a position i that schedules two jobs j, j', we (arbitrarily) distribute j, j' among positions $i, i+1$, thereby moving all jobs from positions $i+1, \ldots, n$ to one higher position. Following this strategy, jobs in higher positions may get moved multiple times.

To bound the contribution in terms of Observation 15, we set up a charging scheme: Each position with two jobs scheduled should be charged to a distinct set of $1/\epsilon$ smaller positions that schedule one job, where ϵ is the accuracy parameter of our algorithm ($1/\epsilon \in \mathbb{N}$). In the following, we will always use the following function SelectCutPosition for RepeatedCut:

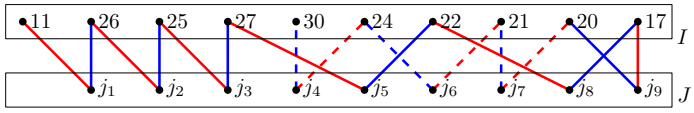
Algorithm 2 SelectCutPosition.

Input: A path $\tilde{P}$ with $|K(\tilde{P})| \geq 1$
Output: A position $i \in K(\tilde{P})$
1: **if** $|K(\tilde{P})| \geq 2/\epsilon + 1$ **then**
2: $I' \leftarrow$ the smallest $2/\epsilon + 1$ positions in $K(\tilde{P})$
3: **return** the position that appears as $(1/\epsilon + 1)$-st position in $\tilde{P}$ of the positions in I'
4: **else**
5: **return** any position in $K(\tilde{P})$

We care about two properties of the position selected by SelectCutPosition. First, when we cut $\tilde{P}$ into P_1, P_2 in line 8 of RepeatedCut, $K(P_1)$,$K(P_2)$ should each contain at least $1/\epsilon$ positions. This way, whichever of these paths becomes integral, there will be $1/\epsilon$ positions that will never be selected by SelectCutPosition in the future. This is important for our charging scheme to have enough positions to charge to. Second, we specifically care about the selected positions being the smallest positions that appear in $K(\tilde{P})$. This essentially allows us to charge each position with two jobs scheduled exclusively to smaller positions, independent of which of the two paths becomes integral.

We represent the charging scheme using a *charging function* (formally defined in Definition 16). Essentially, for a set of positions $\bar{I} \subseteq I$, it charges each position in $\bar{I}$ to its distinct $1/\epsilon$ many smaller positions.

(a) Illustration of μ_l (e.g., $\mu_l(24) = \{16, 23\}$). **(b)** Illustration of μ_{l+1} (e.g., $\mu_{l+1}(26) = \{16, 23\}$).



(c) Path $\tilde{P}_l$ (solid *and* dashed), and path $\tilde{P}_{l+1}$ (solid) after one iteration of the **while**-loop in RepeatedCut.

Figure 3 An update step of Lemma 17 for $\epsilon = 1/2$. The inner positions of $\tilde{P}_l$ are $K(\tilde{P}_l) = \{17, 20, 21, 22, 24, 25, 26, 27\}$. Its $2/\epsilon + 1 = 5$ smallest positions appear in order $22, 17, 20, 21, 24$. $\tilde{P}_l$ is cut at the $(1/\epsilon + 1) = 3$-rd of these positions (20) into two paths, which are then shifted such that the dashed one becomes integral and the solid one becomes $\tilde{P}_{l+1}$. I^+_{l+1} loses positions 21 and 24 compared to I^+_l, as these positions belonged to the dashed path, which became integral. I^+_{l+1} now consists of all remaining $2/\epsilon + 1 = 5$ inner positions $K(\tilde{P}_{l+1}) = \{17, 22, 25, 26, 27\}$. We set $\mu_{l+1}(25) = \mu_l(21)$, $\mu_{l+1}(26) = \mu_l(24)$, and $\mu_{l+1}(27) = \{21, 24\}$ (the lost positions from I^+_l).

▶ **Definition 16.** Let x be a solution to $LP_{\mathcal{I},\mathcal{F}}$ for an instance $\mathcal{I}$ and a fixation $\mathcal{F}$. Let P be a critical path in x. For a set $\bar{I} \subseteq I$, a *charging function* for $\bar{I}$ is a function $\mu : \bar{I} \to \mathcal{P}(I \setminus \bar{I})$ such that for all $i \in \bar{I}$: (1) $|\mu(i)| = 1/\epsilon$, (2) $\forall i' \in \mu(i) : i' < i$ and (3) $\forall i' \in \bar{I} : \mu(i) \cap \mu(i') = \varnothing$.

Consider the l'th iteration of the **while**-loop in RepeatedCut. We define the *charging set* $I^+_l := \bar{I} \cup I^C_l$, where $\bar{I}$ contains the smallest $2/\epsilon + 1$ positions in $K(\tilde{P}_l)$ (or all of them, if $|K(\tilde{P}_l)| < 2/\epsilon + 1$). Similarly, define $I^+_* := I^C_*$.

