Linear-time approximation algorithms for minimum subset sum and subset sum

Liliana Grigoriu[1]

[1]: *Fakultät für Wirtschaftswissenschaften, Wirtschaftsinformatik und Wirtschaftsrecht Universität Siegen, Kohlbettstr. 15, 57068 Siegen , Germany*

Abstract

We present a family of approximation algorithms for minimum subset sum with a worst-case approximation ratio of $1+1/k$ and which run in linear time assuming that is constant. We also present a family of linear-time approximation algorithms for subset sum with worst-case approximation factors of $1-1/(k+1)$ assuming that k is constant. The algorithms use approaches from and improve upon previous literature, where a linear-time 4/5 approximation algorithm for subset sum and a 5/4 linear-time approximation algorithm for minimum subset sum have been provided by Kellerer and by Grigoriu and Briskorn respectively. The maximum number of times a linear-time procedure could be called within the algorithms, which depends on k, is determined computationally for each of the two problems for sample values of k up to $k = 80$. For example, for $k = 10$, this number is 137 for subset sum and 171 for minimum subset sum, for $k = 20$ these numbers are 2712 and respectively 3085, and for $k = 30$ they are 28627 and respectively 31023. We also comment on how the algorithms can be parallelized. The simplicity of the algorithms allows for fast implementation.

Key words: minimum subset sum, subset sum, approximation algorithms, linear-time

1. Introduction

The minimum subset sum problem (see for example Kellerer et al. [6]) asks, given a multiset J of positive integers and a target sum S which subset

[1]Corresponding author; email address: liliana.grigoriu@cs.pub.ro

of J has elements that add up to a sum that is at least S and as close as possible to S. The related subset sum problem (see for example Kellerer et al. [6]) asks, given a multiset J of positive integers and a target sum S, which subset of J has elements that add up to a sum that is at most S and as close to S as possible.

In Kellerer et al. [5] two linear-time 3/4 and 4/5 approximation algorithms for subset sum are presented. Several FPTAS are known for this problem (see Kellerer et al. [6] for an overview). For the minimum subset sum problem, a quadratic approximation algorithm with a worst-case approximation ratio of 5/4 has been proposed in Güntzer and Jungnickel [3], and a linear-time 5/4 approximation algorithm has been presented in Grigoriu and Briskorn [2], where it was used to solve a related scheduling problem. Also, FPTAS for more general problems than minimum subset sum are given in Gens and Levner [1], Kovalyov [7] and Janiak and Kovalyov [4]. In this work we use some ideas that were also used in Kellerer et al. [5] and Grigoriu and Briskorn [2].

In Section 2 we present a strategy to obtain approximation algorithms with a worst-case approximation ratio of $(k+1)/k$, which have a linear time complexity if k is considered constant for minimum subset sum. In Section 3 we present linear-time approximation algorithms for subset sum with approximation factors of $(k+1)/k$ assuming that k is considered constant.

2. Linear-time approximation algorithms for minimum subset sum

To obtain an approximation algorithm that solves the minimum subset sum problem with a worst-case approximation factor of $(k+1)/k$, we first divide the set J into disjoint subsets as follows:

$$J_i = \{j \in J | (i-1)\frac{S}{k} \leq j < \frac{iS}{k}\}$$

and $J_l = \{j \in J | j \geq S\}$.

Next, we present two subroutines which will be used in the procedure $MinSubsetSum(J, S, k)$, which has a worst-case approximation factor of $(k+1)/k$. The subroutine $AddJ1(K)$ adds, if necessary, elements from (the globally visible set) J_1 to a set of jobs $K \subseteq J \setminus J_1$ with $S - \sum_{j \in J_1} j \leq \sum_{j \in K} j \leq \frac{k+1}{k} S$ until the sum of the elements in K is at least S. Then, $AddJ1(K)$ returns the resulting set K, which has the property that the sum

of its elements is at least S and at most $\frac{k+1}{k}S$, as all elements of J_1 are less than $\frac{1}{k}S$.

The second subroutine $Candidate(K, Kbest)$ updates the best found solution $Kbest$ with a newly found solution K if K is better. Here, we assume that the procedure can change $Kbest$.

We denote with $min_i(J_h)$ and with $max_i(J_h)$ the sets of elements with the i smallest and respectively with the i largest values from J_h (with ties broken arbitrarily). The definition assumes that there are at least i elements in J_h. The procedure $MinSubsetSum(J, S)$ is specified in the following.

2.1. General strategy

We give a strategy by which algorithms with an approximation factor of $(k+1)/k$ for the minimum subset sum problem can be built. If k is considered constant the time complexity of these algorithms is linear. The strategy we present mainly generalizes the strategy used in Grigoriu and Briskorn [2], where a 5/4 linear-time approximation algorithm for minimum subset sum was presented in the context of addressing a scheduling problem.

