Research Article

Samer Nofal*

Listing all delta partitions of a given set: Algorithm design and results

https://doi.org/10.1515/comp-2024-0011
received October 15, 2023; accepted May 21, 2024

Abstract: Let \( a \) be a set of \( n \) elements and \( \delta \) be a nonnegative integer. A \( \delta \)-partition of \( a \) is a set of pairwise disjoint nonempty subsets of \( a \) such that the union of the subsets is equal to \( a \) and every subset has a size greater than \( \delta \). We formulate an algorithm for computing all \( \delta \)-partitions of a given \( n \)-element set and show that the algorithm runs in \( O(n) \) space and \( O(n) \) delay time between any two successive outputs of \( \delta \)-partitions of the given set. An application of the notion of \( \delta \)-partitions is illustrated in the following scheduling problem. Suppose a factory has \( n \) machines and \( m \leq n \) jobs to complete daily. Every job can be accomplished by operating at least \( \delta + 1 \) machines. A machine cannot work on multiple jobs simultaneously. According to a utilization policy of the factory’s management, no machine is allowed to be idle, so all machines should be running on some job. Find a daily schedule of the factory’s machines satisfying all the mentioned constraints. Let \( a \) be the set of the factory’s machines. Then, an \( a \)'s \( \delta \)-partition with \( m \) subsets is a legal schedule if every subset (in the \( \delta \)-partition) includes exclusively \( \delta + 1 \) or more machines that run on the same job.

Keywords: constrained set partitions, listing algorithm, delay time

1 Introduction

Scheduling problems are ubiquitous in the real world; see, e.g., previous studies [1–9]. Therefore, exact-solution procedures for scheduling problems have been studied heavily over the years, where seminal methods such as linear programming, integer programming, and constraint programming were designed and analyzed prevalently; likewise, approximate-solution procedures for scheduling problems have been worldwide addressed substantially where influential approaches such as heuristic and metaheuristic algorithms were proposed and other approximation methods such as online algorithms were intensively studied; for further information, we refer the reader to previous studies [10–17], for instance.

In this article, we design and analyze an exact algorithm for computing a given set’s so-called delta partitions. An application of this algorithm can be for a scheduling problem. To illustrate the connection between a scheduling problem and our algorithm of listing all delta partitions of a given set, we first define delta partitions. Let \( a \) be a set of \( n \) elements and \( \delta \) be a nonnegative integer. A partition of \( a \) is a set of pairwise disjoint nonempty subsets of \( a \) such that the union of the subsets is equal to \( a \). A \( \delta \)-partition of \( a \) is a partition of \( a \) such that the size of every subset in the partition is greater than \( \delta \). For instance, the partitions of \( \{a_1, a_2, a_3\} \) are \( \{\{a_1, a_2\}, \{a_3\}\}, \{\{a_1, a_3\}, \{a_2\}\}, \{\{a_1\}, \{a_2, a_3\}\}, \{\{a_1\}, \{a_2\}, \{a_3\}\} \). In contrast, the 1-partition of \( \{a_1, a_2, a_3\} \) is \( \{\{a_1, a_2, a_3\}\} \), which is equal to the 2-partition of \( \{a_1, a_2, a_3\} \). Now, take the following scheduling problem. Suppose a factory has \( n \) machines and \( m \leq n \) jobs to complete daily. Every job can be accomplished by operating at least \( \delta + 1 \) machines. A machine cannot work on multiple jobs simultaneously. According to a utilization policy of the factory’s management, no machine is allowed to be idle, so all machines should be running on some job. Find a daily schedule of the factory’s machines satisfying all the mentioned constraints. Let \( a \) be the set of the factory’s machines. Then, an \( a \)'s \( \delta \)-partition with \( m \) subsets is a legal schedule if every subset (in the \( \delta \)-partition) includes exclusively \( \delta + 1 \) or more machines that run on the same job.

In Section 2, we discuss related work. In Section 3, we demonstrate that all \( \delta \)-partitions of an \( n \)-element set, \( a \), can be listed in \( O(n) \) space and \( O(n) \) delay time between any two successive outputs of \( \delta \)-partitions of \( a \). Finally, we conclude the article in Section 4.

