CONGRESSUS NUMERANTIUM

VOLUME 61

MAY, 1988

WINNIPEG, CANADA

Generating k-combinations from a family of sets

Tom Altman, University of Kentucky
Marek Chrobak, Columbia University
(on leave from the University of Warsaw)
Miroslaw Truszczyński, University of Kentucky

Let U denote a set of n distinct elements and let $S_1, S_2, \ldots, S_m$ be subsets of U. To list all k-combinations C of U, such that $C \subseteq S_i$, for some $1 \leq i \leq m$, an optimal running time algorithm would require at least $O(I+kt)$ steps, where I is the input size and t is the number of k-combinations. In this paper, we present an algorithm for generating such k-combinations. Its running time is bounded by $O(I+kt+mt)$, which is optimal for families of sets S where $m = O(k)$. We also present two algorithms that are optimal for the so called *acyclic* families of sets and for the families of sets whose *thickness* is bounded by 2, (i.e., such that for all $x_i \in U$, x_i is in at most 2 of the S_i's).

Keywords: k-combination, generating algorithms, acyclic family.

1. Introduction

In this paper we investigate algorithms for efficient generation of all k-combinations from a set U of n distinct elements, where $S_1, S_2, \ldots, S_m$ are subsets of U, and a k-combination C of U is generated iff $C \subseteq S_i$, for some $1 \leq i \leq m$. The problem may be reformulated in terms of objects and properties: Let U be a set of objects characterized by some properties $S_1, S_2, \ldots, S_m$; list all k-tuples whose elements share at least one property.

One possible interpretation of this problem may be as follows: Given a set of n nodes and a collection of all cliques in a graph, denoted by $S_1, S_2, \ldots, S_m$; find E - the set of edges in the graph. The analogous interpretation for k-uniform hypergraphs corresponds to the general problem of generating k-tuples.

CONGRESSUS NUMERANTIUM 61(1988), pp.211-220 *

A naive approach for generating the k-combinations would be to examine each S_i separately and append its k-combinations to some global list. The problem that surfaces immediately is the duplication of k-combinations. In the worst case, creation of such a list would require $O(mkt)$ steps. Moreover, the additional time spent to eliminate duplicate entries could take up to $O(mkt \log t)$ steps.

In this paper we present three algorithms for generating all k-combinations from a family of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$. The algorithm which generates the k-combinations for the general case is presented in Section 2. In Section 3, we define the notion of acyclicity in a family of sets $\mathcal{S}$ and present an optimal algorithm for such families. Our third algorithm, which is optimal for the families whose thickness bounded by 2, is described in Section 4.

2. Algorithm for the general case

The algorithm is an extension of the classical recursive algorithm for generating k-combinations of one set (e.g., see [3]). The problem is to implement it efficiently. First we present the algorithm and then we show how it could be implemented to run in time $O(I+kt+mt)$, where $I = \sum_{i=1}^{m} |S_i|$. We assume that each set has at least k elements.

Algorithm I

```
procedure SUBSETS(S, k);
    procedure GENERATE(J, X);
    begin
      if |X| = k then write(X) else
      begin
          x := max( ⋃_{T∈J} T);
          J1 := {T | T ∪ {x} ∈ J, |X ∪ T| ≥ k, or T ∈ J and x ∉ T};
          J2 := {T | T ∪ {x} ∈ J, T ≠ ∅};
          if J1 ≠ ∅ then GENERATE(J1, X);
          GENERATE(J2, X ∪ {x})
      end
    end;
begin
  GENERATE(S, ∅)
end.
```

Figure 1. Recursive generation of k-combinations of $\mathcal{S}$.

The choice of x in the inner block can be arbitrary, but by choosing the maximum value we will generate the k-combinations in antilexicographic order. The main difficulty in analyzing the complexity of this algorithm is that it is not clear how the number of calls of procedure GENERATE depends on t, the number of generated combinations. Consider the tree of calls of GENERATE. If each node had degree 0 or 2 then the number of calls would be at most $2t$. However, some nodes may have degree 1, when the family $\mathcal{T}_2$ happens to be empty. Note that this can happen only when all sets $T \in \mathcal{T}$ contain x and have cardinality $k - |X|$. Below, we show how to avoid this problem.

Now we describe the implementation of Algorithm I. The algorithm works in two phases. In the first phase we construct some data structures.

