An Experimental Study of Test Pattern and Response Compression Techniques for BIST

by

J. Thomas King

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in the
Mathematics and Computer Science
Program

R. Dandapani 8/15/86
(Signature) Date
Adviser

M Hotchkiss August 28, 1986
(Signature) Date
Dean of the Graduate School

Youngstown State University

August, 1986

An Experimental Study of Test Pattern and Response Compression Techniques for BIST

J. Thomas King
Master of Science
Youngstown State University, 1986

In a VLSI environment, Built-in Self-Testing, (BIST), has become a popular option. In BIST, test hardware is on the same chip as the Circuit Under Test, CUT, and offers a faster testing time as compared to convenitonal off-chip testers. BIST hardware consists of a ROM of testing inputs, a ROM of valid outputs, and a comparator usually a row of XOR gates.

In this thesis, a number of methods are given to reduce the amount of ROM space required to store the input set. These methods increase the test set size and require some additional hardware. However, the savings in ROM space make up for the additional cost and size increase. The question of optimal reduction in ROM size is discussed. Finally, a method for compaction of input and output sets taken together is presented.

Acknowledgements

I would like to thank Dr. Dandapani, Dr. Burden, and Dr. Klein for their help with this thesis.

An Experimental Study of Test Pattern and Response Compression Techniques for BIST

by

J. Thomas King

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in the
Mathematics and Computer Science
Program

R. Dandapani
(Signature) 8/15/86
Adviser Date

Sally M. Hotchkiss August 28, 1986
(Signature)
Dean of the Graduate School Date

Youngstown State University

August, 1986

An Experimental Study of Test Pattern and Response Compression Techniques for BIST

J. Thomas King
Master of Science
Youngstown State University, 1986

In a VLSI environment, Built-in Self-Testing, (BIST), has become a popular option. In BIST, test hardware is on the same chip as the Circuit Under Test, CUT, and offers a faster testing time as compared to convenitonal off-chip testers. BIST hardware consists of a ROM of testing inputs, a ROM of valid outputs, and a comparator usually a row of XOR gates.

In this thesis, a number of methods are given to reduce the amount of ROM space required to store the input set. These methods increase the test set size and require some additional hardware. However, the savings in ROM space make up for the additional cost and size increase. The question of optimal reduction in ROM size is discussed. Finally, a method for compaction of input and output sets taken together is presented.

Acknowledgements

I would like to thank Dr. Dandapani, Dr. Burden, and Dr. Klein for their help with this thesis.

Table of Contents

List of Fisures

List of Tables

List of Graphs

Chapter 1

Introduction

The development of VLSI, very large scale integration, has made possible the production of extremely complex microcircuitry on silicon microchips. It would be erroneous to assume this chip, once produced, will always function correctly. Due to physical limits of the materials used and other possible problems that may develop, it is necessary to test the circuitry for valid output at random times. It is possible to exhaustively test the circuit. But for a CUT, circuit under test, with n inputs this requires testing 2^n inputs. This number becomes unwieldy when $n \geq 16$.

An alternative to exhaustive testing is using a fault model and generating a set of test vectors, T, for each fault in the model which adequately tests the circuit. Assume T has N vectors. Let o be the set the CUT will produce corresponding to T if the circuit is functioning properly. The hardware needed to directly store T on chip, called Prestored-T along with CUT is shown in Figure 1. The first ROM contains the set T and has dimensions N * n. A second ROM holds the

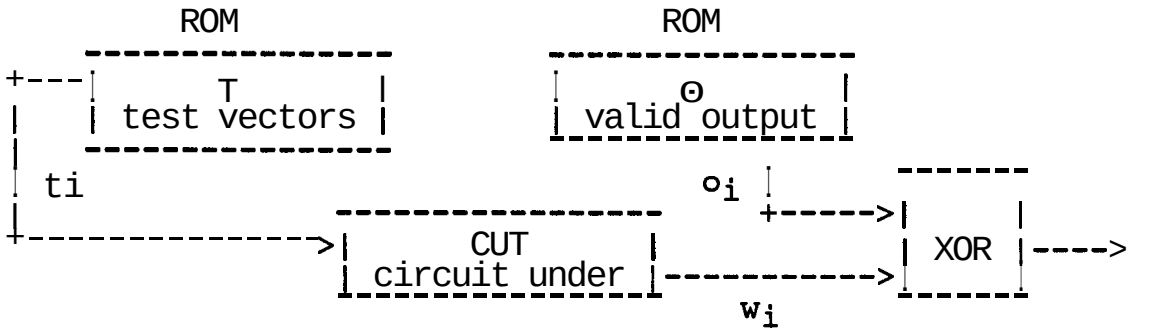


Figure 1. Prestored-T and Prestored-0 for BIST.

set Θ, Prestored-0. The first part of this thesis deals with alternative implementations for T. We will then consider implementation of O and T taken as a single ROM.

When testing the circuit in Figure 1 a test vector, t_i is sent to the CUT. The CUT generates an output w_i, which is compared to the expected output o_i. An error would occur if $o_i \neq w_i$ for any i.

Let $T = \{ t_1, t_2, \ldots, t_N \}$.

If $t_i = a_0 a_1 a_2 \ldots a_{n-1}$, $t_j = b_0 b_1 b_2 \ldots b_{n-1}$, and

$$\theta(a_k, b_k) = \begin{cases} 1 \text{ if } a_k \neq b_k \\ 0 \text{ if } a_k = b_k \end{cases}$$

then Hamming distance is defined as $H(t_i, t_j) = \sum_{k=0}^{n-1} \theta(a_k, b_k)$, i.e., the Hamming distance is the number of positions that

t_i and tj differ.

In [1], a method to reduce the area required to store T on chip was presented. The procedure develops an ordered set $T_1 = \{ s_1, s_2, \ldots, s_M \}$. T_1 contained all of set T, however in T_1, $H(s_i, s_{i+1}) = 1$ for any i. This special property of T_1 makes possible a more efficient storage scheme than Prestored -T. The method is given below as Algorithm P1.

```
Alsorithm P1;
Input T; Output T1;
vector x;
Begin;
x := t1;
place x in T1;
While (T ⊄ T1) do
    Begin;
        Choose ti to be the first vector from T - T1 such
              that H(x,ti) ≤ H(x,tj) for all tj in T - T1.
        Let d = H(x,ti);
        Add d - 1 vectors into T1 so that adjacent pairs
              have Hamming distance of 1;
        Add ti to T1;
        x := ti;
```

End;

End.

Let r = the smallest integer such that $2^r \geq n$. The design of Figure 2 will produce the set T_1 if the register is initialized to the first vector of T_1, and the ROM contains the position of change between successive vectors of T_1. Let M be the size of T_1. The size of the ROM will be r * M. M will determine if there is any actual savings of ROM space. M is dependent on the method used to generate T_1. Algorithm P1 produced savings up to 50% compared to Prestored-T.

```
                                  +-----------------------+
                                  |                       |
-------    -----------    -------      ------------      |
| ROM |--->| Decoder |--->| XOR |<--+--| Register |<--+
-------    -----------    -------   |  ------------
                                    |
                                    +----> CUT
```

Figure 2. The hardware needed to implement Algorithm P1.

The size of T_1 is about twice that of set T. While the increase in the test set size might be viewed as substantial, in a real time analysis it is not very significant. One vector will only require microseconds to be tested. Even if the test size is doubled, total time spent on testing will still be on the order of microseconds.

In Figure 2 additional hardware, specifically a n bit register and a row of n XOR gates, are added to the original hardware. In order for this to be a feasible approach, the ROM savings must offset the cost of this additional hardware. With any test pattern generator, TPG, such as Algorithm P1, the ROM savings must be greater than the cost of the hardware added to the chip. Example 1 illustrates the application of Algorithm P1.

Example 1:

Let T = { 0000,0111,0011 }.

Prestored-T will require a ROM = 0000
0111
0011

3 * 4 = 12 bit ROM.

Algorithm P1 would produce T_1 = { 0000,0010,0011,0111 }.

Assuming the register of Figure 2 is initialized to 0000, the ROM needed would be : 11
10
01

3 * 2 = 6 bits.

Algorithm P1 saved 50% of the original ROM. The natural question to ask is : are there other TPGs than the one given in [1]? We will study this question in this thesis.

Chapter 2

Right Circular Shift Methods

In the TPG of Algorithm P1, r bits are used to store the position of change between successive vectors of T_1. Thus allowing each bit of change to be any of the n positions. One possible way of saving more ROM space would be to use fewer than r bits to represent each position of change. A Right Circular Method, RCM(z), uses $r - z$, $z > 0$, bits to represent the change between adjacent vectors.

Using only $r - z$ bits to represent each change of position presents a problem in 'mobility'. Unlike Algorithm P1, which could complement any of the n positions at any time, a RCM(z) does not have the option to complement each of the n positions at any time. For a RCM(z) to eventually be able to complement all n positions, the position to be complemented must depend on more than the value being held in the $r - z$ bits. While these additional values could come from an outside source such as another ROM, it is desirable to produce the second set of values from the RCM itself.

Figure 3 shows the hardware configuration we will use for RCM(z). It has a shift register which is initialized to all 0's except in one position. The ROM contains the number of positions the shift register must be shifted to form the next vector. Through the position of the single one bit the shift register saves the last position complemented and makes it possible for the RCM(z) to reach all n positions within a certain number of moves.

```
-------     -----------    ------------------
| ROM |---->| Decoder |--->| Shift Register |-----+
-------     -----------    ------------------     |
                                                  |
           +-------------------------------------+
           |
           |       -------
           +------>|     |      ------------
                   | XOR |---->| Register |---+-> CUT
           +------>|_____|      ------------   |
           |                                   |
           |                                   |
           +-----------------------------------+
```

Figure 3. Hardware needed for RCM TPG.

Let POS be the current position of the single one bit in the shift register. Define RANGE(POS, z) = { MOD(POS + i, n)) for $0 \leq i \leq 2^{r-z} - 1$. For a given initial set T, if T_2 is the set of vectors produced by RCM(z) then :

(1) adjacent vectors of T_2 differ in only one bit,

(2) if t_i and t_{i+1} differ in position POS, then t_{i+1} and t_{i+2} differ in a position which is in **RANGE(POS,z)**.

In the following, unless otherwise specified, RCM will refer to **RCM(1)** and **RANGE(POS)** = **RANGE(POS,1)**. The first RCM method is presented next. Let $x = a_0a_1...a_{n-1}$ and $y = b_0b_1...b_{n-1}$. Assume POS = 0 initially in all of the following algorithms. We will show this assumption can be relaxed without adversely affecting the performance of algorithm results.

```
Algorithm P2;
Input T,POS; Output T2;
Vector x;
Begin;
    x := t1;
    Add x to T2;
While ( T ⊄ T2 ) do
     Begin;
          choose ti to be the first vector from T - T2 such
                that H(x,ti) ≤ H(x,tj) for all tj in T - T2;
          Call LINK(POS,x,ti,T2);
          x := ti;
     End
```

```
End.

Algorithm LINK;
Input POS,x,y,T2; Output T2;
Begin;
    While( x ≠ y )do
        Begin;
            Let q be the next differing position between x
                and y from POS in a right circular manner;
            If q is in RANGE(POS) then
                Begin;
                x:=a0a1...ā_q...a_{n-1}; complement position q
                add x to T2;
                POS := q;
                End;
            Else
                Begin;
                    k := MOD(POS + 2^{r-1}-1,n);

                    x := a0a1...ā_k...a_{n-1};
                    add x to T2;
                    x := a0a1...ā_k...a_{n-1};
                    add x to T2;
                    POS := k;
                End
```

```
        End;
End.
```

Example 2:

Let T = {0000,0011}. Applying RCM **Algorithm** P2 would yield T_2 = {0000,0100,0000,0010,0011}. Notice in T_2, 0000 appears twice. T_2 and T_1 are ordered sets and duplication of vectors is required for T_2. For the first occurrence of 0000, POS = 0, but for the second time 0000 appears POS = 1. It is this change in POS-that makes these duplications necessary.

It is worth noting Algorithm **P1** would produce T_1 identical to T2. Assuming the standard register of both methods is initialized to 0000, Algorithm **P1** would require a 3 * 2 = 6 bit ROM, and Algorithm P2 would need a 4 * 1 = 4 bit ROM. Prestored-T needs a 4 * 2 = 8 bit ROM.

Algorithm P2 is the **RCM(1)** extension of Algorithm **P1**. The size of T_2 can be larger than T_1 but T_2 could still use a smaller ROM than T_1 since one less bit is needed to store the position of change. POS was assumed to be **0** initially in Algorithm P2. The following Lemma can be used to eliminate this assumption by modifying T_2.

Lemma 1:

For any value of POS $\neq$ 0 and given vector x, a RCM can produce a POS = 0 with x unchanged by adding 6 or less additional vectors into T_2.

Proof :

Assume position k refers to bit a_k. Let $t_i = a_0 a_1 \ldots a_{n-1}$. If the complementer makes a RCM move to position k to form t_{i+1}, then complements position k again to form t_{i+2}, since a_k complemented twice is a_k then $t_i = t_{i+2}$. We have changed the value of POS without changing the vector.

Assume POS = 1. Move the complementer to the right most position of RANGE(POS). This changes the value of POS to equal MOD($(2^{r-1}-1) + 1, n$) = $MOD(2^{r-1}, n) = 2^{r-1}$. We want to complement position twice thus adding two vectors to T_2. If 0 is now in the RANGE(POS), we can move to position 0 and complement twice, and so the number of added vectors is only four.

If position 0 not in the RANGE(POS), then $n = 2^r$. The right-most position of RANGE(POS) will be: position = $MOD(2^{r-1} + 2^{r-1} - 1, n) = MOD(2^{r-1}(1+1) - 1, n) = MOD(2^r - 1, n) = 2^r - 1$. Moving to this position and complementing twice

would add two more vectors to T_2, and change POS = 2^r-1. The complementer is shifted one position to the right and again complements this position twice. A total of 6 vectors have been added, we have the same vector we started with, and

$$POS = 2^r - 1 + 1 = 2^r = n = 0.$$

Since POS = 1 and n = 2^r is the worst case, for all other values of POS > 1, four or fewer vectors need to be added. For all $1 < POS \leq 2^{r-1}$, 2^{r-1} is in the RANGE(POS). Hence we could move the complementer to 2^{r-1}, and continue the proof as before. For all $2^{r-1} < POS \leq n-1$, $n-1$ is in the RANGE(POS). Therefore the complementer can be moved to $n-1$, and the RANGE(POS) will contain 0.

Q.E.D.

Lemma 1 shows that if we start with POS $\neq$ 0, for any chosen starting vector, POS can be adjusted to 0 without adding a significant number of vectors. This is true as long as N is significantly large. For simplicity we will assume all the following Algorithms begin with POS = 0.

Algorithm P1 can be viewed as RCM(0), but POS has no significance. Algorithms P1 and P2 use the same criterion to select t_i, the Hamming distance. P2 is a natural exten-

sion of **P1.** But for **P2,** the process of linking two vectors is more complicated.

In **P2,** LINK generates the vectors needed to connect x to y under the constraints of RCM and **RANGE(POS).** In LINK the size of set T_2 is enlarged. To keep the size of T_2 as small as possible, LINK should add the minimal number of vectors needed to connect x to y. This is shown to be true in Lemma 3. We will need the following in the proof of Lemma 3.

Let x and y be as defined before. Assume $H(x,y) = m$. Define $C = \{ c_1, c_2, \ldots, c_m \}$ such that $a_{c_i} \neq b_{c_i}$ initially. And c_{i+1} is the next position of difference from c_i in a right circular manner from a given POS. Assume $RANGE(c_i) = RANGE(POS = c_i)$.

<u>Lemma 2:</u>

Let j = the number of elements in $\{ i: c_{i+1}$ is not in $RANGE(c_i)$ for $1 \leq i \leq m-1\}$, then $j < 2$.

<u>Proof:</u>

Assume $j \geq 2$. This implies there exists a sequence, c_i, c_{i+1}, c_k such that c_{i+1} is not in $RANGE(c_i)$, and c_k is

not in RANGE(c_{i+1}) for $k \geq i + 2$. The RANGE(c_i) = R_1 contains 2^{r-1} elements, and the RANGE($c_i + 2^{r-1}$) = R_2 also has 2^{r-1} elements. If the intersection of R_1 and R_2 is nonempty, then R_2 must contain c_i, therefore any possible value for c_k would be in R_1 or R_2 which would be a contradiction. If the intersection of R_1 and R_2 is empty, then the union of these sets would contain $2^{r-1} + 2^{r-1} = 2^r = n$ elements and again for any value for c_k, c_k must be in R_1 or R_2 which is a contradiction.

Q.E.D.

Lemma 3:

Procedure LINK will generate the minimal number of vectors to link x to y for a given POS and RANGE(POS).

Proof:

Assume x and y are as defined previously. Assume H(x,y) = m .Let C = { $c_1, c_2, \ldots, c_m$ } such that $a_{c_i} \neq b_{c_i}$ initially are ordered in a right circular motion from the first value of POS. The execution of LINK will visit positions {c_1, $c_2, \ldots c_{m-1}$} to link x to y.

In the While loop of LINK either q is in RANGE(POS) or is not. Each time q is in RANGE(POS), one vector is added

to T_2. When q is not in the RANGE(POS) then two vectors are added to T2. For any pair of vectors with H distance m, the minimal number of linking vectors required is m-1 since m-1 of the positions of set C must be complemented, and one vector is added to T_2 for each bit complemented.

When connecting x to y, let j = the number of times q is not in the RANGE(POS). The number of times q will be in RANGE(POS) will be m-1. Then the number of vectors LINK produces will be (m -1) + 2 * j. Assume another algorithm exists that uses fewer linking vectors than LINK. Applying Lemma 2, we know j is 0 or 1. Therefore 3 cases exist :

Case(1)

If j is 0 then LINK adds m - 1 vectors to T_2 and so is minimal.