* Corresponding author: Samer Nofal, Department of Computer Science, German Jordanian University, Amman, Jordan, e-mail: samer.nofal@gju.edu.jo

Open Access. © 2024 the author(s), published by De Gruyter. This work is licensed under the Creative Commons Attribution 4.0 International License.
2 Related work

The study of computing set partitions has received significant attention from combinatorial algorithms research communities for a long time until now; see, for example, previous studies [19–24]. Perhaps the earliest work on formal algorithms for computing set partitions goes back to the study of Hutchinson [25], where the partitions of a set with \( n \) elements are represented by certain \( n \)-tuples of positive integers and algorithms were described, which generate the \( n \)-tuples corresponding to all partitions of the given set, all partitions of the given set into \( m \) or fewer sets \( 1 \leq m \leq n \), and all partitions of the given set into exactly \( m \) sets \( 1 \leq m \leq n \).

Ehrlich [26] presented a loopless algorithm for listing all partitions of a given set with constant time between generating one set partition from the previous one. Kaye [27] arranged set partitions in a Gray code list so that each partition is obtained from its immediate predecessor by changing the class of exactly one element. Ruskey [28] developed a Gray code list of the partitions of a set into a fixed number of blocks where each partition is produced in \( O(1) \) amortized time. Er [29] gives a recursive algorithm for generating all partitions of a set with a constant time (on average) per partition generated. The practical performance of the algorithm of Er [29] was enhanced further in the study of Djokic et al. [30]. Mansour and Nassar [31] presented another combinatorial Gray code for the partitions of a given set; also, they constructed a different loopless algorithm for generating all partitions of a given set, which offers a constant time between successive partitions in the construction process. In the study of Kokosski [32], a parallel algorithm is presented for generating all \( n \)-element set partitions where each partition has at least \( m \) parts and such that each partition is generated in \( O(m) \) time. Likewise, Lee et al. [33] proposed another parallel algorithm for set partitioning with cost \( O(nB_n) \), where \( B_n \) is the number of all \( n \)-element set partitions. Williams [34] gives a greedy algorithm for generating set partitions where the algorithm begins with an initial partition and an ordered list of operations and then repeatedly creates a new partition by applying the first possible operation to the most recently created partition. Huemer et al. [35] presented Gray codes for restricted set partitions, what they called non-crossing set partitions. Researchers [36,37] studied Gray codes for a different family of restricted set partitions, what they called nonsetting set partitions.

3 Listing all \( \delta \)-partitions of a given set

Let \( a = \{a_1, a_2, a_3, \ldots, a_n\} \) be a set of elements and \( \delta < n \) be a non-negative integer. We characterize partitions and \( \delta \)-partitions of \( a \) using functions. We say that \( \Gamma : a \rightarrow \{0, 1, 2, \ldots, n\} \) corresponds to a partition of \( a \) (or interchangeably, we say that \( \Gamma \) is a partition of \( a \)) if and only if for all \( x \in a, \Gamma(x) \neq 0 \). Thus, if \( \Gamma \) corresponds to a partition of \( a \), then

\[
\bigcup_{i \in \text{range}(\Gamma)} \{x \in a | \Gamma(x) = i\}
\]

is a partition of \( a \); observe that \( \text{range}(\Gamma) \) is the image set of \( \Gamma \). Likewise, we say that \( \Gamma \) corresponds to a \( \delta \)-partition of \( a \) (or interchangeably, we say that \( \Gamma \) is a \( \delta \)-partition of \( a \)) if and only if \( \Gamma \) corresponds to a partition of \( a \) and for all \( i \in \text{range}(\Gamma) \), \( |\{x \in a | \Gamma(x) = i\}| > \delta \).

Note that in constructing a \( \delta \)-partition of \( a \), for all \( x \in a \), \( \Gamma(x) \) is initially set to 0. But then, later in the construction process, every element of \( a \) must be assigned to a positive integer, as we elaborate throughout the article. In other words, in the process of constructing a \( \delta \)-partition of \( a \), for all \( x \in a \), we denote by \( \Gamma(x) = 0 \) the initial status of \( x \) before eventually assigning \( x \) to some positive integer.

