CSE 430: Operating Systems

Instructor: Dr. Violet R. Syrotiuk

Covering Arrays

Design and Analysis Report

Project 1

Author: Ankit Rathi

ASU ID: 1207543476

Contents

Introduction

What are Covering Arrays?

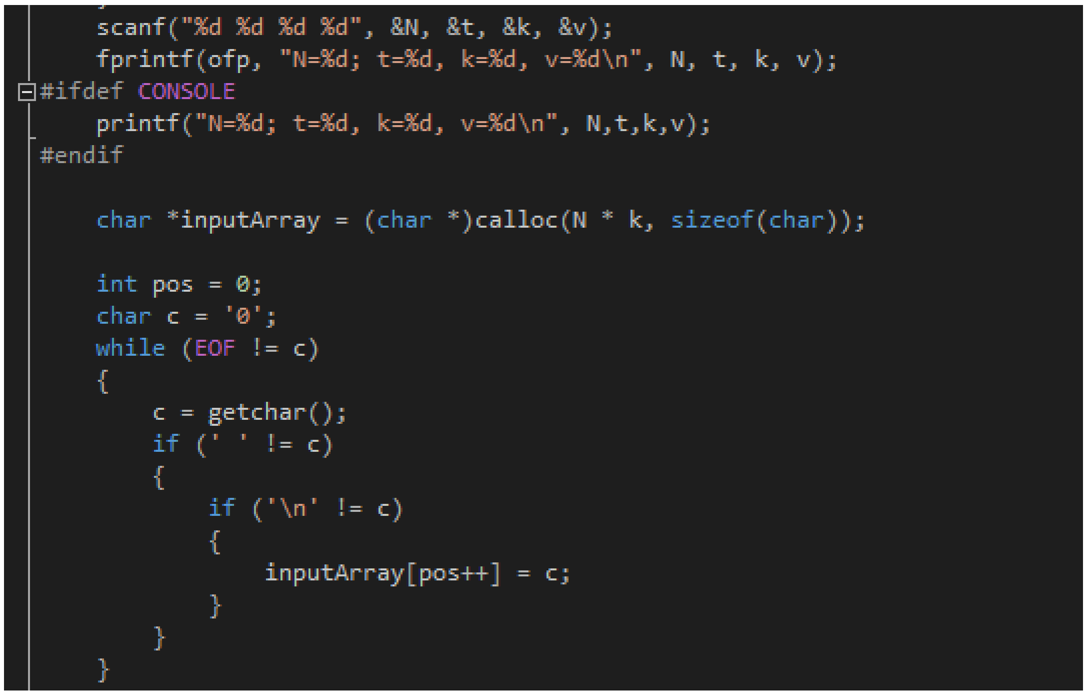
Covering arrays are used in testing software, hardware, composite materials, biological networks, and others. They also form the basis for combinatorial methods to learn an unknown classification function using few evaluations - these arise in computational learning and classification, and hinge on locating the relevant attributes.

A covering array CA(N; t; k; v) is an N x k array where each N x t subarray contains all ordered t-sets on v symbols at least once; t is called the strength of the covering array.

The first milestone of this project is to design and analyze the C/C++ program to check if the given input file is Covering Array or not. If it is Covering Array, the Don't Care Matrix along with the position or the (r,c) co-ordinates of the don't care terms. I have developed the C program for detecting the Covering Array.

Reading the Input file

The program has been designed to read data from the "stdin". To read the data from the "stdin", we have to put the data onto stdin while executing the program. This can be done with the command like "p1_AnkitRathi.o < 01-CA(6;2,5,2)". As per the format of the input line the first line will be in the form of N t k v, each separated by a space. After reading the first line, we have to read the next N lines, each line containing k elements. Each lines has a single space between two elements. This space is eliminated by the program, so even if there are more number of spaces then as expected, the program will still sun as it bypasses any space it encounters. After each line it reads, it ecnounters a new line character indicating it is now beginning to read the next line. This mentioned logic has been implemented in the code as shown below. The program outputs the results onto the console and also generates a output file in the same location as from where it is run.



```
    scanf("%d %d %d %d", &N, &t, &k, &v);
    fprintf(ofp, "N=%d; t=%d, k=%d, v=%d\n", N, t, k, v);
#ifdef CONSOLE
    printf("N=%d; t=%d, k=%d, v=%d\n", N,t,k,v);
#endif

    char *inputArray = (char *)calloc(N * k, sizeof(char));

    int pos = 0;
    char c = '0';
    while (EOF != c)
    {
        c = getchar();
        if (' ' != c)
        {
            if ('\n' != c)
            {
                inputArray[pos++] = c;
            }
        }
    }
```

