RESEARCH REPORT SERIES
(*Computing #2003-01)*

Preorder and Set Covering in the DISCRETE Edit System

Bor-Chung Chen and William E. Winkler

Statistical Research Division
U.S. Bureau of the Census
Washington D.C. 20233

Report Issued: September 10, 2003

Preorder and Set Covering in the DISCRETE Edit System*

Bor-Chung Chen and **William E. Winkler**

Abstract

Most combinatorial problems are very big—bigger than what can be handled efficiently on most fast computers—and hence the development and implementation of fast algorithms is very important. The DISCRETE edit system, based on the Fellegi and Holt model [1976] of editing, contains a combinatorial problem: the set covering problem (SCP). The two major components of the edit system are edit generation and error localization. The SCP is formulated many times in both components. Therefore, an efficient set covering algorithm is critical to the overall performance of the DISCRETE edit system. The preorder of traversing a tree is one of the structures used in the design of a set covering algorithm for the DISCRETE edit system. In this paper, we will describe a simple implementation of the preorder of traversing a tree used in a new set covering algorithm proposed by Chen [1998].

KEY WORDS: Redundant Covers, Subcovers, Integer Programming, Optimization, Lexicographic Ordering, Minimal Change Ordering, Ranking, Unranking, Successor

1 Introduction

The information gathered in any survey may contain inconsistent or incorrect data. These erroneous data need to be revised prior to data tabulations and retrieval. The revisions of the erroneous data should not affect the statistical inferences of the data. One of the important steps of this systematic revision process is computer editing. Fellegi and Holt [1976] provided the underlying basis of developing a computer editing system. The DISCRETE edit system (Winkler and Petkunas [1996]) is designed for general edits of discrete data. It utilizes the Fellegi-Holt model of editing and contains two major components: edit generation and error localization. An edit-generation algorithm, called the EGE algorithm, for the DISCRETE edit system was described in Winkler [1997]. The EGE algorithm is a much faster alternative to Algorithm 1, called the GKL algorithm, of Garfinkel, Kunnathur, and Liepins [1986]. In both of the EGE and GKL algorithms, the set covering routine is invoked many times to generate new implicit edits. Therefore, an efficient algorithm for the set covering problem becomes highly desirable to reduce the computation time of the edit generation. In error localization, the set covering problem, which, in fact, is an integer linear programming

*This paper reports the results of research and analysis undertaken by Census Bureau staff. It has undergone a more limited review than official Census Bureau publications. This report is released to inform interested parties of research and to encourage discussion.

problem, is used to identify the minimum number of fields in an erroneous record to change to pass all the edits. In this paper, an enhancement of the set covering algorithm, proposed by Chen [1998], for the DISCRETE edit system is described. The enhancement significantly reduces the storage requirements of the preorder of traversing a tree while keeping other computations as fast as possible.

The SCP in the DISCRETE edit system is applied twice, one in edit generation and the other in error localization. The first application in error localization is to find the minimal set of fields (the optimal solution) of a failed record to be modified to satisfy all explicit and implicit edits. The SCP is invoked once for each failed record. The second application in edit generation is to find all the minimal sets of edits that are unioned to cover all possible values of a field, called generating field (see Section 2). The second application is NOT to find an optimal solution but to find all prime cover solutions to the SCP. A prime cover is a cover in which each set in the cover is not redundant.

A detailed description of the set covering algorithm is given in Chen [1998]. Its preorder storage required is exponentially increasing with the survey size, measured in the number of fields in a record for error localization or the the number of explicit edits originally specified for edit generation. This storage requirement is generally making the algorithm difficult to be programmed and implemented. In this paper, we will discuss the elimination of the storage required in the set covering algorithm proposed by Chen [1998] and provide some enhancements to the implementation of the algorithm used in the DISCRETE Edit System.

