Generalized Algorithm for Restricted Weak Composition Generation

Generation Algorithm for Second-Order Restricted Weak Compositions

Daniel R. Page

Received: December 1, 2011 / Accepted: July 4, 2012 (Journal of Mathematical Modelling and Algorithms)

Abstract
This paper presents a new algorithm that arrives at a generalized solution for the generation of restricted weak compositions of n-parts. In particular, this generalized algorithm covers many commonly sought compositions such as bounded compositions, restricted compositions, weak compositions, and restricted part compositions. Introduced is an algorithm for generating generalized types of restricted weak compositions called first-order, and second-order restricted weak compositions.

Keywords Restricted Compositions · Restricted Weak Compositions · Generalized Algorithms · Generalized Compositions · Generation Algorithms · Enumeration Algorithms · Integer Compositions · Weak Integer Compositions · Enumeration · Combinatorics · Computational Number Theory

Mathematics Subject Classification (2000) 68R05 · 68R01

1 Introduction

With a long, vast, and rich history, the study of integer compositions (see [4]) has grown in the past decade with the rise of applications in numerous fields such as combinatorics, computational number theory, and computational chemistry. In particular, the spark of interest in the generation of compositions has been of attention lately in the theory of compositions. The term enumeration can apply in two very different ways. These two ways are counting the elements of a set, while the other is to generate by listing the members of a set directly. The primary interest of this paper is the generation of the elements which compose a particular weak integer composition. The reason for this interest in generation instead of counting is that the cardinality of a solution for a generation problem is the same as the solution for a counting problem in this context. Generation problems of this kind take the form of some initial restrictions, with the hopes of finding a means of returning a set of elements

Daniel Page
Winnipeg, Manitoba, Canada
Tel.: +204-802-0155
E-mail: drpage@pagewizardgames.com
Present address: Box 4 GRP555 RR5, Winnipeg, Manitoba, Canada, R2C 2Z2

which meet those restrictions.

Algorithms which can generate restrictive types of compositions have been of interest in the past decade [1,2,4,9,11,12]. I wish to present an algorithm which can enumerate any second-order restricted weak composition. In turn this type of composition can enumerate *most* commonly sought types of *n-compositions*. The types of compositions which of interest in this paper are restricted weak compositons. Before we consider the definition of a restricted weak composition, let us consider the basis of these combinatorial objects, just a typical composition.

Every *n-composition* has two components, namely the *number of parts*, and the *order* of a composition [4]. The *number of parts* is the number of elements in any sequence contained in any given integer composition. The *number of parts* is also known as the length of a composition, which we will denote as n. The *order* of a composition is also known as the sum of a composition. That is, for any sequence of *non-negative integers* $(a_0, a_1, \ldots, a_{n-1})$, the sum of a composition is $\sum_{i=0}^{n-1} a_i = s$. In this paper we will be assuming that for any composition intended to be generated, a given s, n are chosen for a generation. In the literature such compositions are referred to as *n-compositions*. The traditional definition of a composition only allows strictly positive integers but, compositions are only a special case of a type of compositions known as weak composition. Weak Compositions are also referred to as *n-compositions with non-negative integer parts* in the literature.

For readers who are familar with *integer partitions*, a composition is simply a combinatorial object like a partition, except a composition is all the permutations in the additive partitioning of an integer. Since a typical composition is a permutation of all the elements contained in an *integer partition*.

Definition 1 *Given $n \in \mathbb{Z}^+$, and $s \in \mathbb{Z}^+ \cup \{0\}$. A* weak composition $C_{s,n}$ *is the set of any non-negative integer sequences $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$, where $\sigma_i \in \mathbb{Z}^+ \cup \{0\}$, and $\sum_{i=0}^{n-1} \sigma_i = s$.*

For example, $C_{4,3} = \{(4,0,0),(0,4,0),(0,0,4),(3,1,0),(3,0,1),$
$(0,3,1),(1,3,0),(0,1,3),(1,0,3),(2,2,0),$
$(2,0,2),(0,2,2),(1,1,2),(1,2,1),(2,1,1)\}$.

A weak composition $C_{s,n}$ has a cardinality of $|C_{s,n}| = \binom{n+s-1}{n-1}$ [10]. Weak compositions can also be called unrestricted weak compositions, because no restrictions have been placed. Instead of diving right into popular types in study, let us take a more generalized approach which could cover most types of compositions.

Since we know *n-compositions* with positive integer parts are simply a special case of weak compositions, we could restrict the non-negative integer sequences such that, zeroes are not allowed. That is to restrict that value. Many types of compositions restrict the integer values allowed to be used in such integer compositions forming *restricted compositions*. What is desired is a definition of restricted weak compositions so that most families of compositions or weak compositions are covered.

Definition 2 *Given $n \in \mathbb{Z}^+$, and $s \in \mathbb{Z}^+ \cup \{0\}$, a* restricted weak composition *is any subset of a* weak composition $C_{s,n}$.

We should consider restrictions first placed over all the positions because naturally it is less restrictive than if we were to apply restrictions to each part. Examples of types of compositions which restrict all the parts to a single set of non-negative integers are *bounded compositions* [9]. Let us define such a restriction as a first-order restriction. Since we are restricting a set of values, we require a set to contain this information, let us refer to this as

a first-order restricted set.

We can immediately define the first type of restricted weak composition desired for generalization. That is, a first-order restricted weak composition.

Definition 3 *Given* $n \in \mathbb{Z}^+$, $s \in \mathbb{Z}^+ \cup \{0\}$, *and* restricted set $R^1 \subseteq \{0,1,\ldots,s\}$. *A* first-order restricted weak composition $C_{s,n}^{(R^1)^n}$ *is the set of non-negative integer sequences* $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$, *where* $\sigma_i \in R^1$, *and* $\sum_{i=0}^{n-1} \sigma_i = s$.

Using the previous example with a first-order restricted set $R^1 = \{0,2\}$,

$$C_{4,3}^{\{0,2\}^3} = \{(2,2,0),(2,0,2),(0,2,2)\}.$$

With such a generalization, any restriction placed over all the parts may be considered. What is important about this definition is that the restricted set doesn't need to be contigous over a single integer domain. Unlike some recent solutions, the restrictions in this form do not need to be contiguous, and permit any values desired. But, this would lack a covering of any restriction over the parts. Thus, we need another generalization which is even more restricted than first-order restricted weak compositions. That is, for this further generalization, each part can be restricted by some first-order restricted set.

This means the first-order restricted weak composition is a special case of this form where all the restricted sets are *equal* for all elements $r \in R^1$, where $0 \leq r \leq s$. So, we would have n first-order restricted sets $R_0^1, R_1^1, \ldots, R_{n-1}^1$. The option for further generalization would be to consider each restricted set with it's own unique position as an n-tuple. This is defined as the second-order restriction.

Next, we can define the most generalized type of restricted weak composition over parts. This generalization is the second-order restricted weak composition.

Definition 4 *Given* $n \in \mathbb{Z}^+$, $s \in \mathbb{Z}^+ \cup \{0\}$, *and* second-order restricted set $R_n^2 = (R_0^1, R_1^1, \ldots, R_{n-1}^1)$, *where each* first-order restricted set $R_i^1 \subseteq \{0,1,\ldots,s\}$. *A* second-order restricted weak composition $C_{s,n}^{R_n^2}$ *is the set of any non-negative integer sequences* $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$, *where* $\sigma_i \in R_i^1$, *and* $\sum_{i=0}^{n-1} \sigma_i = s$.

For example, with a second-order restricted set $R_3^2 = (\{1,2\},\{0\},\{2,3\})$,

$$C_{4,3}^{R_3^2} = \{(1,0,3),(2,0,2)\}.$$

Additionally, let us introduce the definition of the power second-order restricted weak composition, which we will denote as $\mathscr{C}_{s,n}^{R_n^2}$.

Definition 5 *Given* $n \in \mathbb{Z}^+$, $s \in \mathbb{Z}^+ \cup \{0\}$, *and some* second-order restricted set R_n^2. *The* power second-order restricted weak composition *of* $C_{s,n}^{R_n^2}$ *is,*

$$\mathscr{C}_{s,n}^{R_n^2} = \{(a_0, a_1, \ldots, a_i) \in C_{j,i}^{R_i^2} \mid j \leq s, i \leq n\}.$$

Now with both generalized types of restricted weak compositions, the two generation problems sought can be defined. Both problems have a solution in the form of a generation algorithm in this paper.