Brief Intro about Data Structures Used

Variable Name	**Data Type**	**Size**	**Code**	**Use**
inputArray	Arrays of chars	N * k	char *inputArray = (char *)calloc(N * k, sizeof(char));	Stores the input array read.
elements	Arrays of chars	v	char *elements = (char *)calloc(v, sizeof(char));	Stores the elements needed to generate the t-sets
Symbols	Arrays of chars	(V^t)*t	char *Symbols = (char *)calloc(combinationOfSymbols * t, sizeof(char));	Combination of symbols
SubArrays	Arrays of int	(k choose t) * t	int *SubArrays = (int *)calloc(noOfSubArrays * t, sizeof(int));	This contains the subArrays geenrated
intermediateArray	Arrays of int	noOfSubArrays * combinationOfSymbols	int *intermediateArray = (int *)calloc(noOfSubArrays * combinationOfSymbols, sizeof(int));	This contains the intermediate matrix. To check for the coverage conditions.

finalArray	Arrays of chars	N * k	char *finalArray = (char *)calloc(N*k, sizeof(char));	This array is used to store the final don't care matrix. It is of type char to accommodate * storing as well.

Selection of Data Structures for generating subarrays

In order to verify that the array is a covering array, we must check that every N x t subarray contains all ordered t-sets on v symbols. There are (k choose t) subarrys to check. Recall that (k choose t) = k! / (t! * (k-t)!). Thus, we must check that each of the (k choose t) subarrays contains (or covers) all the ordered t-sets on v.

```
int noOfSubArrays = combination(k, t);
```

```
int *SubArrays = (int *)calloc(noOfSubArrays * t, sizeof(int));
```

Total number of subArrays is (k choose t) which is obtained by calling the function combination() method. Amount of memory needed to store the subarray is depended on total number of subArrays * t. The datatype I have used is int, it can also be changed to unsigned int.

The logic to generate the SubArrays is by the alogorithm as shown below,

```
       if (2 == t)
       {
              int iCount = 0;
              for (iLoopIndex = 1; iLoopIndex <= k; iLoopIndex++)
              {
                     for (jLoopIndex = iLoopIndex + 1; jLoopIndex <= k; jLoopIndex++)
                     {
                            for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
                            {
                                   if (0 == (iCount % t))
                                   {
                                          SubArrays[iCount] = iLoopIndex;
                                   }
                                   else
                                   {
                                          SubArrays[iCount] = jLoopIndex;
                                   }
                                   iCount++;
                            }
                     }
              }
       }
       else if (3 == t)
       {
              int iCount = 0;
              for (iLoopIndex = 1; iLoopIndex <= k; iLoopIndex++)
              {
                     for (jLoopIndex = iLoopIndex + 1; jLoopIndex <= k; jLoopIndex++)
                     {
                            for (kLoopIndex = jLoopIndex + 1; kLoopIndex <= k;
kLoopIndex++)
                            {
```

```
                                        for (lLoopIndex = 0; lLoopIndex < t; lLoopIndex++)
                                        {
                                                if (0 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = iLoopIndex;
                                                }
                                                else if (1 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = jLoopIndex;
                                                }
                                                else if (2 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = kLoopIndex;
                                                }
                                                iCount++;
                                        }
                                }
                        }
                }
        }
```

For t=2 and for t=3, there is separate if conditions to generate the subArrays.

Selection of Data Structures for generating t-sets

Total number of t-sets depends upon the v and t. In general, there are v^t ordered t-sets to cover. So we have to allocate v^t amount of memory to store all the t-sets. Number of elements that will be used to generate t-sets is depended on the total number of symbols available, v.

```
        char *elements = (char *)calloc(v, sizeof(char));
        char *Symbols = (char *)calloc(combinationOfSymbols * t, sizeof(char));
```