We will use the following notations in this paper: $\boldsymbol{a}= (a_1, a_2, \ldots, a_n)$ has n fields. $a_i \in A_i$ for each i $1 \leq i \leq n$, where A_i is the set of possible values or code values which may be recorded in Field i. $|A_i| = n_i$. If $a_i \in A_i^o \subset A_i$, we also say

$$\boldsymbol{a} \in \boldsymbol{A}_i^o = A_1 \times A_2 \times \ldots \times A_{i-1} \times A_i^o \times A_{i+1} \times \ldots \times A_n. \tag{1}$$

The code space is $A_1 \times A_2 \times \ldots \times A_n = \boldsymbol{A}$. A record $\boldsymbol{y}$ fails edit E^r if

$$E^r : \bigcap_{j=1}^{n} \boldsymbol{A}_j^r$$

contains $\boldsymbol{y}$. If A_j^r is a proper subset of A_j, then field j is an entering field of E^r or field j enters E^r.

2 Background

The objective of error localization is to find the minimum number of fields to change if a record fails some of the edits. It can be formulated as a set covering problem. Let $\bar{E} = \{E^1, E^2, \cdots, E^m\}$ be a set of edits failed by a record $\boldsymbol{y}$ with n fields, and consider the set covering problem:

$$\begin{aligned} &\text{Minimize} && \textstyle\sum_{j=1}^{n} c_j x_j \\ &\text{subject to} && \textstyle\sum_{j=1}^{n} a_{ij} x_j \geq 1, \quad i = 1, 2, \cdots, m \end{aligned} \tag{2}$$

$$x_j = \begin{cases} 1, & \text{if field } j \text{ is to be changed;} \\ 0, & \text{otherwise,} \end{cases}$$

where

$$a_{ij} = \begin{cases} 1, & \text{if field } j \text{ enters } E^i; \\ 0, & \text{otherwise,} \end{cases}$$

and c_j is a measure of "confidence" in field j. We need to get $\bar{E}$ from a *complete* set of edits to obtain a meaningful solution to (2). A complete set of edits is the set of explicit (initially specified) edits and all essentially new (the definition of *essentially new* is given later in this section) implied edits derived from them. The dimension of the constraint matrix (a_{ij}) of 0's and 1's associated with (2) is $m \times n$. The size of the preorder forest of the set covering algorithm described in Chen [1998] is $2^n - 1$. The preorder forest is a collection of tree data structures that provide a sequence, called *ranking*, of the n column vectors in the constraint matrix $(a_{ij})_{m \times n}$ to be included in a possible cover solution to (2). The size of a preorder forest is the number of nodes in its collection of tree data structures and is therefore one of the important factors that affect the efficiency of a set covering algorithm.

If $\boldsymbol{x}$ is a prime cover solution to (2) and $K = \{r \mid x_r = 1\} \subset \{1, 2, \cdots, n\}$, then for each $k \in K$ we may change the value of field k to a value from

$$B_k^* = \overline{\bigcup_{j \in J} A_k^j} = \bigcap_{j \in J} \overline{A_k^j}, \tag{3}$$

where $J = \{j \mid 1 \leq j \leq m,$ field k is an entering field of $E^j\}$. The new imputed record $\boldsymbol{y}_1$, which has different value for field $k \; \forall \; k \in K$ from the record $\boldsymbol{y}$, will pass all edits. Note that $B_k^* \neq \emptyset$. If B_k^* were a empty set, then $\bigcup_{j \in J} A_k^j$ would be equal to A_k and an essentially new implicit edit would have been generated and included in the set of $\bar{E}$.

To obtain a *complete* set of edits, implicit edits are needed. Implicit edits may be implied logically from the initially specified edits (or explicit edits). Implicit edits give information about explicit edits that do not originally fail but may fail when a field in a record with an originally failing explicit edit is changed. *Lemma* 1 gives a formulation on how to generate implicit edits.