We assume that:

(1) $\bigcup \mathcal{S} = \{1, 2, \ldots, n\}$,

(2) $|S_i| \geq k$, for each i,

(3) each set S_i is given on input in increasing order (if not, we can use a variation of the bin sort algorithm to properly order the S_i's in time proportional to I).

2.1. Phase I

The whole family $\mathcal{S}$ is represented by an array of lists where the list in the entry j represents the set S_j. Each set S_j, in turn, is a doubly linked list of records R_{ij}. For simplicity we will also denote this list by S_j. Each list S_j has two pointers to the beginning and the end of the list. Lists are sorted with respect to i. A record R_{ij} appears in list S_j only when i belongs to the set S_j, and, besides the links, it contains three entries: i, j, and b_{ij}. The numbers b_{ij} are defined as follows:

$$b_{ij} = \min \{l \mid S_l \cap \{1, 2, \ldots, i\} = S_j \cap \{1, 2, \ldots, i\}\}.$$

Now, we describe how to construct our data structure in time $O(I)$. The preprocessing consists of three steps:

Step 1: Renumber the sets of the family $\mathcal{S}$ so that their lexicographic order is $S_1, \cdots, S_m$ (here we treat sets as increasing sequences of their elements);

Step 2: construct the lists S_j without filling the entries b_{ij};

Step 3: compute the numbers b_{ij}.

Steps 1 and 2 can be done in time $O(I)$. We only need to show how to implement Step 3 (which could have been merged with Step 2, but will be described separately for the sake of exposition).

We use an auxiliary array $P[1..n]$. At the beginning we set $P[i] = nil$ for each i. Then for $j = 1, 2, ..., m$ we scan the list S_j. $P[i]$ will point to the record R_{ij} with the greatest j among the records which have already been scanned. Let $prev(R_{ij})$ be the record which appears on S_j before R_{ij}. Then, to compute the b_{ij}'s of Step 3, we execute the following:

```
for j:=1 to m do
for each R_ij on S_j do
begin
    if P[i] = nil then b_ij := j else
    begin
        find the record R_il that is pointed to by P[i];
        if prev(R_ij) = nil then
            if prev(R_il) = nil then b_ij := b_il else b_ij := j
        else
        begin
            find u such that R_uj = prev(R_ij);
            find v such that R_vl = prev(R_il);
            if v = u and b_uj ≤ l then b_ij := b_il else b_ij := j
        end
    end;
    P[i] := R_ij
end
```

The complexity of Step 3 is trivially $O(I)$. We only have to prove that the numbers b_{ij} will be computed correctly. This can be proved by considering the three cases that follow. Suppose that we are in R_{ij}. We adopt the notation from the above algorithm.

Case 1: $v \neq u$. Then clearly, $b_{ij} = j$.

Case 2: $v = u$ and $b_{uj} > l$. Since $b_{uj} > l$, by induction, the intersections of S_j and S_l with the set $\{1, 2, ..., u\}$ are different. Therefore, their intersections with the set $\{1, 2, ..., i\}$ are also different. Furthermore, there is no set S_r containing i such that $l < r < j$, because of the definition of $P[i]$. Therefore, $b_{ij} = j$.

Case 3: $v = u$ and $b_{uj} \leq l$. It implies that the intersections of the sets S_j and S_l with the set $\{1, 2, ..., u\}$ are equal. Also, from the lexicographic order of the sets in the family S it follows that $j = l + 1$. For suppose that

there is a set S_r with $l < r < j$. If $S_r \subseteq \{1, 2, \ldots, u\}$, then S_r would appear before S_l. The same can be said if S_r contains an x such that $u < x < i$. By the definition of $P[i]$, S_r cannot contain i. The only left possibility is when S_r contains a number greater than i, but then S_r would appear after S_j. Thus we conclude that $j = l+1$. This, in turn, easily implies that $b_{ij} = b_{il}$.

2.2. Phase II

The generating phase is basically Algorithm I in Figure 1, except that we have to handle separately the case when all sets in the family $\mathcal{T}$ have cardinality $k - |X|$. From the ordering of the sets S_j and from the lexicographic order of the family $\mathcal{S}$ it follows that at the beginning of each call to GENERATE we have:

(1) There is x such that each set $T \in \mathcal{T}$ is of the form $S_j \cap \{1, 2, \ldots, x\}$,
(2) $|T \cup X| \geq k$ for each $T \in \mathcal{T}$,
(3) $x \in T$, for some $T \in \mathcal{T}$,
(4) $|X| \leq k-1$,
(5) $X \subseteq \{x+1, x+2, \ldots, n\}$.