Problem 1 *Given* $n \in \mathbb{Z}^+$, $s \in \mathbb{Z}^+ \cup \{0\}$, *and* first-order restricted set R^1, *generate* $C_{s,n}^{(R^1)^n}$, *and return this set.*

Problem 2 *Given* $n \in \mathbb{Z}^+$, $s \in \mathbb{Z}^+ \cup \{0\}$, *and* second-order restricted set $R_n^2 = (R_0^1, R_1^1, \ldots, R_{n-1}^1)$, *generate* $C_{s,n}^{R_n^2}$, *and return this set.*

Problem 2 is of interest in this paper, because an algorithm which can generate second-order restricted weak compositions can construct *many* restricted weak compositions, and most commonly sought types of *n-composition*, including *problem 1*.

The main focus for the solution of *problem 2* is to construct restricted weak compositions as an application. Since the second-order restricted weak composition is defined as the most general form over the restriction of parts, the algorithm outputs a very efficient to prune subset for the construction of any sought restricted weak composition. Restricted weak compositions can take the form of existence problems for particular construction problems. Furthermore, the same principles could be applied to seek out the existence of many other combinatorial constructions.

The ability for an algorithm to allow the defining of subsets of compositions through different restricted sets is a new idea *algorithmically*. Generating this combinatorial object effectively solves any composition generation problem of this form.

2 History of the Problem

Fairly recently researchers have been trying to find methods of *avoiding* or restricting weak compositions. In this section I will only be covering some of the major recent results due to the broadness of research done in this field of generation.

The idea of generating *subsets* of compositions was becoming a topic of interest because in practice, generating a whole weak composition would be impractical. This is due to the *combinatorial growth* of the number of integer sequences found in a composition for large input sizes. In 2000, Timothy Walsh accomplished two tasks which raised awareness of generation algorithms of this kind [12].

1. Modified Klingsberg's Algorithm for weak compositions which was developed with *Knuth-Klingsberg gray codes* originally by restricting each part in the form of a *n-bounded composition* [5,6]. This is also known as an $(m_0, m_1, \ldots, m_{n-1})$-bounded compositions, where $0 \leq m_i \leq s$. This is a significant restriction to include in a composition in contrast to an unrestricted weak composition. This is a type of restricted weak composition.
2. Modified the mechanism in the *Bitner-Ehrlich-Reingold* method contributed by Ehrlich for sequencing *gray codes*, into a form to generate this new modified *gray code* in $O(1)$ time per integer sequence [10].

In terms of our definitions, Walsh's weak composition algorithm for restricted weak compositions constructs a list *looplessly* for $C_{s,n}^{R_n^2}$, where $R_n^2 = (\{0,\ldots,b_0\},\ldots,\{0,\ldots,b_{n-1}\})$. This algorithmic ability to contruct restricted weak compositions is fairly general.

In 2003, Timothy Walsh developed an algorithm which takes his year 2000 result and generalizes it so it can generate *gray codes* in $O(1)$ worse-case time per word for almost all gray codes [13]. Walsh's paper establishes that *gray codes* can infact be used in a practical manner by generalizing all the main techniques of representing *gray codes* [3]. Any *loopless gray code* generation algorithm can be done efficiently using this algorithm. There have been several augmentations of Walsh's algorithm to exhaust or generate types of compositions, or partitions, or permutations which have been developed [1,2,4,7,8,9,11]. To augment this result, all one needs to do is follow the construction for how many *words* are required.

Recently in 2011, there was a very unique take on n-bounded compositions with *gray code* solutions by V. Vajnovszki and R. Vernay [11]. Their result along with developing a

new technique for generating types of permutations was to take the concepts from Walsh's loopless algorithm for bounded compositions, and generalize it further. Their result gave a procedure for constructing b-bounded compositions in their section on restricted compositions [11]. In this construction, each part allows integers in the sequence to be of values $0 \leq l \leq b_i \leq s$, where $[l,k]$ are given. Vajnovszki and Vernay's construction can construct b-bounded compositions of a positive integer s *looplessly*. In terms of our definitions, this is only a *second-order* restricted weak compositions of the form $C_{s,n}^{R_n^2}$, where $R_n^2 = (\{l,(l+1),\ldots,b_0\},\ldots,\{l,(l+1),\ldots,b_{n-1}\})$. Thus far, this is the most recent result.

All modern solutions if applied to *problem 2* with some modifications all suffer an additional $\Omega(nk)$ worst-case time-complexity, where k is the size of the list those algorithms can output. For large n, this can be very costly. This comes with the cost of generating the entire list and we want to avoid generating the entire list if possible with a solution for this problem. This comes from the limitations of not being able to restrict over non-contigous bounds. These algorithms cannot constuct most restricted weak compositions without such modifications, but such an algorithm is what is sought in this paper with my new algorithm. The new algorithm employs a new technique, and does not rely on *gray code* based solutions, nor *recursion* based solutions.

3 Algorithm

3.1 Second-Order Restricted Weak Composition Generation Algorithm

Consider the *Second-Order* Restricted Weak Composition Generation Algorithm.

Problem: Generate the second-order restricted weak composition, $C_{s,n}^{R_n^2}$.
Inputs: Integer arrays $R_0^1,\ldots,R_{n-1}^1$ (restricted set of the *second-order*).
Non-negative integer s (order of the composition).
Non-negative integer n (number of parts of the composition).
Output: *Queue* $Q[s]$, which contains all the elements of $C_{s,n}^{R_n^2}$.

```
Queue compositionSecondOrder(int[ ][ ] R,int s,int n){
    boolean empty := false;
    if((n = 0) ∨ (n < 0) ∨ ((n = 0) ∧ (s > 0)) ∨ (|R| = 0)){
        empty := true;//This is empty set.
    }
    (Alg. 1.1) Preliminaries : Construct Q[0],...,Q[s] and R_map.
    if((n = 1) ∧ (¬empty)){
        (Alg. 1.2) Trivial Case : n = 1.
    }
    if((n = 2) ∧ (¬empty)){
        (Alg. 1.3) Trivial Case : n = 2.
    }
    if((n > 2) ∧ (¬empty){
        (Alg. 1.4) Construction Step : Construct the edge sets.
        (Alg. 1.5) Base Step : Concatenate the first position.
        (Alg. 1.6) Induction Step : Perform (n−2) rounds of generation.
    }
    return Q[s];//Q[s] contains all valid integer sequences for C_{s,n}^{R_n^2}.
}//compositionSecondOrder
```

Algorithm 1: Second-Order Restricted Weak Composition Generation

The idea of this algorithm is to group non-negative integer sequences of length n by their sums, and generate the sequences together *left, to right*. These sequences are generated through constructing n-tuples inductively through *rounds*. Upon termination, the algorithm returns a *queue* containing integer arrays, which represents the expected second-order restricted weak composition that contains n-tuples.

The algorithm consists of three main steps. These steps are the *construction step*, the *base step*, and the *induction step*. The construction step of the algorithm constructs edge-sets from *bit vectors* which represent the second-order restricted set R_n^2.

The algorithm has three inputs. The first parameter is the second-order restricted set modelled by *n integer arrays*. Each row of the array $R[w]$ represents a first-order restricted set R_w^1, $0 \leq w \leq (n-1)$. The last two parameters are the sum s, and the number of parts n.

Let us run through an example to help understand the algorithm. Suppose we wish to generate the second-order restricted weak composition $C_{5,6}^{R_6^2}$. In this example, $R_6^2 = (\{0,1\},\{0,1\},\{1\},\{1,2\},\{1\},\{0\})$. The object sought is the set of all non-negative integer 6-tuples of the form $(a_0,a_1,a_2,a_3,a_4,a_5)$, where, $\sum_{k=0}^{5} a_k = 5$ and $a_k \in R_k^1$, assuming $0 \leq k < 6$.