Case(2)

If j is 1 then LINK will produce m - 1 + 2 = m + 1 vectors. A more efficient linking procedure would have to use m - 1 or m vectors. Case(2) assumes m vectors are used. We know m - 1 vectors would be used when complementing m - 1 positions where x and y differ. The extra vector would force the final H distance to 2 or 0. If the final H distance is 2

then the set could not have been produced by a RCM. If the final H distance = 0, then only m - 1 vectors were needed.

Case(3)

Assume j = 1 and the number of vectors used by the new algorithm is m-1. If j = 1 then in the set C there exists two element c_i and c_{i+1}, such that c_{i+1} is not in RANGE(c_i). The new linking algorithm would have to use m-1 vectors to change m-1 positions of difference. This would mean the new algorithm would have to move from position c_i to c_{i+1} without changing any positions between c_i and c_{i+1}. Since c_{i+1} is not in RANGE(c_i), the sequence of vectors could not follow the RCM constraints implied by Figure 3.

Q.E.D.

P1 and P2 were tested on random sets of 16 bit vectors, (n = 16). The test set sizes were powers of 2 from N = 32 up to N = 1024. Tables 1 and 2 list the performance of Algorithms P1 and P2 respectively in comparison to Prestored-T with respect to ROM size. Programs implementing Algorithms P1 and P2 were run with 15 random test sets for each N. Tables 1 and 2 contain the averages of these results. The outputs of all runs of Algorithms P1 - P7 are listed in Appendix A. All programs run are recorded in Appendix D.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings(%)
32	140	559	-47	-9
64	263	1053	-29	-3
128	470	1881	167	8
256	837	3349	747	18
512	1426	5707	2485	30
1024	2486	9944	6440	40

Table 1. Average Results of Algorithm P1 Compared to Prestored-T.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings(%)
32	160	481	31	6
64	305	914	110	11
128	550	1649	399	20
256	1012	3035	1061	26
512	1860	5581	2611	32
1024	3429	10287	6097	38

Table 2. Average Results of Algorithm P2 Compared to Prestored-T.

The performance of Algorithm P2 is an improvement over Algorithm P1 for smaller values of N. Algorithm P2 is non-optimal in terms of number of vectors added during each loop of the While block. For example, let t_i be a vector in the first vector of $T - T_2$ with H distance from x equal to one, and a position of difference not in RANGE(POS). Let t_j exist in $T - T_2$, $j > i$, with H distance from x equal to one and position of difference in RANGE(POS). Algorithm P2 will choose t_i over tj even though choosing t_j would add no additional vectors to T_2 while choosing t_i would require at least two additional vectors. Algorithm P3, given next, will choose tj over t_i under these conditions.

```
Algorithm P3;
Input T; Output T2;
x := t1;
add x to T2;
While( T ⊄ T2)do
     Begin;
          y := first vector of T - T2;
          For (all ti in T - T2) do
               Begin;
                    If (H(x,ti) = 1 and H(x,y) = 1 and the
                         position of difference of x and y is
```

```
                    not in RANGE(POS) and position of
                    difference of x and t_i is in
                    RANGE(POS)) then
                        y := t_i;
                Else
                    If (H(x,t_i) < H(x,y)) then
                        y := t_i;
            End;
            Call LINK(POS,x,y,T_2);
            x := y;
        End
End.
```

Example 3:

Let T = (0000,0010,0100). Algorithm P2 would choose 0010 as the second vector and produce T_2 = (0000,0100,0000, 0010,0000,0001,0000,1000,0000,0100} needing a 1 * 9 = 9 bit-ROM. Algorithm P3 would pick 0100 as the second vector and output T_2 = (0000,0100,0000,0010) requiring a 1 * 3 = 3 bit ROM, while Prestored-T needs a 3 * 4 = 12 bit ROM. --

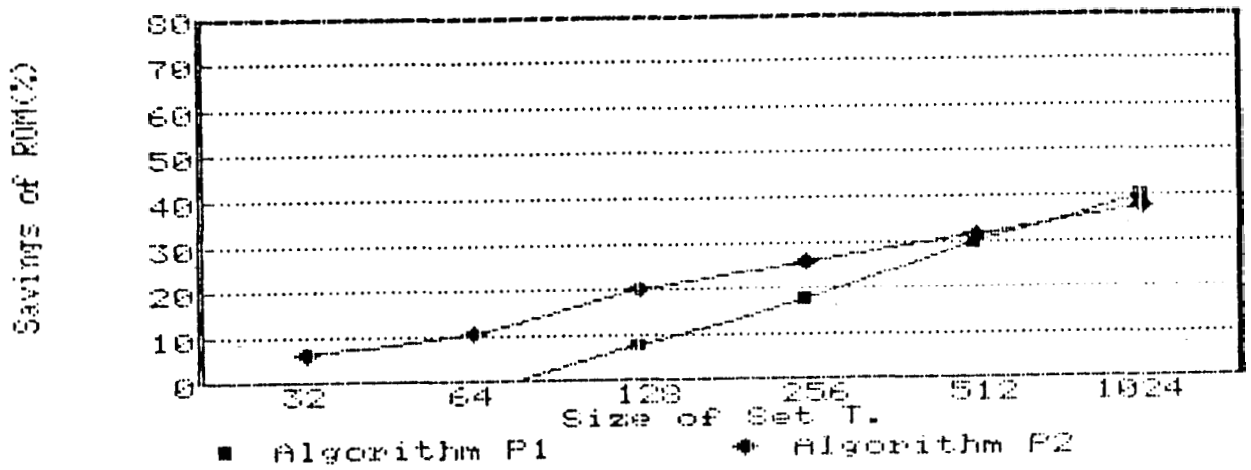
Algorithm P3 was tested on the same 15 sets for each size of N as Algorithms P1 and P2. The averages are presented in Table 3. Few vectors in random test sets are expected to have H distance equal to one. Hence results for Algorithm P3

do not differ appreciably from those of Algorithm **P2**. But for special test sets, Algorithm **P2** would out-perform Algorithm **P3**.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings(%)
32	160	481	31	6
64	1305	914	110	11
128	550	1649	399	20
256	1020	3060	1036	25
512	1877	5630	2562	32
1024	3575	10725	5659	35

Table 3. Average results of **Algorithm P3** compared to **Prestored-T.**

A comparison of average performances of Algorithms **P1** and **P2** is shown in Graph **1**. Since there was little variation between Algorithms **P2** and **P3**, Graph **1** also approximates the relationship between **P3** and **P1**.



Graph 1. Comparison of Average performances of Algorithms P1 and P2.

Algorithm P3 optimizes the selection of vectors with Hamming distance of one. The next algorithm will extend this optimization to all Hamming distances. Define f(x,y,p) to be the number of vectors LINK adds to T_2 when connecting X to y with POS = p.

Algorithm P4;

Input T; Output T_2;

Begin;

x := t_1;

add x to T_2;

While($T \not\subseteq T_2$) do

Begin;

```
        y := first element of T - T2;
        For (all ti in T - T2) do
            Begin;
                If (H(x,y) = H(x,ti)) then
                    If (f(x,ti,POS) < f(x,tj,POS)) then
                        y := ti;
                Else
                    If(H(x,ti) < H(x,y)) then
                        y := ti;
            End;
        Call LINK(POS,x,ti,T2);
        x := y;
    End;
End.
```

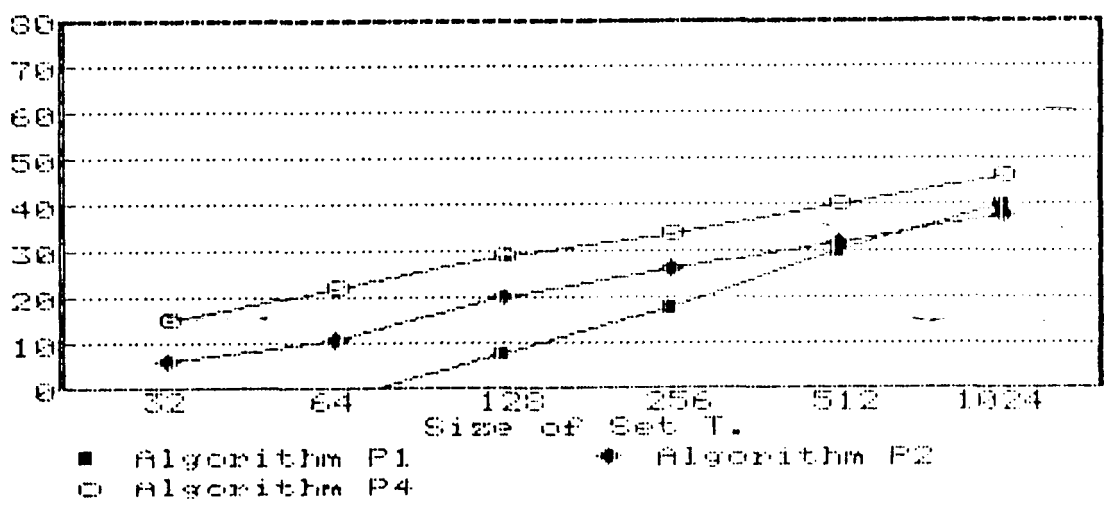
Example 4:

Let T = (0000,1001,0110,0111). Algorithms P2 and P3 applied on T would output T_2 = {0000,1000,1100,1000,1010,1000, 1001,0001,0101,0111,0110) and the new ROM would be 1 * 10 = 10 bits. Algorithm P4 would produce T_2 = (0000,0100,0110,0111, 1111,1011,1001) and a ROM of 1 * 6 = 6 bits. Prestored T would use a 4 * 4 = 16 bit ROM.

The average results from Algorithm **4** are listed in Table 4. Algorithm **P4** shows a substantial improvement over the three previous algorithms. Graph **2** compares the performance of Algorithms P1, P2, and **P4**.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings(%)
32	145	435	77	15
64	266	799	225	22
128	487	1461	587	29
256	899	2696	1400	34
512	1644	4931	3261	40
1024	3000	9000	7384	46

Table **4**. Average results of Algorithm **P4** compared to



Graph **2**. Comparison of Average performances of Algorithms P1, P2, and **P4**.

The test sets had n = 16. For any x and t_i from T - T_2, $1 \leq H(x,t_i) \leq 16$, if T is a large set, we can expect vectors having the same H distance from x to be a common occurrence. But the selection process of Algorithm P4 does not pick the t_i in T - T_2 that requires the least number of added vectors. It is possible for t_i and tj to exist such that $H(x,t_i) < H(x,t_j)$ and yet t_j may require fewer vectors than t_i to be linked to x.

With a RCM-the number of vectors needed to link is dependent on two factors, RANGE(POS) and initial value of POS sent to LINK. To find the t_i from the set T - T_2 requiring the fewest additional vectors f must be used as criteria for selection.

The method presented next, Algorithm P5, uses only the f function to select t_i. The averages are given in Table 5. Algorithm P5 performs more efficiently than Algorithms P1, P2 and P3. But Algorithm P5 was outperformed by Algorithm P4 for N $\leq$ 512. The output of Algorithm P4 is compared to the output of Algorithm P5 in Graph 3.

```
Algorithm P5;
Input POS, T; output T2;
Begin;
x := t1;
add x to T2;
While( T ⊄ T2 ) do
     Begin;
          y := first element of T - T2;
          For (all ti in T - T2) do
               Begin;
                    If (f(x,ti,POS) < f(x,tj,POS)) then
                              y := ti;
               End;
          Call LINK(POS,x,y,T2);
          x := y;
     End;
End.
```

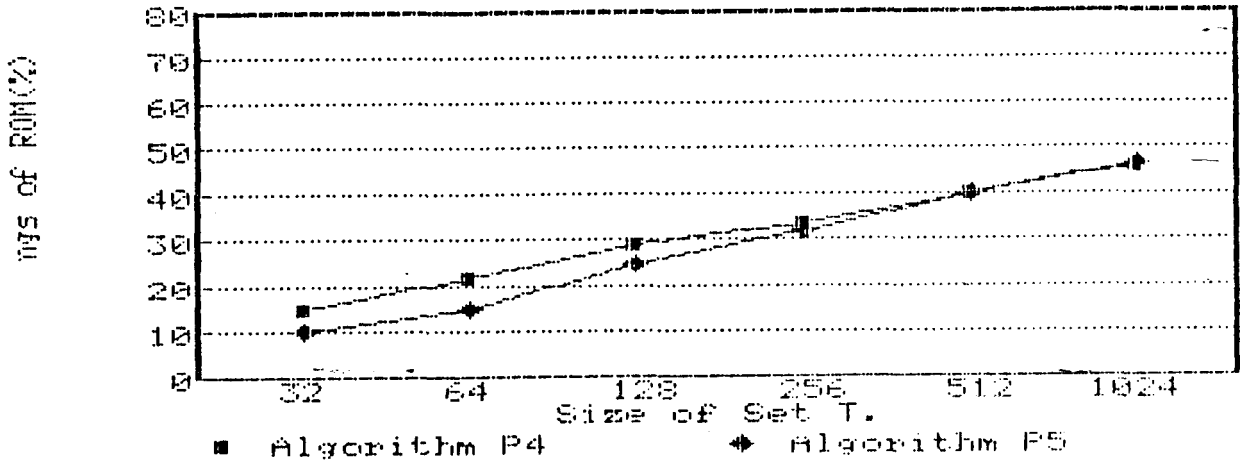
Example 5:

Let T = (0000,0011,0110). Algorithms P2 to P4 would produce T_2 = {0000,0100,0000,0010,0011,0010,1010,0010,0110} requiring a 1 * 8 = 8 bit ROM. Algorithm P5 using the f value as the choosing criteria would produce T_2 = (0000,0100, 0110,0111,1111,0111,0011} which would require a 1 * 6 bit ROM

for implementation. Prestored-T would result in a 3 * 4 = 12 bit ROM.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings(%)
32	153	460	52	10
64	290	871	153	15
128	513	1540	508	25
256	930	2790	1306	32
512	1650	4948	3244	40
1024	2915	8745	7639	47

Table 5. Average results of Algorithm P5 compared to Prestored-T.



Graph 3. Comparison of Average performances of Algorithms P4 and P5.

Algorithm P5 is optimal in that the fewest additional vectors are added each time LINK is called. Whether Algorithm P5 is absolutely optimal in the sense of producing a T_2 with the minimal number of vectors is dealt with in the next chapter. Algorithms P2, P3, P4, P5 represent a progression of GREEDY APPROACH [2] algorithms. However x and t_j can at most differ in n positions. This means the maximum number of vectors required to connect x and tj is n - 1.

For large values of N, we would expect X to find many vectors in T - T_2 requiring the same number of linking vectors. Algorithm P5 chooses the first vector with minimal f value, $f(x,t_j,POS) \leq f(x,t_i,POS)$ for all t_i,t_j in T - T_2 and $i > j$. Example 6 demonstrates this selection process is not always optimal.

Example 6:

Assume for all algorithms the non-shift register in the hardware drawings is initialized to the first vector of the set T.

Let T = {00000000,00110000,00001000,00001100}

Prestored-T would require 8 * 4 = 32 bit ROM.

Algorithm P1 would produce

T_1 = {00000000,00001000,00001100,00000100,00000000,00100000,

00110000) requiring 3 * 6 = 18 bit ROM.

Algorithms P2, P3, and P4 would produce

T_2 = (00000000,00001000,00001100,00001000,10001000,00001000,
00101000,00111000,00110000) requiring 3 * 8 = 24 bit ROM.

Algorithm P5 would produce

T_2 = {00000000,00100000,00110000,00110010,00110000,
01110000,00110000,00010000,00000000,00001000,00001100)
requiring 3 * 10 = 30 bit ROM for implementation.

Example 6 shows Algorithm P5 will in certain situations make a less than optimal choice. It also shows minimizing the H distance can still improve performance. Algorithm P6 uses the same criteria as Algorithm P4, but reverses their precedence. That is, it first minimizes the f value and then the H distance.

<u>Algorithm P6</u>;
<u>Input</u> POS,T; <u>Output</u> T_2;
<u>Begin</u>;
x := t_1;
add x to T_2;
<u>While</u>(T $\not\subseteq$ T_2) <u>do</u>

```
Begin;
    y := first element of T - T2;
    For (all ti in T - T2) do
        Begin;
            If (f(x,ti,POS) = f(x,tj,POS)) then
                If (H(x,y) < H(x,ti)) then
                        y := ti;
            Else
                If (f(x,ti,POS) < f(x,tj,POS)) then
                        y := ti;
        End;
    Link(POS,x,y,T2);
    x := y;
  End;
End;
```

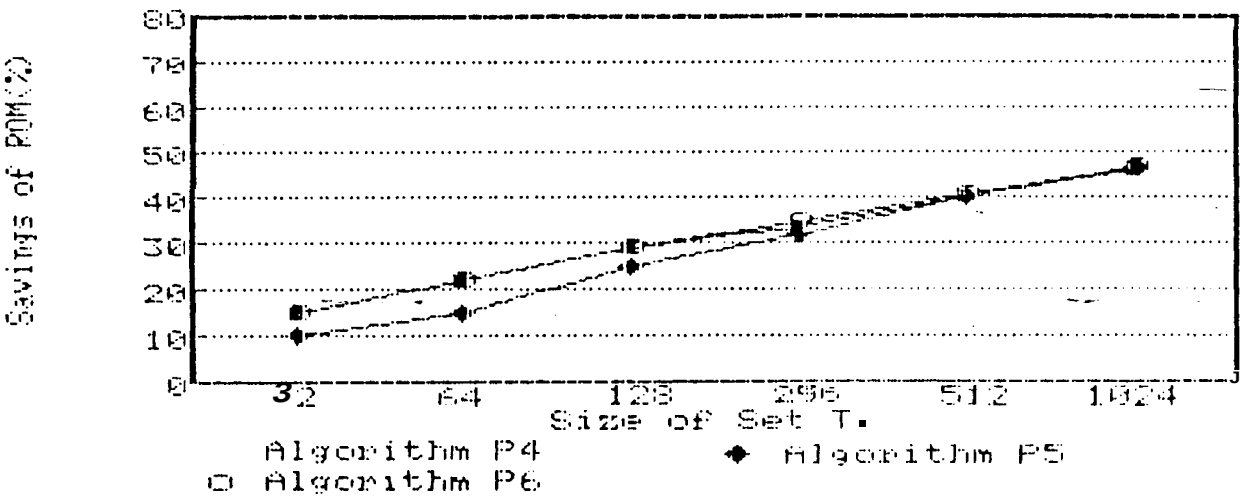
Example 7:

Let T = (0000,1110,0010,0011). Algorithm P4 would produce T_2 = {0000,0100,0000,0010,0011,0010,1010,1110} which would need a 1 * 7 = 7 bit ROM for implementation. Algorithm P5 would produce T_2 = {0000,1000,1100,1110,1111, 0111,0011,0001,0011,0010} requiring a 1 * 9 = 9 bit ROM. Algorithm P6 would produces the same T_2 as Algorithm P4 so then also needs a 7 bit ROM.

