Set Multi-Covering via Inclusion-Exclusion

Qiang-Sheng Hua[1], Yuexuan Wang[2,*], Dongxiao Yu[1], and Francis C.M. Lau[1]

[1]Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong, China
qshua@cs.hku.hk, dxyu@cs.hku.hk, fcmlau@cs.hku.hk

[2]Institute for Theoretical Computer Science, Tsinghua University, Beijing, 100084, China
wangyuexuan@tsinghua.edu.cn

Abstract

Set multi-covering is a generalization of the set covering problem where each element may need to be covered more than once and thus some subset in the given family of subsets may be picked several times for minimizing the number of sets to satisfy the coverage requirement. In this paper, we propose a family of exact algorithms for the set multi-covering problem based on the inclusion-exclusion principle. The presented ESMC (Exact Set Multi-Covering) algorithm takes $O^*((2t)^n)$ time and $O^*((t+1)^n)$ space where t is the maximum value in the coverage requirement set (The $O^*(f(n))$ notation omits a $poly\log(f(n))$ factor). We also propose the other three exact algorithms through different tradeoffs of the time and space complexities. To the best of our knowledge, this present paper is the first one to give exact algorithms for the set multi-covering problem with

* Corresponding author

nontrivial time and space complexities. This paper can also be regarded as a generalization of the exact algorithm for the set covering problem given in [2].

1. Introduction

Recently it has been shown that for some exact algorithms, using the inclusion-exclusion principle can significantly reduce the running time. For example, Björklund et al. have applied the inclusion-exclusion principle to various set covering and set partitioning problems, obtaining time complexities that are much lower than those of previous algorithms [2]. This principle has also been used in some early papers, such as [1] and [7]. By using the Möbius inversion technique which is an algebraic equivalent of the inclusion-exclusion principle, Björklund et al. give a fast algorithm for the subset convolution problem [3] and Nederlof presents a family of fast polynomial space algorithms for the Steiner Tree problem and other related problems [8]. In this paper, we are interested in designing inclusion-exclusion based exact algorithms for the set multi-covering problem [10,11]. This problem is a generalization of the set covering problem in which each element needs to be covered by a specified integer number of times and each set can be picked multiple times in order to satisfy the coverage requirement. It is a bit surprising that only approximation algorithms have so far been proposed for the set multi-covering problem. In fact, by using the same greedy strategy as for the set covering problem, which is to repeatedly add the set containing the largest number of uncovered elements to the cover, one can achieve the same $O(\log n)$ approximation for the problem [10]. Feige shows that the set covering problem can not be approximated better than $\ln n$ unless

$NP \in DTIME(n^{\log\log n})$ [5]. Some parallel approximation algorithms for the set covering problem and its generalizations, such as the set multi-cover problem, the multi-set multi-cover problem and the covering integer programs problem have been presented in [11]. In all these related work on approximation solutions, the set multi-covering problem appears to be no harder than the set covering problem. In this paper, we will see that finding an exact solution for the set multi-covering problem can take much longer time than that for the fastest exact algorithm for the set covering problem [2]. To the best of our knowledge, this present paper is the first one to give exact algorithms for the set multi-covering problem with nontrivial time and space complexities.

The structure of this paper is as follows. In Section 2, we give a formal definition of the set multi-covering problem. In Section 3, we give a brief introduction of the inclusion-exclusion principle and then transform the set multi-covering problem to the problem of counting the number of *k*-tuples that satisfy the integral coverage requirements. We then give four algorithms for counting these numbers of *k*-tuples in Section 4. In Section 5, we give a constructive algorithm for finding the minimum number of sets that meet the coverage requirements. A simple illustrating example for our algorithms is given in the Appendix. We conclude the paper in Section 6.

2. The Set Multi-covering Problem

A summary of the various notations used in this paper and their corresponding definitions is given in Table 2.1. Throughout the paper, we let the union of a *k*-tuple $< s_1,...,s_k >$ which is denoted as $C = \bigcup_{i=1}^{k} s_i$ represent a multi-set. This means

that we just put all the elements in each s_i into the set C without removing duplicated elements.

Table 2.1: Summary of notations and their definitions

Notation	Definition
N	The universe set, where $N = \{1,...,n\}$ and $\|N\| = n$.
F	A family of subsets of N, where $F = \{s_1,...,s_{\|F\|}\}$ and $\|F\|$ is the total number of subsets in F.
T	The integral coverage requirement set, where $T = \{t_1,...,t_n\}$; each $i \in N$ must be covered at least $t_i \geq 1$ times in the picked subsets over F.
t	The maximum integer in the set T, i.e., $t = \max_{1 \leq i \leq n}\{t_i\}$.
$c_k(F)$	The number of k-tuples $< s_1,...,s_k >$ over F such that the union of each k-tuple, i.e., $C = \bigcup_{i=1}^{k} s_i$, satisfy the specified coverage requirement T.
$n_k(X)$	The number of k-tuples $< s_1,...,s_k >$ over F such that each $i \in X$ ($X \subseteq N$) appears at most $(t_i - 1)$ number of times in the set C.
$a(X)$	The number of subsets in F that avoid X.
$b(X,Y)$	The number of subsets in F that include Y but avoid $X \setminus Y$.
$p_q^X(n_1,...,n_{\|X\|})$ or $p_q^X(n_X)$	The number of q-tuples over F such that each $j \in X$ appears n_j times in the union of each q-tuple. For simplicity, we use n_X to denote $\{n_1,...,n_{\|X\|}\}$.

The Set Multi-covering Problem: Let $N = \{1,...,n\}$ be the universe, and F a given family of subsets $\{s_i\}$ over N, and the union of all the subsets in F covers all the elements in N. A legal (k,T) cover is a collection of k subsets over F such that $\bigcup_{i=1}^{k} s_k \geq TN$, where $T = \{t_1,...,t_n\}$ and the inequality means that each $i \in N$ must

appear at least $t_j \geq 1$ times in the union of the k subsets. Note that the k subsets can be non-distinct which means that some subsets in F can be picked several times. The goal of the set multi-covering problem is to find the minimum k to make a legal (k,T) cover.

Remark 1: Since each subset in F can contain each element of N at most once, in order to find a legal (k,T) cover, k must be greater than or equal to t, the maximum integer in the coverage requirement set T, i.e., $k \geq t$. Also, since the union of F covers all the elements in N, we have $k \leq tn$.

3. Counting based Exact Algorithm for the Set Multi-covering Problem

3.1 The Inclusion-Exclusion Principle

FACT 3.1 [folklore]: Let B be a finite set with subsets $A_1, A_2, ..., A_n \subseteq B$. With the convention that $\bigcap_{i \in \varnothing} A_i = B$, the number of elements in B which lie in none of the A_i is

$$|\bigcap_{i=1}^{n} \overline{A_i}| = \sum_{X \subseteq N} (-1)^{|X|} \cdot |\bigcap_{i \in X} A_i| \tag{3.1}$$

3.2 Counting the number of k-tuples

LEMMA 3.2: Let $n_k(X)$ denote the number of k-tuples $< s_1, ..., s_k >$ where for each $j \in X$, the number of j in the set $C = \bigcup_{i=1}^{k} s_k$ is at most $t_j - 1$; then the number of k-tuples that satisfy the coverage requirement T can be computed from the following equation:

$$c_k(F) = \sum_{X \subseteq N} (-1)^{|X|} \cdot n_k(X) \tag{3.2}$$

PROOF: Let B be the set of k-tuples $< s_1,...,s_k >$ from F, and let A_i be the set of k-tuples where element i in the set C appears at most $(t_i - 1)$ times. The left side of Equation 3.1 is the number of k-tuples in which each element i in the universe N is covered at least t_i times, which is represented by $c_k(F)$, the left side of Equation 3.2. Accordingly, $|\bigcap_{i \in X} A_i|$ is the number of k-tuples in which each $j \in X$, which is an element in the set C, appears at most $(t_j - 1)$ times; i.e., $n_k(X) = |\bigcap_{i \in X} A_i|$. By the right side of Equation 3.1, we can derive the right side of Equation 3.2.