Let \( \Gamma_1 \) and \( \Gamma_2 \) be partitions of \( a \). We say that \( \Gamma_1 \) and \( \Gamma_2 \) are identical if and only if

\[
\forall \exists k \text{ such that } \{x | \Gamma_1(x) = j\} = \{x | \Gamma_2(x) = k\}. \tag{1}
\]

If two \( \delta \)-partitions are identical, then it suffices to compute one of them. To this end, we use a rule for mapping elements of \( a \) to positive integers as specified in the following proposition where we prove that to construct a partition \( \Gamma \) of a given set \( a = \{a_1, a_2, a_3, \ldots, a_n\} \), \( a_1 \) needs to be assigned to 1 only and, every \( (a_i)_{i>1} \) needs to be assigned to a positive integer not greater than \( m + 1 \), where \( m \) is a positive integer that satisfies two conditions: (1) for some \( k < i \) it holds that \( m = \Gamma(a_k) \), and (2) for all \( j < i \) it is the case that \( m \geq \Gamma(a_j) \).

Proposition 1. Let \( a = \{a_1, a_2, a_3, \ldots, a_n\} \) be a set of elements, \( \Gamma_1 \) be a partition of \( a \) with \( \Gamma_1(a_1) = \lambda \) for some \( a_i \) such that \( \lambda > \eta + 1 \), where

\[\eta = \begin{cases} 0 & \text{ if } i = 1 \\ m : \exists (a_k)_{i<j}, m \in \Gamma_1 \text{ with } \forall j < i, m \geq \Gamma_1(a_j) & \text{ if } i > 1. \end{cases}\]

Then, there exists \( \Gamma_2 \) identical to \( \Gamma_1 \) such that for every \( (a_i)_{i<j} \) it is the case that \( \Gamma_2(a_i) = \Gamma_1(a_i) \) and \( \Gamma_2(a_j) = \eta + 1 \).

Proof. Let us construct \( \Gamma_2 \) from \( \Gamma_1 \) by the following steps:

1. \( \Gamma_2 \leftarrow \emptyset \);
2. for each \( x \) with \( \Gamma_1(x) = \lambda \), \( \Gamma_2 \leftarrow \Gamma_2 \cup \{(x, \eta + 1)\} \);
3. for each \( x \) with \( \Gamma_2(x) = \eta + 1 \), \( \Gamma_2 \leftarrow \Gamma_2 \cup \{(x, \lambda)\} \);
4. for each \( x \) with \( \Gamma_2(x) \notin \{(\lambda, \eta + 1), (x, \lambda)\} \), \( \Gamma_2 \leftarrow \Gamma_2 \cup \{(x, \Gamma_2(x))\} \).

Notably, this process produces \( \Gamma_2 \) from \( \Gamma_1 \) such that (1) holds, so \( \Gamma_2 \) and \( \Gamma_1 \) are identical. Due to step 2 in the process, \( \Gamma_2(a_i) = \eta + 1 \). However, we need to show that

\[
\forall (a_i)_{i<j} \Gamma_2(a_i) = \Gamma_1(a_i). \tag{3}
\]
Observe $\Gamma(a_i) = \eta + 1$ and $\Gamma_i(a_i) > \eta + 1$. Hence, according to (2), for all $(a_i)_{i < l}$

$$\Gamma_i(a_i) < \Gamma_i(a_i) < \Gamma_i(a_i).$$

(4)

Thus, we note that steps 2 and 3, in the process above do not apply to those elements that are in $(l_\alpha \in a : u < l)$; but step 4 applies to them, and so (3) holds.

As we are concerned with computing the $\delta$-partitions of a set, in the following two propositions, we specify the conditions under which a function $\Gamma$ has a superset $\Gamma' \supseteq \Gamma$ with $\Gamma'$ being a $\delta$-partition of the given set.

**Proposition 2.** Let $a = \{a_1, a_2, a_3, ..., a_n\}$ be a set of elements, $\delta < n$ be a non-negative integer, $\Gamma_i : a \rightarrow \{0, 1, 2, 3, ..., n\}$ be a function, and $\beta$ be a nonempty set such that

$$\beta = \{j > 0 : 0 < |\{x : \Gamma_i(x) = j\}| \leq \delta\}$$

(5)

with

$$|\{x : \Gamma_i(x) = 0\}| < \sum_{j \in \beta} \delta - |\{x : \Gamma_i(x) = j\}| + 1.$$  

(6)

Then, for all $\Gamma_2 \supseteq (\{x, j\} \in \Gamma_1 : j > 0)$, $\Gamma_2$ is not a $\delta$-partition of $a$.

