Faster Minimization of Tardy Processing Time on a Single Machine

Karl Bringmann[1,2] · Nick Fischer[1,2] · Danny Hermelin[3] · Dvir Shabtay[3] · Philip Wellnitz[2]

Abstract

This paper is concerned with the $1||\sum p_j U_j$ problem, the problem of minimizing the total processing time of tardy jobs on a single machine. This is not only a fundamental scheduling problem, but also an important problem from a theoretical point of view as it generalizes the Subset Sum problem and is closely related to the 0/1-Knapsack problem. The problem is well-known to be NP-hard, but only in a weak sense, meaning it admits pseudo-polynomial time algorithms. The best known running time follows from the famous Lawler and Moore algorithm that solves a more general weighted version in $O(P \cdot n)$ time, where P is the total processing time of all n jobs in the input. This algorithm has been developed in the late 60s, and has yet to be improved to date. In this paper we develop two new algorithms for problem, each improving on Lawler and Moore's algorithm in a different scenario.

- Our first algorithm runs in $\tilde{O}(P^{7/4})$ time, and outperforms Lawler and Moore's algorithm in instances where $n = \tilde{\omega}(P^{3/4})$.
- Our second algorithm runs in $\tilde{O}(\min\{P \cdot D_\#, P + D\})$ time, where $D_\#$ is the number of *different* due dates in the instance, and D is the sum of all *different* due dates. This algorithm improves on Lawler and Moore's algorithm when $n = \tilde{\omega}(D_\#)$ or

✉ Dvir Shabtay
  dvirs@bgu.ac.il

[1] Saarland University, Saarland Informatics Campus (SIC), Saarbrücken, Germany

[2] Max Planck Institute for Informatics, Saarland Informatics Campus (SIC), Saarbrücken, Germany

[3] Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva, Israel

$n = \tilde{\omega}(D/P)$. Further, it extends the known $\tilde{O}(P)$ algorithm for the single due date special case of $1||\sum p_j U_j$ in a natural way.

Both algorithms rely on basic primitive operations between sets of integers and vectors of integers for the speedup in their running times. The second algorithm relies on fast polynomial multiplication as its main engine, and can be easily extended to the case of a fixed number of machines. For the first algorithm we define a new "skewed" version of (max, min)-Convolution which is interesting in its own right.

Keywords Single machine scheduling · Tardy processing time · Pseudo-polynomial time algorithm · Fast polynomial multiplication · (max, min)-Convolution

1 Introduction

In this paper we consider the problem of minimizing the total processing times of tardy jobs on a single machine. In this problem we are given a set of n jobs $J = \{1, \ldots, n\}$, where each job j has a *processing time* $p_j \in \mathbb{N}$ and a *due date* $d_j \in \mathbb{N}$. A *schedule* σ for J is a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$. In a given schedule σ, the *completion time C_j* of a job j under σ is given by $C_j = \sum_{\sigma(i) \leq \sigma(j)} p_i$, that is, the total processing time of jobs preceding j in σ (including j itself). Job j is *tardy* in σ if $C_j > d_j$, and *early* otherwise. Our goal is find a schedule with minimum total processing time of tardy jobs. If we assign a binary indicator variable U_j to each job j, where $U_j = 1$ if j is tardy and otherwise $U_j = 0$, our objective function can be written as $\sum p_j U_j$. In the standard three field notation for scheduling problems of Graham [5], this problem is denoted as the $1||\sum p_j U_j$ problem (the 1 in the first field indicates a single machine model, and the empty second field indicates there are no additional constraints).

The $1||\sum p_j U_j$ problem is a natural scheduling problem, which models a basic scheduling scenario. As it includes Subset Sum as a special case (see below), the $1||\sum p_j U_j$ problem is NP-hard. However, it is only hard in the weak sense, meaning it admits pseudo-polynomial time algorithms. The focus of this paper is on developing fast pseudo-polynomial time algorithms for $1||\sum p_j U_j$, improving in several settings on the best previously known solution from the late 60s. Before we describe our results, we discuss the previously known state of the art of the problem, and describe how our results fit into this line of research.

1.1 State of the Art

$1||\sum p_j U_j$ is a special case of the famous $1||\sum w_j U_j$ problem. Here, each job j also has a weight w_j in addition to its processing time p_j and due date d_j, and the goal is to minimize the total weight (as opposed to total processing times) of tardy jobs. This problem has already been studied in the 60s, and even appeared in Karp's fundamental paper from 1972 [6]. The classical algorithm of Lawler and Moore [10] for the problem is one of the earliest and most prominent examples of pseudo-polynomial algorithms, and it is to date the fastest known algorithm even for the special case of $1||\sum p_j U_j$. Letting $P = \sum_{j \in J} p_j$, their result can be stated as follows:

Theorem 1 [10] $1||\sum w_j U_j$ *(and hence also* $1||\sum p_j U_j$*) can be solved in* $O(P \cdot n)$
time.

Note that as we assume that all processing times are integers, we have $n \leq P$,
and so the running time of the algorithm in Theorem 1 can be bounded by $O(P^2)$. In
fact, it makes perfect sense to analyze the time complexity of a pseudo-polynomial
time algorithm for either problems in terms of P, as P directly corresponds to the total
input length when integers are encoded in unary. Observe that while the case of $n = P$
(all jobs have unit processing times) essentially reduces to sorting, there are several
non-trivial cases where n is smaller than P yet still quite significant in the $O(P \cdot n)$
term of Theorem 1. The question this paper addresses is:

> "*Can we obtain an* $O(P^{2-\varepsilon})$ *time algorithm for* $1||\sum p_j U_j$*, for any fixed*
> $\varepsilon > 0$ *?*"

For $1||\sum w_j U_j$ there is some evidence that the answer to the analogous question
should be no. Karp [6] observed that the special case of the $1||\sum w_j U_j$ problem
where all jobs have the same due date d, the $1|d_j = d|\sum w_j U_j$ problem, is essentially
equivalent to the classical 0/1-Knapsack problem. Cygan *et al.* [4] and Künnemann
et al. [9] studied the (min, +)-Convolution problem (see Sect. 2), and conjectured
that the (min, +)-convolution between two vectors of length n cannot be computed
in $\tilde{O}(n^{2-\varepsilon})$ time, for any $\varepsilon > 0$. Under this (min, +)-Convolution Conjecture, they
obtained lower bounds for several Knapsack related problems. In our terms, their result
can be stated as follows:

Theorem 2 [4,9] *There is no* $\tilde{O}(P^{2-\varepsilon})$ *time algorithm for the* $1|d_j = d|\sum w_j U_j$
problem, for any $\varepsilon > 0$*, unless the* (min, +)-*Convolution Conjecture is false. In
particular,* $1||\sum w_j U_j$ *has no such algorithm under this conjecture.*

