The Art of Staying Ahead of Deadlines: Improved Algorithms for the Minimum Tardy Processing Time

Mihail Stoian*

Abstract

We study the fundamental scheduling problem $1\|\sum p_j U_j$. Given a set of n jobs with processing times p_j and deadlines d_j, the problem is to select a subset of jobs such that the total processing time is maximized without violating the deadlines. In the midst of a flourishing line of research, Fischer and Wennmann have recently devised the sought-after $\widetilde{O}(P)$-time algorithm, where $P = \sum p_j$ is the total processing time of all jobs. This running time is optimal as it matches conditional lower bounds based on popular conjectures.

However, P is not the sole parameter one could parameterize the running time by. Indeed, they explicitly leave open the question of whether a running time of $\widetilde{O}(n + \max d_j)$ or even $\widetilde{O}(n + \max p_j)$ is possible. In this work, we show, somewhat surprisingly, that by a refined implementation of their original algorithm, one can obtain the asked-for $\widetilde{O}(n + \max d_j)$-time algorithm.

*University of Technology Nuremberg, Germany, mihail.stoian@utn.de

1 Introduction

The $1\|\sum p_jU_j$ problem is a classical NP-hard problem that generalizes the well-known Subset Sum problem. Given n jobs with processing times p_j and deadlines d_j, the problem is to select a subset of jobs such that the total processing time is maximized under the constraint that their deadlines are observed. Its history goes back to the 55-year old algorithm by Lawler and Moore [9], whose running time reads $O(nP)$, where $P = \sum p_j$.

Multiple Machines. The problem also appears in the multiple-machine flavor as the so-called $P_m\|\sum p_jU_j$ problem: In this setting, there are m machines among which the jobs can be distributed; the task remains the same as before. For simplicity, we assume that m is a constant.

Related Work. Interestingly, the running time of Lawler and Mooore's algorithm remained the state-of-the-art for quite some time: Only recently did Bringmann, Fischer, Hermelin, Shabtay and Wellnitz [2] manage to lower it to $\widetilde{O}(P^{7/4})$ by introducing, as a by-product, a novel convolution – the *skewed convolution*.[1] Following their work, a line of research started to improve either the algorithmic reduction or the skewed convolution itself, resulting in a $\widetilde{O}(P^{7/5})$-time algorithm [8, 11]. Recently, Fischer and Wennmann [5] showed that the skewed convolution can be forgotten altogether: Indeed, they provide a randomized algorithm in time $O(P \log P)$ and, alternatively, a deterministic one in time $O(P \log^{1+o(1)} P)$. In particular, their algorithm is optimal (up to lower-order factors) under the Strong Exponential Time Hypothesis or the Set Cover Hypothesis.

Similar to the Subset Sum problem, our problem also has other parameters that can be used to parametrize the running time by. Prominent examples are $d_{max} = \max d_j$ and $p_{max} = \max p_j$. Prior research also introduced algorithm w.r.t. to these parameters: an $\widetilde{O}(\min\{P \cdot D_{\#}, P + D\})$-time algorithm [2], where $\#D$ is the number of distinct deadlines and D is the sum of all distinct deadlines, and an $\widetilde{O}(n + p_{max}^3)$-time algorithm [8]. Fischer and Wennmann explicitly leave open the question of whether a time of $\widetilde{O}(n + d_{max})$ or even $\widetilde{O}(n + p_{max})$ is possible [5].

Our Results. We show that an $\widetilde{O}(n+d_{max})$-time algorithm does indeed exist. Perhaps surprisingly, our algorithm is a simple refined implementation of Fischer and Wennmann's. Our analysis also extends to the multi-machine setting, resulting in a $\widetilde{O}(n + d_{max}^m)$-time algorithm.

2 Preliminaries

Notation. We use $[n] := \{1, \dots, n\}$ (note that Fischer and Wennmann's notation is 0-based [5]). The interval $[i..j]$ denotes $\{i, \dots, j\}$, while $[i..j)$ denotes $[i, j]\backslash\{j\}$; analogously, define $(i..j]$. Furthermore, we use the standard sumset notation $A+b = \{a+b \mid a \in A\}$ and $A+B = \{a+b \mid a \in A, b \in B\}$.