Any solution that contains an element of J_l is not better than the solution $min_1(J_l)$, which our algorithm will check separately. Any other solution contains a number n_i of elements from J_i for $i \in \{2, 3, \ldots, k\}$, and possibly elements from J_1. We call a tuple $(n_2, \ldots, n_k)$ configuration of all subsets of $J \backslash J_l$ that contain exactly n_i of elements from J_i for all $i \in I$, where $I = \{2, 3, \ldots, k\}$.

Our algorithms build sets Q of elements from $J \setminus J_1$ which fulfill one of the following conditions:

1. it can be proved that they are a $\frac{k+1}{k}$ approximation for the problem instance
2. they can be the optimal solution
3. $\sum_{i \in Q} i < S$ and $\sum_{i \in Q} i + sum_{i \in J_1} i \geq S$. Here, a solution can be obtained by adding elements from J_1 to Q until the sum of elements in Q reaches or exceeds S, at which point we have $S \leq \sum_{i \in Q} \leq \frac{k+1}{k}S$, which implies that Q is a $\frac{k+1}{k}$ approximation of an optimal solution as desired.

We consider all configurations of an optimal solution or a $(k+1)/k$ approximation thereof can have.

Lemma 2.1
We assume that the item with the smallest value from J_l is considered to be a candidate solution. If (in addition to that) all configurations $(n_2, \ldots, n_k)$ which fulfill all of the following properties are considered (that means solutions that correspond to these configurations are found and returned if they are a $(k+1)/k$ approximation or stored as a candidate if they may be the optimal solution), either an optimal solution or a $(k+1)/k$ approximation thereof is found:

1. (a) $\sum_{i \in I} \frac{iS}{k} \cdot n_i + \sum_{j \in J_1} j \geq S$. In addition, we must have
   (b) $\sum_{i \in I} \sum_{j \in max_{n_i} J_i} j + \sum_{j \in J_1} j \geq S$.
2. $\sum_{i \in I} (i-1) n_i < k + \min\{q | n_q > 0\} - 1$.

Proof: We prove the lemma by arguing that no solutions which correspond to configurations that do not fulfill Condition 1 exist and that it is not necessary to consider solutions which correspond to configurations that do not fulfill Condition 2 in case all solutions that correspond to configurations that fulfill Condition 2 are considered. We do not consider in this proof solutions that contain any jobs from J_l (according to the way we defined solutions corresponding to configurations).

If Condition 1.(a) is not fulfilled for a configuration T, then, as a consequence of the ranges for the items of each set J_i, no solutions exist that have configuration T. If Condition 1.(b) is not fulfilled for a configuration T, then, as a consequence of the contents of the sets J_i within the considered problem instance, no solutions exist that have configuration T.

We next argue why it is not necessary to consider sets that have configurations which do not fulfill Condition 2. If any configuration $T = (n_2, \ldots, n_k)$ has the property $\sum_{i \in I} \sum_{j \in min_{n_i} J_i} j \geq \sum_{i \in I} (i-1) n_i \frac{S}{k} \geq S$, which is equivalent to $\sum_{i \in I} (i-1) n_i \geq k$, T is the configuration of a solution. Adding any element to a set with such a configuration would result in a set that fulfills $\sum_{i \in I} (i-1) n_i \geq k + \min\{q | n_q > 0\} - 1$ and will lead to a worse solution. With other words, a solution that has a configuration with the property $\sum_{i \in I} (i-1) n_i \geq k + \min\{q | n_q > 0\} - 1$ can be transformed into a better solution by removing from it an element from $J_{\min\{q | n_q > 0\}}$, and if such a removal is done repeteadly until no further removal of this type can be done, it means that a configuration of a feasible solution that is at least as good (as all configurations considered in the removal process) and that fulfills Condition 2 is found.

Concluding, if all solutions that have configurations that fulfill Conditions 1 and 2 are considered, in addition to the set containing the smallest item from J_l if it exists, either an optimal solution or a (k+1)/k approximation of it is found. $\triangle$

Let $sumJ1 = \sum_{j \in J_1} j$. Our algorithm considers all configurations of interest, i.e. those which fulfill Condition 2 of Lemma 2.1 and returns any $(k+1)/k$ approximation of an optimal solution. We next present the procedure CheckConfiguration that considers each tuple $T = (n_2, \ldots, n_k)$ (which should preferably be chosen to fulfill the properties from Lemma 2.1). We assume that k, $sumJ1$, the sets $J_1, J_2, \ldots J_k$ and the current best candidate solution $Kbest$ are globally visible and can be accessed by the procedure:

CheckConfiguration(T){
(0) Let $I = \{2, 3, \ldots, k\}$, let $Q' = \cup_{i \in I} max_{n_i} J_i$ and $Q = \cup_{i \in I} min_{n_i} J_i$; (if for some $i \in I$, J_i does not contain n_i elements return $\emptyset$;)
(1) If $(\sum_{j \in Q'} j + sumJ1 \geq S)$\{
(2) If $(\sum_{j \in Q} j \geq S)$\{
(a) candidate (Q, Kbest);
(b) If $(\sum_{j \in Q} j \leq \frac{k+1}{k} S)$ return Q; // this step is optional
(c) return $\emptyset$;
}
(3) Else {
(a) If $\sum_{j \in Q} \geq S - SumJ1$, return $AddJ1(Q)$;
(b) For all $i \in I$ do: Iteratively replace in Q elements of $min_{n_i} J_i$ with elements of $max_{n_i} J_i$ until the sum of elements in Q reaches or exceeds $S - SumJ1$, in which case return $AddJ1(Q)$, or until all elements from $min_{n_i} J_i$ are replaced in Q.
}
}
(4) return $\emptyset$;
}

Here, it is assumed that the procedure can change the value of K_{best}. The following Lemma states properties of this procedure.

Lemma 2.2
In case there is no solution that corresponds to the configuration passed as parameter to $CheckConfiguration$, then $CheckConfiguration$ returns the

empty set. Otherwise, it either returns a solution that is (k+1)/k approximation of an optimal solution, or it records the best among the solutions corresponding to the considered configuration if it is the best solution found so far.

Proof: In step (1) the procedure makes sure that we are dealing with a configuration that can lead to a solution. In step (2), it is first checked whether taking the smallest elements corresponding to the configuration (which results in a set that we call Q) results in a solution, and in that case Q is considered as a candidate and stored if it is the best candidate found so far. In such a situation it is not necessary to check other solutions corresponding to the configuration, as these can not be better than Q (and the empty set is returned unless Q is returned in step 2(b)). In case the optional step 2 (b) is included, and if Q is returned in this step, we have $\sum_{j \in Q} j \leq \frac{k+1}{k} S$, which implies that Q is a (k+1)/k approximation of an optimal solution, as the sum of the elements of an optimal solution must be at least S.

In step (3)(a) it is checked whether Q (which has been found in the previous step to not be a solution) leads to a (k+1)/k approximation by adding elements from J_1. In step (3)(b) of the procedure each replacement increases the sum of elements in Q by less than $\frac{1}{k}S$, and thus if S is reached or exceeded by such a replacement the algorithm returns a (k+1)/k approximation of its optimal solution. The same happens in case the $AddJ1$ procedure adds elements to Q, as noted above. The main observation here is that replacing an element within Q from a set J_i (with $i \in I$) with another element from the same set J_i results in a difference between the old and new sum of elements of Q of less than $\frac{1}{k}S$, and that it suffices to consider the largest n_i elements from J_i when replacing elements from J_i in Q to make sure a (k+1)/k approximation is found. The existence of such an approximation results from the previous checks, as we already know from Step (1) that the sum of the elements of Q' is at least $S - sumJ1$.

Thus, in case the configuration can result in a solution, either a (k+1)/k approximation is returned or a candidate which is best possible among solutions corresponding to the considered configuration is recorded if it is the best candidate found so far. Otherwise, the procedure returns $\emptyset$.

We next present the main algorithm:

$\underline{MinSubsetSum(J, S, k)}$

1. Let $SumJ_1 = \sum_{j \in J_1} j$. If $SumJ_1 \geq S$, then return $AddJ1(\emptyset)$.

2. If $|J_l| \geq 1$, then $K_{best} = min_1(J_l)$, else $K_{best} = J$;
3. $T = (0, 0, \ldots, 0)$
4. $X = RecursionMinSubsetSum(T, 2)$
5. if $X \neq \emptyset$ return X, else return K_{best}

Here, the procedure $RecursionMinSubsetSum(T, i)$ uses a subroutine $ConditionMinSubsetSum(T, i)$, which returns *true* if increasing the value of n_i in T results in a *configuration of interest*, that is, in a configuration that fulfills Condition 2 of Lemma 2.1, and which returns *false* otherwise. $RecursionMinSubsetSum(T, i)$ is given as follows (here, we denote with $|X|$ the number of elements in a set X):