Analogous to the situation with $1||\sum w_j U_j$, the special case of $1||\sum p_j U_j$ where
all jobs have the same due date d (the $1|d_j = d|\sum p_j U_j$ problem) is equivalent to
the classical Subset Sum problem. Recently, there has been significant improvements
for Subset Sum resulting in algorithms with $\tilde{O}(T + n)$ running times [2,7], where n is
number of integers in the instance and T is the target. Due to the equivalence between
the two problems, this yields the following result for the $1|d_j = d|\sum p_j U_j$ problem:

Theorem 3 [2,7] $1|d_j = d|\sum p_j U_j$ *can be solved in* $\tilde{O}(P)$ *time.*

On the other hand, due to the equivalence of $1|d_j = d|\sum p_j U_j$ and Subset Sum,
we also know that Theorem 3 above cannot be significantly improved unless the
Strong Exponential Time Hypothesis (SETH) fails. Specifically, combining a recent
reduction from k-SAT to Subset Sum [1] with the equivalence of Subset Sum and
$1|d_j = d|\sum p_j U_j$, yields the following:

Theorem 4 [1] *There is no* $\tilde{O}(P^{1-\varepsilon})$ *time algorithm for the* $1|d_j = d|\sum p_j U_j$ *prob-
lem, for any* $\varepsilon > 0$*, unless SETH fails.*

Nevertheless, Theorem 4 still leaves quite a big gap for the true time complexity of
$1||\sum p_j U_j$, as it can potentially be anywhere between the $\tilde{O}(P)$ time known already

for the special case of $1|d_j = d| \sum p_j U_j$ (Theorem 3), and the $O(Pn) = O(P^2)$ time of Lawler and Moore's algorithm (Theorem 1) for the $1|| \sum w_j U_j$ problem. In particular, the $1|| \sum p_j U_j$ and $1|| \sum w_j U_j$ problems have not been distinguished from an algorithmic perspective so far. This is the starting point of our paper.

1.2 Our Results

The main contribution of this paper is two new pseudo-polynomial time algorithms for $1|| \sum p_j U_j$, each improving on Lawler and Moore's algorithm in a different sense. Our algorithms take a different approach to that of Lawler and Moore in that they rely on fast operators between sets and vectors of numbers.

Our first algorithm improves Theorem 1 in case there are sufficiently many jobs in the instance compared to the total processing time. More precisely, our algorithm has a running time of $\tilde{O}(P^{7/4})$, and so it is faster than Lawler and Moore's algorithm in case $n = \tilde{\omega}(P^{3/4})$.

Theorem 5 $1|| \sum p_j U_j$ can be solved in $\tilde{O}(P^{7/4})$ time.

The algorithm in Theorem 5 uses a new kind of convolution which we coined "Skewed Convolution" and is interesting in its own right. In fact, one of the main technical contributions of this paper is a fast algorithm for the (max, min)-Skewed-Convolution problem (see definition in Sect. 2).

Our second algorithm for $1|| \sum p_j U_j$ improves Theorem 1 in case there are not too many different due dates in the problem instance; that is, $D_\# = |\{d_j : j \in J\}|$ is relatively small when compared to n. This is actually a very natural assumption, for instance in cases where delivery costs are high and products are batched to only few shipments. Let D denote the sum of the different due dates in our instance. Then our second result can be stated as follows:

Theorem 6 $1|| \sum p_j U_j$ can be solved in $\tilde{O}(\min\{P \cdot D_\#, \ P + D\})$ time.

The algorithm in Theorem 6 uses basic operations between sets of numbers, such as the sumset operation (see Sect. 2) as basic primitives for its computation, and ultimately relies on fast polynomial multiplication for its speedup. It should be noted that Theorem 6 includes the $\tilde{O}(P)$ result of Theorem 3 for $1|d_j = d| \sum p_j U_j$ as a special case where $D_\# = 1$ or $D = d$. However, when measuring only in terms of n and D, the running times of the algorithms in [2,7] for the single due date case are $\tilde{O}(D + n)$, which can be significantly faster than $\tilde{O}(P)$.

As a final result we show that the algorithm used in Theorem 6 can be easily extended to the case where we have a fixed number $m \geq 1$ of parallel machines at our disposal. This problem is known as $P_m|| \sum p_j U_j$ in the literature. We show that the complexity of Theorem 6 scales naturally in m for this generalization. This should be compared with the $O(P^m n)$ running time of Lawler and Moore's algorithm.

Theorem 7 $P_m|| \sum p_j U_j$ can be solved in $\tilde{O}(\min\{P^m \cdot D_\#, \ P^m + D^m\})$ time.

1.3 Roadmap

The paper is organized as follows. In Sect. 2 we discuss all the basic primitives that are used by our algorithms, including some basic properties that are essential for the algorithms. We then present our second algorithm in Sect. 3, followed by our first algorithm in Sect. 4. Section 5 describes our fast algorithm for the skewed version of (max, min)-convolution, and is the main technical part of the paper. In Sect. 6 we consider the case of multiple machines and prove Theorem 7, and we conclude with some remarks and open problems in Sect. 7.

2 Preliminaries

In the following we discuss the basic primitives and binary operators between sets/vectors of integers that will be used in our algorithms. In general, we will use the letters X and Y to denote sets of non-negative integers (where order is irrelevant), and the letters A and B to denote vectors of non-negative integers.

Sumsets: The most basic operation used in our algorithms is computing the sumset of two sets of non-negative integers:

Definition 1 *(Sumsets)* Given two sets of non-negative integers X_1 and X_2, the *sumset* of X_1 and X_2, denoted $X_1 \oplus X_2$, is defined by

$$X_1 \oplus X_2 = \{x_1 + x_2 : x_1 \in X_1, x_2 \in X_2\}.$$

Clearly, the sumset $X_1 \oplus X_2$ can be computed in $O(|X_1| \cdot |X_2|)$ time. However, in certain cases we can do better using fast polynomial multiplication. Consider the two polynomials $p_1[\alpha] = \sum_{x \in X_1} \alpha^x$ and $p_2[\beta] = \sum_{x \in X_2} \beta^x$. Then the exponents of all terms in $p_1 \cdot p_2$ with non-zero coefficients correspond to elements in the sumset $X_1 \oplus X_2$. Since multiplying two polynomials of maximum degree d can be done in $O(d \log d)$ time [3], we have the following:

Lemma 1 *Given two sets of non-negative integers $X_1, X_2 \subseteq \{0, \ldots, P\}$, one can compute the sumset $X_1 \oplus X_2$ in $O(P \log P)$ time.*

Set of all Subset Sums: Given set of non-negative integers X, we will frequently be using the set of all sums generated by subsets of X:

Definition 2 *(Subset Sums)* For a given set of non-negative integers X, define the set of all subset sums $\mathcal{S}(X)$ as the set of integers given by

$$\mathcal{S}(X) = \Big\{ \sum_{x \in Y} x : Y \subseteq X \Big\}.$$

Here, we always assume that $0 \in \mathcal{S}(X)$ (as it is the sum of the empty set).