**Proof.** Let $\Gamma_2 = \Gamma_1$, and

$$\pi = \{j > 0 : 0 < |\{x : \Gamma_2(x) = j\}| \leq \delta\},$$

(7)

such that

$$|\{x : \Gamma_2(x) = 0\}| < \sum_{j \in \pi} \delta - |\{x : \Gamma_2(x) = j\}| + 1.$$  

(8)

Now, update $\Gamma_2$ such that for each $y$ with $\Gamma_2(y) = 0$ do

1. $\Gamma_2(y) \leftarrow j$, for some $j \in \pi$;
2. $\pi \leftarrow \{j > 0 : 0 < |\{x : \Gamma_2(x) = j\}| \leq \delta\}.$

Having applied steps 1 and 2 to each $y$ with $\Gamma_2(y) = 0$, and given (8), we note that

$$0 < \sum_{j \in \pi} \delta - |\{x : \Gamma_2(x) = j\}| + 1.$$  

(9)

Now, assume that $\pi = \emptyset$. Then, we rewrite (9) as $0 < 0$, which is impossible. Therefore, $\pi \neq \emptyset$, and so (9) implies that

$$\exists j > 0 \text{ such that } 0 < |\{x : \Gamma_2(x) = j\}| \leq \delta.$$  

(10)

Hence, for all $\Gamma_2 \supseteq (\{x, j\} \in \Gamma_1 : j > 0)$, $\Gamma_2$ is not a $\delta$-partition of $a$.

**Proposition 3.** Let $a = \{a_1, a_2, a_3, ..., a_n\}$ be a set of elements, $\delta < n$ be a non-negative integer, $\Gamma_i : a \rightarrow \{0, 1, 2, 3, ..., n\}$ be a function, and $\beta$ be a nonempty set such that

$$\beta = \{j > 0 : 0 < |\{x : \Gamma_i(x) = j\}| \leq \delta\}$$

(11)

with

$$|\{x : \Gamma_i(x) = 0\}| = \sum_{j \in \beta} \delta - |\{x : \Gamma_i(x) = j\}| + 1.$$  

(12)

Then, for all $\Gamma_2 \supseteq \{\{x, j\} \in \Gamma_1 : j > 0\}$ it holds that

$$\exists y (\Gamma_i(y) = 0 \wedge \Gamma_i(y) = k \wedge k \notin \beta)$$

$$\Gamma_2$$ is not a $\delta$-partition of $a$.

**Proof.** Let $\Gamma_1 = \Gamma_1$. Rewrite (11) and (12) by replacing $\Gamma_i$ with $\Gamma_2$. Thus, for

$$\beta = \{j > 0 : 0 < |\{x : \Gamma_i(x) = j\}| \leq \delta\},$$

(13)

it holds that

$$|\{x : \Gamma_i(x) = 0\}| = \sum_{j \in \beta} \delta - |\{x : \Gamma_i(x) = j\}| + 1.$$  

(14)

Now, for some $y$ with $\Gamma_i(y) = 0$, set $\Gamma_i(y) \leftarrow k$ such that $k \notin \beta$. So, rewrite (15) as

$$|\{x : \Gamma_i(x) = 0\}| < \sum_{j \in \beta} \delta - |\{x : \Gamma_i(x) = j\}| + 1.$$  

(16)

Using Proposition 2, for all $\Gamma_2 \supseteq \{\{x, j\} \in \Gamma_1 : j > 0\}$, $\Gamma_2$ is not a $\delta$-partition of $a$. Recall, $\{\{x, j\} \in \Gamma_1 : j > 0\} \supseteq \{\{x, j\} \in \Gamma_1 : j > 0\}$. □

We now give our procedure listed in Algorithm 1. Let $a = \{a_1, a_2, a_3, ..., a_n\}$ be a set of elements, $\delta < n$ be a non-negative integer, and $\Gamma : a \rightarrow \{0, 1, 2, ..., n\}$ be a function such that for all $x \in a$, $\Gamma(x) = 0$. If Algorithm 1 is called with list_delta_partitions($\Gamma, a_i, \delta$), then the algorithm computes all $\delta$-partitions of $a$.