Lemma 1 (Fellegi and Holt [1976]): If E^r are edits $\forall \; r \in S$, where S is any index set,

$$E^r : \bigcap_{j=1}^{n} \boldsymbol{A}_j^r = F, \quad \forall \quad r \in S. \tag{4}$$

Then, for each i $(1 \leq i \leq n)$, the expression

$$E^* : \bigcap_{j=1}^{n} \boldsymbol{A}_j^* = F \tag{5}$$

is an implied edit, where

$$\boldsymbol{A}_j^* = \bigcap_{r \in S} \boldsymbol{A}_j^r \neq \emptyset \quad j = 1, \cdots, i-1, i+1, \cdots, n$$

$$\boldsymbol{A}_i^* = \bigcup_{r \in S} \boldsymbol{A}_i^r \neq \emptyset.$$

If all the sets A_i^r are proper subsets of A_i, i.e., $A_i^r \neq A_i$ (field i is an entering field of edit E^r) $\forall\ r \in S$, but $A_i^* = A_i$, then the implied edit (5) is called an *essentially new edit*. Field i, which has n_i possible values, is referred to as the *generating field* of the implied edit. The edits $E^r\ \forall\ r \in S$ from which the new implied edit E^* is derived are called *contributing edits*.

Therefore, in order to generate an essentially new implicit edit, we must have the following three conditions:

1. $A_j^* \neq \emptyset, \forall\ j,\ 1 \leq j \leq n$;
2. $A_i^r \neq A_i, \forall\ r \in S$, where $A_i^r \neq \emptyset$;
3. $A_i^* = A_i$.

Conditions 2 and 3 indicates that the set $\{A_i^r \mid r \in S\}$ is a cover of A_i and are the foundations of the following set covering formulation in (6).

Let $\{E^r \mid r \in S\}$ be the set of the s edits with field i entering, then the set covering problem related to the generating field i is

$$\begin{aligned} &\text{Minimize} \quad \textstyle\sum_{r\in S} x_r \\ &\text{subject to} \quad \textstyle\sum_{r\in S} g_{jr}^i x_r \geq 1, \qquad j = 1, 2, \cdots, n_i \\ &\qquad\qquad x_r = \begin{cases} 1, & \text{if } E^r \text{ is in the cover;} \\ 0, & \text{otherwise,} \end{cases} \\ &\qquad\qquad r \in S \end{aligned} \tag{6}$$

where

$$g_{jr}^i = \begin{cases} 1, & \text{if } E^r \text{ contains the } j\text{th element in field } i; \\ 0, & \text{otherwise,} \end{cases}$$

is the jth element in field i of edit E^r ($r \in S$). If $\boldsymbol{x}$ is a prime cover solution to (6) and $K = \{r \mid x_r = 1\} \subset S$, then $\cup_{k\in K} A_i^k = A_i$. A prime cover solution is a nonredundant set of the edits whose ith components cover all possible values of the entering field, which is the generating field to yield an essentially new implicit edit. This paper will concentrate on the enhancement of the set covering algorithm that finds all the prime cover solutions to the SCP (6), which is also referred to as a SCP with constraint matrix $\boldsymbol{G}= (g_{jr}^i)_{n_i \times s}$. The set covering algorithm with the enhancement has been implemented in the DISCRETE edit system. The size of the preorder forest of the set covering algorithm described in Chen [1998] is $2^s - 1$.

3 Preorder and Combinatorial Generation

In the sections hereafter, we will use n to denote the number of column vectors in the constraint matrices $(a_{ij})_{m\times n}$ of (2) and $(g_{jr}^i)_{n_i \times s}$ of (6). We also assume that (a_{ij}) and (g_{jr}^i) are the reduced constraint matrices, meaning that they do not contain any identical column vectors and the optimal solution of (2) or the prime cover solutions of (6) found will not include any column unit vectors.

A preorder sequence (Knuth [1973]) is a traversal of the combinations of the n column vectors of the constraint matrices of (2) and (6) with a tree data structure. The preorder traversal is (a) visiting the root; (b) traversing the left subtree; and (c) traversing the right subtree. An example is illustrated in Figure 1 with $n = 4$, in which the node (12) means the column vectors 1 and 2. A node containing n is called a leaf node. Nodes (24) and (1234) are leaf nodes. Nodes with the same parent are sibling nodes. Nodes (12), (13), and (14) are sibling nodes. Suppose that $S = \{1, 2, \cdots, n\}$ is the set of the ordinal numbers of the n column vectors of a constraint matrix and $\Re$ consists of the $2^n - 1$ nonempty subsets of S. We will use, for example, (123) to denote the subset $\{1, 2, 3\}$. Given a nonempty subset $T \subseteq S$, let us define the *characteristic vector* of T (Kreher and Stinson [1998]) to be the n-tuple

$$\aleph(T) = [x_n, x_{n-1}, \ldots, x_1], \tag{7}$$

where

$$x_i = \begin{cases} 1, & \text{if } i \in T; \\ 0, & \text{otherwise.} \end{cases}$$