The second phase now looks as follows:

```
procedure SUBSETS(S, k);
   procedure GENERATE(T, X);
   begin
    if |T ∪ X| = k for each T ∈ T then PRINT(T, X) else
    begin
        x := max( ⋃_{T∈T} T);
        T1 := {T | T ∪ {x} ∈ T, |X ∪ T| ≥ k, or T ∈ T and x ∉ T};
        T2 := {T | T ∪ {x} ∈ T, T ≠ ∅};
        GENERATE(T1, X);
        GENERATE(T2, X ∪ {x})
    end
   end;
begin
 GENERATE(S, ∅)
end.
```

As it was already noted, each set T is obtained from some S_j by deleting all numbers greater than x. Thus we can represent such T in $\mathcal{T}$ by a number $nr(T) = j$ and the pointer to the last record R_{ij} such that i is still in T. We

also store the cardinality of each set T. Then the family $\mathcal{T}$ will be represented as a list of such records T (again, we use the same name for the set and its representation).

One call of GENERATE($\mathcal{T}, X$) takes $O(m)$ steps. The number x can be found by looking at the pointers to the last elements of the sets T in $\mathcal{T}$. Similarly, we can check the condition in the if-statement and construct $\mathcal{T}_1$ and $\mathcal{T}_2$.

The procedure PRINT($\mathcal{T}, X$) prints out all different sets $T \cup X$ for $T \in \mathcal{T}$. This can be accomplished as follows. Scan the list $\mathcal{T}$ backwards. After printing out some set $T \cup X$ we look at the number b_{ij}, where $j = nr(T)$ and i is the greatest element of $\mathcal{T}$. Then we skip all sets V such that $nr(V) \geq b_{ij}$. In this way, no combination will be generated twice.

Consider now the complexity of the algorithm. Let us look at the tree of recursive calls of the procedure GENERATE. The leaves are calls of PRINT, and the number of leaves is bounded by t, the number of generated combinations. If PRINT is not invoked in a recursive call to GENERATE then both $\mathcal{T}_1$ and $\mathcal{T}_2$ are nonempty and conditions (1)-(5) are satisfied. Hence, each internal node has 2 sons and the number of internal nodes is also bounded by t. Thus the cost of all internal calls is $O(mt)$. If a single call of PRINT generates s k-combinations, then its cost is $O(sk+m)$. Therefore, the overall cost is $O(I+mt+kt)$.

3. Acyclic families of sets

Ideally, an algorithm to generate k-combinations for a family of sets $\mathcal{S}$ would run in $O(I+kt)$ steps. (Certainly, each such algorithm requires at least that many steps since it has to read in its input and print out the generated k-combinations.) The algorithm described in Section 2 has worse time complexity. However, it turns out that for special classes of families of sets $\mathcal{S}$ with some structure, it is possible to take advantage of the structure of $\mathcal{S}$ and design better algorithms than the one for the general case. Their complexity is $O(I+kt)$, hence, they are optimal. In the next two sections we discuss two such classes of families of sets and describe optimal algorithms for each class.

Let U be a finite set. A family of sets $\mathcal{S} \subseteq \mathcal{P}(U)$ is called *acyclic* (e.g., see [2]), if there exists a directed acyclic graph $G = (U,E)$ such that:

(1) the outdegree of each vertex of G is at most 1,
(2) each $S_i \in \mathcal{S}$ is a vertex set of some directed path in G.

An example of an acyclic family $\mathcal{S}$ is shown in Figure 2.

$$U = \{1,2,\ldots,13\}$$
$$\mathcal{S} = \{\{9,4,2,1\}, \{8,3,2,1\}, \{6,2,1\}, \{5,2,1\}, \{13,12\}, \{9,4,2\}, \{11,9,4\}, \{10,5\}, \{11,9\}\}$$

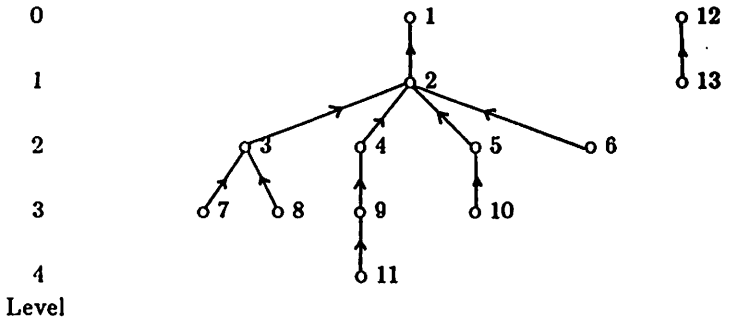

Figure 2. An acyclic family of sets $\mathcal{S}$ and its digraph G.