**Algorithm 1: list_delta_partitions($\Gamma, \rho, a_i, \delta$)**

1. $\Gamma(a_i) \leftarrow \rho$;
2. $\beta \leftarrow \{j > 0 : 0 < |\{x : \Gamma(x) = j\}| \leq \delta\}$;
3. if $\beta \neq \emptyset$ then
4. $\mu \leftarrow \sum_{j \in \beta} \delta - |\{x : \Gamma(x) = j\}| + 1$;
5. if $\mu > |\{x : \Gamma(x) = 0\}|$ then
6. return;
7. if $\mu = |\{x : \Gamma(x) = 0\}|$ then
8. foreach $k \in \beta$ do
9. list_delta_partitions($\Gamma, k, a_{i+1}, \delta$);
10. return;
11. if $i = n$ then
12. report that $\Gamma$ is a $\delta$-partition;
13. return;
14. Let $m \in \{x > 0\}$, report $\Gamma(a_k) = x$ and $\forall j \leq i, \, x \geq \Gamma(a_j)$;
foreach $k \leftarrow 1$ to $m + 1$ do

list_delta_partitions($\Gamma, k, a_{i+1}, \delta$);

return;

For instance, run Algorithm 1 to compute all 1-partitions of $a = \{a_1, a_2, a_3, a_4\}$. Initially, call

\[
\text{list_delta_partitions}((\{a_1, 0\}, (a_2, 0), (a_3, 0), (a_4, 0)), 1, a_1).
\]

Referring to line 15 in Algorithm 1, for $k = 1$, invoke

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 0), (a_3, 0), (a_4, 0)), 1, a_2, 1).
\]

Applying line 15 in Algorithm 1, when $k = 1$, run

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 1), (a_3, 0), (a_4, 0)), 1, a_3, 1).
\]

Following line 15 in Algorithm 1, for $k = 1$, apply

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 1), (a_3, 1), (a_4, 0)), 1, a_4, 1).
\]

Applying lines 11–13 in Algorithm 1, report that

\[
\{\{a_1, 1\}, (a_2, 1), (a_3, 1), (a_4, 1)\}\] is a 1-partition of $a$, and then backtrack to (19). Referring to line 15 in Algorithm 1, for $k = 2$, operate

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 1), (a_3, 1), (a_4, 0)), 2, a_2, 1).
\]

At this stage, as line 6 of Algorithm 1 is performed, backtrack to (19), then to (18). Executing line 15 in Algorithm 1, for $k = 2$, invoke

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 1), (a_3, 0), (a_4, 0)), 2, a_3, 1).
\]

Referring to line 8 in Algorithm 1, for $k = 2$, run

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 1), (a_3, 2), (a_4, 0)), 2, a_4, 1).
\]

Following lines 11–13 in Algorithm 1, report that

\[
\{\{a_1, 1\}, (a_2, 1), (a_3, 2), (a_4, 2)\}\] is a 1-partition of $a$, and then return to (22), (18), then back to (17). Applying line 15 in Algorithm 1, for $k = 2$, operate

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 0), (a_3, 0), (a_4, 0)), 2, a_2, 1).
\]

Referring to line 8 in Algorithm 1, when $k = 1$, execute

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 2), (a_3, 0), (a_4, 0)), 1, a_3, 1).
\]

Performing line 8 in Algorithm 1, for $k = 2$, operate

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 2), (a_3, 1), (a_4, 0)), 2, a_4, 1).
\]

Following lines 11–13, report that $\{\{a_1, 1\}, (a_2, 2), (a_3, 1), (a_4, 2)\}$ is a 1-partition of $a$, and then to (24). Referring to line 8 in Algorithm 1, for $k = 2$, run

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 2), (a_3, 0), (a_4, 0)), 2, a_3, 1).
\]

According to line 8 in Algorithm 1, for $k = 1$, call

\[
\text{list_delta_partitions}((\{a_1, 1\}, (a_2, 2), (a_3, 2), (a_4, 0)), 1, a_4, 1).
\]

Running lines 11–13 in Algorithm 1, report that

\[
\{\{a_1, 1\}, (a_2, 2), (a_3, 2), (a_4, 1)\}\] is a 1-partition of $a$, and next, return to (27), (24), then eventually back to (17). That completes the application of Algorithm 1.

In the next two propositions, we prove the correctness of Algorithm 1.