The results of the test runs for Algorithm P6 are contained in Table 6, and compared with Algorithms P4 and P5 in Graph 4.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings (%)
32	144	433	79	15
64	267	800	224	22
128	484	1451	597	29
256	890	2670	1426	35
512	1615	4844	3348	41
1024	2917	8751	7633	47

Table 6. Average results of Algorithm P6 compared to Prestored-T.



Graph 4. Comparison of Average performances of Algorithms P4, P5 and P6.

Algorithm **P6** performed better than the previous five methods. **Algorithms P6** and **P4** performed very closely in percentage savings. This seems reasonable since we would expect a large percentage of the time the vector with minimal H distance would need the least number of linking vectors and the reverse to be true also. But this is not true in all cases as is reflected by Tables **4** and **6.**

Algorithm **P4** is chosen for comparison because it performed more efficiently most of the time. Algorithm **P6** does show a slight percentage improvement over **P4** for all values of N except **64.** However, a single percentage point savings with N = **1024** saves far more ROM bits than the same increase when N = **64.** Since inefficiencies are more costly when N is. large, Algorithm **P6** is more attractive than **P4.**

Let $k = \lceil \log_2 n - 1 \rceil$. A RCM method saves $n - k$ bits for each t_i in T . A RCM saves ROM space if the number of added is less than $((n - k)/k) * N$. It was observed during the validation of the program run that all the RCMs executed well at the beginning of each test set, the average number of added vectors is low and acceptable. Then the RCM enters a period of poor results increasing the average number of added vectors. The RCM then alternates between periods of good and poor

results. The next algorithm focuses on the periods of poor execution with the intent of shortening their duration.

Suppose with respect to x, t_i and tj require the same number of connecting vectors but t_i has a one bit in position 0 while t_j has a zero bit in position 0. If over 50% of the remaining vectors in $T - T_2$ have an one bit in position 0, then choosing t_i would mean a majority of the remaining vectors in $T - T_2$ match t_j in one position. Algorithm P7 applies this observation when choosing the next vector, hoping to increase the likelihood of there being vectors which match tj in some positions.

Let $$\phi(a,i) = \begin{cases} 1 & \text{If over 50\% of the vectors in } T - T_2 \text{ have bit a in position } i. \\ 0 & \text{otherwise.} \end{cases}$$

Let $t_j = a_0 a_1 \ldots a_{n-1}$ then $W(t_i) = \sum_{i=0}^{n-1} \phi(a_i, i)$ is called the weight of t_j.

<u>Algorithm P7</u>;

<u>Input</u> POS,T; <u>Output</u> T_2;

<u>Begin</u>;

$x := t_1$;

```
add x to T2;
While( T ⊄ T2 ) do
     Begin;
          y := first element of T - T2;
          For (all ti in T - T2) do
               Begin;
                    If (f(x,ti,POS) = f(x,tj,POS))then
                         If (W(y) < W(ti)) then
                              y := ti;
                    Else
                         If (f(x,ti,POS) < f(x,tj,POS))then
                              y := ti;
               End;
     End;
End.
```

Example 8:

Let T = (0000,1110,0010,1111) starting with would mean T - T_2 = (1110,0010,1111). Both 1110 and 0010 require 2 linking vectors. Currently in T - T_2, there are 2 one bits in position 0, 2 one bits in position 1, 3 one bits in position 2, and 1 one bit in position 3. The weight of 1110 is 1+1+1+1 = 4, while the weight of 0010 = 0+0+1+1=2.

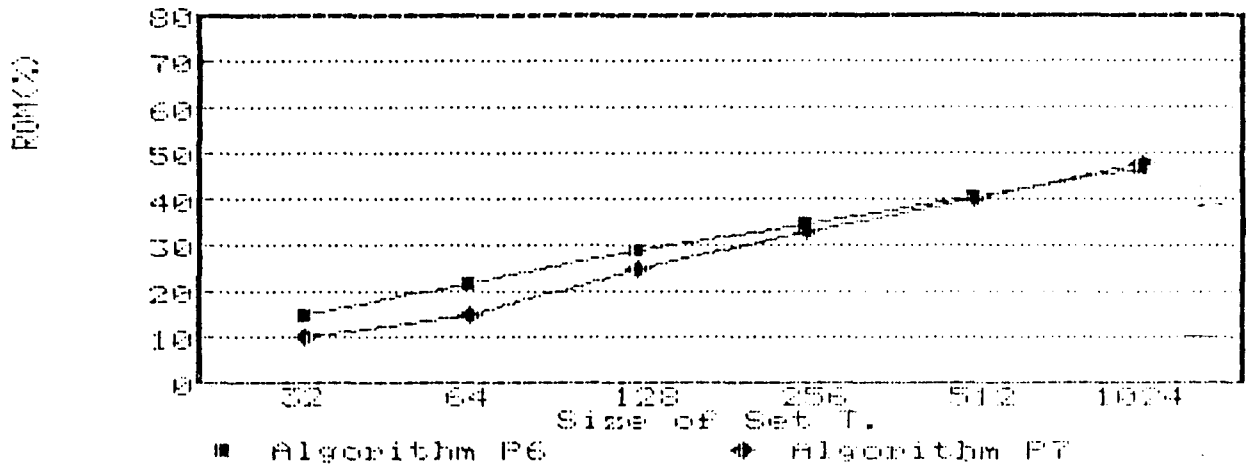
So 1110 is chosen as the next vector from $T - T_2$. The resulting T_2 is equal to (0000,1000,1100,1110,1111,0111,0011,0001, 0011,0010) and a 1 * 9 bit ROM.

Results of Algorithm P7 are listed in Table 7 and a comparison with Algorithm P6 is shown in Graph 5. The rapid improvement in efficiency of Algorithm P7 was expected. The weight of a vector will be more meaningful in a large sample space since the weight is nothing more than a comparison of t_j to the 'average' vector of $T - T_2$. Thus with a small N, the weight has little significance. The results of Table 7 show Algorithm P7 should be implemented when is N large.

One fact has been constant for all of the RCM studied, a test set can be formed for each method such that the method being studied may not yield minimal T_2. So the methods of Algorithms P2 through P7 do not represent optimal solutions. Whether an optimal solution exists may be easier to answer than finding it. This aspect is studied in the next chapter.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings(%)
32	154	463	49	10
64	290	868	156	15
128	513	1541	507	25
256	921	2764	1332	33
512	1628	4883	3308	40
1024	2875	8625	7759	48

Table 7. Average results of Algorithm P7 compared to Prestored-T.



Graph 5. Comparison of Average performances of Algorithms P6 and P7.

Chapter 3

The Existence of an Optimal Solution.

The General Problem

Let $t_i = a_0a_1a_2\ldots a_{n-1}$, $T = \{t_1,t_2,\ldots,t_N\}$, and let r and f be defined as in Chapter 2. The general problem is the minimization of T_2. It can be represented by a weighted graph. Figure 4 represents the general problem with N = 3, and n = 4.

The objective of the algorithms of Chapter 2 is minimization of the size of set T_2 in accordance with conditions of RCM. In Figure 4, the nodes are the elements of T. For each possible value of POS, there is an edge from vector t_j to tj weighted by $f(t_i,t_j,POS)$. Formally, edges can be labeled $(P^k_{ij},F(i,j,p))$, where P^k_{ij} = the ending value of POS after linking t_i to tj with POS = k, and $F(i,j,p) = f(t_i,t_j,$ POS = p), i.e., the number of linking vectors. Therefore minimization of T_2 is equivalent to finding a minimal path for Figure 4.

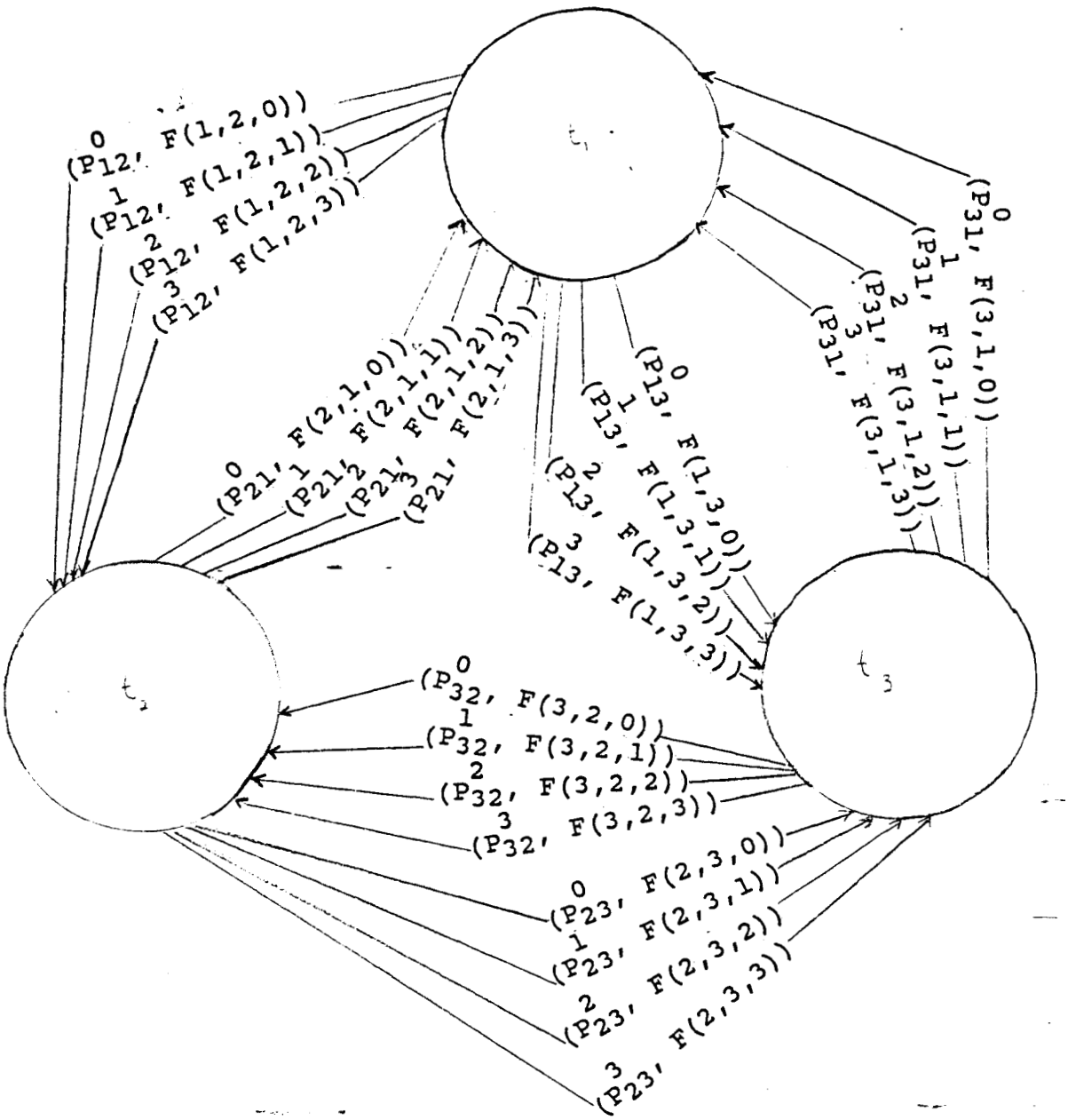


Figure 4. **Representation of the General Problem of RCM with N = 3, and n = 4.**

Applying Lemma 1, irrespective of the starting value of POS, POS can be adjusted to 0 by adding less than six vectors. This allows us to produce from the problem of Figure 4 the problem of Figure 5. The problem of Figure 5 will always start the linking process with POS = 0 and POS will still be 0 when the linking process is completed, (we would have to modify our linking scheme to make the starting and ending value of POS = 0).

The minimization of T_2 involves finding a minimal path passing through all nodes of the graph associated with T_2 (as illustrated by Figure 4). Applying Lemma 1 a subgraph, (as illustrated by Figure 5), is produced. It is believed that the general problem, under the constraints of the hardware design, is a NP complete problem.

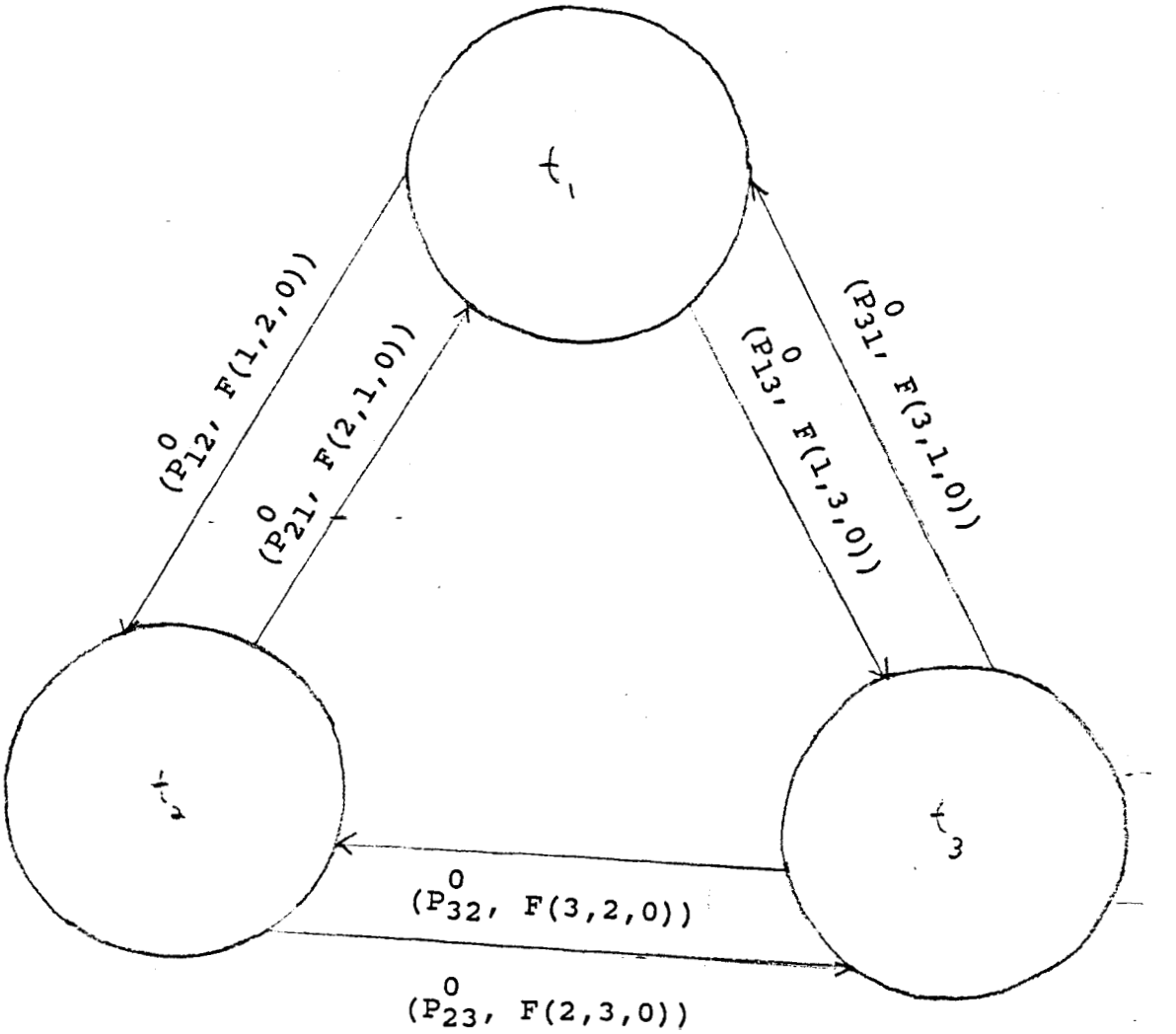


Figure 5. **Applying Lemma 1 to the General Problem with $N = 3$, and $n = 4$.**

We believe if a minimal path can be found for the original graph, then one can be found for the subgraph. However a solution to the subgraph is a Hamilton path and is a NP complete problem. Therefore it is strongly believed though not completely proved at this time, that the NP completeness of the subgraph will imply finding a minimal path for the original graph will also be a NP complete problem.

Even though this problem may be NP complete it might be commercially feasible to investigate. In a real world situation, N and n are fixed so that an exhaustive search may be a feasible option. The algorithms of Chapter 2 are not optimal, however they do save significant amount of ROM space and their execution time is considerably less than exhaustive searching. So as N becomes larger and exhaustive searching becomes more expensive, algorithmsfrom Chapter 2 become more attractive options.

By Lemmas 2 and 3, the linking and comparison processes are of order n. Therefore the Order of Algorithms

P2 through **P7** are of Order N * n. And as N >> n, the Order of **P2** through **P7** becomes N. An exhaustive search procedure would be of order N! and extremely expensive to implement for large values of N. The run times of Algorithms **P2** - **P7** were polynomial, so that their execution should be considerably less than exhaustive searching.

The RCM results are interesting, but not complete. All algorithms presented in chapters 1 and 2 ignore the output set O from Figure 1. Compression of the output set is very important. If for every linking vector added to set T_1 or T_2 an additional output vector is added directly to output set O_1, two problems will develop:

(1) O_1's increase in size from set O will negate any ROM savings the compression method produced.