The *ranking* (or *lexicographic ordering*) on the set of nonempty subsets of S is induced by the ranking of the characteristic vectors. Table 1 illustrates the ranking of the 15 nonempty subsets of $S = \{1, 2, 3, 4\}$. Table 1 also shows their binary representations and the equivalent decimal numbers.

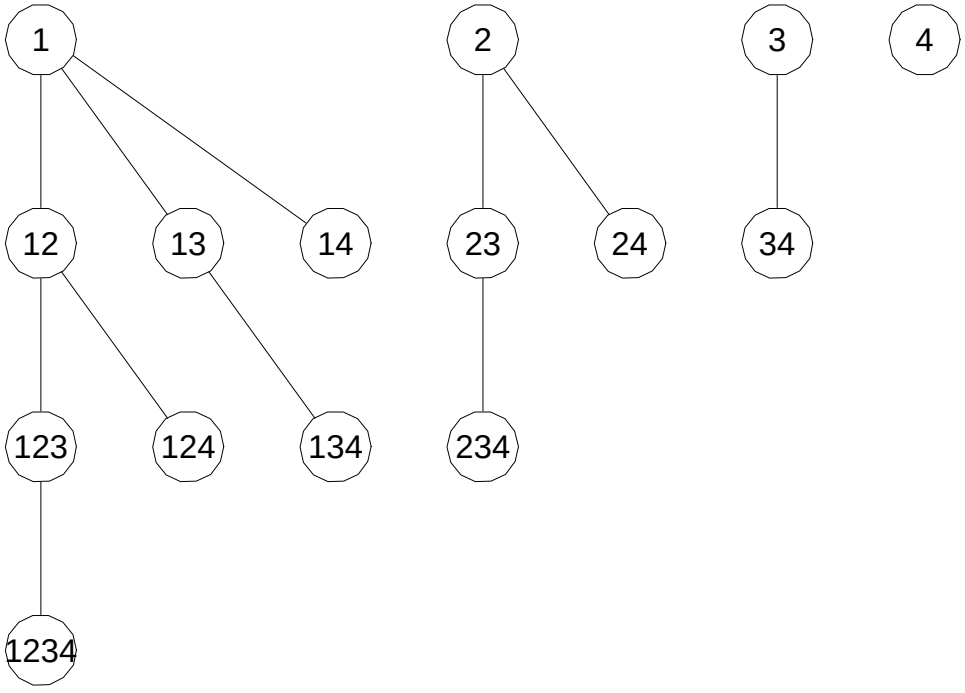

Figure 1: Preorder Tree Data Structure.

The transformation of the characteristic vector of a given node $T \in \Re$ into the equivalent decimal number is given by

Table 1: Ranking and Characteristic Vectors of 15 Nonempty Subsets of $S = \{1, 2, 3, 4\}$

Node (T)	$\aleph(T) = [x_4, x_3, x_2, x_1]$	Binary Representation	Decimal Number $(\Psi(\aleph(T)))$	rank(T)
(1)	[0,0,0,1]	0001	1	1
(12)	[0,0,1,1]	0011	3	2
(123)	[0,1,1,1]	0111	7	3
(1234)	[1,1,1,1]	1111	15	4
(124)	[1,0,1,1]	1011	11	5
(13)	[0,1,0,1]	0101	5	6
(134)	[1,1,0,1]	1101	13	7
(14)	[1,0,0,1]	1001	9	8
(2)	[0,0,1,0]	0010	2	9
(23)	[0,1,1,0]	0110	6	10
(234)	[1,1,1,0]	1110	14	11
(24)	[1,0,1,0]	1010	10	12
(3)	[0,1,0,0]	0100	4	13
(34)	[1,1,0,0]	1100	12	14
(4)	[1,0,0,0]	1000	8	15