$RecursionMinSubsetSum(T, i)$
(1) $T1 = copy(T)$; //it is important to insure that T is not changed;
(2) If ($i < k$) and if ($ConditionMinSubsetSum(T, i + 1) == true$) {
(3) X=RecursionSubsetSum(T, i+1);
If ($X \neq \emptyset$) return X;
}
(4) While ($n_i <= nmax[i]$){
(5) If ($|J_i| \geq n_i + 1$) and if ($ConditionMinSubsetSum(T1, i) == true$)
(6) Replace n_i with $n_i + 1$ in $T1$; (also, update n_i to $n_i + 1$)
(7) X=CheckConfiguration($T1$);
(8) If ($X \neq \emptyset$) return X;
(9) If ($i < k$) and if ($ConditionMinSubsetSum(T1, i+1) == true$){
(10) X=RecursionSubsetSum($T1$, i+1);
(11) If $X \neq \emptyset$ return X;
}
}
(12) Else $n_i = nmax[i] + 1$; // exit the while loop
} // end while
return $\emptyset$
}

Since Condition 1 of Lemma 2.1 depends on the problem instance, it seems necessary to check all configurations which fulfill Condition 2. and for which $\sum_{i \in \{2,\ldots,k\}} n_i \geq 2$, as the case where $\sum_{i \in \{2,\ldots,k\}} n_i \leq 1$ has been checked in step 3. Together with Lemmas 2.1 and 2.2 this implies that the procedure $MinSubsetSum$ returns a $(k + 1)/k$ approximation of the optimal solution,

which we state in the following proposition.

Proposition 2.3
The worst-case approximation factor of $MinSubsetSum(J, S, k)$ when addressing the minimum subset sum problem with positive integers J and target sum S is $(k+1)/k$.

Proof: In Steps 1 $MinSubsetSum$ checks whether there are solutions formed only by elements from J_1, and if this is true, it returns a $(k+1)/k$ approximation of an optimal solution (because the sum of the elements of the returned solution is less than $\frac{k+1}{k}S$). In Step 2, $Kbest$ is initialized to be either the smallest element from J_l if it exists or the whole set J (any solution will be better than the whole set J). The recursion in Step 4 of $MinSubsetSum$ generates all configurations of interest T and calls $CheckConfiguration(T)$ each time a new configuration of this type is found. In Step (3) of $RecursionMinSubsetSum(T, i)$, in case increasing n_{i+1} in T results in a configuration of interest, T is passed unchanged deeper into the recursion, insuring that all configurations of interest with the first i elements of T are considered. At this time no element of T that has an index of $i+1$ or greater has been changed within the initial configuration (passed by the call from Steo 4 of $MinSubsetSum$), and thus all elements of T that have such indices, i.e., $n_{i+1}, \ldots, n_k$ have values 0 before this recursion call.

If in Step (2) $ConditionMinSubsetSum(T, i+1)$ returns $false$, this implies that $ConditionMinSubsetSum(T, i+q)$ for any $q > 1$ also would return false, and that increasing any n_h with $h \geq i+1$ in T does not result in a configuration of interest. Intuitively, this is because if adding a task X to a set H with configuration T does not result in a set that has a configuration of interest (i.e. from which no task can be removed while being sure that a solution results while reasoning only based on the definintions of the sets J_i $(i \in I)$), then adding a task Y that is larger than X to the same set will also not result in a set that has a configuration of interest. This is because if the task that can be removed from $H \cup \{X\}$ is from the initial set H, then the same task can be removed from $H \cup \{Y\}$. In this context also note that, since $k > 0$, and since all elements of T that have indices of at least $i+1$ are 0, $ConditionMinSubsetSum(T, i+1)$ can only return $false$ if T has nonzero elements with indices that are smaller that $i+1$.

In Steps (4) to (12) of $RecursionMinSubsetSum(T, i)$, increasing n_i in T in order to generate other configurations of interest is considered. Any

new generated configuration of interest is checked in Step (7), and passed into deeper levels of the recursion if new configurations of interest can be generated in this way. If CheckConfiguration in Step (7) and the recursion return the empty set the search is continued If a $(k+1)/k$ approximation of the optimal solution is found, it is returned.

According to Lemmas 2.1 and 2.2 $MinSubsetSum(J, S, k)$ either returns an optimal solution as the best found candidate solution or a $\frac{k+1}{k}$ approximation thereof. $\triangle$

The time complexity of the procedure depends on how many configurations of interest are generated in Step 4 of $MinSubsetSum(J, S, k)$, i.e., on how many such configurations exist. Within $MinSubsetSum(J, S, k)$ the procedure $CheckConfiguration(T)$ runs in linear time. Thus, if h is the number of configurations which fulfill Condition 2 of Lemma 2.1, the time complexity of $MinSubsetSum(J, S, k)$ is $O(h|J|)$.