LEMMA 3.3: We can find a legal (k,T) cover if and only if $c_k(F) > 0$.

PROOF: $c_k(F)$ is the number of k-tuples over F that satisfy the coverage requirement T. The number of legal (k,T) covers is the number of k subsets over F that satisfy the coverage requirement T. Since different orderings of the k subsets mean different k-tuples while the (k,T) cover concerned remains the same, we know that only when $c_k(F) > 0$ can there be a legal (k,T) cover. Similarly, if there is a legal (k,T) cover, it guarantees that $c_k(F) > 0$. This finishes the proof.

According to Lemma 3.3, we have the following corollary.

COROLLARY 3.4: The minimum k value to make a legal (k,T) cover is equal to the minimum k value that satisfies $c_k(F) > 0$.

Thus we can transform the set multi-covering problem to the problem of computing $c_k(F)$. By using binary search, since $t \le k \le tn$, the time for solving the set multi-covering problem equals the sum of the times for computing the $O(\log(tn))$ numbers of $c_k(F)$. In the next section, we introduce several algorithms for computing $c_k(F)$ with different time and space complexities.

4. Algorithms for Computing $c_k(F)$

In this section, we show how to compute $c_k(F)$, i.e., to count the number of k-tuples $< s_1,...,s_k >$ over F such that the union of each such k-tuple satisfies the given coverage requirement T.

4.1 How to compute $n_k(X)$

According to Equation 3.2, we know that the crux of computing $c_k(F)$ is to obtain $n_k(X)$, i.e., the number of k-tuples over F such that each $i \in X$ appears at most $(t_i - 1)$ times in the union of every k-tuple. Without loss of generality, we assume $X = \{1,2,...,m\}$, and for the simplicity of notation, we let $n_X = \{n_1, n_2,..., n_m\}$. We then denote $p_q^X(n_X) = p_q^X(n_1, n_2,..., n_m)$, the number of q-tuples over F such that for each $j \in X$ the number of the element j in the union of every q-tuple is n_j. Now since the union of each q-tuple can cover each $j \in X$ at most q times, for each $p_q^X(n_1, n_2,..., n_m)$, we have $n_j \le q$ for each $j \in X$; otherwise, $p_q^X(n_1, n_2,..., n_m)$ equals 0. From these definitions, we can easily obtain the following Equation 4.1. This equation means that, in order to obtain $n_k(X)$, we

should sum all the $p_k^X(n_X)$ values ($\prod_{i=1}^{m} t_i$ of them), where $p_k^X(n_X)$ is from $p_k^X(0,0,...,0)$ to $p_k^X(t_1-1,t_2-1,...,t_m-1)$. Now our problem becomes how to efficiently compute all the $p_k^X(n_X)$ values.

$$n_k(X) = \sum_{\substack{0 \le n_i \le t_i - 1 \\ 1 \le i \le m}} p_k^X(n_X) \tag{4.1}$$

Before delving into the details of calculating all these $p_k^X(n_X)$ values, we need to introduce some notations. We use $a(X)$ to denote the number of sets in F that avoid X where $X \subseteq N$, and $b(X,Y)$ to denote the number of sets in F that include Y but avoid $X \setminus Y$, where $Y \subseteq X$. We show next how to get $a(X)$ for all X and $b(X,Y)$ for all X and Y.

4.2 How to compute all $a(X)$

There are two ways to compute $a(X)$. The first way is to use the fast zeta transform technique introduced in [2]. By using this technique, all $a(X)$ values can be computed in $O^*(2^n)$ time. And since the technique uses a look-up table to store all the interim values including $a(X)$ for all $X \subseteq N$, it requires $O^*(2^n)$ space. The second way is to compute $a(X)$ directly without storing all the interim values into a look-up table. In order to compute $a(X)$ where $X \subseteq N$, we just need to test every subset $S \subseteq N \setminus X$ to see if S is in F, which takes time $O^*(2^{n-|X|})$ by assuming that the membership test in F can be decided in polynomial time and polynomial space. Then for all $X \subseteq N$, the total time for computing $a(X)$ equals $\sum_{X \subseteq N} O^*(2^{n-|X|}) = O^*(\sum_{r=0}^{n} C_n^r 2^{n-r}) = O^*(3^n)$.

4.3 How to compute all $b(X,Y)$

Based on the two different ways of computing $a(X)$, we have two corresponding ways to compute all $b(X,Y)$ for all $Y \subseteq X$ and for all $X \subseteq N$.

For arbitrary X and Y, where $Y \subseteq X$, we let $|X| = m$ and $|Y| = r$ and $r \le m$. Without loss of generality, assume $X = \{1,2,...,m\}$ and $Y = \{1,2,...,r\}$. Then $b(X,Y)$ can be computed via Equation 4.2.

$$b(X,Y) = \sum_{Z \subseteq Y} (-1)^{|Z|} \cdot a(Z \cup (X \setminus Y)) = \sum_{Z \subseteq Y} (-1)^{|Z|} \cdot a(Z \cup \{r+1,...,m\}) \qquad (4.2)$$

Equation 4.2 is obtained by applying the inclusion-exclusion principle. According to Fact 3.1, suppose B is a family of subsets of F which avoid $X \setminus Y$, and let $A_i \subseteq B$ ($i \in Y \subseteq X$) be the family of subsets which further avoid element i. Then the left side of Equation 3.1($|\bigcap_{i=1}^{|Y|} \overline{A_i}|$) is the number of sets in F that cover Y but avoid $X \setminus Y$ which is the value of $b(X,Y)$. Accordingly, the right side of Equation 3.1($|\bigcap_{i \in Z \subseteq Y} A_i|$) is the number of sets in F that avoid $Z \cup (X \setminus Y)$ which is the value of $a(Z \cup (X \setminus Y))$. Thus according to Equation 3.1, we have Equation 4.2. Then we calculate how much time we need to compute all $b(X,Y)$.

First, we do not use a table to store all $a(X)$ values, and the time complexity is given in Lemma 4.1.

Lemma 4.1: For all $Y \subseteq X$ and for all $X \subseteq N$, $b(X,Y)$ can be obtained in $O^*(6^n)$ time and polynomial space.