$$\Psi(\aleph(T)) = \sum_{i=1}^{n} x_i 2^{i-1} \tag{8}$$

We now describe a generation algorithm that will provide the preorder sequence such that

$$a(\texttt{rank}(T)) = \Psi(\aleph(T)), \tag{9}$$

where a is an array with $2^n - 1$ elements. The ranking function, $\texttt{rank}(T)$, is a bijection:

$$\texttt{rank} : \Re \longrightarrow \{1, 2, \cdots, 2^n - 1\}. \tag{10}$$

It defines a total ordering on the elements of $\Re$ by the rule of

$$T_1 < T_2 \Longleftrightarrow \texttt{rank}(T_1) < \texttt{rank}(T_2). \tag{11}$$

For each ranking function, there is a unique unranking function associated with it. This function is also a bijection:

$$\texttt{unrank} : \{1, 2, \cdots, 2^n - 1\} \longrightarrow \Re, \tag{12}$$

which is the inverse function of rank, i.e., we have

$$\texttt{rank}(T) = i \Longleftrightarrow \texttt{unrank}(i) = T \tag{13}$$

for all $T \in \Re$ and all $i \in \{1, 2, \cdots, 2^n - 1\}$. Given a ranking function, rank, a successor function can be defined to satisfy the rule of

$$\texttt{successor}(T_1) = T_2 \Longleftrightarrow \texttt{rank}(T_2) = \texttt{rank}(T_1) + 1, \tag{14}$$

in which $\texttt{successor}(T_1)$ is undefined if $\texttt{rank}(T_1) = 2^n - 1$, i.e.,

$$\texttt{successor}(T_1) = \begin{cases} \texttt{unrank}(\texttt{rank}(T_1) + 1) & \text{if } \texttt{rank}(T_1) < 2^n - 1; \\ \text{undefined} & \text{if } \texttt{rank}(T_1) = 2^n - 1. \end{cases} \tag{15}$$

The generation algorithm is very simple. We need recursive calls to generate the preorder sequence. The generation algorithm is given in **Algorithm 1**.

PREORDER(n) requires

$$\sum_{i=1}^{n} (2^{i-1} - 1) = 2^n - n - 1 \tag{16}$$

multiplications (or left shifts) and $2^n - n - 1$ additions. The efficiency of the algorithm is *the order of* 2^n, i.e., $O(2^n)$. It also requires $2^n - 1$ units of integer storage to hold the array a. For a survey with moderate number of fields, such as $n = 20$, this algorithm is inefficient and requires a huge amount of storage.

Algorithm 1: PREORDER(n)
  $a[1] \leftarrow 1$
  if $n = 1$ **then return**(a)
  else
    begin
      $b \leftarrow$ PREORDER($n - 1$)
      for $k \leftarrow 2^{n-1} + 1$ **to** $2^n - 1$
        begin $a[k] \leftarrow b[k - 2^{n-1}] \times 2$; $a[k - 2^{n-1} + 1] \leftarrow a[k] + 1$ **end**
      return(a)
    end

4 Lexicographic Ordering of Preorder Sequence

Given the characteristic vector x of a node $T \in \Re$, we define $c(x_k)\ \forall\ k \in S$ as following:

$$c(x_k) = \begin{cases} 0 & \text{if } x_i = 0\ \forall\ i \in \{k, k+1, \cdots, n\}; \\ 1 & \text{if } x_k = 1; \\ 2^{n-k} & \text{if } x_k = 0 \text{ and not all } x_i = 0\ \forall\ i \in \{k+1, \cdots, n\}. \end{cases} \tag{17}$$

The algorithm, COUNT(x, n, k), of calculating $c(x_k)$ is given in **Algorithm 2**, which requires at most $n - k + 3$ comparisons. In the algorithms hereafter, $x_i = x[i]\ \forall\ i \in S$.