Earliest-Deadline-First. We employ the key observation due to Lawler and Moore [9] that w.l.o.g. the *early*, i.e., non-tardy, jobs may be scheduled in the sorted order of their deadlines. Thus, the problem reduces to computing a subset of jobs $J^* \subseteq [n]$ that maximizes $\sum_{j \in J^*} p_j$ and observes the deadlines of the jobs within J^*, i.e., $\sum_{i \in J^*: i \le j} p_i \le d_j$ for all $j \in J$.

Lawler and Moore's Algorithm. The algorithm due to Lawler and Moore is a simple dynamic program. Namely, one recursively defines the sets $S_0, \dots, S_n \subseteq [0..P]$ as in the following:

[1] The term $\widetilde{O}(P) = P(\log P)^{O(1)}$ suppresses polylogarithmic factors.

Algorithm 1 Lawler and Moore's algorithm

$$S_0 = \{0\}, \tag{1a}$$
$$S'_j = S_{j-1} + \{0, p_j\}, \tag{1b}$$
$$S_j = S'_j \cap [0..d_j], \tag{1c}$$

where $j \in [n]$.

Note that we follow the two-step construction of S_j proposed by Fischer and Wennmann [5], since it allows for a modular implementation. The optimal value can be then read off as $\max S_n$.

Fischer and Wennmann's Algorithm. The algorithm proposed by Fischer and Wennmann [5] uses the dynamic string data structure [6, 7], recently employed in modular subset sum [1, 3, 10]. Their key observation is that one can use the same data structure to support the deletions in Eq. (3c). In particular, one has to interpret the sets S_j as indicator strings of size P.

Our Approach. However, observe that in Eq. (3c), we anyway *trim* the current set $S_j + \{0, p_j\}$ by its processing times greater than d_j. Hence, if we had a mechanism to only add *valid* processing times, i.e., l.e.q. d_j, we could interpret the sets S_j indeed as indicator strings of size $d_{max} + 1$.[2]

3 A Refined Algorithm

First, observe that, by construction, the following invariant holds: $S_{j-1} \subseteq [0..d_{j-1}]$. The new algorithm works as follows. Instead of directly performing Eq. (3b) as is, we first split S_j into two parts w.r.t. to $d_j - p_j$, i.e., we obtain $L_{j-1} := S_{j-1} \cap [0..d_j - p_j]$ and $R_{j-1} := S_{j-1} \cap (d_j - p_j..d_j]$. We will see later how L_{j-1} and R_{j-1} can be computed via the dynamic string data structure. We then compute $\widetilde{S}_j = L_{j-1} + p_j$. Finally, we need to merge $\widetilde{S}_j$ with both L_{j-1} and R_{j-1} to obtain the final S_j. Better said, we are left with merging $\widetilde{S}_j$ with S_{j-1} to get $S_j = \widetilde{S}_j \cup S_{j-1}$.

Algorithm 2 Refined algorithm

$$S_0 = \{0\}, \tag{2a}$$
$$L_{j-1} = S_{j-1} \cap [0..d_j - p_j], \tag{2b}$$
$$\widetilde{S}_j = L_{j-1} + p_j, \tag{2c}$$
$$S_j = \widetilde{S}_j \cup S_{j-1}, \tag{2d}$$

where $j \in [n]$.

In the following, we outline the implementation details.

Leveraging Dynamic Strings. The dynamic string data structure is an important toolbox in Fischer and Wennmann's algorithm [5]. In the sequel, we use their presentation of the data structure and outline only the operations we will need:

[2] The offset of "+1" is necessary to account for deadlines of value 0.

Lemma 3.1 (Dynamic String Data Structure [6, 7])**.** *The dynamic string data structure maintains a dynamic collection X of non-empty strings, supporting the following operations:*

- `make_string`(x)*: Given any string $x \in \Sigma^+$, insert x into X.*
- `concat`(x_1, x_2)*: Given $x_1, x_2 \in X$, insert the concatenated string x_1x_2 into X.*
- `split`(x, i)*: Given $x \in X$ and $i \in [0..|x|)$, insert $x[0..i]$ and $x(i..|x|]$ into X.*

Let s be the maximum of the total length of all strings and the number of executed operations. Then all operations run in randomized time $O(\log s)$ or in deterministic time $O(\log^{1+o(1)} s)$, except for `make_string` *which takes time $O(|x| + \log s)$ and $O(|x| \cdot \log^{o(1)} s)$, respectively.*

Let u be the string size (this will be set to $d_{max} + 1$ later). We represent a set $S \subseteq [n]$ by its corresponding indicator string $x_S \in \{0, 1\}^u$, i.e., $i \in S$ if and only if $x_S[i] = 1$.