We can use Lemma 1 above to compute $\mathcal{S}(X)$ from X rather efficiently: First, split X into two sets X_1 and X_2 of roughly equal size. Then recursively compute $\mathcal{S}(X_1)$ and

$\mathcal{S}(X_2)$. Finally, compute $\mathcal{S}(X) = \mathcal{S}(X_1) \oplus \mathcal{S}(X_2)$ via Lemma 1. The entire algorithm runs in $\tilde{O}(\sum_{x \in X} x)$ time.

Lemma 2 ([7]) *Given a set of non-negative integers X, with $P = \sum_{x \in X} x$, one can compute $\mathcal{S}(X)$ in $\tilde{O}(P)$ time.*

Convolutions: Given two vectors $A = (A[i])_{i=0}^n$, $B = (B[j])_{j=0}^n$, the $(\circ, \bullet)$-Convolution problem for binary operators $\circ$ and $\bullet$ is to compute a vector $C = (C[k])_{k=0}^{2n}$ with

$$C[k] = \bigcirc_{i+j=k} A[i] \bullet B[j].$$

Throughout this paper we assume that A, B (and C) are integer vectors with entries bounded by $n^{O(1)}$, and with the exceptional values $+\infty$ and $-\infty$. A prominent example of a convolution problem is $(\min, +)$-Convolution discussed above; another similarly prominent example is $(\max, \min)$-Convolution which can be solved in $\tilde{O}(n^{3/2})$ time [8]. For our purposes, it is convenient to look at a *skewed* variant of this problem:

Definition 3 (*Skewed Convolution*) Given two vectors $A = (A[i])_{i=0}^n$, $B = (B[j])_{j=0}^n$, we define the $(\max, \min)$-Skewed-Convolution problem to be the problem of computing the vector $C = (C[k])_{k=0}^{2n}$ where the kth entry in C equals

$$C[k] = \max_{i+j=k} \min\{A[i], B[j] + k\}$$

for each $k \in \{0, \ldots, 2n\}$.

The main technical result of this paper is an algorithm for $(\max, \min)$-Skewed-Convolution that is significantly faster than the naive $O(n^2)$ time algorithm.

Theorem 8 *The $(\max, \min)$-Skewed-Convolution problem can be solved in $\tilde{O}(n^{7/4})$ time.*

3 Algorithm via Sumsets and Subset Sums

In the following section, we provide a proof of Theorem 6 by presenting an algorithm for $1 || \sum p_j U_j$ running in $\tilde{O}(\min\{P \cdot D_{\#}, P + D\})$ time. Recall that $J = \{1, \ldots, n\}$ denotes our input set of jobs, and p_j and d_j respectively denote the processing time and due date of job $j \in \{1, \ldots, n\}$. Our goal is to determine the minimum total processing time of tardy jobs in any schedule for J. Throughout the section we let $d^{(1)} < \cdots < d^{(D_{\#})}$ denote the $D_{\#} \le n$ *different* due dates of the jobs in J.

A key observation for the $1 || \sum p_j U_j$ problem, used already by Lawler and Moore, is that any instance of the problem always has an optimal schedule of a specific type, namely an Earliest Due Date schedule. An Earliest Due Date (EDD) schedule is a schedule $\pi : J \to \{1, \ldots, n\}$ such that

– any early job precedes all late jobs in π, and
– any early job precedes all early jobs with later due dates.

In other words, in an EDD schedule all early jobs are scheduled before all tardy jobs, and all early jobs are scheduled in non-decreasing order of due dates.

Lemma 3 ([10]) *Any* $1||\sum p_j U_j$ *instance has an optimal schedule which is EDD.*

The $D_\#$-many due dates in our instance partition the input set of job J in a natural manner: Define $J_i = \{j : d_j = d^{(i)}\}$ for each $i \in \{1, \ldots, D_\#\}$. Furthermore, let $X_i = \{p_j : j \in J_i\}$ the processing-times of job in J_i. According to Lemma 3 above, we can restrict our attention to EDD schedules. Constructing such a schedule corresponds to choosing a subset $E_i \subseteq J_i$ for each due date $d^{(i)}$ such that $\sum_{j \in E_\ell, \ell \leq i} p_j \leq d^{(i)}$ holds for each $i \in \{1, \ldots, D_\#\}$. Moreover, the optimal EDD schedule maximizes the total sum of processing times in all selected E_i's.

Our algorithm is given in Algorithm 1. It successively computes sets $S_1, \ldots, S_{D_\#}$, where set S_i corresponds to the set of jobs $J_1 \cup \cdots \cup J_i$. In particular, S_i includes the total processing-time of any possible set-family of early jobs $\{E_1, \ldots, E_i\}$. Thus, each $x \in S_i$ corresponds to the total processing time of early jobs in a subset of $J_1 \cup \cdots \cup J_i$. The maximum value $x \in S_{D_\#}$ therefore corresponds to the maximum total processing time of early jobs in any schedule for J. Thus, the algorithm terminates by returning the optimal total weight of tardy jobs $P - x$.

Algorithm 1: SUMSETSCHEDULER(J)

---

1. Let $d^{(1)} < \ldots < d^{(D_\#)}$ denote the different due dates of jobs in J.
2. Compute $X_i = \{p_j : d_j = d^{(i)}\}$ for each $i \in \{1, \ldots, D_\#\}$.
3. Compute $\mathcal{S}(X_1), \ldots, \mathcal{S}(X_{D_\#})$.
4. Let $S_0 = \emptyset$.
5. For $i = 1, \ldots, D_\#$ do

   - Compute $S_i = S_{i-1} \oplus \mathcal{S}(X_i)$.
   - Remove any $x \in S_i$ with $x > d^{(i)}$.

6. Return $P - x$, where x is the maximum value in $S_{D_\#}$.

Correctness of our algorithm follows immediately from the definitions of sumsets and subset sums, and from the fact that we prune out elements $x \in S_i$ with $x > d^{(i)}$ at each step of the algorithm. This is stated more formally in the lemma below.