Algorithm 2: COUNT(x, n, k)
  if $x[k] = 1$ **then return**(1)
  if $x[i] = 0$ for all $i = k, k+1, \cdots, n$ **then return**(0)
  return(2^{n-k})

With respect to the preorder sequence, the rank of a nonempty subset T of S is

$$\texttt{rank}(T) = \sum_{k=1}^{n} c(x_k). \tag{18}$$

An example with $n = 4$ is illustrated in Table 1 and the algorithm, RANK(x, n), of calculating $\texttt{rank}(T)$ is given in **Algorithm 3**. The unranking algorithm, UNRANK(r, n), is given in **Algorithm 4**.

Algorithm 3: RANK(x, n)
  $r \leftarrow 0$
  for $k \leftarrow 1$ **to** n
    $r \leftarrow r +$ COUNT(x, n, k)
  return(r)

Algorithm 3 requires at most

$$\sum_{k=1}^{n}(n-k+3) = \frac{n^2+5n}{2} \tag{19}$$

comparisons and n additions. The efficiency of the algorithm is $O(n^2)$.

Algorithm 4: UNRANK(r, n)
  $t \leftarrow r$
  for $k \leftarrow 1$ **to** n
    begin
      if $t > 2^{n-k}$ **then begin** $x[k] \leftarrow 0$; $t \leftarrow t - 2^{n-k}$ **end**
      else if $t > 0$ **then begin** $x[k] \leftarrow 1$; $t \leftarrow t - 1$ **end**
          else $x[k] \leftarrow 0$
    end
  return(x)

Algorithm 4 requires at most n multiplications, $2n$ comparisons, $5n$ assignments, and $2n$ subtractions. The efficiency of this algorithm is $O(n)$. An implementaion of the set covering algorithm using the preorder sequence is given in **Algorithm 5**.

Algorithm 5: SETCOVER1(n)
  for $r \leftarrow 1$ **to** $2^n - 1$
    begin
      $x \leftarrow$ UNRANK(r, n)
      if x is a cover **then call** STOREPRIMECOVER(x)
    end

Algorithm 5 eliminates the $2^n - 1$ units of integer storage required in Algorithm 1. It still visits every node of the preorder tree structure, that is a total of $2^n - 1$ nodes. Therefore SETCOVER1(n) is not very efficient. The function STOREPRIMECOVER(x) is a subroutine that stores the prime subcovers of x and is given a detailed description in Chen [1998].

5 An Efficient Implementation of Preorder Sequence

In this section, we will discuss a more efficient set covering algorithm using the preorder tree structure. We use the idea of the successor function, `successor`(T), described in Section 3 to develop the function NEXTNODE(x, n), i.e.,

$$\aleph(\texttt{successor}(T)) = \text{NEXTNODE}(x, n), \tag{20}$$

where $x = \aleph(T)$ and $T \subseteq S$. NEXTNODE(x, n) is given in **Algorithm 6**. Algorithm 6 is more efficient than Algorithm 4 because the k in the for loop of UNRANK(x, n) is from 1 to n while of NEXTNODE(x, n) from $n-1$ downto the first k with $x_k = 1$.

In the preorder sequence and the characteristics of the set covering problem, if a node T is a cover, any node of the subtree with root node T will not be a prime cover. Therefore, if node T is a cover, we may skip the visits of the rest of the subtree and move to the next

Algorithm 6: NEXTNODE(x, n)
```
if x[n] = 1 then
  begin
    x[n] ← 0
    for k ← n − 1 downto 1
      if x[k] = 1 then begin x[k] ← 0; x[k + 1] ← x[k + 1] ∨ 1; k ← 0 end
  end
else
  for k ← n − 1 downto 1
    if x[k] = 1 then begin x[k + 1] ← x[k + 1] ∨ 1; k ← 0 end
return(x)
```

sibling of T. **Algorithm 7** provides a very simple implementation of skipping nodes in the subtree. Compared to UNRANK(x, n), NEXTSIBLING(x, n) is also very efficient. If T is a leaf node, NEXTNODE(x, n) = NEXTSIBLING(x, n).

Algorithm 7: NEXTSIBLING(x, n)
```
for k ← n − 1 downto 1
  if x[k] = 1 then begin x[k] ← 0; x[k + 1] ← x[k + 1] ∨ 1; k ← 0 end
return(x)
```

Now, a new set covering algorithm using the preorder tree structure is very straightforward. The algorithm, SETCOVER2(n), is given in **Algorithm 8**.