As inputs for our example, the parameters for the algorithm would be:

- $R := (\{0,1\},\{0,1\},\{1\},\{1,2\},\{1\},\{0\})$,
- $s := 5$,
- $n := 6$.

Now, let us consider the initial set up in order to proceed to the *preliminaries*.

```
Queue[ ] Q := new Queue[s+1];//Q[0],Q[1],...,Q[s]
BitVector[ ] R_map := new BitVector[n][s+1];//n bit vectors of (s+1) bits
for(int i := 0;i < n;i++){
    //For each round ...
    for(int j := 0; j < |R[i]|; j++){
        if((R[i][j] ≥ 0) ∧ (R[i][j] ≤ s)){
            R_map[i][R[i][j]] := 1;//Set bits for elements over the interval [0,s].
        }
    }
}
```

Algorithm 1.1: *Preliminaries:* Construct $Q[0],\ldots,Q[s]$ and R_{map}.

The use of queue data structures are essential to this algorithm. The algorithm uses exactly $(s+1)$ queues to represent the sums from 0 to s, inclusively. The algorithm will transform each integer array $R[i]$ into a bit vector $R_{map}[i]$. The entire array of bit vectors R_{map} represents the second-order restricted set R_n^2, which is given as the input integer array R. Preliminaries of the algorithm will construct n bit vectors, each with $(s+1)$ bits. Each *first-order* restricted set is modelled by each *bit vector*, where each position set in the bit vector represents the presence of a non-negative integer, corresponding to the numeric value of the position. Now continuing our example, we construct queues $Q[0],Q[1],\ldots,Q[5]$ first. Then we construct the array of bit vectors R_{map}, where in our example:

$$R_{map}[0] = (1,1,0,0,0,0,0),$$
$$R_{map}[1] = (1,1,0,0,0,0,0),$$
$$R_{map}[2] = (0,1,0,0,0,0,0),$$
$$R_{map}[3] = (0,1,1,0,0,0,0),$$
$$R_{map}[4] = (0,1,0,0,0,0,0),$$
$$R_{map}[5] = (1,0,0,0,0,0,0).$$

Next, since $n > 2$ in our example, we move directly to the *construction step*. These trivial cases check each of the positions and *enter* valid sequences into $Q[s]$.

```
if(R_map[0][s]){
    int[ ] element := new int[1];
    element[0] := s;
    Q[s].enter(element);
}
```

Algorithm 1.2: *Trivial Case:* $n = 1$.

```
for(int i := 0; i ≤ s; i++){
    if(R_map[0][i] ∧ R_map[1][s − i]){
        int[ ] element := new int[2];
        element[0] := i;
        element[1] := s − i;
        Q[s].enter(element);
    }
}
```

Algorithm 1.3: *Trivial Case:* $n = 2$.

```
int carryTill := −1; //Following this main round, zeroes are allowed to be placed into n − tuples.
for(int i := (n − 2); i > 0; i−−)
    if(¬R_map[i][0] ∧ carryTill = −1){
        carryTill := (i − 1);
    }
}
LinkedList[ ][ ] linkEdges := new LinkedList[n − 2][s + 1];
for(int k := 1; k < (n − 1); k++){
    for(int i := 0; i < s; i++){
        for(int j := s − i; j ≥ 0; j−−){
            if(R_map[k][j]){
                if((carryTill ≤ (k − 1)) ∨ (i + j ≠ s)){
                        linkEdges[k − 1][i].add(i + j); //Add to front of the list.
                }
            }
        }
    }
}
int[ ][ ][ ] edges := new int[n − 2][s + 1][];
//Now copy to the three dimensional array for instant access.
for(int i := 0; i < (n − 2); i++){
    for(int j := 0; j < s; j++){
        Node curr := linkEdges[i][j].head;
        edges[i][j] := new int[ |linkEdges[i][j]| ];
        for(int k := 0; k < |edges[i][j]|; k++){
            edges[i][j][k] := curr.data;
            curr := curr.next();
        }
    }
    edges[i][s] := new int[0];
}
```

Algorithm 1.4: *Construction Step:* Construct the edge-sets.

With the second-order restricted weak composition generation algorithm, the algorithm constructs $(n-2)$ edge sets. As sequences are being constructed, the task of each edge set is to direct the flow of passing along sequences for each *round* between the queues.

In the *induction step*, we use an edge set to reduce the amount of testing we need to do for forming new copies, and placing these new copies into queues for the next round. This is desirable because most restricted weak compositions are not the entire set of the weak composition, but a subset of the weak composition. An edge set reduces the number of tests by eliminating the redundant transitions outside of the restricted set, and having to check each bit of $R_{map}[i]$.

An important optimization is computing $carryTill$. We want $Q[s]$ to store only integer arrays which sum to s. If these sequences were to be placed into $Q[s]$ before approaching the last round, zeroes must be allowed for subsequent positions, or the algorithm must *erase* the data in $Q[s]$. To avoid needing to *erase* at any time sequences in $Q[s]$, if the number of the *main round* is greater than $carryTill$, then if permitted, queues may *enter* arrays into $Q[s]$, as instructed by the edge sets. In our example, $carryTill$ is 3.

For each round, edges are used from $edges[round-1]$. Edges are accessed by *round* to represent each restricted set R_k^1, where $1 \leq k \leq (n-2)$. Now to continue with our example by constructing four edge sets.

Table 1 The edge sets constructed for the induction step in our example.

$edges[0]$	0	1	2	3
0:	0	1		
1:	1	2		
2:	2	3		
3:	3	4		
4:	4			
5:				

$edges[1]$	0	1	2	3
0:	1			
1:	2			
2:	3			
3:	4			
4:				
5:				

$edges[2]$	0	1	2	3
0:	1	2		
1:	2	3		
2:	3	4		
3:	4			
4:				
5:				

$edges[3]$	0	1	2	3
0:	1			
1:	2			
2:	3			
3:	4			
4:	5			
5:				

Each row m of a given edge set represents the edges a given queue $Q[m]$ could $enter$ contained n-tuples into queues $Q[g]$, for each edge g in the row m. The value placed into a sequence for given position is calculated by computing $(g-m)$. Next, we perform the base step of the algorithm.

```
for(int i := 0;i ≤ s;i++){
    int[ ] element := new int[n].
    if(R_map[0][i]){
        element[0] := i;
        Q[i].enter(element);//Let this element enterfor Q[i]_1.
    }
}
```

Algorithm 1.5: *Base Step:* Concatenate the first position.

In the base step, we construct for each non-negative integer $a \in R_0^1$, an n-tuple where a is in the first position, and the remaining positions in the sequence are zeroes. It is assumed when the integer arrays (n-tuples) of length n are instantiated, the n-tuple contains only zeroes as the above pseudo-code describes. If $R_{map}[0][i]$ is set, then the new integer array (n-tuple) $(i,0,0,\ldots,0)$ is *entered* into $Q[i]$ for *round* 1. A queue $Q[i]$ prepared for *round* j will be denoted as $Q[i]_j$. $Q[i]_j$ may also denote the number of non-negative integers presently marked in the sequence excluding the *final round*. In the base step, we are preparing for *round* 1.

Continuing our example, since $R_{map}[0] = (1,1,0,0,0,0)$, we enter the non-negative integer sequences leading with 0, and 1 into $Q[0]$, and $Q[1]$ respectively.

Thus, in each queue we have:

$Q[5]_1 =_B <>_F$
$Q[4]_1 =_B <>_F$
$Q[3]_1 =_B <>_F$
$Q[2]_1 =_B <>_F$
$Q[1]_1 =_B < (1,0,0,0,0,0) >_F$
$Q[0]_1 =_B < (0,0,0,0,0,0) >_F$

Now, we are prepared for the induction step where we perform $(n-2)$ rounds of generation.

```
for(int round := 1; round ≤ (n−2); round++){
    if((¬R_map[round][0]) ∧ (round ≤ (n−2))){
        Q[s].erase(); //Remove all the elements in Q[s].
    }
    for(int i := (s−1); i ≥ 0; i−−){
        if(round = n−2){//Check if Q[i]_{n−2} is reachable to Q[s]_{n−1}
            (Alg. 2.6.1) Final Round : Reachability Test.
        }
        int countDown := |Q[i]|;
        for(int elem := 0; elem < countDown; elem++){
            int[ ] element := Q[i].remove(); //Remove from Q[i]_round to use.
            if(round < (n−2)){//Main induction round
                for(int j := 0; j < |edges[round][i]|; j++){
                    int value := edges[round − 1][i][j];
                    int[ ]copy := copy(element); //Copy of integer array.
                    copy[round] := value − i;
                    Q[value].enter(copy); //Let the copy enter Q[value]_{round+1}.
                }
            }
            else{//The final induction round.
                (Alg. 2.6.2) Final Round : Last round of generation.
            }
        }
    }
}
```

Algorithm 1.6: *Induction Step:* Perform $(n-2)$ rounds of generation.