PROOF: As mentioned earlier, in order to compute $a(X)$ where $X \subseteq N$, we just need to test every subset $S \subseteq N \setminus X$ to see if S is in F, which takes time $O^*(2^{n-|X|})$. For given X and Y, according to Equation 4.2, the time for computing $b(X,Y)$ can be calculated from the formula $\sum_{i=0}^{r} C_r^i \cdot O^*(2^{n-i-m+r})$. By using the Binomial theorem, we have Equation 4.3.

$$\sum_{i=0}^{r} C_r^i \cdot O^*(2^{n-i-m+r}) = O^*(2^{n-m} \cdot 3^r) \tag{4.3}$$

Now for all $Y \subseteq X$, the time for computing $b(X,Y)$ can be calculated through the formula $\sum_{r=0}^{m} C_m^r \cdot O^*(2^{n-m} \cdot 3^r)$. Similarly, by using the Binomial theorem, we have Equation 4.4.

$$\sum_{r=0}^{m} C_m^r \cdot O^*(2^{n-m} \cdot 3^r) = O^*(2^{n+m}) \tag{4.4}$$

Finally, for all $X \subseteq N$, the time for computing $b(X,Y)$ can be calculated through the formula $\sum_{m=0}^{n} C_n^m O^*(2^{n+m})$. Again by the Binomial theorem, we have Equation 4.5.

$$\sum_{m=0}^{n} C_n^m O^*(2^{n+m}) = O^*(6^n) \tag{4.5}$$

According to the computation steps of Equations 4.3, 4.4 and 4.5, since we did not use any look-up table to store the exponential number of $a(X)$ values to speed up the calculation of $b(X,Y)$, the space used is only polynomial. This completes the proof.

Now we give another way to compute all $b(X,Y)$ by using exponential space. Its time and space complexities are given in Lemma 4.2.

Lemma 4.2: For all $Y \subseteq X$ and for all $X \subseteq N$, $b(X,Y)$ can be obtained in $O(4^n)$ time and $O^*(2^n)$ space.

PROOF: As before, by using the fast zeta transform technique introduced in [2], all $a(X)$ values can be computed in $O^*(2^n)$ time and $O^*(2^n)$ space. Then for some given X and Y, according to Equation 4.2, since all $a(X)$ values are known, $b(X,Y)$ can be computed in time 2^r where $r = |Y|$. The time for computing $b(X,Y)$ for all $Y \subseteq X$ equals $\sum_{r=0}^{m} C_m^r \cdot 2^r = 3^m$. Similarly, the time for computing $b(X,Y)$ for all $X \subseteq N$ equals $\sum_{m=0}^{n} C_n^m \cdot 3^m = 4^n$. This finishes the proof.

4.4 Four algorithms for computing all $p_k^X(n_X)$

As mentioned in Section 4.1, we need to compute $\prod_{i=1}^{m} t_i$ $p_k^X(n_X) = p_k^X(n_1, n_2, ..., n_m)$ values, where $0 \le n_i \le t_i - 1$ and $1 \le i \le m$. Without loss of generality, we assume the positive integers in $\{n_1, n_2, ..., n_m\}$ form a set $n_Y = \{n_1, ..., n_r\}$, where $Y = \{1, 2, ..., r\}$ and $0 \le r \le m$. Then from the definitions of $a(X)$ and $b(X,Y)$, we have $p_1^X(n_1, n_2, ..., n_m) = b(X, \{1, 2, ..., r\})$ and $p_1^X(0, 0, ..., 0) = a(X)$. Now for brevity of notation, for any subset $Z = \{r_1, ..., r_i\} \subseteq Y$, we use $(n_X - 1^Z)$ to denote the set $\{n_1, ..., n_{r_1} - 1, ..., n_{r_i} - 1, n_{r_i+1}, ..., n_m\}$, i.e., for all $j \in Z$, the corresponding n_j values are decremented by 1, and for all $j \notin Z$, we keep the corresponding n_j values. Then for $2 \le q \le k$, we use the following recursive function to obtain $p_q^X(n_X)$.

$$p_q^X(n_X) = \sum_{Z \subseteq Y} b(X, Z) \cdot p_{q-1}^X(n_X - 1^Z) \qquad (4.6)$$

Basically, this equation tells us how to calculate the $p_q^X(n_X)$ value when given $p_{q-1}^X(n_X - 1^Z)$ values for all $Z \subseteq Y$. For example, when $Z = \varnothing$, $b(X, \varnothing) = a(X)$ and $p_{q-1}^X(n_X - 1^Z) = p_{q-1}^X(n_X)$. We already know $a(X)$ means the number of sets in F that avoid X, and $p_{q-1}^X(n_X)$ means the number of (q-1)-tuples from F where for each $j \in X$ the number of the element j in the union of every (q-1)-tuple is n_j; thus the product of $a(X)$ and $p_{q-1}^X(n_X)$ is the total number of ways to add a set to each of the $p_{q-1}^X(n_X)$ (q-1)-tuples to make it a q-tuple while keeping n_X unchanged. Similarly, for each nonempty $Z \subseteq Y$, we know $b(X, Z)$ means the number of sets in F that cover Z but avoid $X \setminus Z$, where $Z \subseteq Y \subseteq X$, and $p_{q-1}^X(n_X - 1^Z)$ means the number of (q-1)-tuples from F where for each $j \in X$ the number of the element j in the union of every (q-1)-tuple equals the updated n_j value in the set $(n_X - 1^Z)$; thus the product of $b(X, Z)$ and $p_{q-1}^X(n_X - 1^Z)$ is the total number of ways to add a set to each of the $p_{q-1}^X(n_X - 1^Z)$ (q-1)-tuples to make it a q-tuple while satisfying all the n_j values in the set n_X. Finally, the summation of all these products yields the number of q-tuples from F such that for each $j \in X$ the number of the element j in the union of every q-tuple equals n_j, which is $p_q^X(n_X)$.

So according to Equation 4.6, in order to get all $p_k^X(n_X)$, we need to calculate all $p_q^X(n_X)$ where $1 \le q < k$. We now give four algorithms for computing all $p_k^X(n_X)$.

But first we will analyze the special case where the maximum integer t in the integral coverage requirement set $T = \{t_1, ..., t_n\}$ equals 1. In this case, set multi-

covering becomes the set covering problem. Then as mentioned in Section 4.1, we only need to compute $\prod_{i=1}^{m} t_i = 1$ number of $p_k^X(n_X) = p_k^X(\underbrace{0,...,0}_{m})$ values. This means that the number of positive integers in the set $n_X = \{n_1, n_2, ..., n_m\}$ is zero, i.e., the set Y in Equation 4.6 is an empty set. Accordingly, Equation 4.6 becomes $p_k^X(0,...,0) = b(X,\varnothing) \cdot p_{k-1}^X(0,...,0) = a(X) \cdot p_{k-1}^X(0,...,0)$. Since $p_1^X(0,...,0) = a(X)$, we can obtain $p_k^X(0,...,0) = (a(X))^k$. Finally from Equations 3.2 and 4.1, we obtain $c_k(F) = \sum_{X \subseteq N} (-1)^{|X|} \cdot (a(X))^k$, which is the same as the formula given in [2] for counting the number of *k*-tuples that satisfy the set covering requirement. As discussed in [2], based on whether we use exponential space or not (c.f. Section 4.2), $c_k(F)$ can be computed in $O^*(2^n)$ time and $O^*(2^n)$ space, or can be computed in $O^*(3^n)$ time and polynomial space.

For the following, we assume that the maximum integer t in the integral coverage requirement set $T = \{t_1, ..., t_n\}$ is greater than or equal to 2.

Algorithm 1 for computing all $p_k^X(n_X)$

Input: The value k where $t \le k \le tn$; the set $X = \{1, 2, ..., m\}$; the integral coverage requirement set for X, i.e., $T_X = \{t_1, t_2, ..., t_m\}$. Here T_X is a subset of T, and we use $\min(T_X)$ and $\max(T_X)$ to denote the minimum and the maximum integers respectively in the set T_X.

Output: The values for all $p_k^X(n_X)$.