Algorithm 8: SETCOVER2(n)
```
x[1] ← 1
while Ψ(x) < 2^n do
  if x is a cover then
    begin
      call STOREPRIMECOVER(x)
      x ← NEXTSIBLING(x, n)
    end
  else x ← NEXTNODE(x, n)
```

The efficiency of SETCOVER1(n) and the worst case of SETCOVER2(n) is still $O(2^n)$. However, SETCOVER2(n) is more efficient because (i) NEXTNODE(x, n) and NEXTSIBLING(x, n) in SETCOVER2(n) are more efficient than UNRANK(x, n) in SETCOVER1(n); (ii) SETCOVER2(n) may skip nodes if a cover is found; (iii) in SETCOVER2(n), if a cover is found at a high ranking node (i.e., low ranking number), the number of nodes skipped is enormous.

Table 2: The Modified Gray Code of 15 Nonempty Subsets of $S = \{1, 2, 3, 4\}$

Node (T)	$\aleph(T) = [x_4, x_3, x_2, x_1]$	Binary Representation	Decimal Number ($\Psi(\aleph(T))$)	rank(T)
(1)	[0,0,0,1]	0001	1	1
(12)	[0,0,1,1]	0011	3	2
(123)	[0,1,1,1]	0111	7	3
(1234)	[1,1,1,1]	1111	15	4
(124)	[1,0,1,1]	1011	11	5
(14)	[1,0,0,1]	1001	9	6
(134)	[1,1,0,1]	1101	13	7
(13)	[0,1,0,1]	0101	5	8
(3)	[0,1,0,0]	0100	4	9
(34)	[1,1,0,0]	1100	12	10
(234)	[1,1,1,0]	1110	14	11
(23)	[0,1,1,0]	0110	6	12
(2)	[0,0,1,0]	0010	2	13
(24)	[1,0,1,0]	1010	10	14
(4)	[1,0,0,0]	1000	8	15

The (iii) above can be achieved by sorting the column vectors of the constraint matrix on the number of 1's in descending order. In other words, the lowest ordinal number of the column vector has the highest number of 1's in the vector. With this type of rearrangement of the column vectors, the chance of having a cover at a high ranking node is increased. That will reduce the number of visited nodes in the preorder tree data structure.

6 Discussion

We described the preorder tree data structure to implement the set covering algorithm. This structure fits the algorithms described nicely to eliminate the storage requirements of the preorder tree while maintaining the efficiency of the algorithms. There are alternatives to implement the lexicographic order of the nonempty subsets of S. One of them is to generate the $2^n - 1$ nonempty subsets of S sequentially so that any two consecutive nonempty subsets have distance one (the smallest possible). Here, the distance between two nonempty subsets $T_1, T_2 \subseteq S$ is defined (Kreher and Stinson [1998]) as

$$\texttt{dist}(T_1, T_2) = |T_1 \Delta T_2|, \tag{21}$$

where $T_1 \Delta T_2$ is the *symmetric difference* of T_1 and T_2 and defined as

$$T_1 \Delta T_2 = (T_1 - T_2) \cup (T_2 - T_1). \tag{22}$$

This means that any nonempty subset can be obtained from the previous one by either deleting a single element or adding a single element. Such an ordering of the $2^n - 1$ nonempty

subsets of S is called a *minimum change ordering*. An example with $n = 4$ is given as following

$$(1), (12), (2), (23), (123), (13), (3), (34), (134), (1234), (234), (24), (124), (14), (4) \qquad (23)$$

which is a minimal change ordering. The characteristic vectors of the nonempty subsets in a minimal change ordering form a structure known as a *Gray code*. From the minimal change ordering given above, the following Gray code is obtained:

$$0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000. \quad (24)$$

The Gray code presented does not have the preorder tree structure given in Figure 1. Therefore, we can not take the advantages of the structure described in Section 5 with the Gray code (24). If the Gray code is modified to have the sequence given in Table 2, it is able to fit into the preorder tree structure illustrated in Figure 2. The modified Gray code becomes

$$0001, 0011, 0111, 1111, 1011, 1001, 1101, 0101, 0100, 1100, 1110, 0110, 0010, 1010, 1000. \quad (25)$$