Lemma 17 shows how to obtain a charging function $\mu_* : I^+_* \to \mathcal{P}(I \setminus I^+_*)$ from a charging function $\mu_1 : I^+_1 \to \mathcal{P}(I \setminus I^+_1)$. We do this by updating the charging function with every iteration of RepeatedCut's loop. Figure 3 exemplifies the update of the charging function.

▶ **Lemma 17.** *If there exists a charging function $\mu_1 : I^+_1 \to \mathcal{P}(I \setminus I^+_1)$, then there also exists a charging function $\mu_* : I^+_* \to \mathcal{P}(I \setminus I^+_*)$.*

We now use Observation 15 together with a charging function (of which we assume the existence for now) on a solution x for $LP_{\mathcal{I},\mathcal{F}}$ produced by RepeatedCut to find a solution for $ILP_{\mathcal{I}}$ with not too much more cost. Lemma 18 will allow us to produce such a solution.

▶ **Lemma 18.** *Let x be a solution returned by RepeatedCut for $LP_{\mathcal{I},\mathcal{F}}$, and let μ_* be a charging function for I^+_*. Then we can find a valid solution $\tilde{x}$ in polynomial time for $ILP_{\mathcal{I}}$ such that the contribution of each job increases by a factor of at most $(1 + \epsilon)$ compared to x.*

Consider a solution x returned by RepeatedCut for $LP_{\mathcal{I},\mathcal{F}}$. To be able to apply Lemma 18 and find a solution for $ILP_{\mathcal{I}}$, we need to make sure that we can find a charging function μ_1 for I^+_1. Furthermore, we still need to bound the contribution created by the job j in line 5 of RepeatedCut as of Lemma 14. To do this, we choose a proper fixation $\mathcal{F}^*$ and show that a charging function can then be derived.

▶ **Definition 19.** Let x^* be an optimal solution to $ILP_{\mathcal{I}}$ for an instance $\mathcal{I}$. For $i \in I$, let further $j_i \in J$ and $t_i \in T$ such that $x^*_{j_i,i,t_i} = 1$. Using $M := (2/\epsilon + 1)/\epsilon$, we define the fixation $\mathcal{F}^*$ by

$$X(\mathcal{F}^*) = \{ x_{j_i,i,t_i} \mid i \in [M] \} \qquad J(\mathcal{F}^*) = \{ j \in J \mid p^\wedge_j > \min_{i \in [M]} p^{t_i}_{j_i} \} \qquad I^C(\mathcal{F}^*) = \varnothing$$

▶ **Lemma 20.** *Let x be a solution returned by RepeatedCut for $LP_{\mathcal{I},\mathcal{F}^*}$. Then there exists a charging function μ_1 for I^+_1.*

Essentially, we brute-force which jobs will be scheduled in the last few positions. This will make sure that these positions are not in I_1^+, and as such can be used for the charging function μ_1. Assuming that we brute-forced correctly, an optimal solution will also test all jobs with a larger upper processing time than any of the brute-forced jobs. This is because an optimal solution will always schedule in order of increasing processing times. Finally, we can piece together all of the above lemmas and prove the main theorem (Theorem 4). The detailed proof can be found in [8].

5 Conclusion

We initiated the study of Scheduling with a Limited Testing Budget, where we have a limited budget for testing jobs to potentially decrease their processing time. We provided NP-hardness results, a PTAS, as well as tight bounds for a semi-online (oblivious) setting.

Our results open promising avenues for future research. For the setting where we minimize the total completion time, it remains open whether NP-hardness holds for uniform testing cost. Also, while our LP-rounding-based PTAS achieves the best possible approximation, it remains open whether there is a faster, combinatorial algorithm. Another natural direction would be to consider the case of multiple machines.

Another exciting direction is the following *bipartite matching with testing* problem that generalizes our problem, arising from the graph-theoretic perspective in Section 4.2: Consider a bipartite graph $G \coloneqq (L \cup R, E)$ in which each edge $e \in E$ has a cost c_e that can be reduced to $\check{c}_e$ via a testing operation. Given the possibility to test edges before adding them to the matching, we seek a min-cost perfect matching that respects a given testing budget.

References

1 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows – Theory, algorithms and applications*. Prentice Hall, 1993.

2 Susanne Albers and Alexander Eckl. Explorable uncertainty in scheduling with non-uniform testing times. In *WAOA*, volume 12806 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 2020.

3 Susanne Albers and Alexander Eckl. Scheduling with testing on multiple identical parallel machines. In *WADS*, volume 12808 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2021.