In the *induction step*, the algorithm traverses the queues starting at $i = (s-1)$, and decrements to the left-most queue $Q[0]$, as visualized below. This process will take each queue, and copy valid integer sequences contained, and for each edge based on the edge set, will *enter* the copies in the instructed queues. The queues can be visualized as a line of

queues, sequences never move to the *left*, always to the *right*, or *remain constant*. An important note is that the algorithm doesn't touch $Q[s]$, except when queues to the *left enter* valid non-negative integer sequences. $Q[s]$ will be built inductively through the *induction step*.

Round 1: Start at $i = 4$. Since queues $Q[4], Q[3], Q[2]$ are empty, proceed to $i = 1$. In this round, we are restricted to using any non-negative integer $a_1 \in R_1^1$. That is the restricted set $R_1^1 = \{0, 1\}$.

i:=1
$Q[1]_1 =_B < (1,0,0,0,0,0) >_F$ is being considered:
$Q[5]_2 =_B <>_F$
$Q[4]_2 =_B <>_F$
$Q[3]_2 =_B <>_F$
$Q[2]_2 =_B < (1,1,0,0,0,0) >_F$
$Q[1]_2 =_B < (1,0,0,0,0,0) >_F$
$Q[0]_1 =_B < (0,0,0,0,0,0) >_F$
i:=0
$Q[0]_1 =_B < (0,0,0,0,0,0) >_F$ is being considered:
$Q[5]_2 =_B <>_F$
$Q[4]_2 =_B <>_F$
$Q[3]_2 =_B <>_F$
$Q[2]_2 =_B < (1,1,0,0,0,0) >_F$
$Q[1]_2 =_B < (0,1,0,0,0,0), (1,0,0,0,0,0) >_F$
$Q[0]_2 =_B < (0,0,0,0,0,0) >_F$

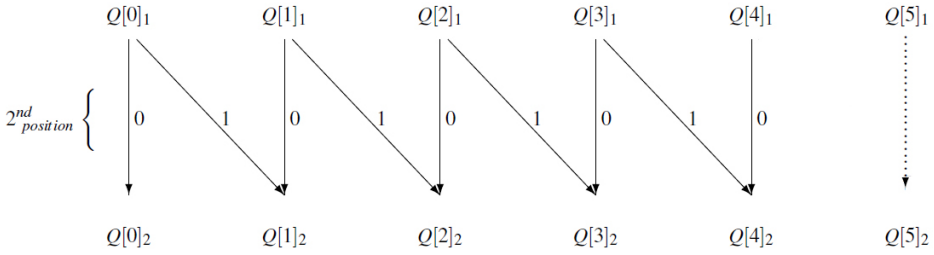

Fig. 1 A visualization of the first main round in our example. The algorithm takes every sequence from each queue, right to the left, and creates copies on each edge to enter.

Round 2: Start at $i = 4$. Since queues $Q[4], Q[3]$ are empty, proceed to $i = 2$. In this round, we are restricted to using any non-negative integer $a_2 \in R_2^1$. That is the restricted set $R_2^1 = \{1\}$.
i:=2
$Q[2]_2 =_B < (1,1,0,0,0,0) >_F$ is being considered:
$Q[5]_3 =_B <>_F$
$Q[4]_3 =_B <>_F$
$Q[3]_3 =_B < (1,1,1,0,0,0) >_F$
$Q[2]_3 =_B <>_F$
$Q[1]_2 =_B < (0,1,0,0,0,0), (1,0,0,0,0,0) >_F$
$Q[0]_2 =_B < (0,0,0,0,0,0) >_F$

i:=1
$Q[1]_2 =_B < (0,1,0,0,0,0),(1,0,0,0,0,0) >_F$ is being considered:
$Q[5]_3 =_B <>_F$
$Q[4]_3 =_B <>_F$
$Q[3]_3 =_B < (1,1,1,0,0,0) >_F$
$Q[2]_3 =_B < (0,1,1,0,0,0),(1,0,1,0,0,0) >_F$
$Q[1]_3 =_B <>_F$
$Q[0]_2 =_B < (0,0,0,0,0,0) >_F$

i:=0
$Q[0]_2 =_B < (0,0,0,0,0,0) >_F$ is being considered:
$Q[5]_3 =_B <>_F$
$Q[4]_3 =_B <>_F$
$Q[3]_3 =_B < (1,1,1,0,0,0) >_F$
$Q[2]_3 =_B < (0,1,1,0,0,0),(1,0,1,0,0,0) >_F$
$Q[1]_3 =_B < (0,0,1,0,0,0) >_F$
$Q[0]_3 =_B <>_F$

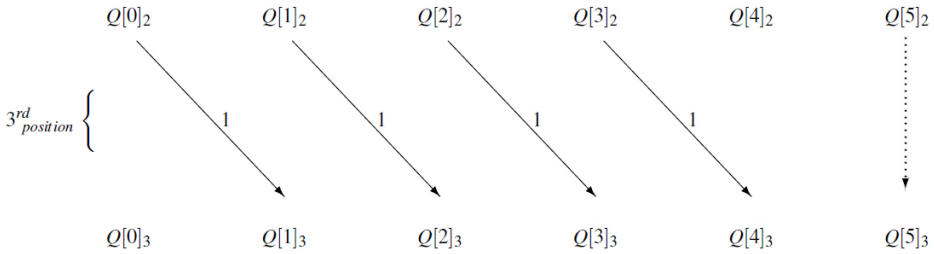

Fig. 2 A visualization of the second main round in our example.

Round 3: Start at $i = 4$. Since the queue $Q[4]_3$ is empty, proceed to $i = 3$. In this round, we are restricted to using any non-negative integer $a_3 \in R_3^1$. Note that, the restricted set $R_3^1 = \{1,2\}$.

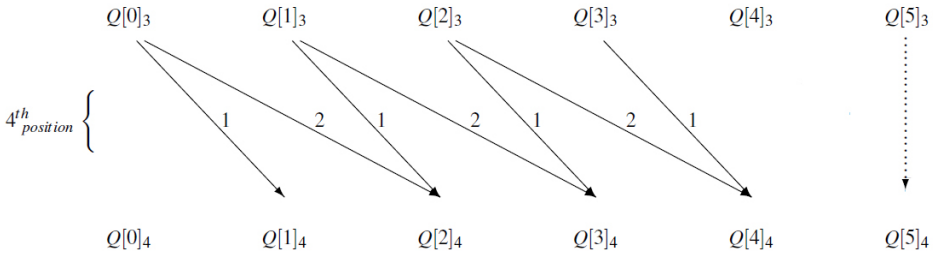

Fig. 3 A visualization of the third main round in our example.