1: For all $X \subseteq N$, by using the fast zeta transform technique given in [2], we compute all $a(X)$ and store them in a look-up table.

2: Based on the first step, for all $Y \subseteq X$ and $X \subseteq N$, we compute all $b(X,Y)$ and store them in another look-up table.

3: For q=2 to k do:

4: By using Equation 4.6, we compute all $p_q^X(n_X)$ from $p_q^X(0,...,0)$ to $p_q^X(\min(q,t_1-1),...,\min(q,t_i-1),...,\min(q,t_m-1))$ and we store all these $p_q^X(n_X)$ values in a look-up table. Here the function $\min(q,t_i-1)$ means choosing the minimum value between q and (t_i-1).

5: End For.

Without storing all of the $p_q^X(n_X)$ values in a table, we have the second algorithm for computing all $p_k^X(n_X)$.

Algorithm 2 for computing all $p_k^X(n_X)$

1: Same as the first step in Algorithm 1.

2: Same as the second step in Algorithm 1.

3: For each of the $\prod_{i=1}^{m} t_i$ number of $p_k^X(n_X)$, where $p_k^X(n_X)$ is from $p_k^X(0,...,0)$ to $p_k^X(t_1-1,...,t_m-1)$, we use Equation 4.6 to compute their values directly without storing any of these values in a table.

Then, without storing all of the $b(X,Y)$ values into a table, we have the third algorithm for computing all $p_k^X(n_X)$.

Algorithm 3 for computing all $p_k^X(n_X)$

1: Same as the first step in Algorithm 1.

2: Same as the third step in Algorithm 2. But since we did not store all the $b(X,Y)$ values into a look-up table, we need to use Equation 4.2 to calculate the $b(X,Y)$ value for each $Y \subseteq X$.

Finally, without storing all of the $a(X)$ values into a table, we have the fourth algorithm for computing all $p_k^X(n_X)$.

Algorithm 4 for computing all $p_k^X(n_X)$

1: Same as the third step in Algorithm 2. But since we did not store all the $a(X)$ and $b(X,Y)$ values into the look-up tables, we need to calculate them on the fly.

With these algorithms for computing all $p_k^X(n_X)$, we can calculate $n_k(X)$ and then $c_k(F)$. We analyze in the following the time and space complexities for calculating $c_k(F)$ through using these four algorithms for computing all $p_k^X(n_X)$.

4.5 Time and space complexities for calculating $c_k(F)$

Theorem 4.3: By using Algorithm 1 for computing all $p_k^X(n_X)$, $c_k(F)$ can be computed in $O^*((2t)^n)$ time and $O^*((t+1)^n)$ space.

PROOF: The first step of Algorithm 1 uses $O^*(2^n)$ time and $O^*(2^n)$ space. For the second step, according to Lemma 4.2, computing all $b(X,Y)$ takes time $O(4^n)$. Obviously there are $\sum_{m=0}^{n} C_n^m 2^m = 3^n$ $b(X,Y)$, so storing all $b(X,Y)$ in a look-up table takes $O^*(3^n)$ space.

In the 'For' loop (step 3 to step 5), we calculate all $p_q^X(n_X)$ from $q=2$ to $q=k$ and store all these $p_q^X(n_X)$ values in a look-up table. So according to Equation 4.6, for each $p_q^X(n_X)$, since all the $b(X,Y)$ values have been stored and so have all the $p_{q-1}^X(n_X)$ values, the time to compute $p_q^X(n_X)$ is $\sum_{j=0}^{r} C_r^j = 2^r$ where r is the number of positive integers in the set n_X. So in order to calculate the total time for calculating all $p_q^X(n_X)$, we just need to count how many $p_q^X(n_X)$ we need to compute.

Since we know the number of positive integers in the set n_x is r, for each q where $2 \le q \le k$, the number of $p_q^x(n_x)$ we need to compute equals $\prod_{i=1}^{r} \min(q, t_i - 1)$, i.e., those $p_q^x(n_x)$ from $p_q^x(\underbrace{1,...,1}_{r},\underbrace{0,...,0}_{m-r})$ to $p_q^x(\underbrace{\min(q,t_1-1),...,\min(q,t_r-1)}_{r},\underbrace{0,...,0}_{m-r})$.

So if $q \le \min(T_x) - 1 \le t - 1$, the number of $p_q^x(n_x)$ we need to compute is q^r, i.e., all $p_q^x(n_x)$ from $p_q^x(\underbrace{1,...,1}_{r},\underbrace{0,...,0}_{m-r})$ to $p_q^x(\underbrace{q,...,q}_{r},\underbrace{0,...,0}_{m-r})$. Similarly, if $t - 1 < q \le k$, the number of $p_q^x(n_x)$ we need to compute equals $\prod_{i=1}^{r}(t_i - 1)$ which is less than $(t-1)^r$, i.e., all $p_q^x(n_x)$ from $p_q^x(\underbrace{1,...,1}_{r},\underbrace{0,...,0}_{m-r})$ to $p_q^x(\underbrace{t_1-1,...,t_r-1}_{r},\underbrace{0,...,0}_{m-r})$. Finally, if $\min(T_x) \le q \le \max(T_x) - 1 \le t - 1$, the number of $p_q^x(n_x)$ we need to compute is at most q^r.

From the above analyses, for a given n_x where the number of positive integers equals r and for all $2 \le q \le k$, the total number of $p_q^x(n_x)$ we have computed is at most:

$$\sum_{q=2}^{t-1} q^r + (k - t + 1) \cdot (t-1)^r \qquad (4.7)$$

As mentioned earlier in this proof, since the time for computing each $p_q^x(n_x)$ is 2^r, the total time for computing all these $p_q^x(n_x)$ is at most

$$2^r \cdot (\sum_{q=2}^{t-1} q^r + (k - t + 1) \cdot (t-1)^r)$$

Then for all n_x where *r*, the number of positive integers in each of them, varies from 0 to *m*, the total time for computing all $p_q^x(n_x)$ is at most:

$$\sum_{r=0}^{m} C_m^r (2^r \cdot (\sum_{q=2}^{t-1} q^r + (k-t+1)\cdot(t-1)^r)) = \sum_{q=2}^{t-1} (2q+1)^m + (k-t+1)\cdot(2t-1)^m$$

Now according to Equation 4.1 which is for computing $n_k(X)$, the total time for computing $n_k(X)$ is less than $\sum_{q=2}^{t-1} (2q+1)^m + (k-t+1)\cdot(2t-1)^m + t^m$, where the last term t^m accounts for the at most t^m number of additions of $p_k^x(n_x)$ to obtain $n_k(X)$.

Finally, according to Equation 3.2 which is for calculating $c_k(F)$, the time for computing $c_k(F)$ is at most:

$$\sum_{m=0}^{n} C_n^m (\sum_{q=2}^{t-1} (2q+1)^m + (k-t+1)\cdot(2t-1)^m + t^m)$$

$$= \sum_{q=2}^{t-2} (2q+2)^n + (k-t+2)\cdot(2t)^n + (t+1)^n$$

Now according to the following helping lemma, Lemma 4.4,

$$\sum_{q=2}^{t-2} (2q+2)^n + (k-t+2)\cdot(2t)^n + (t+1)^n$$

$$= O((t-1)\cdot(2t-2)^n) + (k-t+2)\cdot(2t)^n + (t+1)^n = O^*((2t)^n).$$

Lemma 4.4: For any positive integer *s*, we have

$$(s+1)\cdot(s/2)^n \le \sum_{i=1}^{s} i^n \le (s+1)\cdot s^n / 2.$$

PROOF: First we define a function $f(x) = x^n + (s-x)^n$, where $0 \le x \le s$. By computing the second derivative of $f(x)$, we know $f(x)$ is a convex function. Thus it achieves the largest value at the boundaries of the x values, which are either $x = 0$ or $x = s$. By computing the first derivate of $f(x)$, we find that it achieves its smallest value at $x = s/2$. So we have $2^{1-n}s^n \le f(x) \le s^n$ for all $0 \le x \le s$. Then by replacing x with all its integer values from 0 to s, and summing these inequalities together, we obtain the result. This finishes the proof.