Lemma 4 *Let* $i \in \{1, \ldots, D_\#\}$, *and let* S_i *be the set of integers at the end of the second step of 5(i). Then* $x \in S_i$ *if and only if there are sets of jobs* $E_1 \subseteq J_1, \ldots, E_i \subseteq J_i$ *such that*

- $\sum_{j \in \bigcup_{\ell=1}^{i} E_\ell} p_j = x$, *and*
- $\sum_{j \in \bigcup_{\ell=1}^{i_0} E_\ell} p_j \leq d^{(i_0)}$ *holds for each* $i_0 \in \{1, \ldots, i\}$.

Proof The proof is by induction on i. For $i = 1$, note that $S_1 = \mathcal{S}(X_1) \setminus \{x : x > d^{(1)}\}$ at the end of step 5(1). Since $\mathcal{S}(X_1)$ includes the total processing time of any subset of jobs in J_1, the first condition of the lemma holds. Since $\{x : x > d^{(1)}\}$ includes all integers violating the second condition of the lemma, the second condition holds.

Let $i > 1$, and assume the lemma holds for $i - 1$. Consider some $x \in S_i$ at the end of the second step of $5(i)$. Then by Definition 1, we have $x = x_1 + x_2$ for some $x_1 \in S_{i-1}$ and $x_2 \in \mathcal{S}(X_i)$ due the first step of $5(i)$. By definition of $\mathcal{S}(X_i)$, there is some $E_i \subseteq J_i$ with total processing time x_2. By our inductive hypothesis there is $E_1 \subseteq J_1, \ldots, E_{i-1} \subseteq J_{i-1}$ such that $\sum_{j \in \bigcup_{\ell=1}^{i} E_\ell} p_j = x_1$, and $\sum_{j \in E_\ell, \ell \le i_0} p_j \le d^{(i_0)}$ holds for each $i_0 \in \{1, \ldots, i - 1\}$. Furthermore, by the second step of $5(i)$, we know that $\sum_{j \in E_\ell, \ell \le i} p_j = x \le d^{(i)}$. Thus, $E_1, \ldots, E_i$ satisfy both conditions of the lemma. □

Let us next analyze the time complexity of the SUMSETSCHEDULER algorithm. Steps 1 and 2 can be both performed in $\tilde{O}(n) = \tilde{O}(P)$ time. Next observe that step 3 can be done in total $\tilde{O}(P)$ time using Lemma 2, as $X_2, \ldots, X_{D_\#}$ is a partition of the set of all processing times of J, and these all sum up to P. Next, according to Lemma 1, each sumset operation at step 5 can be done in time proportional to the largest element in the two sets, which is always at most P. Thus, since we perform at most $D_\#$ sumset operations, the merging step requires $\tilde{O}(D_\# \cdot P)$ time, which gives us the total running time of the algorithm above.

Another way to analyze the running time of SUMSETSCHEDULER is to observe that the maximum element participating in the ith sumset is bounded by $d^{(i+1)}$. It follows that we can write the running time of the merging step as $\tilde{O}(D)$, where $D = \sum_{i=1}^{D_\#} d^{(i)}$. Thus, we have just shown that $1 || \sum p_j U_j$ can be solved in $\tilde{O}(\min\{D_\# \cdot P, D + P\})$ time, completing the proof of Theorem 6.

4 Algorithm via Fast Skewed Convolutions

We next present our $\tilde{O}(P^{7/4})$ time algorithm for $1 || \sum p_j U_j$, providing a proof of Theorem 5. As in the previous section, we let $d^{(1)} < \cdots < d^{(D_\#)}$ denote the $D_\# \le n$ different due dates of the input jobs J, and $J_i = \{j : d_j = d^{(i)}\}$ and $X_i = \{p_j : j \in J_i\}$ as in Sect. 3 for each $i \in \{1, \ldots, D_\#\}$.

For a consecutive subset of indices $I = \{i_0, i_0 + 1, \ldots, i_1\}$, with $i_0, \ldots, i_1 \in \{1, \ldots, D_\#\}$, we define a vector $M(I)$, where $M(I)[x]$ equals the latest (that is, maximum) time point x_0 for which there is a subset of the jobs in $\bigcup_{i \in I} J_i$ with total processing time equal to x that can all be scheduled early in an EDD schedule starting at x_0. If no such subset of jobs exists, we define $M(I)[x] = -\infty$.

For a singleton set $I = \{i\}$, the vector $M(I)$ is easy to compute once we have computed the set $\mathcal{S}(X_i)$:

$$M(\{i\})[x] = \begin{cases} d^{(i)} - x & \text{if } x \in \mathcal{S}(X_i) \text{ and } x \le d^{(i)}, \\ -\infty & \text{otherwise.} \end{cases} \tag{1}$$

For larger sets of indices, we have the following lemma.

Lemma 5 *Let $I_1 = \{i_0, i_0 + 1, \ldots, i_1\}$ and $I_2 = \{i_1 + 1, i_1 + 2, \ldots, i_2\}$ be any two sets of consecutive indices with $i_0, \ldots, i_1, \ldots, i_2 \in \{1, \ldots, D_\#\}$. Then for any value*

x we have:

$$M(I_1 \cup I_2)[x] = \max_{x_1+x_2=x} \min\{M(I_1)[x_1], M(I_2)[x_2] - x_1\}.$$

Proof Let $I = I_1 \cup I_2$. Then $M(I)[x]$ is the latest time point after which a subset of jobs $J^* \subseteq \bigcup_{i \in I} J_i$ of total processing time x can be scheduled early in an EDD schedule. Let x_1 and x_2 be the total processing times of jobs in $J_1^* = J^* \cap \left(\bigcup_{i \in I_1} J_i\right)$ and $J_2^* = J^* \cap \left(\bigcup_{i \in I_2} J_i\right)$, respectively. Then $x = x_1 + x_2$. Clearly, $M(I)[x] \leq M(I_1)[x_1]$, since we have to start scheduling the jobs in J_1^* at time $M(I_1)[x_1]$ by latest. Similarly, it holds that $M(I)[x] \leq M(I_2)[x_2] - x_1$ since the jobs in J_2^* are scheduled at latest at $M(I_2)[x_2]$ and the jobs in J_1^* have to be processed before that time point in an EDD schedule. In combination, we have shown that LHS $\leq$ RHS in the equation of the lemma.