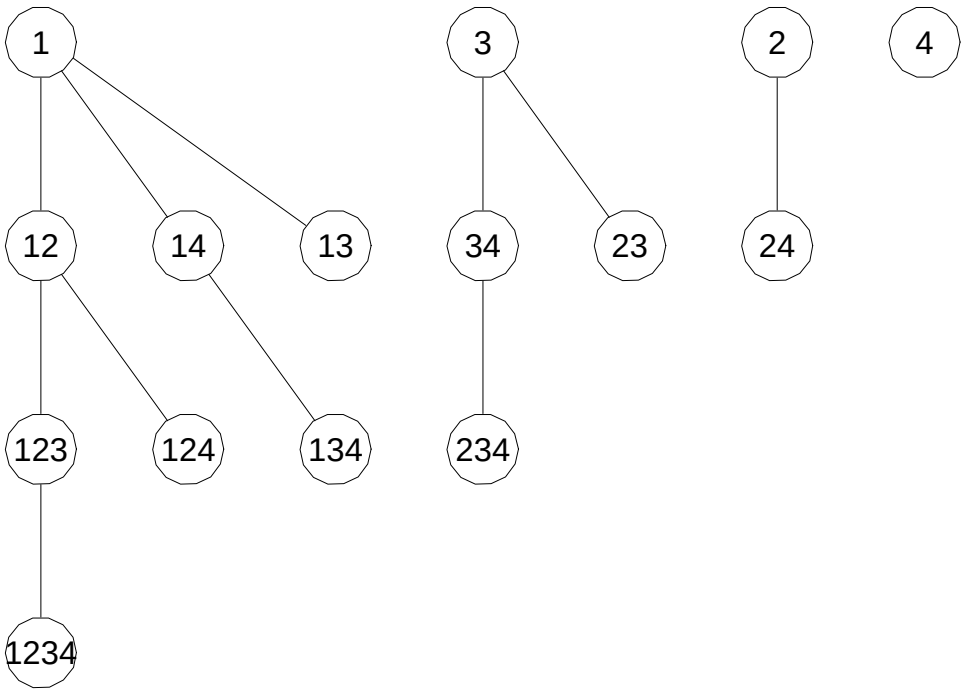

Figure 2: Minimal Change Ordering (The Modified Gray Code).

Comparing the structures in Figures 1 and 2, we realize that the modified Gray code doesn't have the advantage of sorting the column vectors of the constraint matrix based on the number of 1's in the vectors. Therefore, the Gray code and the modified one are inferior to the one given in Figure 1 in terms of the performance of implementing the set covering algorithm.

7 Summary

We have described a new algorithm to implement the preorder sequence used in the set covering algorithm of Chen [1998]. First, we eliminated the storage required in PREORDER(n) to obtain the algorithms, SETCOVER1(n) and SETCOVER2(n). Then, these two algorithms were compared and we concluded that SETCOVER2(n) is potentially more efficient than SETCOVER1(n) as described in Section 5. We also discussed the alternative sequences to the preorder sequence given in Figure 1 and concluded that the alternatives, the Gray code and the modified GRay code in Figure 2, are inferior in the implementaion of the set covering algorithm.

References

[1] B. Chen. Set covering algorithms in edit generation. In *Proceedings of the Statistical Computing Section*, pages 91–96. American Statistical Association, 1998.

[2] I. P. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *Journal of the American Statistical Association*, 71:17–35, 1976.

[3] R. S. Garfinkel, A. S. Kunnathur, and G. E. Liepins. Optimal imputation of erroneous data: Categorical data, general edits. *Operations Research*, 34:744–751, 1986.

[4] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching, Vol. 3*. Addison Wesley, Reading, Massachusetts, 1973.

[5] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press LLC, Boca Raton, Florida 33431, 1998.

[6] W. E. Winkler. Set-covering and editing discrete data. Technical report, Bureau of the Census, 1997.

[7] W. E. Winkler and T. F. Petkunas. The DISCRETE Edit System. Statistical Research Division Research Report, Bureau of the Census, 1996.