The code to generate the ordered t-sets is given below. From the code it can be seen that, the code generation logic goes based on t value. Since I am using a single dimensional array to store the value, for t=2 and v=2, I have used 3 for loops running based on v, v and t. for loop controlled by t, is used to alternate the ordering of 2 D t-sets into single dimension.

```
if (2 == v)
        {
                int iCount = 0;
                elements[0] = '0';
                elements[1] = '1';
                if (2 == t)
                {
                        for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
                                {
                                        for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
                                        {
```

```
						if (0 == (kLoopIndex % t))
						{
							Symbols[iCount] = elements[iLoopIndex];
						}
						else
						{
							Symbols[iCount] = elements[jLoopIndex];
						}
						iCount++;
					}
				}
			}
		}
		else if (3 == t)
		{
			elements[0] = '0';
			elements[1] = '1';
			elements[2] = '2';
			for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
			{
				for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
				{
					for (kLoopIndex = 0; kLoopIndex < v; kLoopIndex++)
					{
						for (lLoopIndex = 0; lLoopIndex < t;
lLoopIndex++)
						{
							if (0 == (lLoopIndex % t))
							{
								Symbols[iCount] =
elements[iLoopIndex];
							}
							else if (1 == (lLoopIndex % t))
							{
								Symbols[iCount] =
elements[jLoopIndex];
							}
							else if (2 == (lLoopIndex % t))
							{
								Symbols[iCount] =
elements[kLoopIndex];
							}
							iCount++;
						}
					}
				}
			}
		}
	}
	else if (3 == v)
	{
		int iCount = 0;
		elements[0] = '0';
		elements[1] = '1';
		elements[2] = '2';
		if (2 == t)
		{
			for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
```

```
                        {
                                for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
                                {
                                        for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
                                        {
                                                if (0 == (kLoopIndex % t))
                                                {
                                                        Symbols[iCount] = elements[iLoopIndex];
                                                }
                                                else
                                                {
                                                        Symbols[iCount] = elements[jLoopIndex];
                                                }
                                                iCount++;
                                        }
                                }
                        }
                }
                else if (3 == t)
                {
                        for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
                                {
                                        for (kLoopIndex = 0; kLoopIndex < v; kLoopIndex++)
                                        {
                                                for (lLoopIndex = 0; lLoopIndex < t; 
lLoopIndex++)
                                                {
                                                        if (0 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = 
elements[iLoopIndex];
                                                        }
                                                        else if (1 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = 
elements[jLoopIndex];
                                                        }
                                                        else if (2 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = 
elements[kLoopIndex];
                                                        }
                                                        iCount++;
                                                }
                                        }
                                }
                        }
                }
        }
```

Selection of Data Structures to track coverage conditions

Once the t-ordered sets and the subarrays are generated, the next step is to keep track of the coverage conditions. Cosider the CA(6;2,5,2) covering arrays shown below, for this, there are 10 subArrays to check. We have to check that each of the 10, 6 x 2 subarrays contains all ordered t-sets on v. Specifically for 6 x 2 subarray defined by pair of columns in

{(1; 2); (1; 3); (1; 4); (1; 5); (2; 3); (2; 4); (2; 5); (3; 4); (3; 5); (4; 5)}.

We have to check that all the 2-sets on v are covered. The orders 2-sets on v are {(0; 0); (0; 1); (1; 0); (1; 1)}. In general, there are v^t ordered t-sets to cover.

		Columns				
		1	2	3	4	5
Rows	1	0	1	1	1	1
	2	1	0	1	0	0
	3	0	1	0	0	0
	4	1	0	0	1	1
	5	0	0	0	0	1
	6	1	1	0	1	0

Table 1: A $CA(6; 2, 5, 2)$ covering array.

First we have to select 6 x 2 (N x t) subarray. In this selected SubArray, we have to check if all the t-sets are present. By checking the ordered t-sets we mark it in the second matrix, or the coverage matrix, referred to as IntermediateMatrix in my code. So for each of the subarray, each of the ordered t-set is checked and the intermediate matrix is updated accordingly.

Size of intermediate matrix/coverage matrix = Number of SubArrays * No of Ordered t-sets.

```
int *intermediateArray = (int *)calloc(noOfSubArrays * combinationOfSymbols,
sizeof(int));
```