To prove that LHS $\geq$ RHS, we construct a feasible schedule for jobs in $\bigcup_{i \in I} J_i$ starting at RHS. Let x_1 and x_2 be the two values with $x_1 + x_2 = x$ that maximize RHS. Then there is a schedule which schedules some jobs $J_1^* \subseteq \bigcup_{i \in I_1} J_i$ of total processing time x_1 beginning at time $\min\{M(I_1)[x_1], M(I_2)[x_2] - x_1\} \leq M(I_1)[x_1]$, followed by a another subset of jobs $J_2^* \subseteq \bigcup_{i \in I_2} J_i$ of total processing time x_2 starting at time $\min\{M(I_1)[x_1], M(I_2)[x_2] - x_1\} + x_1 \leq M(I_2)[x_2]$. This is a feasible schedule starting at time RHS for a subset of jobs in $\bigcup_{i \in I} J_i$ which has total processing time x. □

Note that the equation given in Lemma 5 is close but not precisely the equation defined in Definition 3 for the (min, max)-Skewed-Convolution problem. Nevertheless, the next lemma shows that we can easily translate between these two concepts.

Lemma 6 *Let A and B be two integer vectors of P entries each. Given an algorithm for computing the* (max, min)-*Skewed-Convolution of A and B in $T(P)$ time, we can compute in $T(P) + O(P)$ time the vector $C = A \otimes B$ defined by*

$$C[x] = \max_{x_1+x_2=x} \min\{A[x_1], B[x_2] - x_1\}.$$

Proof Given A and B, construct two auxiliary vectors A_0 and B_0 defined by $A_0[x] = B[x] + x$ and $B_0[x] = A[x]$ for each entry x. Compute the (max, min)-Skewed-Convolution of A_0 and B_0, and let C_0 denote the resulting vector. We claim that the vector C defined by $C[x] = C_0[x] - x$ equals $A \otimes B$. Indeed, we have

$$\begin{aligned}
C_0[x] - x &= \max_{x_1+x_2=x} \min\{A_0[x_1], B_0[x_2] + x\} - x \\
&= \max_{x_1+x_2=x} \min\{A_0[x_1] - x, B_0[x_2]\} \\
&= \max_{x_1+x_2=x} \min\{B[x_1] + x_1 - x, A[x_2]\} \\
&= \max_{x_1+x_2=x} \min\{B[x_1] - x_2, A[x_2]\} \\
&= \max_{x_1+x_2=x} \min\{A[x_1], B[x_2] - x_1\},
\end{aligned}$$

where in the third step we expanded the definition of A_0 and B_0, and in the last step we used the symmetry of x_1 and x_2. $\square$

We are now in position to describe our algorithm called CONVSCHEDULER which is depicted in Algorithm 2. The algorithm first computes the subset sums $\mathcal{S}(X_1), \ldots, \mathcal{S}(X_{D_\#})$, and the set of vectors $\mathcal{M} = \{M_1, \ldots, M_{D_\#}\}$. Following this, it iteratively combines every two consecutive vectors in $\mathcal{M}$ by using the $\otimes$ operation. The algorithm terminates when $\mathcal{M} = \{M_1\}$, where at this stage M_1 corresponds to the entire set of input jobs J. It then returns $P - x$, where x is the maximum value with $M_1[x] > -\infty$; by definition, this corresponds to a schedule for J with $P - x$ total processing time of tardy jobs. For convenience of presentation, we assume that $D_\#$ is a power of 2.

Algorithm 2: CONVSCHEDULER(J)

---

1. Let $d^{(1)} < \ldots < d^{(D_\#)}$ denote the different due dates of jobs in J.
2. Compute $X_i = \{p_j : d_j = d^{(i)}\}$ for each $i \in \{1, \ldots, D_\#\}$.
3. Compute $\mathcal{S}(X_1), \ldots, \mathcal{S}(X_{D_\#})$.
4. Compute $\mathcal{M} = \{M_1 = M(1), \ldots, M_{D_\#} = M(D_\#)\}$.
5. While $|\mathcal{M}| > 1$ do

   - Compute $M_i = M_{2i-1} \otimes M_{2i}$ for each $i \in \{1, \ldots, |\mathcal{M}|/2\}$.

6. Return $P - x$, where x is the maximum value with $M_1[x] > -\infty$.

Correctness of this algorithm follows directly from Lemma 5. To analyze its time complexity, observe that steps 1–4 can be done in $\tilde{O}(P)$ time (using Lemma 2). Step 5 is performed $O(\log D_\#) = O(\log P)$ times, and each step requires a total of $\tilde{O}(P^{7/4})$ time according to Theorem 8, as the total sizes of all vectors at each step is $O(P)$. Finally, step 6 requires $O(P)$ time. Summing up, this gives us a total running time of $\tilde{O}(P^{7/4})$, and completes the proof of Theorem 5 (apart from the proof of Theorem 8).

5 Fast Skewed Convolutions

In the following section we present our algorithm for (max, min)-Skewed-Convolution, and provide a proof for Theorem 8. Let $A = (A[i])_{i=0}^n$ and $B = (B[j])_{j=0}^n$ denote the input vectors for the problem throughout the section. Recall we wish the compute the vector $C^\ell = (C[k])_{k=0}^{2n}$ where

$$C[k] = \max_{i+j=k} \min\{A[i], B[j] + k\}$$

for each $k \in \{0, \ldots, 2n\}$.

We begin by first defining the problem slightly more generally, in order to facilitate our recursive strategy later on. For this, for each integer $\ell \in \{0, \ldots, \log 2n\}$, let $A^\ell = \lfloor A/2^\ell \rfloor$ and $B^\ell = \lfloor B/2^\ell \rfloor$, where rounding is done component-wise. We will

compute vectors $C^\ell = (C^\ell[k])_{k=0}^{2n}$ defined by:

$$C^\ell[k] = \max_{i+j=k} \min\left\{A^\ell[i], B^\ell[j] + \lfloor k/2^\ell \rfloor\right\}.$$

Observe that a solution for $\ell = 0$ yields a solution to the original (max, min)-Skewed-Convolution problem, and for $\ell \geq \log(2n)$ the problem degenerates to (max, min)-Convolution.

We next define a particular kind of additive approximation of vectors C^ℓ. We say that a vector D^ℓ is a *good approximation* of C^ℓ if $C^\ell[k] - 2 \leq D^\ell[k] \leq C^\ell[k]$ for each $k \in \{0, \ldots, 2n\}$. Now, the main technical part of our algorithm is encapsulated in the following lemma.

Lemma 7 *There is an algorithm that computes C^ℓ in $\tilde{O}(n^{7/4})$ time, given A^ℓ, B^ℓ, and a good approximation D^ℓ of C^ℓ.*

We postpone the proof of Lemma 7 for now, and instead show that it directly yields our desired algorithm for (max, min)-Skewed-Convolution:

Proof of Theorem 8 In order to compute $C = C^0$, we perform an (inverse) induction on ℓ: As mentioned before, if $\ell \geq \log(2n)$, then we can neglect the "$+ \lfloor k/2^\ell \rfloor$" term and compute C^ℓ in $\tilde{O}(n^{3/2}) = \tilde{O}(n^{7/4})$ time using a single (max, min)-Convolution computation [8].