After proving the time complexity for calculating $c_k(F)$, we now turn to the space complexity. This is equivalent to finding out the total interim values we have stored in the look-up tables. We know already the total spaces for storing all $a(X)$ and $b(X,Y)$ values are $O^*(3^n)$, and now we only need to know the total number of $p_q^X(n_X)$ we have stored in the table. As given in Equation 4.7, for a given n_X and for all $2 \le q \le k$, the total number of $p_q^X(n_X)$ we have computed is at most $\sum_{q=2}^{t-1} q^r + (k-t+1)\cdot(t-1)^r$. Then for all n_X, the total number of $p_q^X(n_X)$ we have stored is at most:

$$\sum_{r=0}^{m} C_m^r \left(\sum_{q=2}^{t-1} q^r + (k-t+1)\cdot(t-1)^r\right) = \sum_{q=2}^{t-2} (q+1)^m + (k-t+2)\cdot t^m$$

Finally, for all $X \subseteq N$, the total number of $p_q^X(n_X)$ we have stored is at most:

$$\sum_{m=0}^{n} C_n^m \left(\sum_{q=2}^{t-2} (q+1)^m + (k-t+2)\cdot t^m\right) = \sum_{q=2}^{t-2} (q+2)^n + (k-t+2)\cdot(t+1)^n$$

Again, according to Lemma 4.4, we have:

$$\sum_{q=2}^{t-2}(q+2)^n+(k-t+2)\cdot(t+1)^n=O(t^{n+1}+(k-t+2)\cdot(t+1)^n)=O^*((t+1)^n)$$

Since $t\geq 2$, all the time and spaces consumed in the first and the second step of Algorithm 1 can be subsumed in $O^*((2t)^n)$ and $O^*((t+1)^n)$, respectively. This finishes the proof of Theorem 4.3.

Next, we analyze the time and space complexities for calculating $c_k(F)$ by using Algorithm 2.

Theorem 4.5: By using the Algorithm 2 for computing all $p_k^X(n_X)$, $c_k(F)$ can be computed in $O^*((2^k+1)^n)$ time and $O^*(3^n)$ space.

PROOF: First, for computing all $p_k^X(n_X)$, Algorithm 2 chooses to compute each $p_k^X(n_X)$ using the recursive function in Equation 4.6. According to this Equation, for some $X\subseteq N$ where $|X|=m$, we know that each $p_q^X(n_X)$ where $1\leq q<k$ can be called by at most 2^m number of $p_{q+1}^X(n_X)$. From this observation we conclude that after each $p_q^X(n_X)$ has been called by at most 2^m number of $p_{q+1}^X(n_X)$, all the $p_{q+1}^X(n_X)$ values have been calculated. So in order to calculate the total time for calculating all $p_{q+1}^X(n_X)$ (represented as $\sum_{n_X}T(p_{q+1}^X(n_X))$), we have the following two steps. First, we need to compute the time for each $p_q^X(n_X)$ being called by at most 2^m number of $p_{q+1}^X(n_X)$ (the calculating time for $p_q^X(n_X)$ is denoted as $T(p_q^X(n_X))$). According to Equation 4.6, this needs to include the total time for computing $b(X,Z)$ for all $Z\subseteq Y\subseteq X$ (represented as $\sum_Z T(b(X,Z))$) and the 2^m number of product times between $b(X,Z)$ and $p_q^X(n_X+1^Z)$. Second, by summing the calculating times in

the first step for all $p_q^X(n_X)$ we can obtain the upper bound for $\sum_{n_X} T(p_{q+1}^X(n_X))$. Thus we have the following inequality.

$$\sum_{n_X} T(p_{q+1}^X(n_X)) \le \sum_{n_X} (2^m \cdot T(p_q^X(n_X)) + \sum_Z T(b(X,Z)) + 2^m) \tag{4.8}$$

We first calculate $\sum_Z T(b(X,Z))$. Since all $b(X,Y)$ values have been stored in the look-up tables, each look-up takes constant time. So we have $\sum_Z T(b(X,Z)) = O(2^m)$. And since all $p_1^X(n_X)$ values are equivalent to the corresponding $b(X,Y)$ values, we have $\sum_{n_X} T(p_1^X(n_X)) = \sum_Z T(b(X,Z)) = O(2^m)$. Note that, similar to the proof for Theorem 4.3, when $1 \le q < t$, there are $(q+1)^m$ $p_q^X(n_X)$; when $t \le q \le k$, there are t^m $p_q^X(n_X)$. From this observation, by repeatedly using Inequality 4.8, we can obtain the upper bound for the total time for computing all $p_k^X(n_X)$ in terms of the following inequality.

$$\sum_{n_X} T(p_k^X(n_X)) \le 3 \cdot 2^{km} + 2\sum_{i=2}^{t-1} (2^{(k-i)m} \cdot (i+1)^m) + 2\sum_{i=t}^{k-1} (t^m \cdot 2^{(k-i)m}) = O^*(2^{km}) \tag{4.9}$$

Now similar to the proof for Theorem 4.3, according to Equation 4.1 which is for computing $n_k(X)$, the total time for computing $n_k(X)$ equals $O^*(2^{km})$. Then finally, according to Equation 3.2 which is for computing $c_k(F)$, the time for computing $c_k(F)$ equals $\sum_{m=0}^{n} C_n^m (O^*(2^{km})) = O^*((2^k+1)^n)$.

For the space complexity, as have been shown in the beginning of the proof of Theorem 4.3, the spaces we need to store all the $a(X)$ values

and $b(X,Y)$ values are $O^*(2^n)$ and $O^*(3^n)$, respectively. So the total space complexity is $O^*(3^n)$. This ends the proof of Theorem 4.5.

Remark 2: The time complexity given in Theorem 4.5 is a loose upper bound especially for $t \square k \le tn$. The reason is that, in our time complexity analysis, we have assumed that each $p_q^X(n_X)$ is called by 2^m number of $p_{q+1}^X(n_X)$. However, this is not true for every $p_q^X(n_X)$. For example, for some $q \ge t-1$, $p_q^X(t-1,...,t-1)$ can only be called by one $p_{q+1}^X(n_X)$ which is $p_{q+1}^X(t-1,...,t-1)$. For all $t-1 \le q < k$, this counting error is present in each call of $p_q^X(n_X)$ for calculating $p_{q+1}^X(n_X)$. So the time complexity analysis given in the proof for Theorem 4.5 is tighter for smaller k values than for larger k values. Currently we can not come up with a tighter time upper bound analysis for Algorithm 2. Since we will also use Inequality 4.8 for analyzing the time complexities of Algorithms 3 and 4, this remark can also be applied to Theorems 4.6 and 4.7.

Theorem 4.6: By using Algorithm 3 for computing all $p_k^X(n_X)$, $c_k(F)$ can be computed in $O^*((3 \cdot 2^{k-1}+1)^n)$ time and $O^*(2^n)$ space.

