11-2010

Tree-Based Algorithms for Computing *k*-Combinations and *k*-Compositions

Shant Karakashian
University of Nebraska-Lincoln, shantk@cse.unl.edu

Berthe Y. Choueiry
University of Nebraska-Lincoln, choueiry@cse.unl.edu

Tree-Based Algorithms for Computing k-Combinations and k-Compositions

Shant Karakashian and Berthe Y. Choueiry

Constraint Systems Laboratory
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Email: {shantk|choueiry}@cse.unl.edu

TR-UNL-CSE-2010-0009

November 4, 2010

Abstract

In this document, we describe two tree-based algorithms for computing all k-combinations and k-compositions of a finite set.

Contents

1 Introduction

We have developed two algorithms for solving the following combinatorial tasks:

- Given a finite set S and a natural number k, find all subsets of S of size k. In the literature, this problem is called k-subsets and k-combinations.
- Given two natural numbers k, n where $k \leq n$, find all k-compositions of n where a k-composition is an ordered combination of k nonzero natural numbers whose sum is n. Note that in the literature, a k-composition of n can have null numbers. Further, some authors require that the sum of the k numbers to be less or equal to n.

Both algorithms are based on building an intermediary tree data-structure. Using similar tree structures for generating various combinatorial objects under constraints is a "reasonably standard approach" [Hartke 2010]. Algorithms exist in the literature for k-combinations and k-compositions. For example, Wilf [1989] discusses combinatorial Gray codes, and attributes an algorithm for k-compositions to Knuth. Ruskey [1993] shows a bijection between the compositions of Knuth and the combinations of Eades and McKay [1984]. Algorithm pseudocode for those combinatorial problems is reported in Google[1], Section 4.3 and 5.7 of [Ruskey 2010], and [Arndt 2010a; 2010b].

The goal of this document is to report the pseudocode of the algorithms implemented in our software [Karakashian 2010].

2 Generating k-Combinations

Given a non-negative integer c and a set S, COMBINATIONS (Algorithms 1) generates all combinations of size c of the elements of S by calling COMBINATIONSREC (Algorithm 2) on c and s=$|S|$. We implicitly consider that the elements of S are stored in an array and that each (non-negative integer) position of the array corresponds to an element of S.

In particular, Algorithm 2 is a divide-and-conquer algorithm that solves the problem by generating a tree. Given a root node and two non-negative integers c and s, it divides the problem into $(s - c - rest + 1)$ subproblems in Line 2. Each subproblem has a root element that is added as a child to the root of the current problem in Line 4. All solutions in a subproblem include this root element.

[1]Google answers: http://answers.google.com/answers/threadview/id/780070.html.

Algorithm 1: COMBINATIONS(c, S)

```
Input: $c, S$
Output: All subsets of size $c$ of $S$
1 $root \leftarrow \emptyset$
2 $s \leftarrow |S|$
3 CombinationsRec$(c, 0, root, s)$
4 $icombinations \leftarrow$ExtractFromTree$(root)$
5 $combinations \leftarrow$MapPositionsToElements$(icombinations)$
6 return $combinations$
```

The combination size of each subproblem is set to $(c - 1)$, and the subset of the original set with all elements occurring after the root element is assigned as the set of the subproblem. The subproblem is solved by calling the algorithm recursively in Line 4. The recursion ends when c=0 in Line 1. The solution to a subproblem is constructed by adding to each solution from a subproblem the root element of the current problem. The final solution is constructed by calling EXTRACTFROMTREE in Line 4 of Algorithm 1, then mapping back the integers to the corresponding elements of S.

Algorithm 2: COMBINATIONSREC$(c, rest, root, s)$

```
Input: $c, rest, root, s$
1 if $c = 0$ then return
2 for $x \leftarrow rest$ to $(s - c)$ do
3     $node \leftarrow x$
4     add $node$ as a child to $root$
5     CombinationsRec$((c - 1), (x + 1), node, s)$
6 end
```

The output of COMBINATIONSREC and that of an algorithm for generating k-combinations are identical.

The time and space complexity of COMBINATIONS are $\binom{s}{k}$ for a set of s elements. For a graph of $|V|$ vertices, the time complexity is $\Theta(|V|^k)$.

2.1 Illustrating the execution of COMBINATIONS

Figure 1 illustrates the operation of Algorithms 1 and 2 in the call COMBINATIONS(3,{a,b,c,d,e}). The nodes are generated in the lexicographical order of