For the inductive step, let $\ell < \log(2n)$ and assume that we have already computed $C^{\ell+1}$. We construct the vector $D^\ell = 2C^{\ell+1}$, and argue that it is a good approximation of C^ℓ. Indeed, for each entry k, on the one hand, we have:

$$D^\ell[k] = 2C^{\ell+1}[k] = 2 \cdot \max_{i+j=k} \min\{\lfloor A^\ell[i]/2 \rfloor, \lfloor B^\ell[j]/2 \rfloor + \lfloor k/2^{\ell+1} \rfloor\}$$
$$\leq \max_{i+j=k} \min\{A^\ell[i], B^\ell[j] + \lfloor k/2^\ell \rfloor\} = C^\ell[k];$$

and on the other hand, we have:

$$D^\ell[k] = 2C^{\ell+1}[k] = 2 \cdot \max_{i+j=k} \min\{\lfloor A^\ell[i]/2 \rfloor, \lfloor B^\ell[j]/2 \rfloor + \lfloor k/2^{\ell+1} \rfloor\}$$
$$\geq \max_{i+j=k} \min\{A^\ell[i] - 1, B^\ell[j] + \lfloor k/2^\ell \rfloor - 2\} \geq C^\ell[k] - 2.$$

Thus, using D^ℓ we can apply Lemma 7 above to obtain C^ℓ in $\tilde{O}(n^{7/4})$ time. Since there are $O(\log n)$ inductive steps overall, this is also the overall time complexity of the algorithm.   □

It remains to prove Lemma 7. Recall that we are given A^ℓ, B^ℓ, and D^ℓ, and our goal is to compute the vector C^ℓ in $\tilde{O}(n^{7/4})$ time. We construct two vectors L^ℓ and R^ℓ with $2n$ entries each, defined by

$$L^\ell[k] = \max\left\{A^\ell[i_0] : \begin{array}{l} A^\ell[i_0] \leq B^\ell[k - i_0] + \lfloor k/2^\ell \rfloor \text{ and} \\ D^\ell[k] \leq A^\ell[i_0] \leq D^\ell[k] + 2 \end{array}\right\},$$

and

$$R^\ell[k] = \max \left\{ B^\ell[j_0] + \lfloor k/2^\ell \rfloor : \begin{array}{l} B^\ell[j_0] + \lfloor k/2^\ell \rfloor \leq A^\ell[k - j_0] \text{ and} \\ D^\ell[k] \leq B^\ell[j_0] + \lfloor k/2^\ell \rfloor \leq D^\ell[k] + 2 \end{array} \right\}$$

for $k \in \{0, \ldots, 2n\}$. That is, $L^\ell[k]$ and $R^\ell[k]$ respectively capture the largest value attained as the left-hand side or right-hand side of the inner min-operation in $C^\ell[k]$, as long as that value lies in the feasible region approximated by $D^\ell[k]$. Since D^ℓ is a good approximation, the following lemma is immediate from the definitions:

Lemma 8 $C^\ell[k] = \max\{L^\ell[k], R^\ell[k]\}$ *for each* $k \in \{0, \ldots, 2n\}$.

According to Lemma 8, it suffices to compute L^ℓ and R^ℓ. We first focus on computing L^ℓ. The computation of R^ℓ is very similar and we will later point out the necessary changes.

Let $0 < \delta < 1$ be a fixed constant to be determined later. We say that an index $k \in \{0, \ldots, 2n\}$ is *light* if

$$\left| \left\{ i : D^\ell[k] \leq A^\ell[i] \leq D^\ell[k] + 2 \right\} \right| \leq n^\delta.$$

Informally, k is light if the number of candidate entries $A^\ell[i]$ which can equal $C^\ell[k]$ is relatively small (recall that $D^\ell[k] \leq C^\ell[k] \leq D^\ell[k]+2$, as D^ℓ is a good approximation of C^ℓ). If k is not light then we say that it is *heavy*.

Our algorithm for computing L^ℓ proceeds in three main steps: In the first step it handles all light indices, in the second step it sparsifies the input vector, and in the third step it handles all heavy indices:

– *Light indices:* We begin by iterating over all light indices $k \in \{0, \ldots, 2n\}$. For each light index k, we iterate over all entries $A^\ell[i]$ satisfying $D^\ell[k] \leq A^\ell[i] \leq D^\ell[k] + 2$, and set $L^\ell[k]$ to be the maximum $A^\ell[i]$ among those entries with $A^\ell[i] \leq B^\ell[k - i] + \lfloor k/2^\ell \rfloor$. Note that after this step, we have

$$L^\ell[k] = \max \left\{ A^\ell[i_0] : \begin{array}{l} A^\ell[i_0] \leq B^\ell[k - i_0] + \lfloor k/2^\ell \rfloor \text{ and} \\ D^\ell[k] \leq A^\ell[i_0] \leq D^\ell[k] + 2 \end{array} \right\},$$

for each light index k.
– *Sparsification step:* After dealing with the light indices, several entries of A^ℓ become redundant. Consider an entry $A^\ell[i]$ for which $|\{i_0 : A^\ell[i] - 2 \leq A^\ell[i_0] \leq A^\ell[i] + 2\}| \leq n^\delta$. Then all indices k for which $L^\ell[k]$ might equal $A^\ell[i]$ must be light, and are therefore already dealt with in the previous step. Consequently, it is safe to replace $A^\ell[i]$ by $-\infty$ so that $A^\ell[i]$ no longer plays a role in the remaining computation.
– *Heavy indices:* After the sparsification step A^ℓ contains few distinct values. Thus, our approach is to fix any such value v and detect whether $L^\ell[k] \geq v$. To that end, we translate the problem into an instance of (max, min)-Convolution: Let $(A_v^\ell[i])_{i=0}^n$ be an be an indicator-like vector defined by $A_v^\ell[i] = +\infty$ if $A^\ell[i] = v$,

and otherwise $A_v^\ell[i] = -\infty$. We next compute the vector L_v^ℓ defined by $L_v^\ell[k] = \lfloor k/2^\ell \rfloor + \max_{i+j=k} \min\{A_v^\ell[i], B^\ell[j]\}$ using a single computation of (max, min)-Convolution. We choose

$$L^\ell[k] = \max\left\{v : L_v^\ell[k] \geq v \text{ and } D^\ell[k] \leq v \leq D^\ell[k] + 2\right\}$$

for any heavy index k, and claim that $L^\ell[k]$ equals $\max\{A^\ell[i_0] : A^\ell[i_0] \leq B^\ell[k - i_0] + \lfloor k/2^\ell \rfloor\}$.