PROOF: The only difference between Algorithm 2 and Algorithm 3 is that we do not store all the $b(X,Y)$ values in a look-up table in the latter. This will affect the time for computing $\sum_{n_X} T(p_q^X(n_X))$ including the initial $\sum_{n_X} T(p_1^X(n_X))$ and the time for computing $\sum_Z T(b(X,Z))$. Now according to Equation 4.2 which is for computing $b(X,Z)$, since all $a(X)$ values have been stored in the look-up table (c.f. step 1 of Algorithm 3), we know the time for computing $b(X,Z)$ is equal to $2^{|Z|}$.

From this we know the total time for computing all $b(X,Z)$ where $Z \subseteq X$ is equal to $O(\sum_{i=0}^{m} C_m^i 2^i) = O(3^m)$. In addition, as mentioned in the proof for Theorem 4.5, we have $\sum_{n_X} T(p_1^X(n_X)) = \sum_{Z} T(b(X,Z)) = O(3^m)$. Now by repeatedly using Inequality 4.8, we know that the total time for computing all $p_k^X(n_X)$ which is represented as $\sum_{n_X} T(p_k^X(n_X))$ can be calculated from the following inequality.

$$\sum_{n_X} T(p_k^X(n_X)) \le \sum_{i=1}^{t}(3^m \cdot 2^{(k-i)m} \cdot i^m) + \sum_{i=t+1}^{k}(3^m \cdot 2^{(k-i)m} \cdot t^m) + \sum_{i=1}^{t-1}(2^{(k-i)m} \cdot (i+1)^m) + \sum_{i=t}^{k-1}(2^{(k-i)m} \cdot t^m)$$

$$= O^*((3 \cdot 2^{k-1})^m) \qquad (4.10)$$

According to Inequality 4.10, we know the total time for computing $n_k(X)$ is less than $O^*((3 \cdot 2^{k-1})^m)$. Then according to Equation 3.2, we know the time for computing $c_k(F)$ is at most $\sum_{m=0}^{n} C_n^m (O^*((3 \cdot 2^{k-1})^m)) = O^*((3 \cdot 2^{k-1} + 1)^n)$.

For the space complexity, since we only store all the $a(X)$ values in the look-up table, the total space used is also $O^*(2^n)$. This ends the proof of Theorem 4.6.

Theorem 4.7: By using Algorithm 4 for computing all $p_k^X(n_X)$, $c_k(F)$ can be computed in $O^*((2^{k+1}+2)^n)$ time and polynomial space.

PROOF: The only difference between Algorithm 3 and Algorithm 4 is that we did not store all the $a(X)$ values in a look-up table in the latter. Similarly, this will affect the time for computing $\sum_{n_X} T(p_q^X(n_X))$ and the time for computing $\sum_{Z} T(b(X,Z))$. Now according to Equation 4.2 which is for computing $b(X,Z)$, we know the time for computing $b(X,Z)$ is equal to $O^*(3^{|Z|} \cdot 2^{n-m})$ (c.f. Equation 4.3). So for all $Z \subseteq X$,

$\sum_{Z} T(b(X,Z)) = O^*(\sum_{i=0}^{m} C_m^i (3^i \cdot 2^{n-m})) = O^*(2^{n+m})$. Similar to the proof of Theorem 4.5, we have $\sum_{n_X} T(p_1^X(n_X)) = \sum_{Z} T(b(X,Z)) = O^*(2^{n+m})$.

From the above analysis, by repeatedly using Inequality 4.8, we can compute the total time for calculating all $p_k^X(n_X)$ through the following inequality.

$$\sum_{n_X} T(p_k^X(n_X)) \le \sum_{i=1}^{t} (O^*(2^{n+m}) \cdot 2^{(k-i)m} \cdot i^m) + \sum_{i=t+1}^{k} (O^*(2^{n+m}) \cdot 2^{(k-i)m} \cdot t^m)$$

$$+ \sum_{i=1}^{t-1} (2^{(k-i)m} \cdot (i+1)^m) + \sum_{i=t}^{k-1} (2^{(k-i)m} \cdot t^m)$$

$$= O^*(2^{n+m} \cdot 2^{(k-1)m}) \qquad (4.11)$$

Now according to Inequality 4.11, the total time for computing $n_k(X)$ is less than $O^*(2^{n+m} \cdot 2^{(k-1)m})$. Then finally the time for computing $c_k(F)$ is less than

$\sum_{m=0}^{n} C_n^m (O^*(2^{n+m} \cdot 2^{(k-1)m})) = O^*((2^{k+1}+2)^n)$.

For the space complexity, since we did not store all the $a(X)$, $b(X,Y)$ and $p_q^X(n_X)$ values in the look-up tables, the total space used is polynomial. This completes the proof of Theorem 4.7.

5. A Constructive Algorithm for the Set Multi-covering Problem

Although we have computed the minimum number of sets that satisfy the coverage requirement, we have not really constructed these sets. In this section,

we present an algorithm called **ESMC** for picking the minimum number of sets such that each element in the universe is covered by at least the required number of times as specified in the integral coverage requirement set. Before giving this constructive algorithm, we need to define two basic elements pair operations.

5.1 Two basic elements pair operations

We define two kinds of elements pair operations over a series of sets. One is called elements pair separation, which is to divide a set into two sets such that any pair of elements in the original set will fall into two different sets; the other is called elements pair coalition, which is to merge a pair of elements in the same set into a single element. Their formal definitions are given below.

Elements Pair Separation: For any set $s = \{a, b, x_1, ..., x_m\}$ in F which covers a pair of elements a and b, we replace the set s by separating the two elements into two different sets $s_a = \{a, x_1, ..., x_m\}$ and $s_b = \{b, x_1, ..., x_m\}$.

Elements Pair Coalition: For any set $s = \{a, b, x_1, ..., x_m\}$ in F which covers a pair of elements a and b, we replace the set s with the set $s_{ab} = \{ab, x_1, ..., x_m\}$ where the two elements a and b are merged into a new single element ab.

5.2 The constructive algorithm for the set multi-covering problem

We now give a constructive algorithm for finding the minimum number of sets in F that satisfy the integral coverage requirement set T. This algorithm is based on finding the minimum k value such that the value of $c_k(F)$ is greater than zero.

ESMC: Exact Set Multi-Cover Algorithm

Input: A family F of subsets over the universe N; a coverage requirement set T which states the integral coverage requirement for each element in N.

Output: The minimum number of sets from F to satisfy the requirement T.

1: Set $F_{bak} = F$.

2: Calculate the minimum value of k such that $c_k(F) > 0$.

3: Pick any element a in the universe N.

4: Find all the elements $\{x_1, ..., x_m\}$ in N that appear with a in some set in F.

5: Set $F_0 = F$.

6: **For** i=1 to m **do**:

7: $F = F_0$.

8: For the pair of elements (a, x_i), we apply the Elements Pair Separation operation over the set F to generate a new set called F_i.

9: Calculate the value of $c_k(F_i)$.

10: **End For**

11: If all of the $c_k(F_i)$ values where $0 \le i \le m$ are greater than zero, we can deduce that there exists a set in the optimal covering which only covers the element a since otherwise there must exist some x_i whose separation with the element a can make $c_k(F_i) \le 0$. So we just pick this set in F which covers a and contains the least number of elements. We then decrement the value of k by 1 and update the coverage requirement set T, i.e., for all elements x_i in the picked set we decrement each of the corresponding t_i values by 1. Also if any $t_i \le 0$ we remove the element i in the universe set N.