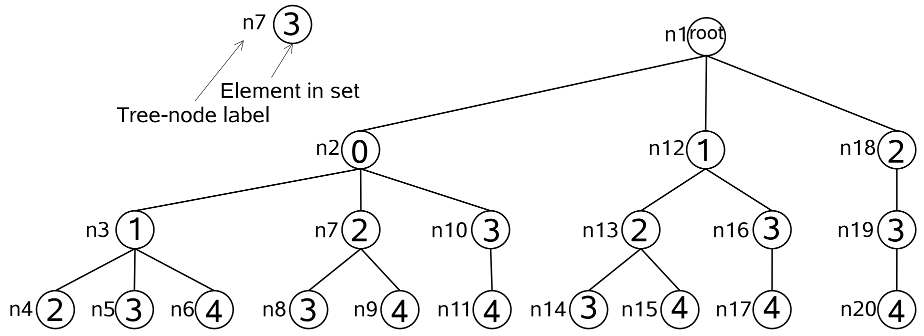

Figure 1: Tree generated by Algorithms 1 and 2 for COMBINATIONS(3,{a,b,c,d,e}).

tree-node labels displayed in Figure 1. The combinations of integers extracted by EXTRACTFROMTREE lie on the paths from the root to leaves of the tree:

$$\begin{aligned}&\{\{0,1,2\},\{0,1,3\},\{0,1,4\},\{0,2,3\},\{0,2,4\},\\&\{0,3,4\},\{1,2,3\},\{1,2,4\},\{1,3,4\},\{2,3,4\}\}\end{aligned} \tag{1}$$

which yields the following combinations:

$$\begin{aligned}&\{\{a,b,c\},\{a,b,d\},\{a,b,e\},\{a,c,d\},\{a,c,e\},\\&\{a,d,e\},\{b,c,d\},\{b,c,e\},\{b,d,e\},\{c,d,e\}\}\end{aligned}$$

2.2 Correctness of COMBINATIONS

Theorem 2.1. COMBINATIONS *is sound and complete.*

Proof. Every generated combination has exactly c distinct elements. The size of the combination is guaranteed by adding to a path in the tree one node at each recursive call and making c recursive calls. At every recursive call, the designated element of the current problem that is added to all solutions of the current problem is excluded from the sets passed to the subproblems. Hence the same element cannot be added more than once to the same set. Moreover, for every element other than the designated element in a problem, a subproblem is generated, such that the first subproblem returns all combinations with the designated element d_1 for the first subproblem, the second subproblem returns all combinations with the designated element d_2 for the second subproblem excluding the d_1, the n^{th} subproblem returns all combinations with designated element d_n excluding the elements $\{d_i | i < n\}$. Consequently no two subproblems return the same combination. Algorithm 2 does not exclude any combination because for every element e in the original set, a subproblem is generated that returns all combinations with the elements ordered after e in the set. The only elements for which no subproblem is generated are the last $(c - 1)$ elements of the original set because no combination of size c exists with fewer than c elements.

Therefore, Algorithm 2 generates all possible combinations of size c from the set given as input to Algorithm 1. $\square$

3 Generating k-Compositions

FIXEDSUMKSTRINGS (Algorithm 3) generates a string of nonzero positive integers of length $size$ such that the sum of the integers in the string is Sum. Algorithm 3 simply creates the root node of a tree structure and passes it to FIXEDSUMKSTRINGSREC (Algorithm 4) with $size$ and Sum as arguments.

Algorithm 3: FIXEDSUMKSTRINGS($size, Sum$)

```
  Input: $size, Sum$
  Output: All strings of strictly positive integers of length $size$ that sum up
          to $Sum$
1 $root \leftarrow \emptyset$
2 FIXEDSUMKSTRINGSREC($size, Sum, root$)
3 $strings \leftarrow$ EXTRACTFROMTREE($root$)
4 return $strings$
```

Algorithm 4 constructs a tree at the root, such that every path from a child of the root to a leaf in the tree is a string of integers with length $size$ and sum equal to Sum. EXTRACTFROMTREE (not presented here because straightforward) retrieves these strings from the tree in Line 3 and returns them at the end.

Algorithm 4 is a divide-and-conquer algorithm, constructing a tree data-structure. Given a root node and values for $size$ and Sum, it divides the problem into $(size - Sum + 1)$ subproblems each of size $(size - 1)$, see Line 6. For each subproblem, an integer x is assigned as the integer at the first position of the

Algorithm 4: FIXEDSUMKSTRINGSREC($size, Sum, root$)

```
Input: $size, Sum, root$
1  if $size = 1$ then
2      $node \leftarrow Sum$
3      add $node$ as a child to $root$
4      return
5  end
6  for $x \leftarrow 1$ to $(Sum - size + 1)$ do
7      $node \leftarrow x$
8      add $node$ as a child to $root$
9      FIXEDSUMKSTRINGSREC($(size - 1), (Sum - x), node$)
10 end
```

string in the corresponding subproblem. This fact is achieved by adding, for each subproblem, a tree node to the root of the current problem with value x in Line 8, which becomes the root of the subproblem. The subproblem is solved by recursively calling the algorithm in Line 9 with the arguments $(size - 1)$, $(Sum - x)$, and the new node. There are $(size - Sum + 1)$ subproblems because only the numbers 1 to $(size - Sum + 1)$ can be used at the current position to form a string of size $size$ and that sums up to Sum. Further, all possible numbers from 1 to $(size - Sum + 1)$ are considered at Line 6 each time Algorithm 4 is called.