On the one hand, if $L_v^\ell[k] \geq v$ then there are indices i and j with $i + j = k$ for which $A^\ell[i] = v$ and $B^\ell[j] + \lfloor k/2^\ell \rfloor \geq A^\ell[i] = v$. Thus, the computed value $L^\ell[k]$ is not greater than

$$L^\ell[k] \leq \max\left\{A^\ell[i_0] : \begin{array}{l} A^\ell[i_0] \leq B^\ell[k - i_0] + \lfloor k/2^\ell \rfloor \text{ and} \\ D^\ell[k] \leq A^\ell[i_0] \leq D^\ell[k] + 2 \end{array}\right\}.$$

On the other hand, for all values v for which $A^\ell[i] = v$ for some $i \in \{0, \ldots, n\}$, we have if $v = A^\ell[i] \leq B^\ell[k - i] + \lfloor k/2^\ell \rfloor$ then $A_v^\ell[i] = -\infty$, which in turn implies that $A_v^\ell[i] \geq B^\ell[k - i] + \lfloor k/2^\ell \rfloor \geq A^\ell[i] = v$. Thus, our selection of $L^\ell[k]$ is also at least as large as

$$L^\ell[k] \geq \max\left\{A^\ell[i_0] : \begin{array}{l} A^\ell[i_0] \leq B^\ell[k - i_0] + \lfloor k/2^\ell \rfloor \text{ and} \\ D^\ell[k] \leq A^\ell[i_0] \leq D^\ell[k] + 2 \end{array}\right\},$$

and hence, these two values must be equal.

We finally argue how to adapt the approach to compute R^ℓ. In the first step, we instead classify an index k as light if $|\{i : D^\ell[k] \leq B^\ell[j] + \lfloor k/2^\ell \rfloor \leq D^\ell[k] + 2\}| \leq n^\delta$. In the same way as before we can compute $R^\ell[k]$ for all light indices k, as well as apply the sparsification step to replace all entries $B^\ell[j]$ which satisfy $|\{j_0 : B^\ell[j] - 2 \leq B^\ell[j_0] \leq B^\ell[j] + 2\}| \leq n^\delta$ by $-\infty$. After the sparsification, the vector B^ℓ contains only few distinct values, and for any such value v we proceed similar to before. Defining B_v^ℓ analogously, we compute $R_v^\ell[k] = \max_{i+j=k} \min\{A^\ell[i], B_v^\ell[j]\}$ and return

$$R^\ell[k] = \max\left\{v + \lfloor k/2^\ell \rfloor : \begin{array}{l} R_v^\ell[k] \geq v + \lfloor k/2^\ell \rfloor \text{ and} \\ D^\ell[k] \leq v + \lfloor k/2^\ell \rfloor \leq D^\ell[k] + 2 \end{array}\right\}$$

for all heavy indices k. One can verify that this choice of $R^\ell[k]$ is correct with exactly the same proof as before.

This completes the description of our algorithm. As we argued its correctness above, what remains is to analyze its time complexity. Note that we can determine in $O(\log n)$ time whether an index k is light or heavy, by first sorting the values in A^ℓ. For each light index k, determining $L^\ell[k]$ can be done in $O(n^\delta)$ time (on the sorted A^ℓ), giving us a total of $\tilde{O}(n^{1+\delta})$ time for the first step. For the second step, we can determine

whether a given entry $A^\ell[i]$ can be replaced with $-\infty$ in $O(\log n)$ time, giving us a total of $\tilde{O}(n)$ time for this step.

Consider then the final step of the algorithm. Observe that after exhausting the sparsification step, A^ℓ contains at most $O(n^{1-\delta})$ many distinct values: For any surviving value v, there is another (perhaps different) value v' of difference at most 2 from v that occurs at least $1/5 \cdot n^\delta$ times in A^ℓ, and so there can only be at most $O(n^{1-\delta})$ such distinct values. Thus, the running time of this step is dominated by the running time of $O(n^{1-\delta})$ (max, min)-Convolution computations, each requiring $\tilde{O}(n^{3/2})$ time using the algorithm of [8], giving us a total of $\tilde{O}(n^{5/2-\delta})$ time for this step.

Thus, the running time of our algorithm is dominated by the $\tilde{O}(n^{1+\delta})$ running time of its first step, and the $\tilde{O}(n^{5/2-\delta})$ running time of its last step. Choosing $\delta = 3/4$ gives us $\tilde{O}(n^{7/4})$ time for both steps, which is the time promised by Lemma 7. Thus, Lemma 7 holds.

6 Multiple Machines

In the following we show how to extend the SUMSETSCHEDULER algorithm of Sect. 3 to the case of a fixed number m multiple machines, the $P_m || \sum p_j U_j$ problem. In this variant, we have m machines at our disposal, and so a schedule σ for the set of jobs J is now a function $\sigma : \{1, \ldots, n\} \to \{1, \ldots, m\} \times \{1, \ldots, n\}$. The first component of $\sigma(j)$ specifies the machine one which j is scheduled, and the second component specifies its order on the machine. The completion time C_j of j is then the sum of all jobs preceding j (including j itself) on the same machine that j is scheduled. Our goal remains to minimize the total processing time of tardy jobs $\sum p_j U_j$.

For extending algorithm SUMSETSCHEDULER, we need to first extend Definition 2 to the case of multiple machines.

Definition 4 For a given set of non-negative integers X, define the set $\mathcal{S}^m(X)$ as the set of m-tuples given by

$$\mathcal{S}^m(X) = \left\{ \left(\sum_{x \in Y_1} x, \ldots, \sum_{x \in Y_m} x \right) \; : \; \begin{array}{l} Y_1, \ldots, Y_m \subseteq X \text{ and } Y_i \cap Y_j = \emptyset \\ \text{for each } i, j \in \{1, \ldots, m\}, i \neq j \end{array} \right\}.$$

Thus, every element in $\mathcal{S}^m(X)$ is an m-tuple $\mathbf{x} = (x_1, \ldots, x_m)$ of non-negative integers, where we interpret x_i as the total processing time on machine $i \in \{1, \ldots, m\}$. We consider component-wise addition between two m-tuples, and define the sumset $X_1 \oplus X_2$ of two sets of m-tuples as in Definition 1; i.e., $X_1 \oplus X_2 = \{\mathbf{x}_1 + \mathbf{x}_2 : \mathbf{x}_1 \in X_1, \mathbf{x}_2 \in X_2\}$.