i:=3
$Q[3]_3 =_B < (1,1,1,0,0,0) >_F$ is being considered:
$Q[5]_4 =_B <>_F$
$Q[4]_4 =_B < (1,1,1,1,0,0) >_F$
$Q[3]_4 =_B <>_F$
$Q[2]_3 =_B < (0,1,1,0,0,0),(1,0,1,0,0,0) >_F$
$Q[1]_3 =_B < (0,0,1,0,0,0) >_F$
$Q[0]_2 =_B <>_F$

i:=2
$Q[2]_3 =_B < (0,1,1,0,0,0),(1,0,1,0,0,0) >_F$ is being considered:
$Q[5]_4 =_B <>_F$
$Q[4]_4 =_B < (0,1,1,2,0,0),(1,0,1,2,0,0),(1,1,1,1,0,0) >_F$
$Q[3]_4 =_B < (0,1,1,1,0,0),(1,0,1,1,0,0) >_F$
$Q[2]_4 =_B <>_F$
$Q[1]_3 =_B < (0,0,1,0,0,0) >_F$
$Q[0]_2 =_B <>_F$

i:=1
$Q[1]_3 =_B < (0,0,1,0,0,0) >_F$ is being considered:
$Q[5]_4 =_B <>_F$
$Q[4]_4 =_B < (0,1,1,2,0,0),(1,0,1,2,0,0),(1,1,1,1,0,0) >_F$
$Q[3]_4 =_B < (0,0,1,2,0,0),(0,1,1,1,0,0),(1,0,1,1,0,0) >_F$
$Q[2]_4 =_B < (0,0,1,1,0,0) >_F$
$Q[1]_4 =_B <>_F$
$Q[0]_2 =_B <>_F$

i:=0
$Q[0]_3 =_B <>_F$ is being considered:
$Q[5]_4 =_B <>_F$
$Q[4]_4 =_B < (0,1,1,2,0,0),(1,0,1,2,0,0),(1,1,1,1,0,0) >_F$
$Q[3]_4 =_B < (0,0,1,2,0,0),(0,1,1,1,0,0),(1,0,1,1,0,0) >_F$
$Q[2]_4 =_B < (0,0,1,1,0,0) >_F$
$Q[1]_4 =_B <>_F$
$Q[0]_4 =_B <>_F$

Since $Q[0]_3$ was empty, the algorithm did not change the queues in $i = 0$ for $round = 3$.

Now, the main rounds of the induction step are completed, now the algorithm moves to the *final round*, because $(6-2) = 4$. This round is two parts, the *reachability test*, and the *last round of generation*. Let us consider a definition for reachability.

Definition 6 *A queue $Q[i]$ is* reachable *to $Q[s]$ if, and only if there exists two non-negative integers $x \in R^1_{n-2}$, $y \in R^1_{n-1}$, such that $s = (i+x+y)$.*

```
boolean reachable := false;
for(int j := 0;(j < |edges[round − 1][i]|) ∧ (¬reachable); j++){
    int valueR := edges[round − 1][i][j];
    if(R_map[n − 1][s − valueR]){
        reachable := true;
    }
}
if(¬reachable){
    Q[i].erase(); //No elements in Q[i]_{n−2} can reach Q[s]_{n−1}.
}
```

Algorithm 1.6.1: *Final Round:* Reachability Test.

It appears $Q[4]$ is the only queue which is reachable in our example. If a queue is not reachable, then the contents of those queues will be deleted by the algorithm. The reachability to $Q[s]$ became important in this example, because majority of the elements sitting in our queues were not going to sum to the value of 5. So the algorithm erases the contents of $Q[0],\ldots,Q[3]$, leaving $Q[4]$ alone.

```
for(int j := 0; j < |edges[round − 1][i]|; j++){
    int value := edges[round − 1][i][j];
    if(R_map[n − 1][s − value]){
        int[ ]copy := copy(element); //Copy of integer array.
        copy[round] := value − i;
        copy[n − 1] := s − value;
        Q[s].enter(copy); //Let the copy enter Q[s]_{n−1}.
    }
}
```

Algorithm 1.6.2: *Final Round:* Last round of generation.

The *final round* of this algorithm takes sequences from $Q[s-1],\ldots,Q[0]$ and places them into $Q[s]$. This step operates exactly like a main round, except the algorithm only creates a copy if copies can reach the last queue, $Q[s]$. This step merges the assignment of two positions because we know the sum of every sequence, and can determine the first $(n-1)$ positions of any sequence by the ordering of the queues. The algorithm appeals to $R_{map}[n-1]$ for the final position, and $R_{map}[n-2]$ for the second last position, to check the validity of a sequence. Now in our example, we have reached $round = 4$. The algorithm proceeds starting at $i = 4$.

i:=4

$Q[4]_4 =_B< (0,1,1,2,0,0),(1,0,1,2,0,0),(1,1,1,1,0,0) >_F$ is being considered:

$Q[5]_5 =_B< (0,1,1,2,1,0),(1,0,1,2,1,0),(1,1,1,1,1,0) >_F$

$Q[4]_5 =_B<>_F$

$Q[3]_4 =_B<>_F$

$Q[2]_4 =_B<>_F$

$Q[1]_4 =_B<>_F$

$Q[0]_4 =_B<>_F$

Since the remaining queues $Q[3],Q[2],Q[1]$, and lastly $Q[0]$ are empty, at the end of the *final round* we have only contents in the final queue, as expected.

i:=0
$Q[0]_4 =_B <>_F$ is being considered:
$Q[5]_5 =_B < (0,1,1,2,1,0),(1,0,1,2,1,0),(1,1,1,1,1,0) >_F$
$Q[4]_5 =_B <>_F$
$Q[3]_5 =_B <>_F$
$Q[2]_5 =_B <>_F$
$Q[1]_5 =_B <>_F$
$Q[0]_5 =_B <>_F$

Following the *final round*, the algorithm returns the elements contained in $Q[5]$. In our example, the elements contained in $Q[5]$ form the second-order restricted weak composition $C_{5,6}^{R_n^2}$, where $R_n^2 = (\{0,1\},\{0,1\},\{1\},\{1,2\},\{1\},\{0\})$.

Now we have covered the generalized algorithm sought to solve both *Problem 1*, and *Problem 2* respectively. That is, the generation algorithm for second-order restricted weak compositions. Next, the *correctness* of the algorithm must be considered.

3.2 Correctness

Lemma 1 *Suppose we are given any main round of the induction step $r \geq 1$, second-order restricted set $R_n^2 = (R_0^1, R_1^1, \ldots, R_{n-1}^1)$, and $s \in \mathbb{Z}^+ \cup \{0\}$. If queues $Q[0], Q[1], \ldots, Q[s]$ contain integer sequences which are valid for the first r positions, then at the end of round r, only valid integer sequences for the first $(r+1)$ positions will be contained in $Q[0], Q[1], \ldots, Q[s]$.*

Proof Let $P(s)$ denote the statement: Given queues $Q[0]_r, Q[1]_r, \ldots, Q[s]_r$ containing integer sequences which are valid for the first r positions, then at the end of round r, only valid non-negative integer sequences for the first $(r+1)$ positions will be contained in $Q[0]_{r+1}, Q[1]_{r+1}, \ldots, Q[s]_{r+1}$.

Base Step ($s = 0$): $P(0)$ says that if we are given the queue $Q[0]_r$ containing only integer sequences which are valid for the first r positions, then at the end of round r, only valid integer sequences for the first $(r+1)$ positions will be in queue $Q[0]_{r+1}$. This is a trivial case because either $Q[0]_r$ only contains one valid integer sequence which sums to 0, or there are no valid sequences in queue $Q[0]_r$. Since $s = 0$, the algorithm only has to check if $0 \in R_r^1$ to check if all the sequences in $Q[0]_r$ are valid. If $0 \in R_r^1$, then the algorithm leaves $Q[0]_r$ alone. If $0 \notin R_r^1$, then the algorithm deletes the contents of the queue because it is invalid. Since the algorithm on this main round for $s = 0$ returns only valid integer sequences in $Q[0]_{r+1}$, $P(0)$ holds.

Base Step ($s = 1$): $P(1)$ in this case states that if we are given the queues $Q[0]_r, Q[1]_r$ which contains valid non-negative integer sequences for the first r positions, then when the algorithm reaches the end of round r, only valid sequences for the first $(r+1)$ positions will be contained in $Q[0]_{r+1}, Q[1]_{r+1}$. Since $s = 1$, the algorithm begins the main round by considering $Q[1]_r$. If $0 \in R_r^1$, then $Q[1]_r$ is left alone. Otherwise the contents of $Q[1]_r$ are deleted by the algorithm. Next, the algorithm proceeds to evaluate the contents of $Q[0]_r$. If any integer sequences are present in $Q[0]_r$, then these sequences are removed one at a time following checking the *second-order* edge set, which models all the allowable positions in R_r^1, for any integer sequence of the round r. Upon each integer sequence removed in $Q[0]_r$, for each

element in R_r^1, a copy of the integer sequence is produced and concatenated with each respective position, and entered into queues $Q[0]_{r+1}$, and $Q[1]_{r+1}$. Since all the non-negative integer sequences are valid for the first $(r+1)$ positions by the definition of a *second-order* restricted weak composition, $P(1)$ holds.