12: Else we pick any i such that $c_k(F_i) \le 0$. Then for the pair of elements $\{a, x_i\}$, we apply the Elements Pair Coalition operation over the set F. Note that the element a has become a new single element (ax_i).

13: Repeat step 4 to step 12 until we have picked a set from F.

14: Set $F = F_{bak}$ and we repeat step 3 to step 13 until $k = 0$.

5.3 Correctness analysis

First, according to step 2, we know that the value of k we choose guarantees that we only use the minimum number of sets to satisfy the coverage requirement. Second, according to step 11, we know that, when we pick a set from F in each step, we can guarantee that the picked set must exist in some optimal legal (k,T) covering sets. From this we also know that, when we pick this set, there must exist a legal $(k-1,T')$ cover where T' is the updated coverage requirement set after picking a subset from F. From the above analysis, we can conclude that we do pick the minimum number of sets from F that satisfies the coverage requirement set T.

5.4 Time and space complexities analyses

The time of the ESMC algorithm can be divided into two parts. The first part is due to step 2, which is to calculate the minimum k value for a legal (k,T) cover. By using binary search, since $t \le k \le tn$, its time corresponds to $O(\log(tn))$ calculations of $c_k(F)$ (c.f. Section 3.2). The second part is due to steps 4 to 12 of the algorithm which is to pick a subset from F. We can easily see that it takes $O(n^2)$ calculations of $c_k(F)$. Since we need to pick k subsets, we need $O(kn^2)$ evaluations of $c_k(F)$ in total. So the overall time complexity is dependent on the time complexity for computing $c_k(F)$. Now according to Theorem 4.3, we have the following corollary.

COROLLARY 5.1: By using Algorithm 1 for computing all $p_k^x(n_x)$, the ESMC algorithm takes $O^*((2t)^n)$ time and $O^*((t+1)^n)$ space where t is the maximum integer in the coverage requirement set T.

Similarly, according to Theorems 4.5, 4.6 and 4.7, we can get the corresponding time and space complexities for the ESMC algorithm. But since the first part of the ESMC algorithm needs to test different k values for finding the minimum k value to make a legal (k,T) cover, the time consumed in this part could be very large depending on which k values we have tested. But since $t \le k \le tn$, we have the following Corollary 5.2 which corresponds to Theorem 4.7. By employing Theorems 4.5 and 4.6 we can obtain similar results as Corollary 5.2 which we omit here.

COROLLARY 5.2: By using Algorithms 4 for computing all $p_k^x(n_x)$, the ESMC algorithm takes $O^*(2^{O(tn^2)})$ time and polynomial space.

6. Conclusion

In this paper, we have generalized the inclusion-exclusion based exact algorithm for the set covering problem to the set multi-covering problem. We have presented a family of exact algorithms to solve the set multi-covering problem through different tradeoffs between the time and space complexities. We have shown that by using more space, the time complexity can be significantly reduced.

Although the simple greedy strategy applied to the set covering problem can be applied to the set multi-covering problem to yield the same approximation ratio $O(\log n)$, our fastest exact algorithm which takes $O^*((2t)^n)$ time

and $O^*((t+1)^n)$ space consumes much more time and space than the currently fastest exact algorithm for the set covering problem which takes $O^*(2^n)$ time and $O^*(2^n)$ space [2]. In addition, if we restrict to polynomial space, the time consumed for the set multi-covering problem is much longer than its set covering counterpart which takes $O^*(3^n)$ time [2].

The following are some possible directions for designing exact algorithms for the set multi-covering problem. First, as mentioned in Remark 2, the time complexity analyses for the Algorithms 2, 3 and 4 for computing all the $p_k^x(n_x)$ values are not tight, so much tighter time complexity analyses for the three algorithms will be needed. Second, it is possible to extend our algorithms to other generalized covering problems, such as multi-set multi-cover [10]. Third, as shown in [4] and [9], some techniques in information theory can help analyze exact algorithms that need counting steps. So it will be interesting to apply this kind of technique to those generalized set covering scenarios. Finally, like what was done by the authors in [6], it might be possible to apply our algorithm to wireless scheduling problems which have drawn increasing attention in the wireless networking community in recent years.

Acknowledgements

This research is supported in part by a Hong Kong RGC-GRF grant (7136/07E), the national 863 high-tech R&D program of the Ministry of Science and Technology of China under grant No. 2006AA10Z216, the National Science Foundation of China under grant No. 60604033, and the National Basic Research

Program of China Grant 2007CB807900, 2007CB807901.

References

[1] E.T. Bax. Inclusion and exclusion algorithms for the Hamiltonian path problem. *Information Processing Letters*, 47(4):203-207,1993.

[2] A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via Inclusion--Exclusion. *SIAM Journal on Computing*, Special Issue for FOCS 2006, to appear.

[3] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets Möbius: fast subset convolution. In *Proc. 39th Annual ACM Symposium on Theory of Computing (STOC)*, San Diego, California, US, June 2007.

[4] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. The travelling salesman problem in bounded degree graphs. In *Proc. 35th ICALP* , Iceland, 2008.

[5] U. Feige. A threshold of ln *n* for approximating set cover. *Journal of the ACM*, 45(4):634-652, 1998.

[6] Q.-S. Hua and F.C.M. Lau. Exact and approximate link scheduling algorithms under the physical interference model. In *Proc. 5th SIGACT-SIGOPS International Workshop on Foundation of Mobile computing* (DIALM-POMC), Toronto, Canada, Aug. 2008.

[7] R.M. Karp. Dynamic programming meets the principle of inclusion-exclusion. *Operations Research Letters*, 1(2):49-51,1982.

[8] J. Nederlof. Fast polynomial-space algorithms using Möbius inversion: Improving on Steiner Tree and related problems, to appear in Proc. ICALP 2009.

[9] J. Radhakrishnan. Entropy and counting. In: *Mishra, J.C. (ed.) IIT Kharagpur Golden Jubilee Volume on Computational Mathematics, Modelling and Algorithms*, Narosa Publishers,New Delhi, 2001.

[10] V.V. Vazirani. Approximation Algorithms. Berlin: Springer, 2003.

[11] S. Rajagopalan and V.V. Vazirani. Primal-dual RNC approximation algorithms for set cover and covering integer programs. *SIAM Journal on Computing*, 28(2):525-540, 1998.

Appendix

In this appendix, we give a very simple example to show how we calculate the value of $c_k(F)$ and how the ESMC algorithm works for the given example.

Suppose the universe $N = \{1,2,3\}$, the family of subsets over N is $F = \{\{1,2\},\{1,3\},\{2,3\}\}$ and the coverage requirement set $T = \{2,1,1\}$. Now we first find the minimum k value to make a legal (k,T) cover. This is equivalent to calculating the minimum k value such that $c_k(F) > 0$. Suppose we first test the case where $k = 2$.

According to Equation 3.2, we have $c_2(F) = \sum_{X \subseteq N} (-1)^{|X|} \cdot n_2(X)$. Now due to Equation 4.1, we have $n_2(X) = \sum_{\substack{0 \le n_i \le t_i - 1 \\ 1 \le i \le |X|}} p_2^X(n_1,...,n_{|X|})$. Then based on these equations we have Table A.1 which is to calculate $n_2(X)$ values for all $X \subseteq N$.