Another way of finding the configurations of interest is, of course, writing out a procedure for each k that sequentially checks all configurations of interest (which fulfill the conditions from Lemma 2.1), which results in linear-time algorithms (if the number of configurations of interest, which is the maximum number of times $CheckConfiguration$ is called, is considered constant). In our opinion this becomes tedious and error-prone as k increases to 5 or more (as we shall see in Section 4, there are 22 configurations in this case). In Grigoriu and Briskorn [2] this was done for $k = 4$.

An easy to see upper bound for the number of configurations which fulfill Condition 2 of Lemma 2.1 is k^{k-1}, since each n_i $(i \in I)$ can only take integral values between 1 and k, and since there are $k-1$ elements of I. Thus, if k is considered constant the algorithm has a linear time complexity of $O(k^{k-1}n)$. This upper bound (k^{k-1}) is very far from being tight, however, as we shall see in Section 4, where we determine the number of configurations for various values of k computationally. We give a somewhat more careful analysis of the time complexity for the algorithms we propose for subset sum in Section 3, where we analytically obtain an upper bound for the number of configurations that must be checked for that problem, which is also very far from being tight. Determining the number of configurations computationally can be done by counting configurations of interest instead of searching for $k+1/k$ approximations in a procedure that uses recursion in a similar way to $MinSubsetSum$. A global variable num_tuples can be initialized to 0 and a tuple T set to be the tuple $(0, 0, ..., 0)$ with $k-1$ elements before entering the

recursion as shown in Step 4 of $MinSubsetSum$. Within the recursion $X = CheckConfiguration(T1)$ in Step (7) of $RecursionMinSubsetSum$ needs to be replaced with a statement that increases a global variable that represents the number of configurations of interest found so far, i.e. $num\_tuples = num\_tuples + 1$. Also, $RecursionMinSubsetSum$ does not need to return anything in this context, and adequate changes to Steps (3),(10), (11) need to be made, while Step (8) needs to be removed. After the recursion ends, all configurations of interest have been generated and counted, and the value of num_tuples contains the maximum number of tuples that need to be checked for the considered value of k.

3. Linear-time approximation algorithms for the subset sum

We next address the subset sum problem with the purpose of describing a family of linear-time algorithms with worst-case approximation ratios of $k/(k+1)$. Here, given a (multi)set J of positive integers, and a target sum S, we aim to find subsets Q of J with element sums that do not exceed S and are either closest possible to S (and thus Q is an optimal solution) or are at least $\frac{k}{k+1}S$, in which case Q is a $\frac{k}{k+1}$ approximation of an optimal solution, as the elements of an optimal solution can not add up to more than S .

We use an approach that is similar to the one used in the previous section, and split J into multiple subsets. These subsets are created as follows when addressing the subset sum problem with an input multiset of items (positive integers) J and target sum S. Let $S' = \frac{k}{k+1}S$. We assume that $J \subseteq [0, S]$, as items that are greater than S can not be part of a solution. If $J \not\subseteq [0, S]$ items from J that are not at most S can be removed in a preliminary step. The subsets we use are:

- $J_1 = J \cap [0, \frac{S'}{k+1}]$
- $J_i = J \cap (\frac{(i-1)S'}{k+1}, \frac{iS'}{k+1}]$

It is clear that any subset Q of J the items of which add up to at least S' and at most S is a $\frac{k}{k+1}$ approximation of an optimal solution, as the items of an optimal solution can not add up to more than S. Also, if no solution exists the items of which add up to at least S', the optimal solution can not have this property either, and thus candidate sets that add up to less than S' must also be considered. Among these, it suffices to store at any time the best candidate set found thus far.

Like for minimum subset sum we use subroutines $AddJ_1$ and $Candidate$.

The subroutine $AddJ_1(K)$ adds elements from J_1 to a set of jobs $K \subseteq J \setminus J_1$ with $S' - \sum_{j \in J_1} j \leq \sum_{j \in K} j \leq S$ while checking (in arbitrary order) for each element from J_1 whether it can be added to K without causing the sum of elements in K to exceed S, in which case the element is added to K (else the element is not added). Then, $AddJ_1(K)$ returns the resulting set K, which has the property that the sum of its elements is at least S' and at most S, as all elements of J_1 are at most $\frac{1}{k}S'$.

The second subroutine $Candidate(K, Kbest)$ updates the best found solution $Kbest$ with a newly found solution K if K is better than $Kbest$.

We use the same concept of configuration that was used in the previous section, while the term solution from that definition refers to solutions of subset sum, i.e. sets $Q \subseteq J$ with a sum of elements that is at most S.