Inductive Step ($P(k-1) \rightarrow P(k)$): Fix some $k > 1$. $P(k-1)$ denotes the statement: Given queues $Q[0]_r, Q[1]_r, \ldots, Q[k-1]_r$ containing integer sequences which are valid for the first r positions, then at the end of the round r, only valid non-negative integer sequences for the first $(r+1)$ positions will be contained in $Q[0]_{r+1}, Q[1]_{r+1}, \ldots, Q[k-1]_{r+1}$.

Assume that $P(k-1)$ holds. We wish to demonstrate the statement $P(k)$: Given queues $Q[0]_r, Q[1]_r, \ldots, Q[k-1]_{r+1}, Q[k]_r$ containing integer sequences which are valid for the first r positions, then at the end of the round r, only valid non-negative integer sequences for the first $(r+1)$ positions will be contained in $Q[0]_{r+1}, Q[1]_{r+1}, \ldots, Q[k-1]_{r+1}, Q[k]_{r+1}$.

We begin with knowing queues $Q[0]_r, Q[1]_r, \ldots, Q[k-1]_r, Q[k]_r$ having valid non-negative integer sequences for the first r positions. Recall that in a *main rounds* of the *induction step* of round r we only allow positions which are contained in R_r^1. This is modelled by the algorithm through the use of the edge sets, which is constructed by the bit vectors $R_{map}[k]$ for round k. This matches the definition of a second-order restricted weak composition, because only a position at the $(r+1)^{th}$ position will be considered valid if that non-negative integer is in R_r^1.

Recall that each queue $Q[i]_r$ in our array of queues represents all valid non-negative integer sequences of the first $(r+1)$ positions which sum to i. For each queue $Q[i]_r$, there is the possibility of entering sequences upon removal into queues $Q[i+m]_{r+1}$, where $m \in R_r^1$. That is, for any potential queue $Q[i]_r$, the entering of sequences following concatenation upon removal are over at most $Q[i]_{r+1}, Q[i+1]_{r+1}, \ldots, Q[k-1]_{r+1}, Q[k]_{r+1}$. This means queues $Q[i]_{r+1}, Q[i+1]_{r+1}, \ldots, Q[k-1]_{r+1}$ have valid non-negative integer sequences. By the induction hypothesis, queues $Q[0]_{r+1}, Q[1]_{r+1}, \ldots, Q[k-1]_{r+1}$ all have valid non-negative integer sequences for the first $(r+1)$ positions, when $s = (k-1)$, but not $s = k$ yet.

We wish to demonstrate $P(k)$ holds. Queue $Q[k]_r$ contains all valid integer sequences which have sum of k. If $0 \in R_r^1$, then we do not delete the contents of $Q[k]_r$, for $Q[k]_{r+1}$. The induction hypothesis is not enough to sufficiently produce the *main round r* since every queue $Q[m]_r$ must have the ability to enter sequences into $Q[k]_{r+1}$. If $(s-m) \in R_r^1$, then include in the edge set an edge leading to $Q[k]_{r+1}$ from $Q[m]_r$. Thus, given valid integer sequences for queues $Q[0]_r, Q[1]_r, \ldots, Q[k]_r$, we have valid integer sequences for each queue $Q[m]_{r+1}$. So $P(k)$ holds.

By mathematical induction, we have proven that for all $s > 1$, the statement $P(s)$ is true. Therefore, if queues $Q[0], Q[1], \ldots, Q[s]$ contain integer sequences which are valid for the first r positions, then at the end of round r, only valid integer sequences for the first $(r+1)$ positions will be contained in $Q[0], Q[1], \ldots, Q[s]$ in the algorithm.□

Lemma 2 *Suppose we are given any* second-order restricted set R_n^2, *and* $s \in \mathbb{Z}^+ \cup \{0\}$. *If queues* $Q[0], Q[1], \ldots, Q[s]$ *contain non-negative integer sequences which are valid for the first position, then at the end of main round* $r > 0$, *only valid integer sequences for the first* $(r+1)$ *positions will be contained in* $Q[0], Q[1], \ldots, Q[s]$.

Proof Let $P(r)$ denote the statement: Given queues $Q[0]_r, Q[1]_r, \ldots, Q[s]_r$ containing non-negative integer sequences which are valid for the first position, then at the end of main round $r > 0$, only valid integer sequences for the first $(r+1)$ positions will be contained in queues $Q[0]_{r+1}, Q[1]_{r+1}, \ldots, Q[s]_{r+1}$.

Base Step ($r = 1$): $P(1)$ says that if we are given queues $Q[0]_1, \ldots, Q[s]_1$ which contain valid integer sequences of length 1, then at the end of round 1, only valid integer sequences of length 2 will be contained in the queues $Q[0]_2, \ldots, Q[s]_2$. Applying *Lemma 1*, this case holds because we are given valid integer sequences of length 1. The algorithm will construct valid integer sequences of length 2 in this main round as a consequence. Thus, $P(1)$ holds.

Inductive Step ($P(d-1) \rightarrow P(d)$): Fix some $d > 1$. $P(d-1)$ denotes the statement: Given queues $Q[0]_{d-1}, Q[1]_{d-1}, \ldots, Q[s]_{d-1}$ containing non-negative integer sequences which are valid for the first position, then at the end of main round $(d-1) > 0$, only valid integer sequences for the first d positions will be contained in $Q[0]_d, Q[1]_d, \ldots, Q[s]_d$.

Assume that $P(d-1)$ holds. We wish to demonstrate the statement $P(d)$: Given queues $Q[0]_d, Q[1]_d, \ldots, Q[s]_d$ containing non-negative integer sequences which are valid for the first position, then at the end of main round $d > 0$, only valid integer sequences for the first $(d+1)$ positions will be contained in $Q[0]_{d+1}, Q[1]_{d+1}, \ldots, Q[s]_{d+1}$.

By the induction hypothesis, $P(d-1)$ says that the algorithm will construct valid integer sequences with d non-negative integers. By *Lemma 1*, the algorithm constructs valid non-negative integer sequences of length $(d+1)$ in a main round d in the algorithm. Thus, $P(d)$ holds.

By mathematical induction, we have proven that for all $r \geq 1$, the statement $P(r)$ is true. Therefore, if queues $Q[0], Q[1], \ldots, Q[s]$ contain non-negative integer sequences that are valid for the first position, then at the end of main round $r > 0$, only valid integer sequences for the first $(r+1)$ positions will be contained in $Q[0], Q[1], \ldots, Q[s]$.□

Theorem 1 *The* second-order restricted weak composition *generation algorithm upon termination returns the sought* second-order restricted weak composition $C_{s,n}^{R_n^2}$.

Proof The second-order restricted weak composition generation algorithm constructs non-negative integer sequences through the *base step*, and the *induction step*. The induction step is comprised of two steps, the *main rounds*, and the *final round*. Upon termination of the algorithm, $Q[s]$ is returned which contains the sought set. We wish to show that this is $C_{s,n}^{R_n^2}$. Let us first consider the *base step*.

The *base step* of the algorithm creates empty non-negative integer sequences of length n for each non-negative integer $v \in R_0^1$, where $0 \leq v \leq s$. Each sequence with v at the first position in the sequence is placed into $Q[v]$ respectively. Thus all queues $Q[0]_1, \ldots, Q[s]_1$ contain all valid integer sequences of length one. It is important to note that each queue represents the sums of the sequences contained in them. For instance, all the sequences in $Q[2]$ would contain all sequences which sum to add to 2.

Next, the *induction step* of the algorithm is begun following the construction of the *edge-set* in the *construction step*. Let us consider the *main rounds*. We can apply *Lemma 2* since we know all the sequences contained in queues $Q[0]_1, Q[1]_1, \ldots, Q[s]_1$ are valid for the first position. The algorithm conducts $(n-3)$ main rounds before the *final round*. As a consequence *Lemma 2* says that every queue will contain valid non-negative integer sequences for the first $(n-2)$ positions.