Table A.1: Calculating $n_2(X)$ for all $X \subseteq N$

X	$n_2(X)$
$\varnothing$	$p_2^{\varnothing}(\varnothing)$
{1}	$p_2^{\{1\}}(0) + p_2^{\{1\}}(1)$
{2}	$p_2^{\{2\}}(0)$
{3}	$p_2^{\{3\}}(0)$
{1,2}	$p_2^{\{1,2\}}(0,0) + p_2^{\{1,2\}}(1,0)$
{1,3}	$p_2^{\{1,3\}}(0,0) + p_2^{\{1,3\}}(1,0)$
{2,3}	$p_2^{\{2,3\}}(0,0)$
{1,2,3}	$p_2^{\{1,2,3\}}(0,0,0) + p_2^{\{1,2,3\}}(1,0,0)$

The next step is to compute all the $p_2^X(n_1,...,n_{|X|})$ values on the right side of Table A.1. By combining Equation 4.6 which computes $p_q^X(n_1,...,n_{|X|})$ and Equation 4.2 which computes $b(X,Y)$, we have Table A.2.

Table A.2: Calculating $p_2^X(n_1,...,n_{|X|})$ for all $X \subseteq N$

X	$n_2(X)$
$\varnothing$	$p_2^{\varnothing}(\varnothing) = b(\varnothing,\varnothing)\cdot p_1^{\varnothing}(\varnothing) = a(\varnothing)\cdot a(\varnothing)$ =3*3=9.
{1}	(1): $p_2^{\{1\}}(0) = b(\{1\},\varnothing)\cdot p_1^{\{1\}}(0)$ $= a(\{1\})\cdot b(\{1\},\varnothing)$ $= a(\{1\})\cdot a(\{1\})$ =1*1=1; (2): $p_2^{\{1\}}(1) = b(\{1\},\varnothing)\cdot p_1^{\{1\}}(1) + b(\{1\},\{1\})\cdot p_1^{\{1\}}(0)$ $= a(\{1\})\cdot b(\{1\},\{1\}) + b(\{1\},\{1\})\cdot b(\{1\},\varnothing)$ $= a(\{1\})\cdot [a(\varnothing) - a(\{1\})] + [a(\varnothing) - a(\{1\})]\cdot a(\{1\})$ =1*(3-1)+(3-1)*1=4; (3): $p_2^{\{1\}}(0) + p_2^{\{1\}}(1)$ =1+4=5.
{2}	$p_2^{\{2\}}(0) = b(\{2\},\varnothing)\cdot p_1^{\{2\}}(0)$ $= a(\{2\})\cdot b(\{2\},\varnothing)$ $= a(\{2\})\cdot a(\{2\})$ =1*1=1.
{3}	$p_2^{\{3\}}(0) = b(\{3\},\varnothing)\cdot p_1^{\{3\}}(0)$ $= a(\{3\})\cdot b(\{3\},\varnothing)$ $= a(\{3\})\cdot a(\{3\})$ =1*1=1.
{1,2}	(1): $p_2^{\{1,2\}}(0,0) = b(\{1,2\},\varnothing)\cdot p_1^{\{1,2\}}(0,0)$ $= a(\{1,2\})\cdot b(\{1,2\},\varnothing)$ $= a(\{1,2\})\cdot a(\{1,2\})$ =0*0=0; (2): $p_2^{\{1,2\}}(1,0) = b(\{1,2\},\varnothing)\cdot p_1^{\{1,2\}}(1,0) + b(\{1,2\},\{1\})\cdot p_1^{\{1,2\}}(0,0)$ $= a(\{1,2\})\cdot b(\{1,2\},\{1\}) + b(\{1,2\},\{1\})\cdot b(\{1,2\},\varnothing)$ $= a(\{1,2\})\cdot [a(\{2\}) - a(\{1\}\cup\{2\})] + [a(\{2\}) - a(\{1\}\cup\{2\})]\cdot a(\{1,2\})$ =0*(1-0)+(1-0)*0=0; (3): $p_2^{\{1,2\}}(0,0) + p_2^{\{1,2\}}(1,0)$ =0+0=0.
{1,3}	(1): $p_2^{\{1,3\}}(0,0) = a(\{1,3\})\cdot a(\{1,3\})$ =0*0=0;

	(2): $p_2^{\{1,3\}}(1,0) = b(\{1,3\},\varnothing)\cdot p_1^{\{1,3\}}(1,0) + b(\{1,3\},\{1\})\cdot p_1^{\{1,3\}}(0,0)$ $= a(\{1,3\})\cdot b(\{1,3\},\{1\}) + b(\{1,3\},\{1\})\cdot b(\{1,3\},\varnothing)$ =0*1+1*0=0; (3): $p_2^{\{1,3\}}(0,0) + p_2^{\{1,3\}}(1,0)$ =0+0=0.
{2,3}	$p_2^{\{2,3\}}(0,0) = a(\{2,3\})\cdot a(\{2,3\})$ =0*0=0.
{1,2,3}	(1): $p_2^{\{1,2,3\}}(0,0,0) = a(\{1,2,3\})\cdot a(\{1,2,3\})$ =0*0=0; (2): $p_2^{\{1,2,3\}}(1,0,0)$ $= b(\{1,2,3\},\varnothing)\cdot p_1^{\{1,2,3\}}(1,0,0) + b(\{1,2,3\},\{1\})\cdot p_1^{\{1,2,3\}}(0,0,0)$ $= a(\{1,2,3\})\cdot b(\{1,2,3\},\{1\}) + b(\{1,2,3\},\{1\})\cdot a(\{1,2,3\})$ =0*0+0*0=0; (3): $p_2^{\{1,2,3\}}(0,0,0) + p_2^{\{1,2,3\}}(1,0,0)$ =0+0=0.

Having calculated all the $n_2(X)$ values which are shown on the right side of Table A.2, we can obtain $c_2(F) = \sum_{X\subseteq N}(-1)^{|X|}\cdot n_2(X) = 9-5-1-1+0+0+0-0 = 2 > 0$, which means that there are two *2*-tuples that can satisfy the coverage requirement. Since the maximum integer in the coverage requirement set *T* is 2, we know the minimum *k* value we need to pick is 2. Actually, by calculating the $c_1(F)$ value, which is $c_1(F) = \sum_{X\subseteq N}(-1)^{|X|}\cdot n_1(X) = 3-3-1-1+0+0+0-0 = -2 < 0$, we can also conclude that the minimum *k* value is 2 since picking one set from *F* does not meet the coverage requirement.

Now according to the ESMC algorithm, we briefly show in the following how to pick the two sets that can satisfy the coverage requirement *T*.

First, according to step 3, we pick the element *1* in the universe *N*. Then we can find the elements $\{x_1 = 2, x_2 = 3\}$ that can appear with *1* in some subsets in *F*. Now according to step 6 to step 10, we obtain $F_1 = \{\{1\},\{2\},\{1,3\},\{2,3\}\}$ and $F_2 = \{\{1,2\},\{1\},\{3\},\{2,3\}\}$. From this we can calculate $c_2(F_1) \le 0$ and $c_2(F_2) \le 0$. Then according to step 12, we choose to merge the elements pair $(1,2)$. Now since the new single element (12) does not appear with any other elements in the set *F*, we have $m = 0$. Then since $c_2(F_0) = c_2(F) = 2 > 0$, according to step 11, we

just pick the first subset in F which is $\{1,2\}$. Similarly, we can pick the second subset in F which is $\{2,3\}$. This finishes the execution of the ESMC algorithm.