Acyclicity of $\mathcal{S}$ can be determined in linear time using an algorithm by Dietz et al. [1]. Below, we present an algorithm which generates k-combinations for acyclic families in time $O(I+kt)$, where $I = \sum_{i=1}^{m} |S_i|$ is the input size and kt is the output size (t is the number of k-combinations to be generated).

Algorithm II

1. Find the acyclic organization of $\mathcal{S}$
   (i.e., construct the appropriate digraph G).

2. If necessary, relabel the sets in $\mathcal{S}$, producing $S_1, S_2, \ldots, S_m$, so that if the top element of S_r is at a "higher" level (i.e., has lesser depth) than the top element of S_t, then $r < t$. This is always possible because $\mathcal{S}$ is acyclic.

3. Represent each set S_i as $S_i = \{x_1, \ldots, x_p\}$, where $p = |S_i|$, so that $x_1 \cdots x_p$ is a directed path in G. Compute j_i so that

$$j_i = \begin{cases} \min\{j : x_j \in S_i \cap \bigcup_{r<i} S_r\} & \text{if } S_i \cap \bigcup_{r<i} S_r \neq \emptyset \\ p+1 & \text{otherwise} \end{cases}$$

   (That j_i is well defined follows from acyclicity of $\mathcal{S}$ and from the way the sets in $\mathcal{S}$ are labeled in Step 2.)

4. **for** $i := 1$ **to** m **do**
begin
Generate all k-combinations contained in S_i but in no S_t, $t < i$.
Use a modification of the classical algorithm to generate k-combinations in the lexicographic order (with respect to the labeling $x_1, \ldots, x_p$ of the elements in S_i) that stops as soon as all k-combinations of S_i have been generated, or when the next k-combination to be generated starts with x_{j_i}.
end

Figure 3. An algorithm that generates k-combinations from an acyclic $\mathcal{S}$.

Step 1 can be computed in time $O(I)$ using the algorithm of Dietz et al. [1]. Once G is known, the relabeling phase of Step 2 consists of simply ordering the sets according their top elements. This can be accomplished with a bin sort, hence, requiring $O(m+n)$ steps (ties are broken arbitrarily). Observe that the sets of $\mathcal{S}$ in Figure 2 are already ordered in a proper fashion.

Once the acyclic organization for $\mathcal{S}$ is known, arrangements $\{x_1, \ldots, x_p\}$ of sets S_i, as well as the index j_i, discussed in Step 3 can be found in time proportional to I. Observe that the elements of each $S_i \in \mathcal{S}$ in Figure 2 are also properly ordered.

To see that the computation time of Step 4 is $O(m+kt)$, observe that the algorithm that generates k-combinations in the lexicographic order uses $O(k)$ steps to generate each combination (see e.g., [3]). The only change we have to make is to add one more conditional statement that will check whether the first element of the next combination to be generated is x_{j_i}. This increases the cost of generating each combination but only by some constant c. Hence, the computational time of Step 4 is $O(m+kt)$ and the whole algorithm works in time $O(I+kt)$, as claimed.

To justify correctness of then above algorithm, we prove the following fact.

Proposition 3.1. Algorithm II generates all valid k-combinations of $\mathcal{S}$, each exactly once.

Proof. Consider some valid k-combination C. Let i be the smallest integer such that $C \subseteq S_i = \{x_1, \ldots, x_p\}$. Then, $C \not\subseteq \bigcup_{r<i} S_r$ and C contains an element x_q, with $q < j_i$, (where j_i is defined in Step 3). Hence, C will be generated in the i^{th} iteration of the loop of Step 4.