The code to generate the coverage matrix is listed below.

```
       for (iLoopIndex = 0; iLoopIndex < noOfSubArrays; iLoopIndex++)
       {
              int firstColumnIndex = SubArrays[t * iLoopIndex];
              int secondColumnIndex = SubArrays[t * iLoopIndex + 1];
              int thirdColumnIndex = SubArrays[t * iLoopIndex + 2];
              char *tempInputArray = (char *)calloc(N*t,sizeof(char));
              for (jLoopIndex = 0; jLoopIndex < N; jLoopIndex++)
              {
                     if (2 == t)
                     {
                            tempInputArray[t*jLoopIndex] = inputArray[k*jLoopIndex +
firstColumnIndex - 1] ;
                            tempInputArray[t*jLoopIndex + 1] = inputArray[k*jLoopIndex +
secondColumnIndex - 1];
                            for (kLoopIndex = 0; kLoopIndex < combinationOfSymbols;
kLoopIndex++)
                            {
                                   if ( (tempInputArray[t * jLoopIndex] ==
Symbols[t*kLoopIndex]) && (tempInputArray[t * jLoopIndex + 1] == Symbols[t*kLoopIndex +
1]))
                                   {
                                          intermediateArray[(combinationOfSymbols *
iLoopIndex) + kLoopIndex]++;
```

```
                                        }
                                }
                        }
                        else if (3 == t)
                        {
                                tempInputArray[t*jLoopIndex] = inputArray[k*jLoopIndex +
firstColumnIndex - 1];
                                tempInputArray[t*jLoopIndex + 1] = inputArray[k*jLoopIndex +
secondColumnIndex - 1];
                                tempInputArray[t*jLoopIndex + 2] = inputArray[k*jLoopIndex +
thirdColumnIndex - 1];
                                for (kLoopIndex = 0; kLoopIndex < combinationOfSymbols;
kLoopIndex++)
                                {
                                        if ((tempInputArray[t*jLoopIndex] ==
Symbols[t*kLoopIndex]) && (tempInputArray[t*jLoopIndex + 1] == Symbols[t*kLoopIndex + 1])
&& (tempInputArray[t*jLoopIndex + 2] == Symbols[t*kLoopIndex + 2]))
                                        {
                                                intermediateArray[(combinationOfSymbols *
iLoopIndex) + kLoopIndex]++;
                                        }
                                }
                        }
                }
                free(tempInputArray);
        }
```

After the coverage matrix is generated, we need to inspect its elements. Those elements which are 1 are very important ones and they need to be preserved with more care then the ones with higher counts in the coverage matrix.

```
        char isCoveringArray = '1';
#ifdef DEBUG
        fprintf(ofp, "Intermediate Matrix\n");
#endif
        for (iLoopIndex = 0; iLoopIndex < noOfSubArrays ; iLoopIndex++)
        {
                for (jLoopIndex = 0; jLoopIndex < combinationOfSymbols; jLoopIndex++)
                {
#ifdef DEBUG
                        fprintf(ofp, "%d\t", intermediateArray[combinationOfSymbols *
iLoopIndex + jLoopIndex]);
#endif
                        if (0 == intermediateArray[combinationOfSymbols * iLoopIndex +
jLoopIndex])
                        {
                                isCoveringArray = '0';
                        }
                }
#ifdef DEBUG
                fprintf(ofp, "\n");
#endif
        }
        if ('0' == isCoveringArray)
        {
                fprintf(ofp, "No, It is Not a Covering Array\n");
#ifdef CONSOLE
                printf("No, It is Not a Covering Array\n");
#endif
        }
        else
        {
                fprintf(ofp, "Yes, It is a Covering Array\n");
#ifdef CONSOLE
                printf("Yes, It is a Covering Array\n");
#endif
        }
```

Selection of Data Structures to track Don't Care positions

The name of the array used to keep track of don't care positions is called finalArray in my code. This array is of size N * k * sizeof(char). I have used to type char to store "*". This is same as the dimensions of the input array except for the fact that, the input array was represented by int, this is represented as char to accommodate the "*".

```
char *finalArray = (char *)calloc(N*k, sizeof(char));
```

To find possible "don't care" positions, it suffices to determine the numbers of times that the v^t * (k choose t) t-way interactions are covered. For each of the N x k entries, check whether the entry appears in any t-way interaction that is covered only once. If not, it is a possible "don't care" position. While conceptually simple, this requires space proportional to v^t * (k choose t), which is too much in practice. Instead, initially mark each of the N x k entries as a possible "don't care." Then for each of the (k choose t) sets of columns in turn, use a vector of length v^t to record the number of times each of the t-way interactions arises in the t chosen columns.

Then for each that arises only once, mark all t positions in it to be no longer "don't care." This requires

only N x k + v^t space, but still requires time proportional to t x N x (k choose t). At the same time, one can verify that the array is in fact a covering array, by ensuring that every t-way interaction is seen at least once. Unfortunately, if we change any one of the possible "don't care" positions to *, some recomputation is then needed.