To initialize the data structure, we build 0^u by inserting the word $w = 0$ and self-concatenating it $\log u$ times. This takes time $O(\log^2 u)$.

Then, to obtain, L_{j-1}, we split $x_{S_{j-1}}$ at position $d_j - p_j$. Afterward, we can compute $\widetilde{S}_j$ by shifting $x_{L_{j-1}}$ by p_j positions. Namely, we perform $\texttt{concat}(0^{p_j}, x_{L_{j-1}})$, where the string 0^{p_j} can be obtained by splitting off the above pre-computed 0^u via $\texttt{split}(0^u, p_j)$. We finally trim the resulting string to length u via $\texttt{split}(0^{p_j}x_{L_{j-1}}, u)$. Merging $\widetilde{S}_j$ with S_{j-1} is now a bit-wise OR of $x_{\widetilde{S}_j}$ and $x_{S_{j-1}}$ and can be implemented in the same way as by Fischer and Wennmann [5].

Running Time Analysis. Our running time analysis closely follows that by Fischer and Wennmann [5], with the key difference that, using our new implementation, we manage to bound the number of insertions by $d_{max} + 1$ instead of $2P + 1$:

Lemma 3.2. *It holds that* $\sum_{i \in [n]} |\widetilde{S}_j \setminus S_{j-1}| \leq d_{max} + 1$.

Proof. A benefit of the new algorithm is that, compared to Ref. [5], we do not need to split w.r.t. d_j since the set S_j is, *by construction*, already filtered out of processing times greater than d_j. Hence, for any $x \in [0..d_{max}]$, once $x \in \widetilde{S}_j \setminus S_{j-1}$, then $x \in S_j, \ldots, S_n$ and, hence, we have

$$|\{i \in [n] \mid x \in \widetilde{S}_j \setminus S_{j-1}\}| \leq 1.$$

With this observation, the cumulative number of insertions reads

$$\sum_{i \in [n]} |\widetilde{S}_j \setminus S_{j-1}| = \sum_{x \in [0..d_{max}]} |\{i \in [n] \mid x \in \widetilde{S}_j \setminus S_{j-1}\}| \leq d_{max} + 1.$$

□

Using the dynamic string data structure with string-size $u = d_{max}+1$, we obtain an $\widetilde{O}(n+d_{max})$-time algorithm.

Retrieving the Schedule. We have shown that we can compute the optimal value in time $\widetilde{O}(n + d_{max})$ by a refined implementation of the original algorithm. Obtaining the optimal schedule can be, indeed, done in a similar way, since the sets themselves are after each iteration the same as in Fischer and Wennmann's algorithm. We skip the details and refer the reader to Ref. [5, Sec. 3.2].

4 Generalizing to Multiple Machines

Our refined analysis can be extended to the setting of m machines. In that context, the original dynamic program slightly changes, but the core idea remains the same. In particular, we can define similar sets $S_0, \ldots, S_n$ which are this time subsets of $[0..P]^m$, i.e., the i-th entry of each vector corresponds to the i-th machine:

Algorithm 3 Lawler and Moore's Algorithm for m machines

$$S_0 = \{0\}, \tag{3a}$$
$$S'_j = S_{j-1} + \{0, p_j \cdot e_1, \ldots, p_j \cdot e_m\}, \tag{3b}$$
$$S_j = S'_j \cap [0..d_j]^m, \tag{3c}$$

for each $j \in [n]$ and $e_1, \ldots, e_m$ are the standard unit vectors in $\mathbb{Z}^m$.

A feasible schedule of total processing time t exists if and only if there exists $s \in S_n$ such that $s_1 + \ldots + s_m = t$ [9]. It is now easy to see that we can apply our refined algorithm for the single-machine setting to this setting as well. The key idea is to apply our trimming strategy point-wise, for each of the m (independent) entries of the vectors.

Algorithm 4 Refined algorithm for m machines

$$S_0 = \{0\}, \tag{4a}$$
$$L_{j-1} = S_{j-1} \cap [0..d_j - p_j]^m, \tag{4b}$$
$$\widetilde{S}_j = L_{j-1} + \{0, p_j \cdot e_1, \ldots, p_j \cdot e_m\}, \tag{4c}$$
$$S_j = \widetilde{S}_j \cup S_{j-1}, \tag{4d}$$

where $j \in [n]$.