**Proposition 4.** Let $a = \{a_1, a_2, a_3, \ldots, a_n\}$ be a set of elements, $\delta < n$ be a non-negative integer, $\Gamma : a \rightarrow \{0, 1, 2, \ldots, n\}$ be a function such that for all $x \in a$ it holds that $\Gamma(x) = 0$, and let Algorithm 1 be invoked with

\[
\text{list_delta_partitions}(\Gamma, 1, a_1, \delta).
\]

If line 12 of Algorithm 1 is executed, then $\Gamma$ is a $\delta$-partition of $a$.

**Proof.** Let $\Gamma^{(i)}$ denote the mappings of $\Gamma$ at some state $i$ of Algorithm 1. From now on, whenever we say “state,” we mean a state of Algorithm 1. The algorithm enters a new state whenever line 1 is applied. Observe that the algorithm might report a $\delta$-partition at state $n + 1$; see line 11 in Algorithm 1. Thus, we focus on the states that lead to a $\delta$-partition. For the initial state, it is the case that

\[
\forall x \Gamma^{(i)}(x) = 0. \quad (29)
\]

And, for every $i \in \{0, 1, 2, 3, \ldots, n\}$ it holds that

\[
\Gamma^{(i+1)} = (\Gamma^{(i)} \cup \{(a_i+1, 0)\}) \cup \{(a_i+1, \rho)\},
\]

for some positive integer $\rho$, see line 1 in Algorithm 1. We need to prove that $\Gamma$ can grow to a $\delta$-partition for every state as described in Propositions 2 and 3. Starting with the case described in Proposition 2, we need to prove that for every state $i \in \{0, 1, 2, 3, \ldots, n\}$, it holds that
Let $a = \{a_1, a_2, a_3, \ldots, a_n\}$ be a set of elements, $\delta < n$ be a non-negative integer, $\Gamma : a \rightarrow \{0, 1, 2, \ldots, n\}$ be a function such that for all $x \in a$ it holds that $\Gamma(x) = 0$, and let Algorithm 1 be invoked with

\[ \text{list}_\Delta \text{p}_\Delta (\Gamma, 1, a_1, \delta). \]

Then, the algorithm computes every $\Delta$-partition of $a$.

**Proof.** Assume that $\varphi$ is a $\Delta$-partition of $a$, but $\varphi$ is not computed by the algorithm. Thus, for every state $s$

\[ \varphi \text{ is not identical to } \Gamma(s). \]

We consider the case where $\varphi$ is not identical to any $\Delta$-partition computed by the algorithm; otherwise, there is no need to compute such $\varphi$ as demonstrated in Proposition 1. Our purpose is to establish a contradiction with (41). Thus, we define

\[ \Lambda_i^{(r)} = \{1, 2, 3, \ldots, m + 1\} \]

to be the set of positive integers for some $a_i \in a$ at some state $r$ such that $m = 0$ if $i = 1$; for $i > 1$, $m = \Gamma^{(r)}(a_i)$ for some $k < i$ such that $m > \Gamma^{(r)}(a_{i-1})$; for all $j < i$. Additionally, we define $a_i^{(r)} \subseteq \Lambda_i^{(r)}$ to be the set of positive integers for some $a_i \in a$ at some state $l$ such that for every state $s$ it is the case that

\[ \forall a_i \in \Gamma(I)(a_i) \in a_i^{(s)} \iff \Gamma(s) \text{ is a } \Delta \text{-partition of } a. \]

Since $\varphi$ is a $\Delta$-partition of $a$, it is the case that for all positive integers $i$

\[ 1 \leq \varphi(a_i) \leq m + 1, \]

where $m = 0$ if $i = 1$; for $i > 1$, $m = \varphi(a_i)$ for some $k < i$ such that $m > \varphi(a_{i-1})$ for all $j < i$. Considering (42) and (44), we note that

\[ \exists s \forall a_i \varphi(a_i) \in \Lambda_i^{(s)}. \]

This is consistent with Proposition 1, implemented in the algorithm on lines 15–16. Referring to (43),

\[ \exists s \forall a_i \varphi(a_i) \in a_i^{(s)}; \]

otherwise, $\varphi$ is not a $\Delta$-partition as established in Propositions 2 and 3 and implemented on lines 2–10 in the algorithm. Following (43),

\[ \exists s \forall a_i \varphi(a_i) = \Gamma^{(s)}(a_i) \quad \text{and} \quad \Gamma^{(s)} \text{ is a } \Delta \text{-partition of } a. \]

This is consistent with Proposition 4, which shows the correctness of Algorithm 1. Subsequently,

\[ \exists s \varphi = \Gamma^{(s)} \quad \text{and} \quad \Gamma^{(s)} \text{ is a } \Delta \text{-partition of } a. \]

See the contradiction between (48) and (41). \qed
Now, we demonstrate the space complexity of Algorithm 1.