Given a tuple $T = (n_2, n_3, \ldots, n_k)$, the best solution found so far, K_{best} (passed by reference), k, S', and assuming the procedure can access K_{best} (the best candidate solution found so far), k, S' the sets $J_1, J_2, \ldots, J_k$, S, and $SumJ_1$, the sum of all elements from J_1, which is calculated before calling it, the procedure we propose works as follows:

CheckConfigSubsetSum(T){
(1) If for some $i \in \{2, 3, \ldots, k\}$, J_i does not contain n_i elements return $\emptyset$;
Let $I = \{2, 3, \ldots, k\}$, $Q = \cup_{i \in I} min_{n_i} J_i$ and $Q' = \cup_{i \in I} max_{n_i} J_i$;
(2) If $\sum_{j \in Q} j > S$ return $\emptyset$;
(3) If $\sum_{j \in Q'} j + SumJ_1 \geq S'$ {
(4) (a) If $\sum_{j \in Q} \geq S' - SumJ1$, return $AddJ_1(Q)$;
(b) For $i \in I$ { Iteratively replace in Q elements of $min_{n_i} J_i$ with elements of $max_{n_i} J_i$ until the sum of elements in Q reaches or exceeds S', in which case return $AddJ_1(Q)$, or until all elements from $min_{n_i} J_i$ are replaced in Q.
}
(c) return $AddJ_1(Q')$;
}
(5) Else candidate($Q' \cup J_1$, K_{best});
(6) return $\emptyset$;
}

The procedure CheckConfigSubsetSum(T, K_{best}, k, S') returns the empty set in case no solutions with configuration T exist in step (2). If such solutions exist, it returns a $(k+1)/k$ approximation of the optimal solution in case a solution with a sum of elements at least S' and at most S with configuration

T exists, else, i.e. if all solutions with configuration T have element sums that are smaller than S', the solution with configuration T with the maximum sum of elements is considered and stored in case it is the best solution found so far in Step (5). Note that in Step 4 (b) each time an element is exchanged the sum of elements of the resulting set Q can not exceed S, since the difference between two elements of any subset J_i is less than $S'/(k+1)$ and since before any exchange the sum of elements in Q is less than S' (as exchanges take place only until the sum of elements in Q reaches or exceeds S'). If the procedure did not return before Step (4)(c) it means that $\sum_{j \in Q'} < S'$, and we have checked in Step (3) that Q' can form a solution with elements from J_1. Thus, in Step (4)(c) a solution with an element sum at least S', which is also $\frac{k+1}{k}$ approximation of the optimal solution, is returned.

We next state a property that all configurations of solutions must have.

Lemma 3.1
Only configurations $(n_2, \ldots, n_k)$ with

$$\sum_{i=2}^{k} n_i(i-1) \leq k \tag{1}$$

can correspond to solutions.

Proof: Suppose that a configuration $(n_2, \ldots, n_k)$ of a solution Q does not fulfill Inequality (1). Then we have $\sum_{i=2}^{k} n_i(i-1) \geq k+1$. Also, $\sum_{j \in Q} j \geq \sum_{i=2}^{k} min_{n_i} J_i > \sum_{i=2}^{k} \frac{(k-1)n_i}{k+1} S = \frac{\sum_{i=2}^{k}(k-1)n_i}{k+1} S \geq \frac{k+1}{k+1} S$, a contradiction, since we assumed that Q is a solution. $\triangle$

We shall call the configurations that fulfill Inequality (1) *feasible* configurations. If k is a constant, the following procedure is a linear-time approximation algorithm for subset sum, assuming the number of steps to find all configurations that fulfill Inequality (1) runs in O(number of feasible configurations). The number of feasible configurations can be upper bounded by $2 \cdot (k-1)^{\lfloor k/2 \rfloor}$, since each af the $k-1$ elements of a configuration tuple except for n_2 can only take values between 0 and $\lfloor k/2 \rfloor$ by Inequality (1), and the number of configurations where $n_2 \geq \lfloor \frac{k}{2} \rfloor$ is at most the number of values of n_2 within this range ($\lceil \frac{k}{2} \rceil$) times the number tuples $(n_3, \ldots n_k)$ which together with an $n_2 \geq \lfloor \frac{k}{2} \rfloor$ can form a feasible configuration. In this second number we must have for $i \geq 3$ that $n_i \leq \lfloor (k - \lfloor \frac{k}{2} \rfloor)/2 \rfloor = \lfloor \frac{\lceil k/2 \rceil}{2} \rfloor \leq \lceil k/4 \rceil$,

else Inequality (1) is not fulfilled. As there are $k-2$ elements of the tuple $(n_3, \ldots n_k)$, which can take values between 0 and $(\lceil \frac{k}{4} \rceil$, the number h of feasible configurations with $n_2 \geq \lfloor k/2 \rfloor$ fulfills $h < \lceil \frac{k}{2} \rceil \cdot (k-2)^{\lceil \frac{k}{4} \rceil}$, which is less than $(k-1)^{\lfloor k/2 \rfloor}$, since $k \geq 4$. We consider $k \geq 4$, since linear-time approximation algorithms similar to the ones we propose with $k = 3$ and $k = 4$ have been presented in Kellerer et al. [5]. Thus an algorithm which addresses all tuples and needs $O(1)$ steps on average to find each feasible configuration has a time complexity of $O(n * (k-1)^{\lfloor k/2 \rfloor})$. We conjecture that a more careful analysis can yield a possibly significantly better time complexity, however, this suffices for an argument that the proposed algorithms are linear if k is constant .