We can similarly bound the number of insertions:

Lemma 4.1. *It holds that* $\sum_{j\in[n]} |\widetilde{S}_j \setminus S_{j-1}| \leq (d_{max} + 1)^m$.

Proof. We closely follow our proof for Lemma 3.2. Namely, we make the observation that for any $x \in [0..d_{max}]^m$, once $x \in \widetilde{S}_j \setminus S_{j-1}$, then $x \in S_j, \ldots, S_n$. Hence, we have

$$|\{j \in [n] \mid x \in \widetilde{S}_j \setminus S_{j-1}\}| \leq 1.$$

Therefore, we can bound the total number of insertions by

$$\sum_{j\in[n]} |\widetilde{S}_j \setminus S_{j-1}| = \sum_{x\in[0..d_{max}]^m} |\{j \in [n] \mid x \in \widetilde{S}_j \setminus S_{j-1}\}| \leq (d_{max} + 1)^m.$$

□

The rest of the algorithm, in particular the adapted multi-dimensional dynamic string data structure, remains the same as in Fischer and Wennmann [5].

5 Open Problems

We settled an open question explicitly asked by Fischer and Wennmann [5]. Somewhat surprisingly, we obtained it by slightly modifying their original algorithm, resulting in a $\widetilde{O}(n + d_{max})$-time algorithm. A pressing open question is whether the problem can be solved in $\widetilde{O}(n + p_{max})$-time. Indeed, it would be (again) surprising if the same algorithm could be adapted to this setting. However, such an algorithm would have a wider implication, since it would also represent a break-through for Subset Sum with small items. In spite of that, the recent break-through by Chen, Lian, Mao and Zhang [4], namely an $\widetilde{O}(n + \sqrt{wt})$-time algorithm for the Subset Sum problem, shows that a similar running time for the $1\|\sum p_j U_j$ problem may not be completely out of reach.

References

[1] Kyriakos Axiotis, Arturs Backurs, Karl Bringmann, Ce Jin, Vasileios Nakos, Christos Tzamos, and Hongxun Wu. *Fast and Simple Modular Subset Sum*, pages 57–67. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976496.6, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976496.6, doi:10.1137/1.9781611976496.6. 2

[2] Karl Bringmann, Nick Fischer, Danny Hermelin, Dvir Shabtay, and Philip Wellnitz. Faster minimization of tardy processing time on a single machine. *Algorithmica*, 84(5):1341–1356, may 2022. doi:10.1007/s00453-022-00928-w. 1

[3] Jean Cardinal and John Iacono. *Modular Subset Sum, Dynamic Strings, and Zero-Sum Sets*, pages 45–56. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976496.5, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976496.5, doi:10.1137/1.9781611976496.5. 2

[4] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. An improved pseudopolynomial time algorithm for subset sum, 2024. arXiv:2402.14493. 5

[5] Nick Fischer and Leo Wennmann. Minimizing tardy processing time on a single machine in near-linear time, 2024. arXiv:2402.13357. 1, 2, 3, 4, 5

[6] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. *Optimal Dynamic Strings*, pages 1509–1528. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975031.99, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611975031.99, doi:10.1137/1.9781611975031.99. 2, 3

[7] Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 1657–1670, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519935.3520061. 2, 3

[8] Kim-Manuel Klein, Adam Polak, and Lars Rohwedder. *On Minimizing Tardy Processing Time, Max-Min Skewed Convolution, and Triangular Structured ILPs*, pages 2947–2960. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611977554.ch112, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611977554.ch112, doi:10.1137/1.9781611977554.ch112. 1

[9] E. L. Lawler and J. M. Moore. A functional equation and its application to resource allocation and sequencing problems. *Management Science*, 16(1):77–84, 1969. URL: http://www.jstor.org/stable/2628367. 1, 4

[10] Krzysztof Potepa. Faster deterministic modular subset sum. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 76:1–76:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.ESA.2021.76, doi:10.4230/LIPICS.ESA.2021.76. 2

[11] Baruch Schieber and Pranav Sitaraman. Quick minimization of tardy processing time on a single machine. In *Algorithms and Data Structures: 18th International Symposium, WADS 2023, Montreal, QC, Canada, July 31 – August 2, 2023, Proceedings*, page 637–643, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-38906-1_42. 1