**Proposition 6.** Let $a = \{a_1, a_2, a_3, \ldots, a_n\}$ be a set of elements, $\delta < n$ be a non-negative integer, $\Gamma : a \rightarrow \{0, 1, 2, \ldots, n\}$ be a function such that $\Gamma(x) = 0$ for all $x \in a$, and let Algorithm 1 be invoked with list_delta_partitions($\Gamma, 1, a_1, \delta$). Then, Algorithm 1 runs in $O(n)$ space.

**Proof.** Besides the linear-space stack required to execute the recursion of Algorithm 1 and the linear space imposed by using the function $\Gamma$ inputted to the algorithm, additional space is required. Note, the set $\beta$ (line 2 in the algorithm) can be implemented instead as a function $\mathcal{B} : \{1, 2, 3, \ldots, n\} \rightarrow \{0, 1, 2, 3, \ldots, n\}$ such that in the initial state of the algorithm we set $\mathcal{B}(j) = 0$ for all $j$. Thus, we can update $\mathcal{B}(j) = \mathcal{B}(j) + 1$ in constant time whenever we map an element of $a$ to some subset $j$, see line 1 in the algorithm. Consequently, for any state of the algorithm, for any $j$, it holds that

$$\mathcal{B}(j) = |\{x : \Gamma(x) = j\}|.$$  

(49)

Now, we show that $\mu$ (line 4 in the algorithm) can be computed in constant time. Hence, in the initial state of Algorithm 1, we set $M = 0$. Whenever we apply $\mathcal{B}(k) = \mathcal{B}(k) + 1$ for some $k$, $M$ is updated subsequently (in constant time) as follows:

$$M = \begin{cases} M - 1 & \text{if } \mathcal{B}(k) > 1 \\ M + \delta & \text{if } \mathcal{B}(k) = 1 \end{cases}$$  

(50)

We now prove inductively that $M$ is equivalent to $\mu$ employed by Algorithm 1 at line 4. For the initial state of Algorithm 1, it is obvious that $M(0) = \mu(0)$. Next, we demonstrate that for all states $i$, it is the case that

$$M(i) = \mu(i) \Rightarrow M(i+1) = \mu(i+1).$$  

(51)

Suppose $M(i) = \mu(i)$. According to line 4 in the algorithm, we write

$$M(i) = \mu(i) = \sum_{j \leq \delta} \delta - |\{x : \Gamma^{(j)}(x) = j\}| + 1.$$  

(52)

If we set $\Gamma^{(i+1)}(x) = k$ for some $x \in a$ at some state $i + 1$ where $\mathcal{B}^{(i)}(k) > 0$, then $\mathcal{B}^{(i+1)}(k) > 1$. Therefore, using (50), we have

$$M^{(i+1)} = M^{(i)} - 1 = \mu^{(i)} - 1.$$  

(53)

Observe

$$|\{x : \Gamma^{(i+1)}(x) = k\}| = |\{x : \Gamma^{(i)}(x) = k\}| + 1.$$  

(54)

Thus,

$$M^{(i+1)} = \mu^{(i)} - 1 = -1 + \sum_{j \leq \delta} \delta - |\{x : \Gamma^{(j)}(x) = j\}| + 1.$$  

(55)

Using (54), (55) can be rewritten as

$$M^{(i+1)} = \sum_{j \leq \delta} \delta - |\{x : \Gamma^{(i+1)}(x) = j\}| + 1 + 1.$$  

(56)

For the second case of (50), if we set $\Gamma^{(i+1)}(x) = k$ for some $x \in a$ at some state $i + 1$ with $\mathcal{B}^{(i)}(k) = 0$, then $\mathcal{B}^{(i+1)}(k) = 1$. Therefore, using (50), we write

$$M^{(i+1)} = M^{(i)} + \delta = \mu^{(i)} + \delta.$$  

(57)

Observe

$$\mu^{(i+1)} = \sum_{j \leq \delta} \delta - |\{x : \Gamma^{(i+1)}(x) = j\}| + 1.$$  

(58)

Subsequently,

$$\mu^{(i+1)} = (\delta - |\{x : \Gamma^{(i)}(x) = k\}| + 1) + \mu^{(i)} = \delta + \mu^{(i)}$$  

(59)

$$M^{(i+1)} = M^{(i)} + \delta = M^{(i+1)}.$$  

Note that

$$|\{x : \Gamma^{(i+1)}(x) = k\}| = \mathcal{B}^{(i+1)}(k) = 1,$$  

(60)

Next, in the last proposition, we illustrate the time complexity of Algorithm 1.