To efficiently compute the sumset $X_1 \oplus X_2$ when X_1 and X_2 are sets of m-tuples we use multivariate polynomial multiplication. Let $p_1[\alpha_1, \ldots, \alpha_m] = \sum_{(x_1, \ldots, x_m) \in X_1} \Pi_{i=1}^m \alpha_i^{x_i}$ and $p_2[\beta_1, \ldots, \beta_m] = \sum_{(x_1, \ldots, x_m) \in X_2} \Pi_{i=1}^m \beta_i^{x_i}$. Then the exponents of all terms in $p_1 \cdot p_2$ with non-zero coefficients correspond to elements in the sumset $X_1 \oplus X_2$. Since multiplying two m-variate polynomials of maximum degree d on each variable can be reduced to multiplying two univariate polynomials of maximum degree $O(d^m)$ using Kronecker's map (see e.g. [11]), we obtain the following:

Lemma 9 *Given two sets of m-tuples of non-negative integers $X_1, X_2 \subseteq \{0, \ldots, P\}^m$, one can compute the sumset $X_1 \oplus X_2$ in $O(P^m \log P)$ time.*

Using the same divide and conquer approach used for Lemma 2, we can use Lemma 9 above to compute $\mathcal{S}^m(X)$ from X. The same analysis used for Lemma 2 will give us a running time of $O(P^m \log P)$ instead of $O(P \log P)$.

Lemma 10 *Given a set of non-negative integers X, with $P = \sum_{x \in X} x$, one can compute $\mathcal{S}^m(X)$ in $\tilde{O}(P^m)$ time.*

The algorithm now proceeds in an analogous manner to the single machine case. First we partition the set of jobs J into $J_1, \ldots, J_{D_\#}$ according to the $D_\#$ different due dates $d^{(1)}, \ldots, d^{(D_\#)}$, and we let $X_i = \{p_j : j \in J_i\}$ for each $i \in \{1, \ldots, D_\#\}$. This can be done in $O(n) = O(P)$ time. We then compute $\mathcal{S}^m(X_1), \ldots, \mathcal{S}^m(X_{D_\#})$ in $\tilde{O}(P^m)$ time using Lemma 10. Finally, we compute $S_0, S_1, \ldots, S_{D_\#} \subseteq \{0, \ldots, P\}^m$ starting from $S_0 = \emptyset$, and then iteratively computing S_i by $S_i = S_{i-1} \oplus \mathcal{S}^m(X)$, where elements in S_i with a component strictly larger than $d^{(i)}$ are discarded from future computations. The time complexity of this last final step is $\tilde{O}(\min\{P^m \cdot D_\#, D^m\})$, using a similar analysis to the one done in Sect. 3. This completes the proof of Theorem 7.

7 Discussion and Open Problems

In this paper we presented two algorithms for the $1||\sum p_j U_j$ problem; the first running in $\tilde{O}(P^{7/4})$ time, and the second running in $\tilde{O}(\min\{P \cdot D_\#, P + D\})$ time. Both algorithms provide the first improvements over the classical Lawler and Moore algorithm in 50 years (which can also solve the more general $1||\sum w_j U_j$), and use more sophisticated tools such as polynomial multiplication and fast convolutions. Moreover, both algorithms are very easy to implement given a standard ready made FFT implementation for fast polynomial multiplication. Nevertheless, there are still a few ways which our results can be improved or extended:

- *Multiple machines*: As we showed in Sect. 6, the SUMSETSCHEDULER algorithm can easily be extended to the multiple parallel machine case, giving us a total running time of $\tilde{O}(\min\{P^m \cdot D_\#, P^m + D^m\})$. We do not know how to obtain a similar extension for algorithm CONVSCHEDULER. In particular, there is no reason to believe that $P_m||\sum p_j U_j$ cannot be solved in $\tilde{O}(P^m)$ time, or even better, either by extending algorithm CONVSCHEDULER or by a completely different approach.
- *Even faster skewed convolutions*: We have no indication that our algorithm for (max, min)-Skewed-Convolution is the fastest possible. It would interesting to see whether one can improve its time complexity, say to $\tilde{O}(P^{3/2})$. Naturally, any such improvement would directly improve Theorem 5. Conversely, one could try to obtain some sort of lower bound for the problem, possibly in the same vein as Theorem 2. Improving the time complexity beyond $\tilde{O}(P^{3/2})$ seems difficult as this would directly imply an improvement to the (max, min)-Convolution problem. Indeed, let A, B be a given (max, min)-Convolution instance and construct vectors A_0, B_0 with $A_0[i] = N \cdot A[i]$ and $B_0[j] = N \cdot B[j]$ for $N = 2n + 1$. If C_0 is the (max, min)-Skewed-Convolution of A_0 and B_0 (that is, $C_0[k] =$

$\max_{i+j=k} \min\{A_0[i], B_0[j] + k\}$), then the vector C with $C[k] = \lfloor C_0[k]/N \rfloor$ is the (max, min)-Convolution of A and B.

- *Other scheduling problems*: Can the techniques in this paper be applied to any other interesting scheduling problems? A good place to start might be to look at other problems which directly generalize Subset Sum.

References

1. Abboud, A., Bringmann, K., Hermelin, D., Shabtay, D.: SETH-based lower bounds for subset sum and bicriteria path. In: Proceedings of of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 41–57 (2019)
2. Bringmann, K.: A near-linear pseudopolynomial time algorithm for subset sum. In: Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1073–1084 (2017)
3. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
4. Cygan, Marek, Mucha, Marcin, Wegrzycki, Karol, Wlodarczyk, Michal: On problems equivalent to (min, +)-convolution. ACM Trans. Algorithms **15**(1), 14:1-14:25 (2019)
5. Graham, Ronald L.: Bounds on multiprocessing timing anomalies. SIAM J. Appl. Math. **17**(2), 416–429 (1969)
6. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, pp. 85–103. Springer, Berlin (1972)
7. Koiliaris, K., Xu, C.: A faster pseudopolynomial time algorithm for subset sum. In: Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1062–1072 (2017)
8. Kosaraju, S.R.: Efficient tree pattern matching. In: Proceedings of the 30th annual symposium on Foundations Of Computer Science (FOCS), pp. 178–183 (1989)
9. Künnemann, M., Paturi, R., Schneider, S.: On the fine-grained complexity of one-dimensional dynamic programming. In: Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP), pp. 21:1–21:15 (2017)
10. Lawler, Eugene L., Moore, James M.: A functional equation and its application to resource allocation and sequencing problems. Manage. Sci. **16**(1), 77–84 (1969)
11. Pan, Victor Y.: Simple multivariate polynomial multiplication. J. Symb. Comput. **18**(3), 183–186 (1994)