Suppose now that some combination C is generated twice. Then it must be generated in two different iterations of the loop of Step 4. Suppose, the first iteration C is generated in is iteration i and the next one is iteration i'. Then, since C is generated in iteration i', it contains an element with index smaller than $j_{i'}$ (where $j_{i'}$ is computed for $S_{i'}$ in Step 3), hence, it is not contained in $\bigcup_{r<i'} S_r$, a contradiction as it is generated earlier in the iteration i, for some $i < i'$.

Applying Algorithm II to $\mathcal{S}$ of Figure 2, with $k=2$, will produce the following output:

$$U = \{1,2,\ldots,13\}$$
$$\mathcal{S} = \{\{9,4,2,1\}, \{8,3,2,1\}, \{6,2,1\}, \{5,2,1\}, \{13,12\}, \{9,4,2\}, \{11,9,4\}, \{10,5\}, \{11,9\}\}$$

$\{9,4\}$, $\{9,2\}$, $\{9,1\}$, $\{4,2\}$, $\{4,1\}$, $\{2,1\}$, $\{8,3\}$, $\{8,2\}$, $\{8,1\}$, $\{3,2\}$,
$\{3,1\}$, $\{6,2\}$, $\{6,1\}$, $\{5,2\}$, $\{5,1\}$, $\{13,12\}$, $\{11,9\}$, $\{11,4\}$, $\{10,5\}$.

Figure 4. The generated 2-combinations of $\mathcal{S}$.

4. Bounded thickness families

A family of sets $\mathcal{S}$ is said to have its *thickness bounded by* d, if for all $x_i \in U$, x_i belongs to at most d of the S_i's. Below, we present an algorithm which generates k-combinations for the families of sets whose thickness is bounded by 2. The running time this algorithm is, again, $O(I+kt)$.

Algorithm III

for $i := 1$ **to** m **do**
begin

1. Find the intersections, $T_1, \ldots, T_{i-1}$,
   of S_i with $S_1, \ldots, S_{i-1}$, respectively ;

2. Arrange the elements of S_i so that each T_j $(j < i)$
   and $S_i - (T_1 \cup \cdots \cup T_{i-1})$ are segments of consecutive elements,
   with $S_i - (T_1 \cup \cdots \cup T_{i-1})$ preceding each T_j ;

3. Generate all k-combinations contained in S_i,
   but not contained in T_j, for all $j < i$;

end

Figure 5. Algorithm to generate k-combinations from $\mathcal{S}$ of thickness 2.

It is clear that Algorithm III will only generate valid k-combinations (i.e., contained in some S_i) and there will be no duplicates.

To perform Step 3 we again use the classical algorithm that generates all k-combinations of a given set in the lexicographic order (we will refer to this algorithm as algorithm A) and modify it so we never generate a k-combination contained in any T_j. To accomplish this, if the next combination to be generated by algorithm A is $C = \{x_{i_1}, \ldots, x_{i_k}\}$, where $i_1 < \cdots < i_k$ and $C \subseteq T_j$ then the modified version of A prints out $C' = \{x_{i_1}, \ldots, x_{i_{k-1}}, x'\}$, where x' is the first element of T_{j+1}. To check that $C \subseteq T_j$ and to find x' efficiently we have to compute first and last elements of each segment T_j when the sequential arrangement of objects in which each T_j is a segment is created in Step 2.

At each iteration i, Step 1 requires time proportional to $|S_i|$ (since any element of S_i can be in at most one other set, and we could represent $\mathcal{S}$ by a collection of linked lists of elements of each set and by associating with each element the list of 1 or 2 sets it belongs to), hence, the m iterations of Step 1 can be performed in time $O(I)$. Similarly, the m iterations of Step 2 require only $O(I)$ comparisons. Finally, the m iterations of Step 3 require time proportional to $O(kt)$, therefore, the running time of Algorithm III is $O(I+kt)$.

References

1. Dietz, P., Furst, M., and Hopcroft, J.E., "A Linear Time Algorithm for the Generalized Consecutive Retrieval Problem," Technical Report TR 79-386, Dept. of Computer Science, Cornell University, 1979.
2. Lipski, W., "Information Storage and Retrieval - Mathematical Foundations II (Combinatorial Problems)," *Theoretical Computer Science*, vol. 3, pp. 183-212, 1976.
3. Nijenhuis, A. and Wilf, H.S., *Combinatorial Algorithms*, Academic Press, New York, 1975.