(2) How will the testing circuit be able to differentiate valid test and output vectors from the don't care vectors produced from the compression method implemented?

These problems must be answered if the algorithms presented in this thesis are to have any practical value.

Chapter 4

A Method to Differentiate Valid and Invalid Results

A simple way to simultaneously compact Prestored-T and Prestored-0 is to combine the two ROMs into a single ROM, V, as shown in Figure 6.

```
   ROM V
-----------------
|       :       |
|   T   :   O   |
|       :       |
-----------------
```

Figure 6: Combined Prestored-T and Prestored-0

ROM V would have N vectors, the same as T and O. A vector v_i from V would be $v_i = t_i o_i$ where t_i is the ith vector from set T and o_i is the corresponding valid output vector from O. Any compaction method we have studied could be used on set V as we did on set T. Let vectors from T be k bits, and vectors from O be s bits. Then vectors from V would be n = k + s bits.

Suppose a RCM was applied to set **v** producing set v^1. The hardware needed to implement this scheme is almost identical to Figure 3 with a few additions as illustrated in Figure 7.

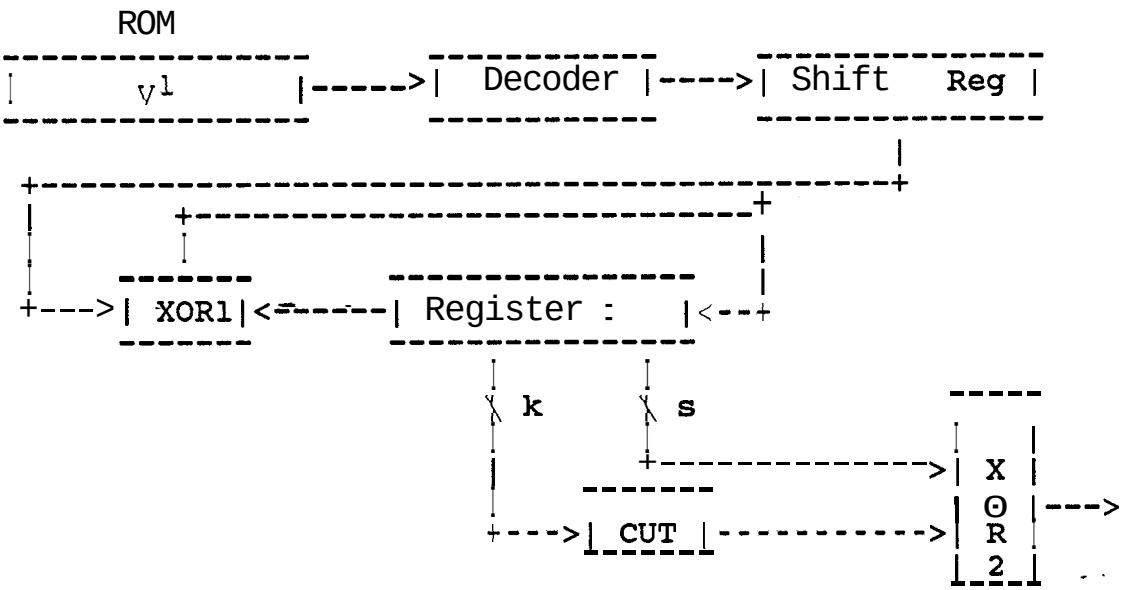


Figure 7. The hardware needed to implement RCM compression of Prestored-T and Prestored-0.

The hardware still does not have a way to distinguish valid output from the invalid output produced by the RCM employed. A straight forward approach would be to place an extra bit on the end of each vector in v^1. Let x_i be a vector of v^1. If $x_i = v_k$ for some v_k in set **v** then the extra bit would be 0, otherwise the bit would be 1. This extra bit would be placed into an OR gate with the output from the gate'

of XOR2 in Figure 7.

This method performed poorly. The RCM saving was not enough to offset the additional flag bit cost on each vector in V^1. A flag bit will be used to identify valid from invalid results. However, instead of adding an extra bit onto each vector of V^1, the flag bit will be incorporated into one of the bit positions.

For simplicity, assume the flag bit is placed into position 0, the first position of the vectors. First divide V into two disjoint sets V_1 and V_0. V_1 contains all vectors of V having a one bit in position 0, while V_0 contains all vectors of V having a zero bit in position 0. The union of sets V_1 and V_0 is V.

Assume we start with V_1 and apply a RCM with a special condition, the first position, position 0, is ignored from the linking process. Thus the right circular motion will be

1, to 2,3,4,...,n-1, to 1,2,... .

Assume the RCM on V_1 forms set W_1. The complementing procedure does not use position 0 thus allows the flag to be placed into position o. For V_1, during the RCM procedure, a one bit is placed in the first position if the vector was an element

of v_1, and a zero bit is placed in position 0 if the vector is a linking vector.

In set W_1, for w_i in W_1, if w_i has a one in position 0, then it is an element of v_1 and will generate valid output. If w_i has a zero in the first position, it is a linking vector and will produce output to be ignored. This allows the circuit to distinguish valid from invalid vectors in W_1.

The process is then duplicated with set v_0 but with reverse logic for the flags. Assume W_2 is produced when the RCM uses v_0 as input. The RCM must place a zero bit in position 0 if the vector was an element of v_0, and places an one bit if the vector was created to link. So for a w_i in W_2, if position 0 is 0 then this vector will produce a valid output in the CUT, otherwise the vector's output should be ignored.

The hardware necessary for this approach is presented in Figure 8. All the additional hardware not found in Figure 7 is used to distinguish valid and invalid results. Register 1 is initialized to the first vector of W_1. Register 2 is initialized to the last vector of W_1. The JK flip-flop is set at start up of test mode.

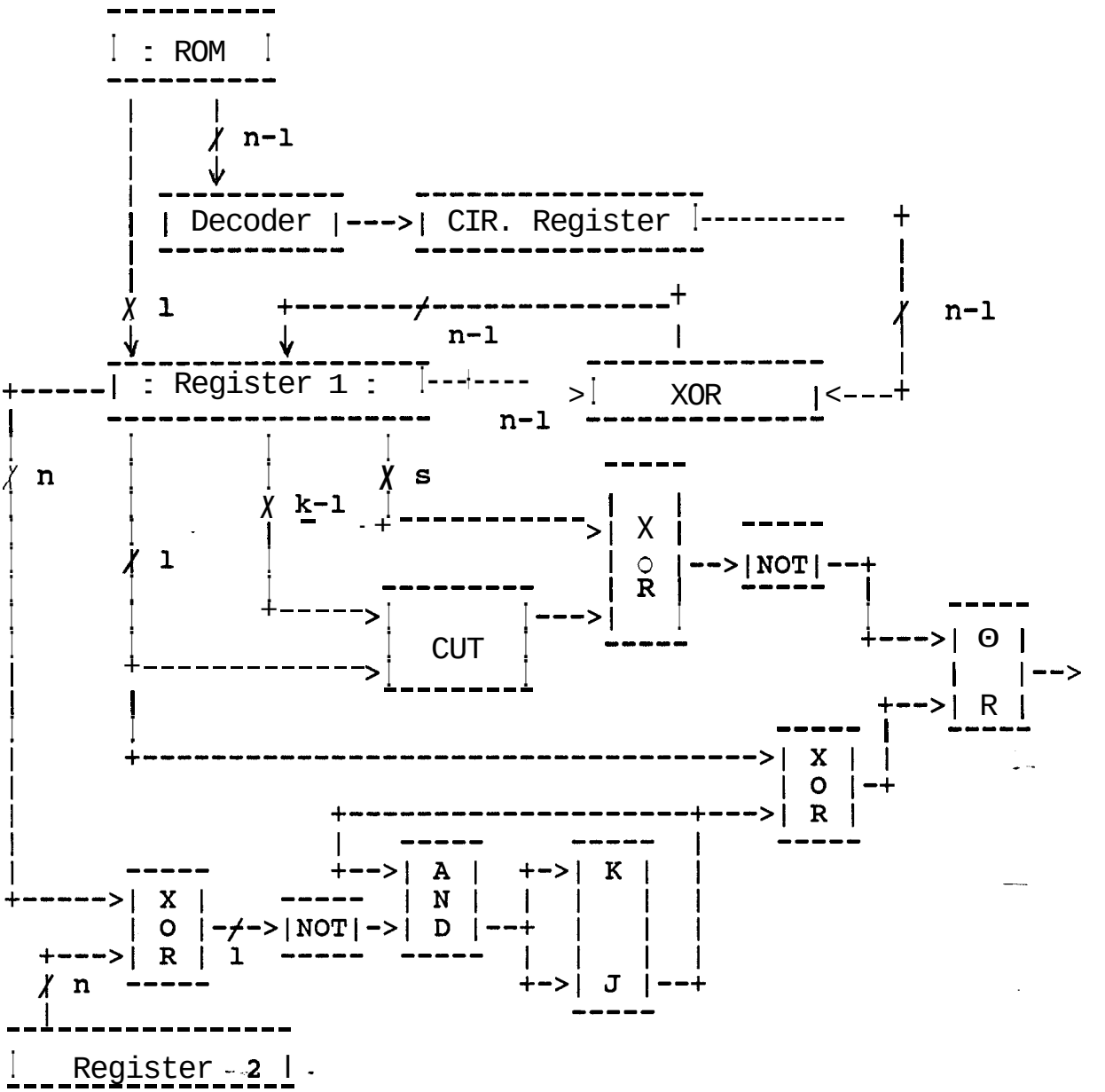


Figure 8. Hardware conception for valid/invalid distinction compaction of Prestored-T and Prestored-0.

Example 9. Assume we use the **RCM** of Algorithm 2.

Let V = (00000000,10000001,00001110,11001001,01000100)

then V_1 = {10000001,11001001},

V_0 = {00000000,00001110,01000100}

For V_1, the linking would involve vectors 0000001,1001001.

Linking would yield 0000001,1000001,1001001.

Therefore W_1 = (10000001,01000001,11001001)

For V_0, the linking would use 0000000,0001110,1000100.

The first vector chosen would be 0000000, since it has the smallest H distance from the last vector of V_1.

Linking produces 1000001,1000000,0000000,1000000,1001000,
1000000,1000100,0000100,0001100,0001110.

and W_2 = (11000001,11000000,00000000,11000000,11001000,
11000000,01000100,10000100,10001100,00001110}.

The criteria for vector selection was H distance as in Algorithm P2. Assuming the main register is initialized to the first vector of W_1, the **ROM** produced for this scheme is

3 * 13 = 39 bits **ROM**

```
0 00  W1
1 11

1 00  W2
1 10
0 01
1 00
1 11
1 11
0 01
1 01
1 11
1 11
0 10
```

Prestored-T would require a 5 * 8 = 40.

From Figure 7 and Example 9 the logic of this method can be seen. While register 1 is producing vectors from W_1, the signal from the JK ff will be 1. If the first bit of register 1 is one, the XOR gate receiving the JK signal will produce a 0 signal to the final OR gate which means the CUT XORed with the expected output will determine the output of the OR gate. If the first bit of the register is 0 then the output of the OR gate will be 1 in effect ignoring the CUT comparison.

The contents of register 2 are constantly compared to the contents of register 1. When a match is found, i.e., register 1 contains the last element of W_1, a complement signal changes the output of the JK flip-flop to 0 for the rest of the testing period. This forces a reverse logic with respect to the first bit of register 1. Hence, a 0 bit now in register 1's first bit will mean the CUT comparison will determine the final output, and a one bit will mean the final output will always be 1.

Algorithm P8 presents the steps to this method.

```
Alsorithm P8;
Input V,POS; Output V2;
Begin;
    Divide V into disjoint sets V1 and V0;
    Let X := the first element of V1;
    Place X in V2
    While ( V1 ⊄ V2 )do
      Begin;
         Let y be chosen the next vector by some RCM criteria
               from V2 - V1;
         Link x to y ignoring the first bit of both vectors
              and place a 0 bit in the first position of the.
              linking vectors and place them in V2;
         place y in V2;
         x := y;
      End;
   While (V0 ⊄ V2) do
      Begin;
         Let y be chosen the next vector by some RCM criteria
               from V2 - V0;
         Link X to y ignoring the first bit of both vectors
              and place a 1 bit in the first position of the
```

```
            linking vectors and place them in V2;
        add y to V2;
        X := y;
    End;
End.
```

Algorithm P8 was tested on the random sets the seven previous algorithms were tested on. The averages of the runs are in Table 8, RCM of Algorithm P7 was employed.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings(%)
32	157	628	-116	-22
64	298	1092	-68	-6
128	493	1972	76	4
256	960	3840	256	7
512	1688	6752	1440	18
1024	3001	12004	4380	27

Table 8. Average Results of Algorithm P8, using position 0 for flag bit, compared to Prestored-T.

It was hoped the exclusion of the first bit would decrease the number of vectors produced by a RCM method. On average, this goal was achieved. However the decrease was not

enough to completely make up for the additional space of the flag bits.

It may be possible to decrease the size of V_2 by choosing a position other than the first to imbed the flag bit. When V is divided into V_1 and V_0, the choices the RCM has for the next vector are reduced. The best case would be if all vectors of V had zeros, or if all had ones. Therefore the last series of experimental runs consisted of applying the idea of Algorithm P8 on the position containing the most 1 bits. The results of Table 9 were run using Algorithm P7 as the chosen RCM.

Size of T	Size of T_1	New ROM Size	Bits Saved	Savings(%)
32	145	580	-68	-13
64	273	1092	-68	-6
128	493	1972	76	4
256	870	3480	616	18
512	1553	6212	1980	25
1024	2743	10972	6097	37

Table 9. Average Results of Algorithm P8, using the position with the greatest number of one bits, compared to Prestored-T.

Chapter 5

Conclusions

Prestored-T and Prestored-0 are major costs of BIST. We wanted methods reducing the ROM space of Prestored-T. The results of the algorithms presented in chapter 2 are encouraging. They showed cognitive approaches to compaction of Prestored-T can improve the efficiency of ROM storage with very little extra cost. None of the algorithms presented are minimal as there are sets for which they will produce a less than optimal solution. The problem of finding a minimal solution for Prestored-T through RCM on T maybe a NP complete problem. Thus a minimal solution may not be obtainable except through exhaustive searching. But RCMs offer a substantial ROM savings and require very little extra hardware.

In order to include output and a method of output validity distinction, some efficiency is sacrificed. However the results of Tables 8 and 9 show savings are still possible. A flag bit used to distinguish valid/invalid output requires considerable ROM space thus hurting savings.

Algorithm P8 reduced the number of vectors in T_2 by reducing the number of bits involved in linking. There are several possible ways of improving P8. Careful selection of where to place the flag bit and employing a different RCM are two possibilities. The first alternative was tested, and Table 9 shows a definite improvement.

The second option would be an easy inexpensive possibility. Some other possible TPG worth investigating are RCM(z), $z > 1$. The graph representation of the General problem offers an interesting possibility of approaching this problem with a minimal spanning tree as a solution. There would have to be changes in the hardware, but a solution of order n already exists for this approach. This thesis has looked at a number of possible solutions, but many more possibilities exist.

APPENDIX A

Complete Listing of Results of Algorithms P1-P7

Trial	Size of T_1	New ROM Size	Bits Saved	Savings(%)
1	134	536	-24	- 5
2	141	564	-52	-10
3	140	560	-48	- 9
4	135	540	-28	- 5
5	140	560	-48	- 9
6	135	540	-28	- 5
7	143	572	-60	-12
8	140	560	-48	- 9
9	146	584	-72	-14
10	142	568	-56	-11
11	142	568	-56	-11
12	139	556	-44	- 9
13	140	560	-48	- 9
14	136	544	-32	- 6
15	142	568	-56	-11
Average	140	589	-47	- 9

Table Al. Algorithm P1 Size of T = 32.

Trial	Size of T_1	New ROM Size	Bits Saved	Savings(%)
1	154	462	50	10
2	163	489	23	5
3	158	474	38	7
4	163	489	23	4
5	160	480	32	6
6	155	465	47	10
7	169	507	5	1
8	158	474	38	7
9	164	492	20	4
10	164	492	20	4
11	162	486	26	5
12	157	471	41	8
13	162	486	26	5
14	154	462	50	10
15	164	492	20	4
Average	160	481	31	6

Table A2. Algorithm P1 Size of T = 32.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	154	462	50	10
2	163	489	23	5
3	158	474	38	7
4	163	489	23	4
5	160	480	32	6
6	155	465	47	9
7	169	507	5	1
8	158	474	38	7
9	164	492	20	4
10	164	492	20	4
11	162	486	26	5
12	157	471	41	8
13	162	486	26	5
14	154	462	50	10
15	164	492	20	4
Average	161	481	31	6

Table A3. Algorithm P3 Size of T = 32.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	141	423	89	17
2	147	441	71	14
3	143	429	83	16
4	151	453	59	12
5	142	426	86	17
6	140	420	92	18
7	150	450	62	12
8	148	444	68	13
9	145	435	77	15
10	146	438	74	14
11	143	429	83	16
12	144	432	80	16
13	142	426	86	17
14	148	444	68	13
15	145	435	77	15
Average	145	435	77	15

Table A4. Algorithm P4 Size of T = 32.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	156	468	44	8
2	153	459	53	10
3	156	468	44	8
4	153	459	53	10
5	150	450	62	12
6	150	450	62	12
7	156	468	44	10
8	150	450	62	12
9	152	456	56	11
10	151	453	59	12
11	156	468	44	8
12	153	459	53	10
13	147	441	71	14
14	161	483	29	6
15	158	474	38	7
Average	153	460	52	10

Table A5. Algorithm P5 Size of T = 32.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	141	423	89	17
2	145	435	77	15
3	143	429	83	16
4	150	450	62	12
5	142	426	86	17
6	140	420	92	18
7	151	453	59	12
8	145	435	77	15
9	145	435	77	15
10	143	429	83	16
11	143	429	83	16
12	144	432	80	16
13	142	426	86	17
14	145	435	77	15
15	145	435	77	15
Averages	144	433	79	15

Table A6. Algorithm P6 Size of T = 32.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	152	456	56	11
2	156	468	44	9
3	156	468	44	9
4	155	465	57	9
5	153	459	53	10
6	154	462	50	10
7	157	471	41	8
8	150	450	62	12
9	149	447	65	13
10	152	456	56	11
11	152	456	56	11
12	164	492	20	4
13	154	462	50	10
14	153	459	53	10
15	156	468	44	9
Averages	154	463	49	10

Table A7. Algorithm P7 Size of T = 32.