The algorithm to implement this logically can be seen in the piece of code presented below.

```
if ('1' == isCoveringArray)
        {
                int firstColumnIndex = 0;
                int secondColumnIndex = 0;
                int thirdColumnIndex = 0;

                char *finalArray = (char *)calloc(N*k, sizeof(char));
                memset(finalArray, '0', (N*k*sizeof(char)));
                for (iLoopIndex = 0; iLoopIndex < noOfSubArrays; iLoopIndex++)
                {
```

```
			for (jLoopIndex = 0; jLoopIndex < combinationOfSymbols; 
jLoopIndex++)
			{
				int positionOfOnes = 0;
				int p1_row = 0;
				int p1_column = 0;
				if (1 == intermediateArray[combinationOfSymbols * iLoopIndex + 
jLoopIndex])
				{
					if (2 == t)
					{
						positionOfOnes = combinationOfSymbols * 
iLoopIndex + jLoopIndex;
						p1_row = iLoopIndex;
						p1_column = jLoopIndex;
#ifdef DEBUG
						fprintf(ofp, "%d(%d,%d),", positionOfOnes, 
p1_row, p1_column);
						fprintf(ofp, "{%d,%d}", SubArrays[t*iLoopIndex], 
SubArrays[t*iLoopIndex + 1]);
						fprintf(ofp, "{%c,%c}\n", Symbols[t*jLoopIndex], 
Symbols[t*jLoopIndex + 1]);
#endif
						firstColumnIndex = 0;
						secondColumnIndex = 0;
						char *tempInputArray = (char *)calloc(N*t, 
sizeof(char));
						int jLoopIndex_tempInputArray = 0;
						int tempIndex = 0;
						for (jLoopIndex_tempInputArray = 0; 
jLoopIndex_tempInputArray < N; jLoopIndex_tempInputArray++)
						{
							firstColumnIndex = SubArrays[t * 
iLoopIndex];
							secondColumnIndex = SubArrays[t * 
iLoopIndex + 1];

	tempInputArray[t*jLoopIndex_tempInputArray] = 
inputArray[k*jLoopIndex_tempInputArray + firstColumnIndex - 1];

	tempInputArray[t*jLoopIndex_tempInputArray + 1] = 
inputArray[k*jLoopIndex_tempInputArray + secondColumnIndex - 1];
						}
						for (tempIndex = 0; tempIndex < N; tempIndex++)
						{
#ifdef DEBUG
							fprintf(ofp, "%c%c==%c%c\t", 
Symbols[t*jLoopIndex], Symbols[t*jLoopIndex + 1], tempInputArray[t*tempIndex], 
tempInputArray[t*tempIndex + 1]);
#endif
							if ((tempInputArray[t*tempIndex] == 
Symbols[t*jLoopIndex]) && (tempInputArray[t*tempIndex + 1] == Symbols[t*jLoopIndex + 1]))
							{
#ifdef DEBUG
								fprintf(ofp, "Found at Row %d", 
tempIndex + 1);
#endif
```

```
								finalArray[k*tempIndex +
firstColumnIndex - 1]++;
								finalArray[k*tempIndex +
secondColumnIndex - 1]++;
							}
							else
							{
#ifdef DEBUG
								fprintf(ofp, "Not Found at Row
%d", tempIndex + 1);
#endif
							}
#ifdef DEBUG
							fprintf(ofp, "\n");
#endif
						}
#ifdef DEBUG
						printMatrix(finalArray, N, k);
#endif
						free(tempInputArray);
					}
					else if (3 == t)
					{
						positionOfOnes = combinationOfSymbols *
iLoopIndex + jLoopIndex;
						p1_row = iLoopIndex;
						p1_column = jLoopIndex;
#ifdef DEBUG
						fprintf(ofp, "%d(%d,%d),", positionOfOnes,
p1_row, p1_column);
						fprintf(ofp, "{%d,%d,%d}",
SubArrays[t*iLoopIndex], SubArrays[t*iLoopIndex + 1], SubArrays[t*iLoopIndex + 2]);
						fprintf(ofp, "{%c,%c,%c}\n",
Symbols[t*jLoopIndex], Symbols[t*jLoopIndex + 1], Symbols[t*jLoopIndex + 2]);
#endif
						firstColumnIndex = 0;
						secondColumnIndex = 0;
						thirdColumnIndex = 0;
						char *tempInputArray = (char *)calloc(N*t,
sizeof(char));
						int jLoopIndex_tempInputArray = 0;
						int tempIndex = 0;
						for (jLoopIndex_tempInputArray = 0;
jLoopIndex_tempInputArray < N; jLoopIndex_tempInputArray++)
						{
							firstColumnIndex = SubArrays[t *
iLoopIndex];
							secondColumnIndex = SubArrays[t *
iLoopIndex + 1];
							thirdColumnIndex = SubArrays[t *
iLoopIndex + 2];

	tempInputArray[t*jLoopIndex_tempInputArray] =
inputArray[k*jLoopIndex_tempInputArray + firstColumnIndex - 1];

	tempInputArray[t*jLoopIndex_tempInputArray + 1] =
inputArray[k*jLoopIndex_tempInputArray + secondColumnIndex - 1];
```

```
	tempInputArray[t*jLoopIndex_tempInputArray + 2] = 
inputArray[k*jLoopIndex_tempInputArray + thirdColumnIndex - 1];
					}
					for (tempIndex = 0; tempIndex < N; tempIndex++)
					{
#ifdef DEBUG
						fprintf(ofp, "%c%c%c==%c%c%c\t", 
Symbols[t*jLoopIndex], Symbols[t*jLoopIndex + 1], Symbols[t*jLoopIndex + 2], 
tempInputArray[t*tempIndex], tempInputArray[t*tempIndex + 1], tempInputArray[t*tempIndex 
+ 2]);
#endif
						if ((tempInputArray[t*tempIndex] == 
Symbols[t*jLoopIndex]) && (tempInputArray[t*tempIndex + 1] == Symbols[t*jLoopIndex + 1]) 
&& (tempInputArray[t*tempIndex + 2] == Symbols[t*jLoopIndex + 2]))
						{
#ifdef DEBUG
							fprintf(ofp, "Found at Row %d", 
tempIndex + 1);
#endif
							finalArray[k*tempIndex + 
firstColumnIndex - 1]++;
							finalArray[k*tempIndex + 
secondColumnIndex - 1]++;
							finalArray[k*tempIndex + 
thirdColumnIndex - 1]++;
						}
						else
						{
#ifdef DEBUG
							fprintf(ofp, "Not Found at Row 
%d", tempIndex + 1);
#endif
						}
#ifdef DEBUG
						fprintf(ofp, "\n");
#endif
					}
#ifdef DEBUG
					printMatrix(finalArray, N, k);
#endif
					free(tempInputArray);
				}
			}
		}
	}
```