The recursion ends when $size$=1 in Line 1, which corresponds to the last position of the string in the original problem. A leaf node with value Sum is added to the root of the current problem in Line 3.

3.1 Illustrating the execution of FIXEDSUMKSTRINGS

Figure 2 illustrates the operation of Algorithm 3 in the call FIXEDSUMKSTRINGS(3,5). Line 6 of Algorithm 4 is called with the values of 1, 2 then 3 for x, as shown in the first level of the tree in Figure 2. The recursive calls in Line 9 yield the tree shown in Figure 2.

EXTRACTFROMTREE extracts the strings by traversing the tree, yielding the strings:

$$\{\{1,1,3\},\{1,2,2\},\{1,3,1\},\{2,1,2\},\{2,2,1\},\{3,1,1\}\}. \tag{2}$$

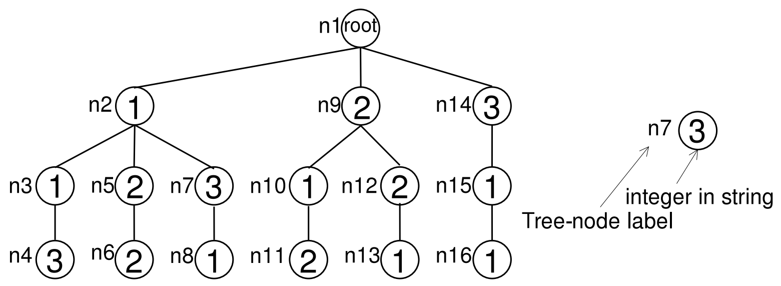

Figure 2: Tree generated by Algorithms 3 and 4 for FIXEDSUMKSTRINGS$(3, 5)$.

3.2 Correctness of FIXEDSUMKSTRINGS

Theorem 3.1. FIXEDSUMKSTRINGS *is sound and complete.*

Proof. Let Sum_o be the original value of Sum given as input to Algorithm 3. We consider that the root of the tree is at depth zero and that its children are at depth one. When the recursive call is made on a node at a depth d in the tree (see Line 9), the value of the second argument in the call is equal to $(Sum_o - \sum_{i=1}^{(d-1)} x_i)$, where x_i is the value of the node at depth i in the path from root to the current node. This fact guarantees that the sum of the integers in each generated string does not exceed Sum_o. Moreover, when $size = 1$, the value stored in the leaf is $x_d = Sum_o - \sum_{i=1}^{(d-1)} x_i$. Consequently, the sum of the values stored in the nodes along a path from the root to a leaf is $\sum_{i=1}^{d} x_i = \sum_{i=1}^{(d-1)} x_i + x_d = Sum_o$, which is the required sum for the original problem.

The algorithm enumerates all possible strings with $size$ and Sum given as input by considering all possible values for each position in the string. Therefore, the algorithm returns all possible strings of the given size and sum. □

3.3 Complexity of FIXEDSUMKSTRINGS

Proposition 3.2 (Number of strings returned by FIXEDSUMKSTRINGS.)**.** *The number of strings returned by* FIXEDSUMKSTRINGS$(size, Sum)$ *is*

$$\mathcal{O}\left((Sum - size)^{(size-1)}\right). \quad (3)$$

Proof. Each string is given by a path from the root to a leaf. Hence, the number of strings is equal to the number of leaves in the tree. The depth of the tree generated by Algorithm 4 is $size$. The maximum branching factor of the tree is $(Sum - size + 1)$ for all nodes except the nodes at depth $(size - 1)$, which have a single child. Hence, there are at most $(Sum - size)^{(size-1)}$ leaves in the tree. $\square$

Proposition 3.3. *Assuming that a node is added in constant time, it follows from Proposition 3.2 that the time complexity of* FIXEDSUMKSTRINGS *is*

$$\mathcal{O}\left((sum - size)^{(size-1)}\right). \tag{4}$$

References

Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*, chapter Combinations. Academic Press, London, UK, 2010.

Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*, chapter Compositions. Academic Press, London, UK, 2010.

Peter Eades and Brendan McKay. An Algorithm for Generating Subsets of Fixed Size with a Strong Minimal Change Property. *Information Processing Letters*, 19:131–133, 1984.

Stephen G. Hartke. Personal communication, 2010.

Shant Karakashian. An Implementation of An Algorithm for Generating All Connected Subgraphs of a Fixed Size. Software (Version Oct2010), Constraint Systems Laboratory, University of Nebraska-Lincoln, Lincoln, NE, 2010.

Frank Ruskey. Simple Combinatorial Gray Codes Constructed by Reversing Sublists. In *Fourth International Symposium on Algorithms and Computation (ISSAC 93)*, pages 201–208, 1993.

Frank Ruskey. Combinatorial Generation. Unpublished manuscript from Citeseer, 2010.

Herbert S. Wilf. *Combinatorial Algorithms: An Update*. SIAM CBMS-NSF 55, 1989.