Trial	Size of T_1	New ROM Size	Bits Saved	Savings(%)
1	264	1056	-32	-3
2	263	1052	-28	-3
3	265	1060	-36	-4
4	267	1068	-44	-4
5	258	1032	- 8	-1
6	260	1040	-16	-2
7	264	1056	-32	-3
8	264	1056	-32	-3
9	264	1056	-32	-3
10	264	1056	-32	-3
11	262	1048	-24	-2
12	262	1048	-24	-2
13	262	1048	-24	-2
14	266	1064	-40	-3
15	262	1048	-24	-2
Average	263	1053	-29	-3

Table A7. Algorithm P1 , Size of T = 64.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	308	924	100	10
2	315	945	79	8
3	305	915	109	11
4	307	921	103	10
5	298	894	130	13
6	300	900	124	12
7	308	924	100	10
8	306	918	106	10
9	302	906	118	12
10	302	906	118	12
11	298	894	130	13
12	304	912	112	11
13	306	918	106	10
14	304	912	112	11
15	306	918	106	10
Average	305	914	110	11

Table A8. Algorithm P2 , Size of T = 64.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	308	924	100	10
2	315	945	79	8
3	305	915	109	11
4	307	921	103	10
5	298	894	130	13
6	300	900	124	12
7	308	924	100	10
8	306	918	106	10
9	302	906	118	12
10	302	906	118	12
11	298	894	130	13
12	304	912	112	11
13	306	918	106	10
14	304	912	112	11
15	306	918	106	10
Average	305	914	110	11

Table A9. Algorithm P3 , Size of T = 64.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	271	813	211	21
2	267	801	223	22
3	265	795	229	22
4	271	813	211	21
5	265	795	229	22
6	265	795	229	22
7	267	801	223	22
8	264	792	232	23
9	268	804	220	21
10	266	798	226	22
11	260	780	244	24
12	268	804	220	21
13	266	798	226	22
14	269	807	217	21
15	261	783	241	24
Average	266	799	225	22

Table A10. Algorithm P4 , Size of T = 64.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	285	855	169	17
2	295	885	139	14
3	293	879	145	14
4	291	873	151	15
5	299	897	127	12
6	286	858	166	16
7	284	852	172	17
8	292	876	148	14
9	285	855	169	17
10	287	861	163	16
11	281	843	181	18
12	298	894	130	13
13	296	888	136	13
14	295	885	139	14
15	290	870	154	15
Average	290	871	153	15

Table A12. Algorithm P5, Size of T = 64.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	269	807	217	21
2	267	801	223	22
3	265	795	229	22
4	269	807	217	21
5	265	795	229	22
6	266	798	226	22
7	269	807	217	21
8	264	792	232	23
9	270	810	214	21
10	266	798	226	22
11	260	780	244	24
12	268	804	220	21
13	270	810	214	21
14	269	807	217	21
15	261	783	241	24
Average	267	800	224	22

Table A13. Algorithm P6, Size of T = 64.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	282	846	178	17
2	291	873	151	15
3	280	840	184	18
4	291	873	151	15
5	299	897	127	12
6	285	855	169	17
7	276	828	196	19
8	286	858	166	16
9	291	873	151	15
10	299	897	127	12
11	288	864	160	16
12	298	894	130	13
13	291	873	151	15
14	290	870	154	15
15	295	885	139	14
Average	289	868	156	15

Table A14. Algorithm P7, Size of T = 64.

Trial	Size of T,	New ROM Size	Bits Saved	Savings(%)
1	471	1884	164	8
2	472	1888	160	8
3	472	1888	160	8
4	465	1860	188	9
5	470	1880	168	8
6	465	1860	188	9
7	472	1888	160	8
8	482	1928	120	6
9	468	1872	176	9
10	466	1864	184	9
11	474	1896	152	7
12	470	1880	168	8
13	461	1844	204	10
14	474	1896	152	7
15	473	1892	156	8
Average	470	1881	167	8

Table A15. Algorithm P1, Size of T = 128.

Trial	Size of T	New ROM Size	Bits Saved	Savings(%)
1	549	1647	401	20
2	558	1674	374	18
3	562	1686	362	18
4	541	1623	425	21
5	550	1650	398	19
6	547	1641	407	20
7	546	1638	410	20
8	566	1698	350	17
9	544	1632	416	20
10	540	1620	428	21
11	560	1680	368	18
12	540	1620	428	21
13	539	1617	431	21
14	552	1656	392	19
15	549	1647	401	20
Average	550	1649	399	20

Table A16. Algorithm P2, Size of T = 128.

Trial	Size of T_1	New ROM Size	Bits Saved	Savings(%)
1	549	1647	401	20
2	558	1674	374	18
3	562	1686	362	18
4	541	1623	425	21
5	550	1650	398	19
6	547	1641	407	20
7	550	1650	398	19
8	566	1698	350	17
9	544	1632	416	20
10	540	1620	428	21
11	560	1680	368	18
12	540	1620	428	21
13	539	1617	431	21
14	552	1656	392	19
15	549	1647	401	20
Average	550	1649	399	19

Table A17. Algorithm P3, Size of T = 128.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	490	1470	578	28
2	479	1437	611	30
3	479	1437	611	30
4	498	1494	554	27
5	490	1470	578	28
6	482	1446	602	29
7	479	1437	611	30
8	479	1437	611	30
9	490	1470	578	28
10	486	1458	590	29
11	495	1485	563	27
12	487	1461	587	29
13	495	1485	563	27
14	487	1461	587	29
15	491	1473	575	28
Average	487	1461	587	29

Table A18. Algorithm P4, Size of T = 128.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	504	1512	536	26
2	506	1518	530	26
3	511	1533	515	25
4	491	1473	575	28
5	515	1545	503	25
6	524	1572	476	23
7	522	1566	482	24
8	516	1548	500	24
9	513	1539	509	25
10	513	1539	509	25
11	530	1590	458	22
12	516	1548	500	24
13	499	1497	551	27
14	523	1569	479	23
15	515	1545	503	25
Average	513	1540	508	25

Table A19. Algorithm P5, Size of T = 128.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	482	1446	602	29
2	484	1452	596	29
3	482	1446	602	29
4	486	1458	590	29
5	481	1443	605	30
6	484	1452	596	29
7	479	1437	611	30
8	479	1437	611	30
9	487	1461	587	29
10	480	1440	608	30
11	482	1446	602	29
12	486	1458	590	29
13	487	1461	587	29
14	487	1461	587	29
15	488	1464	584	29
Average	483	1451	597	26

Table A20. Algorithm P6, Size of T = 128.

Trial	Size of T_A	New ROM Size	Bits Saved	Savings(%)
1	505	1515	533	26
2	511	1533	515	25
3	508	1524	524	26
4	510	1530	518	25
5	523	1569	479	23
6	519	1557	491	24
7	506	1518	530	26
8	517	1551	497	24
9	508	1524	524	26
10	509	1527	521	25
11	529	1587	461	23
12	523	1569	479	23
13	512	1536	512	25
14	507	1521	527	26
15	519	1557	491	24
Average	514	1541	507	26

Table A21. Algorithm P7, Size of T = 128.

Trial	Size of T_1	New ROM size	Bits Saved	Savings(%)
1	833	3332	764	19
2	839	3356	740	18
3	837	3348	748	18
4	839	3356	740	18
5	829	3316	780	19
6	827	3308	788	19
7	828	3312	784	19
8	841	3364	732	18
9	832	3328	768	19
10	846	3384	712	17
11	842	3368	728	18
12	841	3364	732	18
13	840	3360	736	18
14	844	3376	720	18
15	841	3364	732	18
Averages	837	3349	747	18

Table A22. Algorithm P1, Size of T = 256.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	1017	3051	1045	26
2	1011	3033	1063	26
3	1011	3033	1063	26
4	1013	3039	1057	26
5	997	2991	1105	27
6	1019	3057	1039	25
7	998	2994	1102	27
8	1031	3093	1003	24
9	1002	3006	1090	27
10	1018	3054	1042	25
11	1014	3042	1054	26
12	1017	3051	1045	26
13	1010	3030	1066	26
14	1002	3006	1090	27
15	1017	3051	1045	26
Averages	1012	3035	1061	26

Table A23. Algorithm P2, Size of T = 256.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	1024	3072	1024	25
2	1019	3057	1039	25
3	1018	3054	1042	25
4	1025	3075	1021	25
5	1012	3036	1060	26
6	1026	3078	1018	25
7	1016	3048	1048	26
8	1025	3075	1021	25
9	1016	3048	1048	26
10	1016	3048	1048	26
11	1028	3084	1012	25
12	1027	3081	1015	25
13	1010	3030	1066	26
14	1014	3042	1054	26
15	1024	3072	1024	25
Averages	1020	3060	1036	26

Table A24. Algorithm P3, Size of T = 256.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	881	2643	1453	36
2	900	2700	1396	34
3	909	2727	1369	33
4	890	2670	1426	35
5	888	2664	1432	35
6	882	2646	1450	36
7	900	2700	1396	34
8	906	2718	1378	34
9	903	2709	1387	34
10	902	2706	1390	34
11	902	2706	1390	34
12	910	2730	1366	33
13	905	2715	1381	34
14	902	2706	1390	34
15	899	2697	1399	34
Averages	899	2696	1400	34

Table A25. Algorithm P4, Size of T = 256.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	914	2742	1354	33
2	930	2790	1306	32
3	932	2796	1300	32
4	914	2742	1354	33
5	939	2817	1279	31
6	926	2778	1318	32
7	937	2811	1285	31
8	926	2778	1318	32
9	934	2802	1294	32
10	945	2835	1261	30
11	930	2790	1306	32
12	948	2844	1252	31
13	931	2793	1303	32
14	915	2745	1351	33
15	930	2790	1306	32
Averages	930	2790	1306	32

Table A26. Algorithm P5, Size of T = 256.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	908	2724	1372	34
2	868	2604	1492	36
3	894	2682	1414	35
4	896	2688	1408	34
5	882	2646	1450	35
6	883	2649	1447	35
7	896	2688	1408	34
8	898	2694	1402	34
9	886	2658	1438	35
10	905	2715	1381	34
11	887	2661	1435	35
12	884	2652	1444	35
13	890	2670	1426	35
14	887	2661	1435	35
15	887	2661	1435	35
Averages	890	2670	1426	35

Table A27. Algorithm P7, Size of T = 256.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	914	2742	1354	33
2	928	2784	1312	32
3	919	2757	1339	33
4	924	2772	1324	32
5	929	2787	1309	32
6	927	2781	1315	32
7	927	2781	1315	32
8	900	2700	1396	34
9	923	2769	1327	32
10	925	2775	1321	32
11	917	2751	1345	33
12	919	2757	1339	33
13	922	2766	1330	32
14	917	2751	1345	33
15	931	2793	1303	32
Averages	921	2764	1332	33

Table A28. Algorithm P7, Size of T = 256.

Trial	Size of T_1	New ROM size	Bits Saved	Savings(%)
1	1425	5700	2492	30
2	1430	5720	2472	30
3	1437	5748	2444	30
4	1426	5704	2488	30
5	1434	5736	2456	30
6	1416	5664	2528	31
7	1428	5712	2480	30
8	1427	5708	2484	30
9	1427	5708	2484	30
10	1431	5724	2468	30
11	1429	5716	2476	30
12	1431	5724	2468	30
13	1419	5676	2516	31
14	1416	5664	2528	31
15	1424	5696	2496	30
Averages	1427	5707	2485	30

Table A29. Algorithm P1, Size of T = 512.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	1843	5529	2663	33
2	1886	5658	2534	31
3	1856	5568	2624	32
4	1868	5604	2588	32
5	1870	5610	2582	32
6	1828	5484	2708	33
7	1854	5562	2630	32
8	1869	5607	2585	32
9	1871	5613	2579	31
10	1885	5655	2537	31
11	1863	5589	2603	32
12	1861	5583	2609	32
13	1863	5589	2603	32
14	1848	5544	2648	32
15	1838	5514	2678	33
Averages	1860	5581	2611	32

Table A30. Algorithm P2, Size of T = 512.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	1878	5634	2558	31
2	1869	5607	2585	32
3	1890	5670	2522	31
4	1900	5700	2492	30
5	1871	5613	2579	31
6	1846	5538	2654	32
7	1899	5697	2495	30
8	1863	5589	2603	32
9	1879	5637	2555	31
10	1872	5616	2576	31
11	1879	5637	2555	31
12	1876	5628	2564	31
13	1878	5634	2558	31
14	1856	5568	2624	32
15	1892	5676	2516	31
Averages	1877	5630	2562	31

Table A31. Algorithm P3, Size of T = 512.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	1633	4899	3293	40
2	1638	4914	3278	40
3	1649	4947	3245	40
4	1675	5025	3167	39
5	1645	4935	3257	40
6	1648	4944	3248	40
7	1640	4920	3272	40
8	1658	4974	3218	39
9	1640	4920	3272	40
10	1635	4905	3287	40
11	1638	4914	3278	40
12	1647	4941	3251	40
13	1616	4848	3344	41
14	1653	4959	3233	39
15	1639	4917	3275	40
Averages	1644	4931	3261	40

Table A32. Algorithm P4, Size of T = 512.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	1631	4893	3299	40
2	1654	4962	3230	39
3	1687	5061	3131	38
4	1640	4920	3272	40
5	1659	4977	3215	39
6	1644	4932	3260	40
7	1648	4944	3248	40
8	1660	4980	3212	39
9	1654	4962	3230	39
10	1650	4950	3242	40
11	1630	4890	3302	40
12	1613	4839	3353	41
13	1636	4908	3284	40
14	1671	5013	3179	39
15	1661	4983	3209	39
Averages	1649	4948	3244	40

Table A33. Algorithm P5, Size of T = 512.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	1593	4779	3413	42
2	1626	4878	3314	40
3	1633	4899	3293	40
4	1599	4797	3395	41
5	1610	4830	3362	41
6	1617	4851	3341	41
7	1632	4896	3296	40
8	1625	4875	3317	40
9	1618	4854	3338	41
10	1621	4863	3329	41
11	1608	4824	3368	41
12	1583	4749	3443	42
13	1614	4842	3350	41
14	1614	4842	3350	41
15	1629	4887	3305	40
Averages	1615	4844	3347	41

Table A34. Algorithm P6, Size of T = 512.

Trial	Size of T_2	New ROM size	Bits Saved	Savings(%)
1	1632	4896	3296	40
2	1627	4881	3311	40
3	1615	4845	3347	41
4	1626	4878	3314	40
5	1626	4878	3314	40
6	1647	4941	3251	40
7	1621	4863	3329	41
8	1634	4902	3290	40
9	1648	4944	3248	40
10	1612	4836	3356	41
11	1639	4917	3275	40
12	1615	4845	3347	41
13	1616	4848	3344	41
14	1637	4911	3281	40
15	1623	4869	3323	41
Averages	1628	4884	3308	40

Table A35. Algorithm P7, Size of T = 512.

Trial	Size of T_1	New ROM Size	Bits Saved	Savings(%)
1	2496	9984	6400	40
2	2506	10024	6360	39
3	2486	9944	6440	40
4	2484	9936	6448	40
5	2479	9916	6404	40
6	2495	9980	6512	40
7	2468	9872	6448	40
8	2473	9936	6448	40
9	2473	9892	6492	40
10	2487	9948	6436	40
11	2494	9976	6408	40
12	2482	9928	6456	40
13	2470	9880	6504	40
14	2494	9976	6408	40
15	2487	9948	6436	40
Average	2486	9944	6440	40

Table A36. Algorithm P1 Size of T = 1024.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	3440	10320	6064	38
2	3450	10350	6034	37
3	3420	10260	6124	38
4	3430	10290	6094	38
5	3417	10251	6133	38
6	3427	10281	6103	38
7	3400	10200	6184	38
8	3456	10368	6016	37
9	3443	10329	6055	37
10	3429	10287	6097	38
11	3428	10284	6100	38
12	3418	10254	6130	38
13	3406	10218	6166	38
14	3428	10284	6100	38
15	3447	10341	6043	37
Average	3429	10287	6097	38

Table A37. Algorithm P2 Size of T = 1024.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	3573	10719	5665	35
2	3555	10665	5719	35
3	3589	10767	5617	35
4	3552	10656	5728	35
5	3547	10641	5743	36
6	3511	10533	5851	36
7	3600	10800	5584	35
8	3572	10716	5668	35
9	3574	10722	5662	35
10	3543	10629	5755	36
11	3614	10842	5542	34
12	3543	10629	5755	36
13	3690	11070	5314	33
14	3575	10725	5659	35
15	3580	10740	5644	35
Average	3575	10725	5659	35

Table A38 Algorithm P3 Size of T = 1024.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	3024	9072	7312	45
2	2982	8946	7438	46
3	2986	8958	7426	46
4	3031	9093	7291	45
5	3044	9132	7252	45
6	3021	9063	7321	45
7	2973	8919	7465	46
8	2987	8961	7423	46
9	2976	8928	7456	46
10	3009	9027	7357	45
11	3016	9048	7336	45
12	2990	8970	7414	46
13	2954	8862	7522	46
14	3003	9009	7375	46
15	2997	8991	7393	46
Average	3000	9000	7384	46

Table A39 Algorithm P4 Size of T = 1024.