Lastly, the *final round* is really a two step version of one of the *main rounds*. This is due to the property that if one knows the first $(n-1)$ positions of the sequence, then you can determine the last non-negative integer, since we are given the sum s of the composition. The *final round* of the *induction step* in this algorithm will apply a condition only allowing

sequences $(a_0, a_1, \ldots, a_{n-2}, a_{n-1})$ to be valid only if $(s - value) \in R^1_{n-1}$, where *value* is the queue directed to if the algorithm were to place $(n-1)^{st}$ position with from the *edge-set*. Really this means that it is identical to a *main round* only with this additional condition. Applying *Lemma 1*, the algorithm is given valid integer sequences of length $(n-2)$, and will construct valid integer sequences of length $(n-1)$. By applying the described condition, the algorithm will include the final position's non-negative integer along with the $(n-1)^{st}$ position's entry upon creation, and place only those which meet the final condition into $Q[s]_{n-1}$. This means that the sequences contained in $Q[s]$ following the *final round* will be all non-negative integer sequences $(a_0, a_1, \ldots, a_{n-1})$, which contain for each position, $a_i \in R^1_i$. By definition, this set is a second-order restricted weak composition. At this point, the algorithm terminates, and returns $Q[s]$.

Therefore, the second-order restricted weak composition generation algorithm returns the sought second-order restricted weak composition $C^{R^2_n}_{s,n}$. □

4 Conclusions

There exists a new generalized algorithm for the generation of restricted weak compositions as demonstrated. That is, an algorithm for generating second-order restricted weak compositions. The second-order restricted weak composition generation algorithm covers *many* types of restricted weak compositions. This algorithm can be augmented with a loop following the execution to seek any restricted weak composition with the smallest known subsets, due to the algorithm being the most general solution over the restriction of the parts. The second-order restricted weak composition generation algorithm can enumerate restrictive types of compositions such as any part-wise restricted composition, bounded compositions, part-wise bounded composition, and more because this algorithm can use non-continuous restricted sets.

Given the number of parts $n \in \mathbb{Z}^+$, the sum $s \in \mathbb{Z}^+ \cup \{0\}$, and some second-order restricted set R^2_n, the worst-case time-complexity of the algorithm is $\Theta(n(|\mathscr{C}^{R^2_{n-1}}_{s,n-1}| + s^2))$, where $0 \leq |\mathscr{C}^{R^2_{n-1}}_{s,n-1}| \leq \binom{n+s}{n-1}$. If the sought second-order restricted weak composition is a weak composition, then the worst-case time-complexity is $O(s|C_{s,n}|)$. In terms of time-complexity, this algorithm performs on par with modern solutions to more specific problems, but has a far more general structure. Furthermore, this algorithm is very efficient for restrictive types of compositions over *non-contiguous* restrictions over the parts for large s, or n.

The space-complexity of the algorithm in the worst-case is $\Theta(n(|\mathscr{C}^{R^2_{n-1}}_{s,n-1}| + s^2))$, but most instances consume less space than the worst-case. The worst-case space-complexity for the construction of a weak composition is $O(n|C_{s,n}|)$. In contrast to most modern solutions which rely on generating an entire list, this algorithm can perform in most instances significantly better in space-complexity due to the avoidance of generating the entire list of a weak composition. The algorithm uses no recursion, but makes use of loops.

In turn, the algorithm is simple to implement. The algorithm only requires the use of simple data structures to carry out the entire algorithm and could be simple to understand and implement in contrast to many modern solutions. Such an algorithm is adaptable due to it's integration of returning a queue which contains all the elements within the defined restricted weak composition. For example, such an adaptation could be to solve particular restricted weak composition counting problems, or used by an application.

The structure of this algorithm allows for trivial augmentations that allow for the gen-

eration of specific compositions. It is important to mention that the algorithm can also be easily specialized for specific problems by using very particular restricted sets for any given purpose for such generation algorithms. The power of the algorithm is that any restriction or rules can be placed by constructing restricted sets and using them as input. The second-order restricted weak composition returned by this algorithm can be used to obtain any specific restricted weak compositions efficiently. This can be achieved in $\Omega(nk)$ for the smallest such k for restrictions over the parts. The ability to construct restricted sets by the means of *another* algorithm, or for a sole purpose allows for any generalization one could want in a restricted weak composition.

Therefore the second-order restricted weak composition generation algorithm can generate any second-order restricted weak composition of length n, and can be augmented for efficient generation of any restricted weak composition.

Acknowledgements Thank you Dr. John van Rees for supervising this research in Summer 2011.

5 Appendix

The following section includes source code for the second-order restricted weak composition generation algorithm, the components needed for this implementation, and example executions of the algorithm.

The following inputs and outputs were tested on a Toshiba Notebook with a 2.00 GHz Intel Pentium Processor, with 4GB RAM on the Windows 7 operating-system. The source code is in Java. This section begins with essential pieces to have the algorithm function, then the algorithm, and then a few examples.

The following is source code for the array copying which is essential to the algorithm. This class is written in Java.

```
class ArrayCopy{
  public static int[] copy(int[] a){
    int[]b = new int[a.length];
    for(int i=0;i<b.length;i++){
      b[i] = a[i];
    }
    return b;
  }
}
```

The following is source code for the queue used for the algorithm. This class is written in Java.

```
class Queue{
  class NodeQ{

    public int[]element;
    public NodeQ next;

    public NodeQ(int[]element,NodeQ next){
      this.element = element;
      this.next = next;
    }

    public NodeQ(int[]element){
```

```
      this.element = element;
      this.next = null;
    }
    public void setNext(NodeQ next){
      this.next = next;
    }

  }
  NodeQ head;//head of the queue
  NodeQ tail;//tail of the queue
  long size;
  public Queue(){

    this.head = null;
    this.tail = null;
    this.size = 0;



  }
  //enter operation
  public void enter(int[]element){

    NodeQ node = new NodeQ(element);
    if(head == null){
      this.head = node;
      this.tail = node;
    }
    else{
      this.tail.setNext(node);
      tail = node;
    }
    if(element != null){
      this.size = this.size+1;
    }
  }
  //to erase an entire queue's contents.
  public void erase(){
    this.head = null;
    this.tail = null;
    this.size = 0;
  }
  //to remove an element from the queue.
  public int[] remove(){
    int[]data;
    if(head == null){
      data = null;

    }
    else{
      NodeQ node = head;
      data = head.element;
      head = head.next;
      if(head == tail){
        tail = head;
      }
    }
    if(data != null){
      this.size = this.size -1;
    }
```

```
    return data;
  }
  //obtain the size of the queue.
  public long size(){
    return this.size;
  }
  //for printing the queue's contents
  public String toString(){
    String str = "";
    NodeQ ptr = head;
    while(ptr != null){
      if(ptr.element != null){
        str+="(";
        for(int i=0;i<ptr.element.length;i++){
          str+=ptr.element[i];
          if(i<ptr.element.length-1){
            str+=",";
          }
        }

        str+=")\n";
        ptr = ptr.next;
      }
    }
    str += "";
    return str;
  }
}
```

The following is source code for the node, and linked list used the source code of the algorithm. This is implemented in Java.

```
class Node{
  int data;
  Node next;
  public Node(int data){
    this.data = data;
    this.next = null;
  }
  public void setNext(Node next){
    this.next = next;
  }
}
class LinkedList{
  Node head;//the head of the linked list
  int numNodes;//number of nodes.

  //constructor
  public LinkedList(){
    this.head = null;
    numNodes = 0;
  }
  //insert at the head
  public void add(int data){
    Node newNode = new Node(data);
    numNodes +=1;
    if(head == null){
      head = newNode;
    }
```

```
    else{
      newNode.setNext(head);
      head = newNode;
    }
  }
  //size of the linked list.
  public int size(){
    return numNodes;
  }
}
```