**Proposition 7.** Let $a = \{a_1, a_2, a_3, \ldots, a_n\}$ be a set of elements, $\delta < n$ be a non-negative integer, $\Gamma : a \rightarrow \{0, 1, 2, \ldots, n\}$ be a function such that $\Gamma(x) = 0$ for all $x \in a$, and let Algorithm 1 be invoked with list_delta_partitions($\Gamma, 1, a_1, \delta$). Then, Algorithm 1 runs in $O(n)$ delay time between any two successive outputs of $\delta$-partitions of $a$.

**Proof.** Let $\Gamma$ be a $\delta$-partition listed by Algorithm 1 (line 12). If there are further $\delta$-partitions of $a$ not listed yet by the algorithm, then Algorithm 1 returns (line 13) to assign an element $a_i$ to another subset $\Gamma(a_i) < k \leq m + 1$ (see lines 15 and 16). Recall that $m + 1$ is the only empty subset because, according to line 14, $m$ is the greatest integer representing a nonempty subset among all nonempty subsets of the current partition $\Gamma$. Now, if $a_i$ is assigned to a nonempty subset $\Gamma(a_i) < k \leq m$, then $\beta$ (line 2) remains the same or becomes smaller, and consequently, $\mu$ (line 4) remains the same or becomes less; thus, the condition of line 5 remains false and subsequently line 6 is not executed, which means
that the algorithm proceeds to the next element $a_{i+1}$. But, if $a_i$ is assigned to the only empty subset $m + 1$, then $\beta$ might expand (see line 2) and so $\mu$ might increase (see line 4), which might make the condition of line 5 true and subsequently line 6 being executed; if line 6 is executed, then the algorithm returns to the caller instance of the procedure \texttt{list\_delta\_partitions} (line 16) to re-assign $a_i$ to another subset. But since the loop (at lines 15 and 16) has finished (recall we were in the last round when $k = m + 1$), we return (line 17) further to the earlier instance of the procedure \texttt{list\_delta\_partitions} where we were processing $a_{i-1}$. After processing $a_{i-1}$, we continue to $a_i$ to start the loop at lines 15 and 16 by assigning $a_i$ to a nonempty subset $k = 1$. In summary, for each $(a_i)_{i \in [m]}$, at most, two subset assignments are tried by Algorithm 1 until the next $\delta$-partition of $a$ is found. So far, we have discussed the cases in which the algorithm is invoked at line 16. If the procedure \texttt{list\_delta\_partitions} is invoked at line 9, then line 6 will never be executed in the subsequent stages. This is because, at line 9, we invoke the procedure \texttt{list\_delta\_partitions} to assign an element of $a$ to a nonempty subset exclusively (see line 8); note that we illustrated earlier that line 6 might be run if and only if an element is assigned to an empty subset.

4 Conclusion

We formulated and analyzed an algorithm listing all $\delta$-partitions of a given $n$-element set. We discussed the correctness of the algorithm and illustrated its complexity in terms of running space and delay time between any two successive outputs of $\delta$-partitions. We proved that the delay time of the algorithm is $O(n)$ and its running space is $O(n)$. Although the algorithm can be applied to solve a scheduling problem, as discussed in the article, one can employ other solution methods to the scheduling problem, such as integer programming or constraint programming. Future research might investigate randomizing our algorithm to see whether the expected delay time of the algorithm can be better than $O(n)$. Likewise, whether the $\delta$-partitions of a given $n$-element set can be computed in running space better than $O(n)$ is yet to be resolved.

Funding information: The author states no funding involved.

Author contributions: Samer Nofal is the only author of this article.

Conflict of interest: The author states no conflict of interest.

Data availability statement: The study uses no data.

References