Trial	Size of T_2	New ROM Size	Bits Saved	Savings(%)
1	2880	8640	7744	48
2	2873	8619	7765	48
3	2900	8700	7684	47
4	2865	8595	7789	48
5	2872	8616	7768	48
6	2900	8700	7684	47
7	2866	8598	7786	48
8	2879	8637	7747	48
9	2863	8589	7795	48
10	2848	8544	7840	48
11	2893	8679	7705	48
12	2869	8607	7777	48
13	2871	8613	7771	48
14	2877	8631	7753	48
15	2871	8613	7771	48
Average	2875	8625	7759	48

Table A42. Algorithm P7 Size of T = 1024.

APPENDIX B

Complete Results of Algorithm P8, Both Versions.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	155	620	-108	-21
2	164	656	-144	-28
3	159	636	-124	-24
4	162	648	-136	-26
5	151	604	-92	-17
6	149	596	-84	-16
7	161	644	-132	-25
8	151	604	-92	-17
9	154	616	-104	-20
10	156	624	-112	-21
11	159	636	-124	-24
12	163	652	-140	-27
13	160	640	-128	-25
14	163	652	-140	-27
15	155	620	-108	-21
Average	157	628	-116	-22

Table B1: Results for Algorithm P8, first version,

Size of T = 32.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	143	572	-60	-11
2	148	592	-80	-15
3	161	644	-132	-25
4	139	556	-44	-8
5	139	556	-44	-8
6	151	604	-92	-17
7	149	596	-84	-16
8	145	580	-68	-13
9	146	584	-72	-14
10	137	548	-36	-7
11	144	576	-64	-12
12	141	564	-52	-10
13	148	592	-80	-15
14	148	592	-80	-15
15	143	572	-60	-11
Average	145	580	-68	-13

Table B2: Results for Algorithm P8, second version,

Size of T = 32.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	294	1176	-152	-14
2	302	1208	-184	-17
3	302	1208	-184	-17
4	306	1224	-200	-19
5	301	1204	-180	-17
6	301	1204	-180	-17
7	293	1172	-148	-14
8	302	1208	-184	-17
9	299	1196	-172	-16
10	294	1176	-152	-14
11	295	1180	-156	-15
12	295	1180	-156	-15
13	294	1176	-152	-14
14	291	1164	-140	-13
15	303	1212	-188	-18
Average	298	1192	-168	-16

Table B3: Results for Algorithm P8, first version,

Size of T = 64.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	275	1100	-76	-7
2	268	1072	-48	-4
3	268	1072	-48	-4
4	268	1072	-48	-4
5	267	1068	-44	-4
6	270	1080	-56	-5
7	265	1060	-36	-3
8	283	1132	-108	-10
9	269	1076	-52	-5
10	276	1104	-80	-8
11	274	1096	-72	-7
12	294	1176	-152	-14
13	269	1076	-52	-5
14	280	1120	-96	-9
15	268	1072	-48	-4
Average	273	1092	-68	-6

Table B4: Results for Algorithm P8, second version,

Size of T = 64.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	510	2040	8	1
2	527	2108	-60	-2
3	526	2104	-56	-2
4	518	2072	-24	-1
5	525	2100	-52	-2
6	528	2112	-64	-3
7	527	2108	-60	-2
8	511	2044	4	1
9	531	2124	-76	-3
10	535	2140	-92	-4
11	522	2088	-40	-1
12	520	2080	-32	-1
13	524	2096	-48	-2
14	526	2104	-56	-2
15	534	2136	-88	-4
Average	520	2080	-32	-1

Table B5: Results for Algorithm P8, first version,

Size of T = 128.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	516	2064	-16	0
2	470	1880	168	9
3	531	2124	-76	-3
4	476	1904	144	8
5	483	1932	116	6
6	484	1936	112	6
7	487	1948	100	5
8	531	2124	-76	-3
9	483	1932	116	6
10	474	1896	152	8
11	476	1904	144	8
12	498	1992	56	3
13	496	1984	64	4
14	499	1996	52	3
15	485	1940	108	6
Average	493	1972	76	4

Table B6: Results for Algorithm P8, second version,

Size of T = 128.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	966	3864	232	6
2	964	3856	240	6
3	979	3916	180	5
4	964	3856	240	6
5	931	3724	372	10
6	951	3804	292	8
7	971	3884	212	6
8	947	3788	308	8
9	971	3884	212	6
10	961	3844	252	7
11	971	3884	212	6
12	970	3880	216	6
13	948	3792	304	8
14	957	3828	268	7
15	950	3800	296	8
Average	960	3840	256	7

Table B7: Results for Algorithm P8, first version,

Size of T = 256.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	851	3404	692	17
2	892	3568	528	13
3	896	3584	512	13
4	852	3408	688	17
5	859	3436	660	17
6	866	3464	632	16
7	870	3480	616	16
8	861	3444	652	16
9	871	3484	612	15
10	868	3472	624	16
11	861	3444	652	16
12	880	3520	576	15
13	873	3492	604	15
14	863	3452	644	16
15	884	3536	560	14
Average	870	3480	616	16

Table B8: Results for Algorithm P8, second version,

Size of T = 256.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	1714	6856	1336	17
2	1694	6776	1416	18
3	1702	6808	1384	17
4	1668	6672	1520	19
5	1682	6728	1464	18
6	1674	6696	1496	19
7	1664	6656	1536	19
8	1711	6844	1348	17
9	1662	6648	1544	19
10	1688	6752	1440	18
11	1686	6744	1448	18
12	1653	6612	1580	20
13	1716	6864	1328	17
14	1729	6916	1276	16
15	1683	6732	1460	18
Average	1688	6752	1440	18

Table B9: Results for Algorithm P8, first version,

Size of T = 512.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	1573	6292	1900	24
2	1558	6232	1960	24
3	1581	6324	1868	23
4	1547	6188	2004	25
5	1551	6204	1988	25
6	1551	6204	1988	25
7	1531	6124	2068	26
8	1546	6184	2008	25
9	1558	6232	1960	24
10	1559	6236	1956	24
11	1545	6180	2012	25
12	1545	6180	2012	25
13	1535	6140	2052	26
14	1533	6132	2060	26
15	1577	6308	1884	23
Average	1553	6212	1980	25

Table B10: Results for Algorithm P8, second version,

Size of T = 512.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	2968	11872	4512	28
2	2973	11892	4492	28
3	3206	12824	3560	22
4	2967	11868	4516	28
5	2999	11996	4388	27
6	2952	11808	4576	28
7	3050	12200	4184	26
8	2996	11984	4400	27
9	2982	11928	4456	28
10	2980	11920	4464	28
11	2972	11888	4496	28
12	3008	12032	4352	27
13	3011	12044	4340	27
14	2999	11996	4388	27
15	2957	11828	4556	28
Average	3001	12004	4380	27

Table B11: Results for Algorithm P8, first version,

Size of T = 1024.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	2726	10904	5480	34
2	2745	10980	5404	33
3	2754	11016	5368	33
4	2741	10964	5420	34
5	2721	10884	5500	34
6	2730	10920	5464	34
7	2745	10980	5404	33
8	2751	11004	5380	33
9	2749	10996	5388	33
10	2737	10948	5436	34
11	2741	10964	5420	34
12	2755	11020	5364	33
13	2716	10864	5520	34
14	2744	10976	5408	34
15	2788	11152	5232	32
Average	2743	10972	5412	34

Table B12: Results for Algorithm P8, second version,

Size of T = 1024.

APPENDIX C

Results with Test Sets of n=31, N = 1024.

This appendix contains the results of most of the algorithms presented in this thesis run on 10 sets of n = 31 bit vectors with N = 1024. Algorithm P1 was chosen for comparison. The others were chosen because they yielded the best for the other test sets.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	7689	38445	-6701	-21
2	7674	38370	-6626	-20
3	7691	38455	-6711	-21
4	7697	38485	-6741	-21
5	7676	38380	-6636	-20
6	7703	38515	-6771	-21
7	7690	38450	-6706	-21
8	7653	38265	-6521	-20
9	7697	38485	-6741	-21
10	7697	38485	-6741	-21
Average	7687	38435	-6691	-21

Table C1: Results for Algorithm P1 on n = 31, N = 1024.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	6822	27288	4456	15
2	6817	27268	4476	15
3	6809	27236	4508	15
4	6809	27236	4508	15
5	6820	27280	4464	15
6	6811	27244	4500	15
7	6807	27228	4516	15
8	6805	27220	4524	15
9	6815	27260	4484	15
10	6797	27188	4556	15
Average	6811	27244	4500	15

Table C2: Results for Algorithm P4 on n = 31, N = 1024.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	7699	30796	948	3
2	7730	30920	824	3
3	7733	30932	812	3
4	7723	30892	852	3
5	7701	30804	940	3
6	7724	30896	848	3
7	7725	30900	844	3
8	7716	30864	880	3
9	7733	30932	812	3
10	7737	30948	796	3
Average	7722	30888	856	3

Table C3: Results for Algorithm P5 on n = 31, N = 1024.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	6822	27288	4456	15
2	6817	27268	4476	15
3	6809	27236	4508	15
4	6809	27236	4508	15
5	6820	27280	4464	15
6	6811	27244	4500	15
7	6807	27228	4516	15
8	6805	27220	4524	15
9	6815	27260	4484	15
10	6798	27192	4552	15
Average	6811	27244	4500	15

Table C4: Results for Algorithm P6 on n = 31, N = 1024.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	7712	30848	896	3
2	7706	30824	920	3
3	7680	30720	1024	4
4	7721	30884	860	3
5	7691	30764	980	4
6	7700	30800	944	3
7	7694	30776	968	4
8	7681	30724	1020	4
9	7689	30756	988	4
10	7756	31024	720	3
Average	7703	30812	932	3

Table C5: Results for Algorithm P7 on n = 31, N = 1024.

Trial	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
1	7596	37980	−6236	−19
2	7617	38085	−6341	−19
3	7580	37900	−6156	−19
4	7584	37920	−6176	−19
5	7577	37885	−6141	−19
6	7614	38070	−6326	−19
7	7614	38070	−6326	−19
8	7592	37960	−6216	−19
9	7586	37930	−6186	−19
10	7604	38020	−6276	−19
Average	7596	37980	−6236	−19

Table C6: Results for Algorithm P8 on n = 31, second version and N=1024.

The poor results were probaly caused by the small fraction of the total set the test sets contain. One thousand and twenty four is less than one percent of the complete set of 2^{31} vectors.