4 Richard Bruce, Michael Hoffmann, Danny Krizanc, and Rajeev Raman. Efficient update strategies for geometric computing with uncertainty. *Theory Comput. Syst.*, 38(4):411–423, 2005.

5 J. Bruno, E.G. Coffman Jr., and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Comm. ACM*, 17:382–387, 1974.

6 João Manuel Paiva Cardoso, José Gabriel de Figueired Coutinho, and Pedro C Diniz. *Embedded computing for high performance: Efficient mapping of computations using customization, code transformations and compilation*. Morgan Kaufmann, 2017.

7 Qingyun Chen, Sungjin Im, Benjamin Moseley, Chenyang Xu, and Ruilong Zhang. Min-max submodular ranking for multiple agents. *CoRR*, abs/2212.07682, 2022. `arXiv:2212.07682`.

8 Christoph Damerius, Peter Kling, Minming Li, Chenyang Xu, and Ruilong Zhang. Scheduling with a limited testing budget, 2023. `arXiv:2306.15597`.

9 J. Du and J.Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Discrete Math.*, 2(4):473–487, 1989.

10 Christoph Dürr, Thomas Erlebach, Nicole Megow, and Julie Meißner. An adversarial model for scheduling with testing. *Algorithmica*, 82(12):3630–3675, 2020.

11 Evangelia Gergatsouli and Christos Tzamos. Online learning for min sum set cover and pandora's box. In *ICML*, volume 162 of *Proceedings of Machine Learning Research*, pages 7382–7403. PMLR, 2022.

12 Marc Goerigk, Manoj Gupta, Jonas Ide, Anita Schöbel, and Sandeep Sen. The robust knapsack problem with queries. *Comput. Oper. Res.*, 55:12–22, 2015.

13 Mingyang Gong, Randy Goebel, Guohui Lin, and Eiji Miyano. Improved approximation algorithms for non-preemptive multiprocessor scheduling with testing. *Journal of Combinatorial Optimization*, 44(1):877–893, 2022.

14 Anupam Gupta, Amit Kumar, Viswanath Nagarajan, and Xiangkun Shen. Stochastic load balancing on unrelated machines. *Math. Oper. Res.*, 46(1):115–133, 2021.

15 Anupam Gupta, Amit Kumar, Viswanath Nagarajan, and Xiangkun Shen. Stochastic makespan minimization in structured set systems. *Math. Program.*, 192(1):597–630, 2022.

16 Anupam Gupta, Benjamin Moseley, and Rudy Zhou. Minimizing completion times for stochastic jobs via batched free times. *CoRR*, abs/2208.13696, 2022. `arXiv:2208.13696`.

17 Anupam Gupta and Viswanath Nagarajan. A stochastic probing problem with applications. In *IPCO*, volume 7801 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 2013.

18 Varun Gupta, Benjamin Moseley, Marc Uetz, and Qiaomin Xie. Greed works - online algorithms for unrelated machine stochastic scheduling. *Math. Oper. Res.*, 45(2):497–516, 2020.

19 Leslie A. Hall, Andreas S. Schulz, David B. Shmoys, and Joel Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22(3):513—-544, 1997.

20 Michael Hoffmann, Thomas Erlebach, Danny Krizanc, Matús Mihalák, and Rajeev Raman. Computing minimum spanning trees with uncertainty. In *STACS*, volume 1 of *LIPIcs*, pages 277–288. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2008.

21 Simon Kahan. A model for data in motion. In *STOC*, pages 267–277. ACM, 1991.

22 R.M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Plenum, 1972.

23 Adam Kasperski and Pawel Zielinski. On the approximability of robust spanning tree problems. *Theor. Comput. Sci.*, 412(4-5):365–374, 2011.

24 Adam Kasperski and Paweł Zieliński. Robust discrete optimization under discrete and interval uncertainty: A survey. *Robustness analysis in decision aiding, optimization, and analytics*, pages 113–143, 2016.

25 Parul Kudtarkar, Todd F DeLuca, Vincent A Fusaro, Peter J Tonellato, and Dennis P Wall. Cost-effective cloud computing: a case study using the comparative genomics tool, roundup. *Evolutionary Bioinformatics*, 6:EBO–S6259, 2010.

26 Nicole Megow, Julie Meißner, and Martin Skutella. Randomization helps computing a minimum spanning tree under uncertainty. *SIAM J. Comput.*, 46(4):1217–1240, 2017.

27 Chris Olston and Jennifer Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *VLDB*, pages 144–155. Morgan Kaufmann, 2000.

28 Martin Weitzman. *Optimal search for the best alternative*, volume 78(8). Department of Energy, 1978.

29 Gang Yu and Panagiotis Kouvelis. Complexity results for a class of min-max problems with robust optimization applications. In *Complexity in numerical optimization*, pages 501–511. World Scientific, 1993.