To generate all feasible configurations one can start with the configuration $(0, 0, \ldots, 0)$ and increase each element via a recursive search, as we show next:

SubsetSum(J,S,k){
(1) $S' = k/(k+1)S$;
Generate the sets J_1, $J_2, \ldots, J_k$;
Let $SumJ_1 = \sum_{j \in J_1} j$;
Record the number of elements in each J_i in a visible vector: $nmax[i] = |J_i|$;
(2) $n_i = 0$ for all $i \in \{2, \ldots k\}$;// this forms the tuple T
(3) $K_{best} = \emptyset$;
(4) X=RecursionSubsetSum$(T, 2)$;
(5) If $(X \neq \emptyset)$ return X;
(6) Else return K_{best};
}

Here, the procedure RecursionSubsetSum from Step (4) uses a subroutine $ConditionSubsetSum(T, i)$, which returns *true* if increasing the value of n_i in T results in a configuration that fulfills Inequality (1) and which returns *false* otherwise. It is given as follows:

RecursionSubsetSum(T, i){
(1) $T1 = copy(T)$; //it is important to insure that T is not changed;
(2) If ($i < k$ and $ConditionSubsetSum(T, i+1)$) {
(3) X=RecursionSubsetSum(T, $i+1$);
If $(X \neq \emptyset)$ return X;

```
    }
(4) While (n_i <= nmax[i]) {
(5)      If there are at least n_i+1 elements in J_i and if (ConditionSubsetSum(T, i))
{
(6)           Replace n_i with n_i + 1 in T1; (also, update n_i to n_i + 1)
(7)           X=CheckConfigSubsetSum(T1);
(8)           If (X ≠ ∅) return X;
(9)           If (i < k and ConditionSubsetSum(T1, i + 1)) {
(10)             X=RecursionSubsetSum(T1, i+1);
(11)             If X ≠ ∅ return X;
              }
         } // end If from step (5)
(12)     Else n_i = nmax[i] + 1; // will exit the while loop
   } // end while
return ∅
}
```

RecursionSubsetSum does not change its input parameter T, even though it is passed on to deeper recursion levels in Step (3).

Within the while loop that starts in Step (4), for the i received as an argument by RecursionSubsetSum, all n_i-s with $n_i \leq nmax[i]$ that fulfill Inequality (1) together with (a copy of) the passed along tuple are checked in Step (7). The tuple T received as an argument in the recursion has been checked before entering the recursion (also in Step (7)). The initial tuple made only of zeroes is considered when K_{best} is set to $\emptyset$ in the calling procedure SubsetSum (in its Step (3)). With other words, each time a new tuple is created (this happens only in Step (6) of the procedure RecursionSubsetSum), it is checked immediately afterwards, before it is passed along deeper into the recursion. If a solution which is a $(k+1)/k$ approximation is found, it is immediately passed back to the calling procedure which then immediately passes it back to its calling procedure until it can be returned by SubsetSum as a final result.

If the condition from Step (5) of RecursionSubsetSum is not fulfilled, then replacing n_i with any greater number in $T1$ does not result in a feasible configuration, and therefore, it is not necessary to consider increasing n_i in $T1$ any further. This also implies that it is not necessary to consider increasing values of n_j with $j > i$ in $T1$ any further, as this would also result in $T1$ not fulfilling Inequality (1).

Thus, after being called by SubsetSum, the procedure RecursionSubsetSum checks all feasible configurations (unless they require more elements from a set J_i than the set contains) by using the procedure CheckConfigSubsetSum, and as a consequence of the properties of CheckConfigSubsetSum on which we elaborated above, SubsetSum returns a $k/(k+1)$ approximation of an optimal solution. SubsetSum also runs in linear time in case k is considered a constant.

It may be better in steps (2) and (9) to also check whether increasing n_{i+1} in T and respectively in $T1$ results in a feasible configuration, in order to prevent unnecessary recursion steps. Here, we have a tradeoff between having a lot of useless checks and cutting off a big part of the recursion tree.