APPENDIX D

PLI Programs Used.

```
P2:PROC OPTIONS(MAIN);
 DCL (NOB,NUV,SL,SR,BB,POS) FIXED BIN(31);
 DCL INPUT FILE INPUT;
 GET LIST(NUV,NOB,POS,BB);
 GET FILE(INPUT) LIST(NUV,NOB,POS,BB);
 SL=0;
 BEGIN;
 DCL 1 ORIG(NW),
      2 VECTOR          CHAR(NOB),
      2 USED            BIT(1),
     1 LVECTOR(NOB),
      2 LX CHAR(NOB),
     (DONE,OLD,BFLAG,INRANGE)BIT(1),
      KEY               CHAR(NOB),
      AN                CHAR(1),
      YOU               FILE INPUT,
      NUMBS             FILE STREAM OUTPUT ENV(F BLKSIZE(80)),
     (VCOUNT,LINK,COUNT,RANGE)    ENTRY,
      HEX               ENTRY RETURNS (CHAR(16)),
      ICOUNT            FIXED BIN(31),
      Z                 FIXED DEC(10,4),
     (I,J,KK)           FIXED BIN(31),
(PP,TCNT,LOGG,RN,MINCNT,
                MINPTR,MVCNT,CCNT,LPTR)FIXED BIN(31);
 LOGG=0;TCNT=1;MVCNT=0;ICOUNT=0;BFLAG='0'B;
 DO WHILE(2**LOGG<NOB);LOGG=LOGG+1;END;
 RN=2**(LOGG-BB);
 DO 1=1 TO NUV;
     GET LIST(PP);  USED(I)='0'B;
     VECTOR(I)=SUBSTR(HEX(PP,NOB),1,NOB);
 END;
 KEY-VECTOR(1);     USED(1)='1'B;          DONE='0'B;
 ICOUNT=ICOUNT+1;
 PUT FILE(NUMBS) LIST(1);
 KK=0;
 JX=-1;
 DO WHILE ("DONE);
   MVCNT=NOB+1;
   JX=JX+1;
```

```
      DO I =2 TO NUV;
        IF ^USED(I) THEN DO;
           CALL COUNT(KEY,VECTOR(I),NOB,CCNT);
           IF CCNT < MVCNT THEN DO;
              MVCNT=CCNT;MINPTR=I;
           END;
        END;
      END;
      PP=POS;
      CALL LINK(KEY,VECTOR(MINPTR),LVECTOR,NOB,PP,LPTR,RN);
      DO 1=1 TO LPTR-1;
         OLD='0'B;J=1;
         DO WHILE(J<=NUV & "OLD);
           IF ^USED(J) THEN DO;
               IF VECTOR(J)=LX(I) THEN DO;
                   USED(J)='1'B;OLD='1'B;
               END;
           END;
           IF "OLD THEN J=J+1;
         END;
         KK=MOD(KK+1,3);
         ICOUNT=ICOUNT+1;
         IF OLD THEN DO;
           PUT EDIT('''||LVECTOR(I)||'*'''||',')
                    (COL((1) *KK+1),A);
           PUT FILE(NUMBS) LIST(J) ;NOB+4
         END;
         ELSE
           PUT EDIT('''||LVECTOR(I)||''''||',')
          (COL((1) *KK+1),A);
      END;
      KEY=VECTOR(MINPTR);USED(MINPTR)='1'B;
      TCNT=TCNT+LPTR-1;
      DONE='1'B;
     DO I=1 TO NUV;IF ^USED(I) TN
                      DONE='0'B;END;
    END;
    PUT EDIT('OLD INFO',NUV,NOB,NOB*NUV)
    (COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),F(10));
    PUT EDIT('NEW INFO',TCNT,LOGG,TCNT*(LOGG-1))
    (COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),F(10));
   Z=(NOB*NUV-TCNT*(LOGG-1));
    Z=Z/(NOB*NUV)*100;
    PUT EDIT('SAVINGS',NOB*NUV-TCNT*(LOGG-1),Z)
      (COL(1),A,COL(10),F(7),COL(20),F(7.4),COL(30),F(10));
    END;
  END P2;
```

```
P3:PROC OPTIONS(MAIN);
DCL (NOB,NUV,SL,SR,BB,POS) FIXED BIN(31);
DCL INPUT FILE INPUT;
GET LIST(NUV,NOB,POS,BB)
GET FILE(INPUT) LIST(NUV,NOB,POS,BB);
SL=0;
BEGIN;
DCL 1 ORIG(NW),
     2 VECTOR          CHAR(NOB),
     2 USED            BIT(1),
    1 LVECTOR(NOB),
     2 LX CHAR(NOB),
    (DONE,OLD,BFLAG,INRANGE)BIT(1) ,
     KEY               CHAR(NOB),
     AN                CHAR(1),
     YOU               FILE INPUT,
     NUMBS             FILE STREAM OUTPUT ENV(F BLKSIZE(80)),
    (VCOUNT,LINK,COUNT       )    ENTRY,
     HEX               ENTRY RETURNS (CHAR(16)),
     ICOUNT            FIXED BIN(31),
     Z                 FIXED DEC(10,4),
    (I,J,KK)           FIXED BIN(31),
    (PP,TCNT,LOGG,RN,MINCNT,MINPTR,
     MVCNT,CCNT,LPTR) FIXED BIN(31);
LOGG=0;TCNT=1;MVCNT=0;ICOUNT=0;BFLAG='0'B;
DO WHILE(2**LOGG<NOB);LOGG=LOGG+1;END;
RN=2**(LOGG-BB);
DO I=1 TO NUV;
    GET LIST(PP);  USED(I)='0'B;
    VECTOR(I)=SUBSTR(HEX(PP,NOB),1,NOB);
END;
KEY=VECTOR(1);     USED(1)='1'B;          DONE='0'B;
ICOUNT=ICOUNT+1;
PUT FILE(NUMBS) LIST(1);
KK=0;
JX=-1;
DO WHILE ("DONE);
  INRANGE= '0'B;
  MVCNT=NOB+1;
  JX=JX+1;
  DO I =2 TO NUV;
    IF ^USED(I) THEN DO;
       CALL COUNT(KEY,VECTOR(I),NOB,CCNT);
       IF CCNT = 1 & ^INRANGE THEN DO;
           CALL RANGE(KEY,VECTOR(I),NOB,POS,INRANGE);
   IF INRANGE THEN DO; MVCNT=CCNT;MINPTR=I; END;
END;
       ELSE DO;
          IF CCNT < MVCNT THEN DO;
```

```
                MVCNT=CCNT;MINPTR=I;
            END;
          END;
        END;
      END;
      PP=POS;
      CALL LINK(KEY,VECTOR(MINPTR),LVECTOR,NOB,PP,LPTR,RN);
      DO 1=1 TO LPTR-1;
         OLD='0'B;J=1;
         DO WHILE(J<=NUV & "OLD);
           IF ^USED(J) THEN DO;
               IF VECTOR(J)=LX(I) THEN DO;
                   USED(J)='1'B;OLD='1'B;
               END;
           END;
           IF ^OLD THEN J=J+1;
         END;
         KK=MOD(KK+1,3);
         ICOUNT=ICOUNT+1;
         IF OLD THEN DO;
          PUT EDIT('''||LVECTOR(I)||'*'''||',')
                  (COL((1)*KK+1),);
           PUT FILE(NUMBS)LIST(J);NOB+4
         END;
         ELSE
           PUT EDIT('''||LVECTOR(I)||''''||',')
                  (COL((1)*KK+1),A);
      END
      KEY=VECTOR(MINPTR);USED(MINPTR)='1'B;
      TCNT=TCNT+LPTR-1;
      DONE='1'B;  DO 1=1 TO NUV;IF ^USED(I) THEN DONE='0'B;END;
    END;
    PUT EDIT('OLD INFO',NUV,NOB,NOB*NUV)
     (COL(1),A,COL(10),F(7),CL(20), F(7),COL(30),F(10));
    PUT EDIT('NEW INFO',TCNT,LOGG,TCNT*(LOGG-1))
       (COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),F(10));
  Z=(NOB*NUV-TCNT*(LOGG-1));
    Z=Z/(NOB*NUV)*100;
    PUT EDIT('SAVINGS',NOB*NUV-TCNT*(LOGG-1),Z)
      (COL(1),A,COL(10),F(7),COL(20),F(7.4),COL(30),F(10));
    END;
```

```
RANGE:PROC(A,B,N,P,F);
DCL (A,B) CHAR(*),
      F                 BIT(1),
     (N,P,P1,P2,I) FIXED BIN(31);
P1=P;
DO I=1 TO 15; P1=I+P; IF P1>N THEN P1=P1-N;
   IF SUBSTR(A,P1,1) ^= SUBSTR(B,P1,1) THEN P2=P1;
END;
IF P2<P THEN P2=P2+N;
IF P2-P > N/2 THEN F='1'B;
ELSE              F='0'B;
RETURN;
END RANGE;
END P3;
```

```
P4:PROC OPTIONS(MAIN);
DCL (NOB,NUV,SL,SR,BB,POS) FIXED BIN(31);
DCL INPUT FILE INPUT;
GET LIST(NUV,NOB,POS,BB);
GET FILE(INPUT)LIST(NUV,NOB,POS,BB);
BEGIN;
DCL 1 ORIG(NUV),
     2 VECTOR          CHAR(NOB),
     2 USED            BIT(1),
    1 LVECTOR(NOB),
     2 LX CHAR(NOB),
    (DONE,OLD,BFLAG,INRANGE)BIT(1),
     KEY               CHAR(NOB),
     AN                CHAR(1),
     YOU               FILE INPUT,
     NUMBS             FILE STREAM OUTPUT ENV(F BLKSIZE(80)),
    (VCOUNT,LINK,COUNT,RANGE)    ENTRY,
     ICOUNT            FIXED BIN(31),
     Z                 FIXED DEC(10,4),
    (I,J,KK) FIXED BIN(31),
    (PP,TCNT,LOGG,RN,MINCNT,
     MINPTR,MVCNT,CCNT,LPTR) FIXED BIN(31);
LOGG=0;TCNT=1;MVCNT=0;ICOUNT=0;
DO WHILE(2**LOGG<NOB);LOGG=LOGG+1;END;
RN=2**(LOGG-BB);
SR=RN;   SL=0;
DO 1=1 TO NUV; GET LIST(VECTOR(I));USED(I)='0'B; END;
KEY=VECTOR(1);     USED(1)='1'B;          DONE='0'B;
ICOUNT=ICOUNT+1;
 KK=0;
DO WHILE ("DONE);
  MINCNT=NOB+1;
  DO I =2 TO NUV;
    IF ^USED(I) THEN DO;
      CALL COUNT(KEY,VECTOR(I),NOB,CCNT);
      SELECT;
        WHEN(CCNT<MINCNT)DO;
            MINCNT=CCNT;MINPTR=I;
            CALL VCOUNT(KEY,VECTOR(I),NOB,MVCNT,POS,RN);
        END;
        WHEN(CCNT=MINCNT)DO;
            CALL VCOUNT(KEY,VECTOR(I),NOB,CCNT,POS,RN);
            IF CCNT< MVCNT THEN  DO;
               MVCNT=CCNT;MINPTR=I;
            END;
        END;
        OTHERWISE;
      END;
    END;
```

```
    END;
    PP=POS;
    IF SL>0 THEN DO;
      BACK:DO I=SL TO 1 BY -1;
           PP=POS-I;IF PP<=0 THEN PP=PP+NOB;
           IF SUBSTR(KEY,PP,1)^=SUBSTR(VECTOR(MINPTR),PP,1)
             THEN DO;
              BFLAG='1'B; LEAVE BACK;
           END;
      END BACK;
      IF ^BFLAG THEN  PP=POS;
    END;
    CALL LINK(KEY,VECTOR(MINPTR),LVECTOR,NOB,PP,LPTR,RN);
    POS=PP;
    DO I=1 TO LPTR-1;
       OLD='0'B;J=1;
       DO WHILE(J<=NUV & "OLD);
         IF ^USED(J) THEN DO;
             IF VECTOR(J)=LX(I) THEN DO;
                 USED(J)='1'B;OLD='1'B;
             END;
         END;
         IF "OLD THEN J=J+1;
       END;
       KK=MOD(KK+1,3);
       ICOUNT=ICOUNT+1;
       IF OLD THEN DO;
         PUT EDIT('''||LVECTOR(I)||'*'''||',')
                 (COL((NOB+4)*KK+1),A);
         PUT FILE(NUMBS) LIST(J);
       END;
       ELSE
         PUT EDIT('''||LVECTOR(I)||''''||',')
                 (COL((NOB+4)*KK+1),A);
    END;
    KEY=VECTOR(MINPTR);USED(MINPTR)='1'B;
    TCNT=TCNT+LPTR-1;
    DONE='1'B;DO 1=1 TO NUV;IF ^USED(I) THEN DONE='0'B;END;
  END;
  PUT EDIT('OLD INFO',NUV,NOB,NOB*NUV)
      (COL(1),A,COL(10),F(7),COL(0),f(7),COL(30),F(10));
  PUT EDIT('NEW INFO',ICOUNT,LOGG,ICOUNT*(LOGG-1))
          (COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),F(10));
  Z=(NOB*NUV-ICOUNT*(LOGG-1));
  Z=Z/(NOB*NUV)*100;
  PUT EDIT('SAVINGS',NOB*NUV-ICOUNT*(LOGG-1),Z)
     (COL(1),A,COL(10),F(7),COL(20),F(7.4),COL(30),F(10));
END;
END P4;
```

```
P4:PROC OPTIONS(MAIN);
 DCL (NOB,NUV,SL,SR,BB,POS) FIXED BIN(31);
 DCL INPUT FILE INPUT;
 GET LIST(NUV,NOB,POS,BB);
 GET FILE(INPUT) LIST(NUV,NOB,POS,BB);
 SL=0;
 BEGIN;
 DCL 1 ORIG(NW),
       2 VECTOR          CHAR(NOB),
       2 USED            BIT(1),
      1 LVECTOR(NOB),
       2 LX CHAR(NOB),
      (DONE,OLD,BFLAG,INRANGE)BIT(1),
       KEY               CHAR(NOB),
       AN                CHAR(1),
       YOU               FILE INPUT,
       NUMBS             FILE STREAM OUTPUT ENV
                         (F BLKSIZE(80)),
      (VCOUNT,LINK,COUNT,RANGE)    ENTRY,
       HEX               ENTRY RETURNS (CHAR(16)),
       ICOUNT            FIXED BIN(31),
       Z                 FIXED DEC(10,4),
      (I,J,KK)           FIXED BIN(31),
      (PP,TCNT,LOGG,RN,MINCNT,
       MINPTR,MVCNT,CCNT,LPTR) FIXED BIN(31);
 LOGG=0;TCNT=1;MVCNT=0;ICOUNT=0;BFLAG='0'B;
 DO WHILE(2**LOGG<NOB);LOGG=LOGG+1;END;
 RN=2**(LOGG-BB);
 DO 1=1 TO NW;
     GET LIST(PP);  USED(I)='0'B;
     VECTOR(I)=SUBSTR(HEX(PP,NOB),1,NOB);
 END;
 KEY=VECTOR(1);     USED(1)='1'B;          DONE='0'B;
 ICOUNT=ICOUNT+1;
 PUT FILE(NUMBS)LIST(1);
 KK=0;
 JX=-1;
 DO WHILE ("DONE);
   MVCNT=NOB+1;
   JX=JX+1;
   DO I =2 TO NUV;
     IF ^USED(I) THEN DO;
        CALL VCOUNT(KEY,VECTOR(I),NOB,CCNT,POS,RN);
        IF CCNT < MVCNT THEN DO;
           MVCNT=CCNT;MINPTR=I;
        END;
     END;
   END;
   PP=POS;
```

```
    END;
    CALL LINK(KEY,VECTOR(MINPTR),LVECTOR,NOB,PP,LPTR,RN);
    POS=PP;
    /* REPOSITION POS, DONE IN LLINK                    */
    /* PRINT VECTOR  ALL OF LINK AND VECTOR(MINPTR), MINPTR
      DO 1=1 TO LPTR-1;
       OLD='0'B;J=1;
       DO WHILE(J<=NUV & "OLD);
         IF ^USED(J) THEN DO;
             IF VECTOR(J)=LX(I) THEN DO;
                 USED(J)='1'B;OLD='1'B;
             END;
         END;
         IF ^OLD THEN J=J+1;
      END;
       KK=MOD(KK+1,3);
       ICOUNT=ICOUNT+1;
       IF OLD THEN DO;
         PUT EDIT('''||LVECTOR(I)||'*'''||',')(COL(1),A);
         PUT FILE(NUMBS)LIST(J);
       END;
       ELSE
         PUT EDIT('''||LVECTOR(I)||''''||',')(COL(1),A);
    END;
    KEY=VECTOR(MINPTR);USED(MINPTR)='1'B;
    TCNT=TCNT+LPTR-1;
     DONE='1'B;  DO I=1 TO NUV;IF ^USED(I) THEN DONE='0'B;-
END;    END;
  PUT EDIT('OLD INFO',NUV,NOB,NOB*NUV)(COL(1),A,COL(10),-
F(7),COL(20),
            F(7),COL(30),F(10));
   PUT EDIT('NEW INFO',ICOUNT,LOGG,ICOUNT*(LOGG-1))
              (COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),-
F(10));    Z=(NOB*NUV-ICOUNT*(LOGG-1));
   Z=Z/(NOB*NUV)*100;
   PUT EDIT('SAVINGS',NOB*NUV-ICOUNT*(LOGG-1),Z)
     (COL(1),A,COL(10),F(7),COL(20),F(7.4),COL(30),F(10));
   END;
END P5;
```

```
P6:PROC OPTIONS(MAIN);
DCL (NOB,NUV,SL,SR,BB,POS) FIXED BIN(31);
DCL INPUT FILE INPUT;
GET LIST(NUV,NOB,POS,BB);
GET FILE(INPUT)LIST(NW,NOB,POS,BB);
BEGIN;
DCL 1 ORIG(NUV),
     2 VECTOR          CHAR(NOB),
     2 USED            BIT(1),
    1 LVECTOR(NOB),
     2 LX CHAR(NOB),
    (DONE,OLD,BFLAG,INRANGE)BIT(1) ,
     KEY               CHAR(NOB),
     AN                CHAR(1),
     YOU               FILE INPUT,
     NUMBS             FILE STREAM OUTPUT ENV(F BLKSIZE(80)),
    (VCOUNT,LINK,COUNT,RANGE)    ENTRY,
     ICOUNT            FIXED BIN(31),
     Z                 FIXED DEC(10,4),
    (I,J,KK) FIXED BIN(31),
    (PP,TCNT,LOGG,RN,MINCNT,
     MINPTR,MVCNT,CCNT,LPTR) FIXED BIN(31);
LOGG=0;TCNT=1;MVCNT=0;ICOUNT=0;
DO WHILE(2**LOGG<NOB);LOGG=LOGG+1;END;
RN=2**(LOGG-BB);
SR=RN;   SL=0;
DO I=1 TO NUV;     GET LIST(VECTOR(I));  USED(I)='0'B;
END;
KEY-VECTOR(1);     USED(1)='1'B;          DONE='0'B;
ICOUNT=ICOUNT+1;
KK=0;
DO WHILE ("DONE);
  MINCNT=NOB+1;
  DO I =2 TO NUV;
      IF ^USED(I) THEN DO;
      CALL VCOUNT(KEY,VECTOR(I),NOB,MVCNT,POS,RN);
      SELECT;
        WHEN(MVCNT<MINCNT)DO;
           MINCNT=MVCNT;MINPTR=I;
           CALL COUNT(KEY,VECTOR(I),NOB,CCNT);
        END;
        WHEN(MVCNT=MINCNT)DO;
           CALL COUNT(KEY,VECTOR(I),NOB,MVCNT);
           IF MVCNT< CCNT THEN  DO;
              CCNT=MVCNT;MINPTR=I;
           END;
```

```
        END;
        OTHERWISE;
      END;
    END;
  END;
  PP=POS;
  CALL LINK(KEY,VECTOR(MINPTR),LVECTOR,NOB,PP,LPTR,RN);
  POS=PP;
  DO 1=1 TO LPTR-1;
     OLD='0'B;J=1;
     DO WHILE(J<=NUV & "OLD);
       IF ^USED(J) THEN DO;
           IF VECTOR(J)=LX(I) THEN DO;
               USED(J)='1'B;OLD='1'B;
           END;
       END;
       IF "OLD THEN J=J+1;
     END;
     KK=MOD(KK+1,3);
     ICOUNT=ICOUNT+1;
     IF OLD THEN'DO;
       PUT EDIT('''||LVECTOR(I)||'*'''||',')
               (COL((NOB+4)*KK+A);
       PUT FILE(NUMBS)LIST(J);
     END;
     ELSE
       PUT EDIT('''||LVECTOR(I)||''''||',')
                (COL((NOB+4)*KK+1),A);
  END;
  KEY=VECTOR(MINPTR);USED(MINPTR)='1'B;
  TCNT=TCNT+LPTR-1;
  DONE='1'B;DO 1=1 TO NUV;IF ^USED(I) THEN DONE='0'B;END;
END;
PUT EDIT('OLD INFO',NUV,NOB,NOB*NUV)
       (COL(1),A,COL(10),F(7)COL20),F(7),COL(30),F(10));
PUT EDIT('NEW INFO',ICOUNT,LOGG,ICOUNT*(LOGG-1))
        (COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),F(10));
Z=(NOB*NUV-ICOUNT*(LOGG-1));
Z=Z/(NOB*NUV)*100;
PUT EDIT('SAVINGS',NOB*NUV-ICOUNT*(LOGG-1),Z)
  (COL(1),A,COL(10),F(7),COL(20),F(7.4),COL(30),F(10));
END;
END P6;
```

```
 P7:PROC OPTIONS(MAIN);
DCL (NOB,NUV,SL,SR,BB,POS) FIXED BIN(31);
DCL INPUT FILE INPUT;
GET LIST(NUV,NOB,POS,BB);
GET FILE(INPUT) LIST(NUV,NOB,POS,BB);
SL=0;
BEGIN;
DCL 1 ORIG(NUV),
     2 VECTOR          CHAR(NOB),
     2 USED            BIT(1),
    1 LVECTOR(NOB),
     2 LX CHAR(NOB),
    (DONE,OLD,BFLAG,INRANGE)BIT(1),
     KEY               CHAR(NOB),
     AN                CHAR(1),
     YOU               FILE INPUT,
     NUMBS             FILE STREAM OUTPUT ENV(F BLKSIZE(80)),
    (VCOUNT,LINK,COUNT,RANGE,WEIGHT)    ENTRY,
     HEX               ENTRY RETURNS (CHAR(31)),
     ICOUNT            FIXED BIN(31),
     Z                 FIXED DEC(10,4),
    (I,J,KK,ONECNT(NOB),VLEFT,WMINCNT,WCNT) FIXED BIN(31),
   (PP,TCNT,LOGG,RN,MINCNT,
    MINPTR,MVCNT,CCNT,LPTR) FIXED BIN(31);
    LOGG=0;TCNT=1;MVCNT=0;ICOUNT=0;BFLAG='0'B;
DO WHILE(2**LOGG<NOB);LOGG=LOGG+1;END;
RN=2**(LOGG-BB);
DO I=1 TO NOB; ONECNT(I)=0; END;
DO 1=1 TO NUV;
    GET LIST(PP);  USED(I)='0'B;
    VECTOR(I)=SUBSTR(HEX(PP,NOB),1,NOB);
    DO J=1 TO NOB;
       IF SUBSTR(VECTOR(I),J,1)='1' THEN
           ONECNT(J)=ONECNT(J)+1;
    END;
END;
KEY=VECTOR(1);     USED(1)='1'B;          DONE='0'B;
ICOUNT=ICOUNT+1;
PUT FILE(NUMBS) LIST(1);
KK=0; VLEFT=NUV;
DO WHILE ("DONE);
  VLEFT=VLEFT-1;
  MVCNT=NOB*2+1;
  JX=JX+1;
  DO I =2 TO NUV;
    IF ^USED(I) THEN DO;
       CALL VCOUNT(KEY,VECTOR(I),NOB,CCNT,POS,RN);
       SELECT;
          WHEN( CCNT < MVCNT ) DO;
```

```
            MVCNT=CCNT;MINPTR=I;
            CALL WEIGHT(ONECNT,VECTOR(I),WMINCNT,VLEFT);
          END;
          WHEN(CCNT=MVCNT) DO;
            CALL WEIGHT(ONECNT,VECTOR(I),WCNT,VLEFT);
             IF (WCNT > WMINCNT) THEN DO;
                MVCNT=CCNT;MINPTR=I;WMINCNT=WCNT;
            END;
          END;
          OTHERWISE;
        END;
      END;
    END;
    PP=POS;
    CALL LINK(KEY,VECTOR(MINPTR),LVECTOR,NOB,PP,LPTR,RN);
    POS=PP;
    DO 1=1 TO LPTR-1;
       OLD='0'B;J=1;
       DO WHILE(J<=NUV & "OLD);
         IF ^USED(J) THEN DO;
             IF VECTOR(J)=LX(I) THEN DO;
                 USED(J)='1'B;OLD='1'B;
             END;
         END;
         IF "OLD THEN J=J+1;
       END;
       KK=MOD(KK+1,3);
       ICOUNT=ICOUNT+1;
       IF OLD THEN DO;
         PUT EDIT('''||LVECTOR(I)||'*'''||',')
                 (COL((1) *KK+1A);
         PUT FILE(NUMBS) LIST(J) ;NOB+4
       END;
       ELSE
         PUT EDIT('''||LVECTOR(I)||''''||',')
                 (COL((1)*KK+1)
    END;
    KEY=VECTOR(MINPTR);USED(MINPTR)='1'B;
    TCNT=TCNT+LPTR-1;
    DONE='1'B;  DO 1=1 TO NUV;IF ^USED(I) THEN DONE='0'B;END;
  END;
  PUT EDIT('OLD INFO',NUV,NOB,NOB*NUV)
          (COL(1),A,COL(10),F(7)OL(20),F(7),COL(30),F(10));
  PUT EDIT('NEW INFO',ICOUNT,LOGG,ICOUNT*(LOGG-1))
          (COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),F(10));
  Z=(NOB*NUV-ICOUNT*(LOGG-1));
  Z=Z/(NOB*NUV)*100;
  PUT EDIT('SAVINGS',NOB*NUV-ICOUNT*(LOGG-1),Z)
    (COL(1),A,COL(10),F(7),COL(20),F(7.4),COL(30),F(10));
  END;
  END P7;
```

```
 P8:PROC OPTIONS(MAIN);
DCL (NOB,NUV,SL,SR,BB,POS) FIXED BIN(31),
     INPUT              FILE INPUT;
GET LIST(NUV,NOB,POS,BB);
GET FILE(INPUT) LIST(NUV,NOB,POS,BB);
 NOB=NOB-1;
SL=0;
BEGIN;
DCL 1 ORIG(NW),
     2 FLAG             CHAR(1), 2 VECTOR        CHAR(NOB),
     2 USED             BIT(1),
     VX                 CHAR(NOB+1),
    YOU FILE INPUT,
    1 LVECTOR(NOB),   2 LX CHAR(NOB),
     FFLAG              CHAR(1),
    (DONE,OLD,BFLAG,HDONE,INRANGE)BIT(1),
     KEY                CHAR(NOB),
     NUMBS              FILE STREAM OUTPUT ENV(F BLKSIZE(80)),
    (VCOUNT,LINK,COUNT,RANGE,WEIGHT)    ENTRY,
     HEX                ENTRY RETURNS (CHAR(31)),
     ICOUNT             FIXED BIN(31),
     Z                  FIXED DEC(10,4),
     TC CHAR(1),
    (RNI,J,KK,ONECNT(NOB),VLEFT,WMINCNT,WCNT) FIXED
BIN(31),
    (FC,POS                                   ) FIXED
BIN(31),
    (PP,TCNT,LOGG,RN,MINCNT,
       MINPTR,MVCNT,CCNT,LPTR) FIXED BIN(31);
CCNT=1;MVCNT=0;ICOUNT=0;BFLAG='0'B;HDONE='0'B;LOGG=0;FC=0;
DO WHILE(2**LOGG<(NOB+1));LOGG=LOGG+1;END;
RN=2**(LOGG-BB);
DO 1=1 TO NOB; ONECNT(I)=0; END;
DO 1=1 TO NUV;
    GET LIST(PP); USED(I)='0'B;
    VX=SUBSTR(HEX(PP,NOB+1),1,NOB+1);FLAG(I)=
                     SUBSTR(VX,NOB+11);
/*PUT EDIT(I,VX)(COL(1),F(3),COL(5),A); */
    VECTOR(I)=SUBSTR(VX,1,NOB);
    IF FLAG(I)='1' THEN FC=FC+1;
    DO J=2 TO NOB;
     IF SUBSTR(VX,J,1)='1' THEN ONECNT(J-1)=ONECNT(J-1)+1;
    END;
END;
POS=0;MAX=FC;DO 1=1 TO NOB;
  IF ONECNT(I) > MAX THEN DO; MAX=ONECNT(I); POS=I;END;
  END;
/*DISPLAY('MOST COMMON?');*/
RNI=0;
IF POS ^= 0 & RNI=1 THEN DO;
```

```
      PUT LIST(FC,ONECNT(POS));
      SELECT(POS);
         WHEN (1) DO;
           DO I=1 TO NUV; TC=FLAG(I); FLAG(I)=
                        SUBSTR(VECTOR(I)1,1);
                SUBSTR(VECTOR(I),1,1)=TC;END;
           ONECNT(1)=FC;
         END;
         WHEN (NOB) DO;
           DO I=1 TO NUV; TC=FLAG(I); FLAG(I)=
                      SUBSTR(VECTOR(I)NOB,1);
                SUBSTR(VECTOR(I),NOB,1)=TC;END;
           ONECNT(NOB)=FC;
         END;
         OTHERWISE  DO;
           DO I=1 TO NUV; TC=FLAG(I); FLAG(I)=
                      SUBSTR(VECTOR(I)POS,1);
             VECTOR(I)=FLAG(I)||SUBSTR(VECTOR(I),1,POS-1)||
                                SUBSTR(VECTOR(I),POS+1);END;
           DO I= POS TO 2 BY -1; ONECNT(I)=ONECNT(I-1);END;
           ONECNT(1)=FC;
         END;
   END;
END;
KEY=VECTOR(1);  USED(1)='1'B;    DONE='0'B; FFLAG=FLAG(1);
/* PUT EDIT(1,KEY)(COL(1),F(3),COL(5),A);*/
ICOUNT=ICOUNT+1;
/*PUT FILE(NUMBS)LIST(1);*/
KK=0; VLEFT=NUV;
DO WHILE (^DONE);
  VLEFT=VLEFT-1;
  MVCNT=NOB*2+1;
  JX=JX+1;
  DO I =2 TO NUV;
    IF ^USED(I)  & FLAG(I)=FFLAG THEN DO;
       CALL VCOUNT(KEY,VECTOR(I),NOB,CCNT,POS,RN);
       SELECT;
          WHEN( CCNT < MVCNT ) DO;
            MVCNT=CCNT;MINPTR=I;
            CALL COUNT(KEY,VECTOR(I),NOB,WMINCNT);
          END;
          WHEN(CCNT=MVCNT) DO;
            CALL COUNT(KEY,VECTOR(I),NOB,WCNT);
            IF (WCNT < WMINCNT) THEN DO;
               MVCNT=CCNT;MINPTR=I;WMINCNT=WCNT;
            END;
          END;
          OTHERWISE;
       END;
    END;
```

```
  END;
  PP=POS;
  IF SL>0 THEN DO;
    BACK:DO I=SL TO 1 BY -1;
         PP=POS-I;IF PP<=0 THEN PP=PP+NOB;
         IF SUBSTR(KEY,PP,1)^=
           SUBSTR(VECTOR(MINPTR),PP,1) THEN DO;
            BFLAG='1'B; LEAVE BACK;
         END;
    END BACK;
    IF ^BFLAG THEN  PP=POS;
  END;
  CALL LINK(KEY,VECTOR(MINPTR),LVECTOR,NOB,PP,LPTR,RN);
  POS=PP;
  DO 1=1 TO LPTR-1;
     OLD='0'B;J=1;
     DO WHILE(J<=NUV & "OLD);
       IF ^USED(J) THEN DO;
           IF VECTOR(J)=LX(I) THEN DO;
               USED(J)='1'B;OLD='1'B;
           END;
       END;
       IF "OLD THEN J=J+1;
     END;
     KK=MOD(KK+1,3);
     ICOUNT=ICOUNT+1;
     IF OLD THEN DO;
       PUT EDIT(MINPTR)(COL(1),F(5));
       PUT EDIT('''''||LVECTOR(I)||'*'''||',')(COL(1),A);
       PUT FILE(NUMBS)LIST(J);NOB+4KK+1,
     END;
     ELSE
       PUT EDIT('''''||LVECTOR(I)||''''||',')(COL(1),A);
     END;
  KEY=VECTOR(MINPTR);USED(MINPTR)='1'B;
  TCNT=TCNT+LPTR-1;
  HDONE='1'B;
  DONE='1'B;  DO 1=1 TO NW;
     IF FLAG(I)=FFLAG & ^USED(I) THEN HDONE='0'B; --
     IF ^USED(I) THEN DONE='0'B;END;
  IF HDONE THEN DO;IF FFLAG='0' THEN FFLAG='1';
  ELSE FFLAG='0'; END;
END;
PUT EDIT('OLD INFO',NUV,(NOB+1),(NOB+1)*NUV)
(COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),F(10));
PUT EDIT('NEW INFO',ICOUNT,LOGG,ICOUNT*(LOGG-1))
        (COL(1),A,COL(10),F(7),COL(20),F(7),COL(30),F(10));
Z=((NOB+1)*NUV-ICOUNT*(LOGG-1));
Z=Z/((NOB+1)*NUV)*100;
```

```
 PUT EDIT('SAVINGS',(NOB+1)*NUV-ICOUNT*(LOGG-1),Z)
   (COL(1),A,COL(10),F(7),COL(20),F(7.4),COL(30),F(10));
  END;
 END P8;


LINK:PROC (A1,A2,AA,LN,POS,LPTR,RN);
DCL (I,KK,LN,LPTR,PPOS,POS,LAST,CNT,NNUM,NUM,RN) FIXED
     BIN(31),
    (A1,A2)   CHAR(*),
    1  AA(*),
       2 AVECTOR CHAR(*);
  PPOS=POS;
  LPTR=1;
 I=POS;
 DO WHILE(A1^=A2);
   IF SUBSTR(A1,I,1)=SUBSTR(A2,I,1) THEN DO;
     KK=MOD(PPOS+RN,LN);
     IF KK>LN THEN KK=KK-LN;
     KK=KK-1;
     IF KK<1  THEN KK=KK+LN;
     IF (I=KK)THEN DO;
       IF SUBSTR(A2,I,1)='1' THEN
          AVECTOR(LPTR)=SUBSTR(A1,1,I-1)||'0'||
                        SUBSTR(A1,I+1);
       ELSE
          AVECTOR(LPTR)=SUBSTR(A1,1,I-1)||'1'||
                        SUBSTR(A1,I+1);
       LPTR=LPTR+1;
       AVECTOR(LPTR)=A1;
       LPTR=LPTR+1;
       PPOS=I;I=I-1;
     END;
    END;
    ELSE  DO;
        AVECTOR(LPTR)=SUBSTR(A1,1,I-1)||SUBSTR(A2,I,1)||
                                        SUBSTR(A1,I+1);
         LPTR=LPTR+1;
       CNT=CNT+1;
       PPOS=I;
    END;
    A1=SUBSTR(A1,1,I-1)||SUBSTR(A2,I,1)||SUBSTR(A1,I+1);
    I=MOD(I+1,LN);
    IF 1=0 THEN I=LN;
  END;
POS=PPOS;
RETURN;
END LINK;
```

```
RRAN:PROC OPTIONS(MAIN);
DCL (IC,NOB,NUV,I,J,K)        FIXED BIN(31),
     FOUND              BIT(1),
    (HEX, CHECKR)        ENTRY,
     RANDOM             ENTRY RETURNS (FIXED BIN(31));
/* NOB=NUMBER OF BITS
   NUV=NUMBER OF VECTORS  */
I=1;
GET LIST (NOB,NUV,X);
PUT EDIT (NUV,NOB,I,I)
 (COL(1),F(4),COL(6),F(3),COL(10),F(1),COL(12),F(1));
BEGIN;
DCL AA(NUV) CHAR(NOB),
    BB(NUV) FIXED BIN(31);
DO 1=1 TO NUV;
   SELECT;
    WHEN(MOD(I,2)=0)DO;
     X=X+I;
     X=LOG(X);
     J=X;
     X=X-J;
     J=X*10**(NOB/3+.5);
   END;
   OTHERWISE DO;
     X=X+I;
     X=LOG(X);
     J=X;
     X=X-J;
     J=X*10**(NOB/3+.5);
   END;
   END;
   FOUND='1'B;
   IC=0;
   DO WHILE(FOUND);
     J=MOD(J,2**NOB);
     FOUND='0'B;
     DO K=1 TO I-1;
        IF BB(K)=J THEN FOUND='1'B;
     END;
 /*        CALL CHECK(FOUND,J,BB,I-1);*/
     IF FOUND THEN  DO;
         X=1.198762*X;
         X=COS(X);
         J=X;X=X-J;J=X*10**(NOB/3+.9);
     END;
     IF IC>2000 THEN DO;
        PUT LIST('INFINITE LOOP',J,IC,RANDOM(J,I,NOB));
        STOP;
     END;
```

```
      IC=IC+1;
    END;
    BB(I)=J;
/*  CALL HEX(J,AA(I),NOB);
    AA(I)=SUBSTR(AA(I),3)||SUBSTR(AA(I),1,2);
    KK=MOD((NOB+3)*MOD((I-1),3),80);
    IF 80-KK<NOB+3 THEN KK=0;
    PUT EDIT('''||AA(I)||'''||',')(COL(KK+1),A);*/
    KK=MOD((11 )*MOD( (I-1), 7),80);
    IF 80-KK<11 THEN KK=0;
    PUT EDIT(BB(I),',')(COL(KK+1),F(10,0),COL(KK+11),A);
 END;
 END;
 END RRAN;


COUNT:PROC(A1,A2,NOB,NOC);
DCL (A1,A2) CHAR(*) ,(NOC,NOB,II) FIXED BIN(31);
NOC=0;
DO II=1 TO NOB;
  IF SUBSTR(A1,II,1) ^= SUBSTR(A2,II,1) THEN  NOC=NOC+1;
END;
RETURN;END COUNT;


WEIGHT:PROC(COUNTS,VECTOR,VALUE,VLEFT);
 DCL (COUNTS(*),VALUE,VLEFT,I,L) FIXED BIN(31),
      (FCOUNT,FLEFT) FIXED DEC(8,3),
      VECTOR CHAR(*);
 IF VLEFT=0 THEN DO; PUT DATA(VLEFT,COUNTS); STOP; END;
 L=LENGTH(VECTOR); VALUE=0;
 FLEFT=VLEFT;
 DO I=1 TO L;
    FCOUNT=COUNTS(I);
    IF SUBSTR(VECTOR,I,1)='1' THEN
       IF (FCOUNT/FLEFT) > .5 THEN VALUE=VALUE+1;
    ELSE
       IF (FCOUNT/FLEFT) < .5 THEN VALUE=VALUE+1;  --
 END;
 END WEIGHT;
```

```
HEX:PROC(J,NOB) RETURNS (CHAR(31));
DCL (I,J,K,NOB) FIXED BIN (31);
K=J;
BEGIN;
DCL A   CHAR(NOB) VARYING;
A='';
DO I=NOB-1 TO 0 BY -1;
 IF 2**I <=K THEN DO;
   K=K-2**I;
   A=A||'1';
 END;
 ELSE
   A=A||'0';
END;
IF K^=0 THEN DO;
  PUT LIST ('ERROR IN HEX');
  PUT DATA(A,J);
  STOP;
END;
RETURN(A);
END;
END HEX;
```

```
HVAL:PROC(A,N)RETURNS (FIXED BIN(31));
DCL A(*) CHAR(1), (I,J,K,N,SUM) FIXED BIN(31);
SUM=0;
DO 1=0 TO N-1;
  IF A(I+1)='1' THEN SUM=SUM+2**(N-1-I);
END;
RETURN(SUM);
END HVAL;
```

```
VCOUNT:PROC(KEY,VECTOR,NOB,CCNT,POS,RN);
DCL (KEY,VECTOR) CHAR(*),
    A1 CHAR(1),
    YOU FILE INPUT,
    (LAST,NOB,I,J,KK,JJ,POS,PPOS,RN,CCNT) FIXED BIN(31);
PPOS=POS; LAST=POS;CCNT=0;
DO 1=0 TO NOB-1;
   JJ=PPOS+I;
   IF JJ>NOB THEN JJ=JJ-NOB;
   KK=JJ-LAST;
   IF KK<0   THEN KK=KK+NOB;
   IF SUBSTR(KEY,JJ,1) ^=SUBSTR(VECTOR,JJ,1) THEN DO;
```

```
      SELECT;
        WHEN(KK<RN) CCNT=CCNT+1;
        WHEN(KK<2*RN-1 & KK >=RN) CCNT=CCNT+2;
        OTHERWISE DO;
         CCNT=CCNT+4;
        END;
      END;
      LAST=JJ;
    END;
END;
/*DISPLAY('IN VCOUNT');
GET FILE(YOU) EDIT(A1)(COL(1),A(1));
SELECT(A1);
  WHEN('S') STOP;
  WHEN('D') PUT(DATA);
  OTHERWISE;
END;*/
RETURN;
END VCOUNT;
```