After the matrix is obtained, we check for 0's in the counting matrix, if any place is 0, it means it is a don't care term and can be replaced with *.

```
	//Converting 0 to * for printing in the final Matrix
	for (iLoopIndex = 0; iLoopIndex < N; iLoopIndex++)
	{
		for (jLoopIndex = 0; jLoopIndex < k; jLoopIndex++)
		{
			if ('0' == finalArray[k*iLoopIndex + jLoopIndex])
```

```
                                {
                                        finalArray[k*iLoopIndex + jLoopIndex] = '*';
                                }
                                else
                                {
                                        finalArray[k*iLoopIndex + jLoopIndex] =
inputArray[k*iLoopIndex + jLoopIndex];
                                }
                        }
                }
```

Output the result

After obtaining the matrix, I print the finalarray, which is also the don’t care matrix. To get the position of don’t care terms, I compare the elements in the matrix with *, those row and columns are printed out to obtain the positions.

```
                fprintf(ofp, "Don't Care Matrix\n");
#ifdef CONSOLE
                printf("Don't Care Matrix\n");
#endif
                printMatrix(finalArray, N, k);
                //Printing the co-ordinates of Don't Care positions
                fprintf(ofp, "Don't Care position\n");
#ifdef CONSOLE
                printf("Don't Care position\n");
#endif
                iCount = 0;
                for (iLoopIndex = 0; iLoopIndex < N; iLoopIndex++)
                {
                        for (jLoopIndex = 0; jLoopIndex < k; jLoopIndex++)
                        {
                                if ('*' == finalArray[k*iLoopIndex + jLoopIndex])
                                {
                                        iCount++;
                                        fprintf(ofp, "(%d,%d)\t", iLoopIndex + 1, jLoopIndex +
1);
#ifdef CONSOLE

                                        printf("(%d,%d)\t", iLoopIndex + 1, jLoopIndex + 1);
#endif
                                }
                        }
                }
                fprintf(ofp,"\nTotal Number of Don't Cares :: %d\n",iCount);
```