The following is source code for the second-order restricted weak composition generation algorithm in Java. Along with the given classes, the algorithm in the form of a static method uses the BitSet class.

```
public static void compositionSecondOrder(int[][]R,int s, int n){
  boolean empty = false;
  int carryTill=-1;//following this main round, zeroes are allowed to be placed into n-tuples.
  if((n==0) || (n<0) || (s<0) || ((n==0) && (s>0)) || (R.length == 0)){
    empty = true;//this is empty set
  }
  //Construction Step: Construct Q[0],...,Q[s] and R_map
  Queue[] Q = new Queue[s+1];//Q[0],Q[1],...,Q[s]
  for(int i=0;i<s+1;i++){
    Q[i] = new Queue();
  }
  BitSet[] R_map = new BitSet[n];//n Bit vector of (s+1) bits.
  for(int i=0;i<n;i++){
    R_map[i] = new BitSet(s+1);
  }
  for(int i=0;i<n;i++){
    //For each round...
    for(int j=0;j<R[i].length;j++){
      if((R[i][j] >=0) && (R[i][j] <= s)){
        R_map[i].set(R[i][j]);//Set bits for elements over the interval [0,s].
      }
    }
  }

  if((n==1) && (!empty)){
    //Trivial Case: n=1
    if(R_map[0].get(s)){
      int[] element = new int[1];
      element[0] = s;
      Q[s].enter(element);
    }
  }
  if((n==2) && (!empty)){
    //Trivial Case: n=2
    for(int i=0;i<=s;i++){
      if(R_map[0].get(i) && R_map[1].get(s-i)){
        int[]element = new int[2];
        element[0] = i;
        element[1] = s-i;
        Q[s].enter(element);
      }
    }
  }
  if((n>2) && (!empty)){
```

```
//Construction Step: Construct the edge sets.

//check the furthest we can clear the rightmost queue Q[s].
for(int i=(n-2);i>0;i--){
  if(!R_map[i].get(0) && carryTill == -1){
    carryTill=(i-1);
  }
}
LinkedList[][] linkEdges = new LinkedList[n-2][s+1];
for(int i=0;i<n-2;i++){
  for(int j=0;j<s+1;j++){
    linkEdges[i][j] = new LinkedList();
  }
}
for(int k=1;k<(n-1);k++){
  for(int i=0;i<s;i++){
    for(int j=s-i;j>=0;j--){
      if(R_map[k].get(j)){
        if(!(carryTill>(k-1))||!(i+j == s)){
        linkEdges[k-1][i].add(i+j);//Add to front of the list.
        }
      }
    }
  }
}
int[][][] edges = new int[n-2][s+1][];
//Now copy to the three dimensional array for instant access.
for(int i=0;i<(n-2);i++){
  for(int j=0;j<s;j++){
    Node curr = linkEdges[i][j].head;
    edges[i][j] = new int[linkEdges[i][j].size()];
    for(int k=0;k<edges[i][j].length;k++){
      edges[i][j][k] = curr.data;
      curr = curr.next;
    }
  }
  edges[i][s] = new int[0];
}
//Base Step: Concatenate the first position.
for(int i=0;i<=s;i++){
  int[]element = new int[n];
  if(R_map[0].get(i)){
    element[0] = i;
    Q[i].enter(element);//Let this element enter for Q[i]_1.
  }
}
//Induction Step: Perform (n-2) rounds of generation.
for(int round=1;round<=(n-2);round++){
  for(int i=(s-1);i>=0;i--){
    if(round == (n-2)){//Check if Q[i]_(n-2) is reachable to Q[s]_(n-1)
      //Reachability Test
      boolean reachable = false;
      for(int j=0;(j<edges[round-1][i].length)&&(!reachable);j++){
        int valueR = edges[round-1][i][j];
        if(R_map[n-1].get(s-valueR)){
          reachable = true;
        }
      }
      if(!reachable){
        Q[i].erase();//No elements in Q[i]_(n-2) can reach Q[s]_(n-1).
```

```
          }
        }
        int countDown = (int)Q[i].size();
        for(int elem = 0;elem<countDown;elem++){
          int[]element = Q[i].remove();//Remove from Q[i]_round to use.
          if(round < (n-2)){//Main induction round
            for(int j=0;j<edges[round-1][i].length;j++){
              int value = edges[round-1][i][j];
              int[] copy = ArrayCopy.copy(element);//Copy of integer array.
              copy[round] = value-i;
              Q[value].enter(copy);//Let the copy enter Q[value]_(round+1).
            }
          }
          else{
            //Final Round: Last round of generation.
            for(int j=0;j<edges[round-1][i].length;j++){
              int value = edges[round-1][i][j];
              if(R_map[n-1].get(s-value)){
                int[]copy = ArrayCopy.copy(element);//Copy of integer array.
                copy[round] = value-i;
                copy[n-1] = s-value;
                Q[s].enter(copy);//Let the copy enter Q[s]_(n-1).
              }
            }
          }
        }
      }

    }
  }
  //Now we are done.
  System.out.println(Q[s]);//output C_{s,n}^{R_n^2}
}//compositionSecondOrder
```

Following, here are three input, and output examples for the second-order restricted weak composition generation algorithm along with expected times.

Algorithm - Example 1

Input:

```
int s = 8;
int n = 5;
int[][]R = {{0,1,2},{2,4},{0,2},{0,3},{0,1,2}};
compositionSecondOrder(R,s,n);
```

Output:

```
(2,4,2,0,0)
(1,4,2,0,1)
(2,4,0,0,2)
(2,2,2,0,2)
(0,4,2,0,2)
(1,4,0,3,0)
(1,2,2,3,0)
(2,2,0,3,1)
(0,4,0,3,1)
```

```
(0,2,2,3,1)
(1,2,0,3,2)
```

Running Time (before printing): 0.008s
Total Running Time: 0.010s

Algorithm - Example 2

Input:

```
int s = 5;
int n = 6;
int[][]R = {{0,1},{0,1},{1},{1,2},{1},{0}};
compositionSecondOrder(R,s,n);
```

Output:

```
(1,1,1,1,1,0)
(1,0,1,2,1,0)
(0,1,1,2,1,0)
```

Running Time (before printing): 0.003s
Total Running Time: 0.004s

Algorithm - Example 3

Input:

```
int s = 41;
int n = 15;
int[][]R = {{0,1},{0,1},{1},{1,2},{1},{0},{0,1},{2,3},{4,5},{2,3},
{1,5},{5,6},{0,1},{0,1},{5,10}};
compositionSecondOrder(R,s,n);
```

Output:

```
(1,1,1,2,1,0,1,3,5,3,5,6,1,1,10)
```

Running Time (before printing): 0.008s
Total Running Time: 0.009s

References

1. S. Bacchelli, E. Barcucci, E. Grazzini, and E. Pergola. Exhaustive generation of combinatorial objects by eco. *Acta Informatica*, (40):585–602, 2004.
2. A. Bernini, E. Grazzini, E. Pergola, and R. Pinzani. A general exhaustive generation algorithm for gray structures. *Acta Informatica*, (44):361–376, 2007.
3. F. Gray, 1953. Pulse code communication, US Patent 2632058.
4. S. Heubach and T. Mansour. *Combinatorics of Compositions and Words*. CRC Press, 2010.
5. P. Klingsberg. A gray code for compositions. *Journal of Algorithms*, 3(1):41–44, 1981.
6. D. Knuth. *The Art of Computer Programming, vol. 4*. Addison Wesley, 2005.
7. T. Mansour and G. Nassar. Gray codes, loopless algorithm and partitions. *Journal of Mathematical Modelling and Algorithms*, (7):291–310, 2008.
8. T. Mansour and G. Nassar. Loop-free gray code algorithms for the set of compositions. *Journal of Mathematical Modelling and Algorithms*, (9):343–345, 2010.
9. J. Opdyke. A unified approach to algorithms generating unrestricted and restricted integer compositions and integer partitions. *Journal of Mathematical Modelling and Algorithms*, 9(1):53–97, 2010.
10. E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall Inc., 1977.
11. V. Vajnovszki and R. Vernay. Restricted compositions and permutations: From old to new gray codes. *Information Processing Letters*, 111(13):650–655, 2011.
12. T. Walsh. Loop-free sequencing of bounded integer compositions. *Journal of Combinatorial Mathematics and Combinatorial Computing*, (33):323–345, 2000.
13. T. Walsh. Generating gray codes in o(1) worse-case time per word. *DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE, PROCEEDINGS*, 2731:73–88, 2003.