4. Remarks on the time complexity

In order to determine the maximum number of tuples that may be necessary to check in order to ensure a $\frac{k}{k+1}$ approximation for subset sum, we ran a recursive procedure that generates and counts all tuples that fulfill Inequality (1), and obtained the results listed in Table 2.

We also ran a procedure to determine the number of tuples that may be necessary to check in order to solve minimum subset sum with a worst-case approximation factor of $\frac{k+1}{k}$ (and an accuracy of $\frac{1}{k}$) using the method from section 2, that is, the tuples that fulfill Condition 2 of Lemma 2.1, and obtained the results from Table 1. In order to get an insight on how the number of configurations that need to be checked grows as k grows we calculated the values $nt(k+10)/k$ where it was possible using our results and included them in Tables 1 and 2.

The procedures are very likely to stop before checking all tuples, depending on when a solution with the desired acurracy is found, and depending on whether the subsets J_i of each instance contain enough jobs to allow for the building of sets that correspond to all configurations. In order to have enough jobs in each set J_i to accomodate all configurations, a problem instance must have at least $\lceil k \rceil$ jobs in J_2, $\lceil k/2 \rceil$ jobs in J_3, and $\lceil k/(i-1) \rceil$ jobs in J_i for $i \in \{4, \ldots, k\}$, and thus it is not necessary for J to have very many elements to fulfill this condition.

We note that the growth rate of the number of configurations that need to be checked when k grows by 10 becomes smaller as k grows. The results suggest that our algorithm can be used to compute solutions with worst-case accuracies such as 0.02 or even 0.015 in reasonable time, especially if

k	5	10	20	30	40	50	60	70	80
$\epsilon = \frac{1}{k}$	0.2	0.1	0.05	0.0(3)	0.025	0.02	0.01(6)	0.0143	0.125
$nt(k)$	22	171	3085	31023	227822	1353100	6874382	30941870	126345140
$\frac{nt(k)}{nt(k-10)}$			18.042	10.056	7.344	5.94	5.0805	4.5010	4.0833
time(sec)	< 0.01	< 0.01	0.02	0.22	2.01	14.31	89.1	449.26	2081.56

Table 1: Maximum number of tuples ($nt(k)$) that may need to be checked for minimum subset sum, which were obtained computationally for various values of k.

k	5	10	20	30	40	50	60	70	80
$\epsilon = \frac{1}{k+1}$	0.1(6)	0.(09)	0.04762	0.03226	0.0244	0.019609	0.0164	0.01409	0.01235
$nt(k)$	17	137	2712	28627	215306	1295969	6639347	30053952	123223637
$\frac{nt(k)}{nt(k-10)}$			19.8	10.56	7.5212	6.02	5.1232	4.52	4.1001
time(sec)	< 0.01	< 0.01	0.01	0.21	1.79	13.21	82.49	425.88	1902.8

Table 2: Maximum number of tuples ($nt(k)$) that may need to be checked for subset sum, which were obtained computationally for various values of k

parallelization is used to divide the load on multiple processors.

The times in seconds for generating tuples are listed in the fourth rows of Tables 1 and 2, and we note that up to $k = 40$ less than 2 seconds are needed to generate all tuples. The experiments were performed on an Intel Core I7-6500U with 8 GB RAM.

Parallel versions of the algorithms that split the work on any number of processors can be derived by assigning a set of configuration prefixes to each processor. For example, if there are two processors, one can check configurations with $n_2 \in \{2h | h \in \mathbb{N}, 2h \leq k\}$, and the other configurations with $n_2 \in \{2h + 1 | h \in \mathbb{N}, 2h + 1 \leq k\}$.

References

[1] G. Gens and E. Levner. Fast approximation algorithm for job sequencing with deadlines. *Discrete Applied Mathematics*, 3:313 – 318, 1981.

[2] L. Grigoriu and D. Briskorn. Scheduling jobs and maintenance activities subject to job-dependent machine deteriorations. *Journal of Scheduling, doi: 10.1007/s10951-016-0502-0*, 2016.

[3] M. Güntzer and D. Jungnickel. Approximate minimization algorithms for the 0/1 knapsack and subset-sum problem. *Operations Research Letters*, 26:55–66, 2000.

[4] A. Janiak and M. Kovalyov. Single machine scheduling subject to deadlines and resource dependent processing times. *European Journal of Operational Research*, 94:284–291, 1996.

[5] H. Kellerer, R. Mansini, and M. Speranza. Two linear approximation algorithms for the subset-sum problem. *European Journal of Operational Research*, 120:289–296, 2000.

[6] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, 2004.

[7] M. Kovalyov. A rounding technique to construct approximation algorithms for knapsack and partition type problems. *Applied Mathematics and Computer Science*, 6:101 – 113, 1996.