APPENDIX E

Results with Actual Test Sets.

The following tables are results of Algorithms P1-P8 on actual test sets. Algorithms P1 - P7 are compared in the first two tables to two test sets. The first test set has 27 vectors while the second test set has 46 vectors. Algorithm P8 is then applied to the Prestored-T Prestored-0 in the third table.

Program	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
P1	79	316	62	17
P2	95	285	93	25
P3	106	318	60	16
P4	93	279	99	27
P5	101	303	75	20
P6	93	279	99	27
P7	94	282	96	26

Table El: Algorithms P1-P7 applied to an actual Test set, N = 27, n = 14.

Program	Size of T_2	New ROM SIZE	Bits Saved	Savings(%)
P1	124	496	148	23
P2	152	456	188	30
P3	162	486	158	25
P4	162	486	158	25
P5	148	444	200	32
P6	150	450	194	31
P7	144	432	212	33

Table E2: Algorithms P1-P7 applied to an actual Test set, N = 46, n = 14.

size of T	Size of T_2	New ROM SIZE	Bits saved	Savings(%)
27	53	265	15	6
46	142	710	230	25

Table E3: Algorithms P8 applied to an actual test set and valid output, N = 46, n = 20.

References

[1] R. Dandapani, J. H. Patel and J. A. Abraham, "Design of Test Pattern Generators for Built In Test", Digest, 1984 IEEE Test Conference, pp. 315-319, October 1984.

[2] M. N. S. Swamy and K. Thukasiraman, Graphs, Networks, and Alsorithms, (New York : John Wiley and Sons, 1984), PP. 291-293.

[3] Sara Baase, Computer Alsorithms. Introduction to Desisn and Analysis, (Mass.: Addison-Wesley Publishing Company, Jan. 1963), pp. 255-263.