Algorithm to perform column and row wise Binary to Decimal based compression and decompression schemes for given adjacency matrix of a graph.

Pradeep Gangwar (IHM2016501)[1] Prakhar Chaturvedi (IHM2016001)[2] Praphull Mishra (IWM2016005)[3]

***Abstract*— In this assignment we will obtain a compressed and condensed form of adjacency matrix representation of a graph. We have performed Binary to Decimal based compression techniques and observe the results.**

I. INTRODUCTION AND LITERATURE SURVEY

Graphs are mathematical structures that represent pairwise relationship between objects. They perform important role in modeling data. They are useful in solving many real life applications such as social networks, transportation networks, document-link graphs and many more. Let us introduce the important notations and definitions.

A. Basic Definitions

1) *Graph:* Graph is a pair G=(V,E) where,
   - V is a set of vertices (or nodes), and
   - E $\subseteq$ (VxV) is a set of edges.

   A graph may be directed or undirected, if graph is undirected, then adjacency relation defined by edges is symmetric.
2) *Simple Graph:* A simple graph is an undirected, unweighted graph, containing no self loops and multiple edges.
3) *Adjacency Matrix:* Adjacency matrix of a undirected graph is a matrix B of dimension NxN, where N is the number of vertices such that $B_{i,j}$=1 if there is a edge from vertex *i* to vertex *j*, and 0 otherwise.

B. Objective

Here we will use Binary to Decimal based compression technique to represent the adjacency matrix representation of a graph in condensed form and then we will use some decompression technique to expand the condensed form to adjacency matrix.

II. MOTIVATION

Graph are very important data structures. They are used in network related algorithms, routing, finding relation among objects, shortest paths and many more real-life applications. On E-Commerce websites graphs are used to show recommendations.Connecting with friends on social media, where each user is a vertex and the connection between two users is represented by an edge, is also an important application of graphs. The adjacency matrix and incidence matrix representation of a graph have a very high space complexity. So for above mentioned real life problems, where the size of graph is large and the cost of allocating the space is more, these representations become costly. This motivates us to represent our graphs in some condensed and compressed form, so that space complexity should not be a problem and we can focus on other important aspects.

III. ALGORITHM DESIGN

This part of the report can be further divided into following sections.

A. Input Format

For out input we generate several random matrices containing 0s and 1s and store it in a file $undirec\_test.txt$ for undirected graph, $direc\_test.txt$ for directed graph and use it as input in our main function.

B. Compressing the matrix

- $Encode$ function after getting the matrix as the input traverses through whole array row/column wise.
- It picks up a row and converts its binary form to decimal form.
- Since CPP can hold maximum of 64 bit so after traversing 64 bits we reset the value of the integer and start again for rest of the bits of the row.
- The numbers generated for each row are $n/64$.
- Finally it returns an array that contains all the decimal numbers as the compressed form of matrix.

C. Decompressing the matrix

- $Decode$ function after getting the compressed array as the input traverses through whole array.
- The numbers generated for each row are $n/64$.
- It picks up desired amount of numbers assigned for each row.
- It converts the decimal numbers back to binary format and puts them back into the matrix.

D. Algorithm

Algorithm 1 Compression

```
1: procedure ENCODE(a[][],n)   ▷ Compression function
2:     condensed ← []
3:     for i = 0 to n do
4:         power ← 0
5:         number ← 0
6:         for j = 0 to n do
7:             number ← number + a[i][j] * 2^power
8:             if power = 64 then
9:                 condensed.push(number)
10:                number ← 0
11:                power ← 0
12:            power++
13:        condensed.push(number)
14:    return condensed
```

Algorithm 2 Decompression

```
1: procedure DECODE(encoded[],decoded[][],numrow, n)
2:     for i = 0 to n do
3:         for k = 0 to numrow do
4:             number ← encoded[i * numrow + k]
5:             for j = 0 to n do
6:                 decoded[i][j] ← number%2
7:                 number ← number/2
```

E. Output Format

The output consists of an array that contains decimal numbers. Since we have use CPP programming language so for each row numbers stored will be $n/64$, where n is the number of rows in matrix. CPP can hold maximum 64 bit number. After decompression we obtain the same matrix back as we provided in the input.

IV. ANALYSIS

A. Analysis: Compression

The function $encode(a[][],n)$ traverses through the whole matrix row/column wise and converts them into decimal numbers. For each row the number of $decimalnumbers$ we get is $n/64$ because CPP can hold as big as 64 bit number only. So after parsing 64 bits we reset our number to 0 and start afresh from next bit.

1) Best Case Analysis - (n <5): When the matrix is of size less than 5 we will have to traverse through the entire matrix row/column wise in order to obtain the desired result.

$time_{best(n<5)} \propto 4n^2 + 2n + k$ instructions.(k-constant)

$=> time_{best(n<5)} \propto n^2$

$=> O(n^2)$ - A Quadratic Time Computation

2) Worst Case Analysis : As discussed above in the worst case scenario also we will have to traverse through the entire matrix column wise in order to obtain the desired result. So,the worst case time complexity is equal to best case time complexity.

$time_{worst} \propto 4n^2 + 2n + k$ instructions.(k-constant)

$=> time_{worst} \propto n^2$

$=> O(n^2)$ - A Quadratic Time Computation

$=> t_{best} = t_{worst}$

3) Average Case Analysis : We know that

$$time_{best} <= t_{average} <= t_{worst}$$

But from above

$t_{best} = t_{worst}$

So

$=> t_{best} = t_{avg} = t_{worst}$

B. Analysis: Decompression

The function $decode(encoded[],decoded[][],numrow,n)$ traverses through the whole matrix row/column wise and converts decimals into binary format. $numrow$ contains the number of decimals designated for each row which is in turn equal to $n/64$. For each row we pick $numrow$ amount of decimal numbers and after converting them to binary feed them into matrix sequentially.

1) Best Case Analysis - (n <5): When the matrix is of size less than 5 we will have $n/64$ i.e. $numrow = 1$ number per row. So encoded array contains five numbers i.e. one per row. The inner loop runs five times since it sets five bits per row to get the original matrix back. So it has $N * N$ complexity.

$time_{best(n<5)} \propto 4n^2 + 3n + k$ instructions.(k-constant)

$=> time_{best(n<5)} \propto n^2$

$=> O(n^2)$ - A Quadratic Time Computation

2) Worst Case Analysis : As discussed above in the worst case scenario also we will have to traverse through the entire array and set the value of each cell. So,the worst case time complexity is equal to best case time complexity.

$time_{worst} \propto 4n^2 + 3n + k$ instructions.(k-constant)
$=> time_{worst} \propto n^2$

$=> O(n^2)$ - A Quadratic Time Computation

$=> t_{best} = t_{worst}$

3) Average Case Analysis : We know that

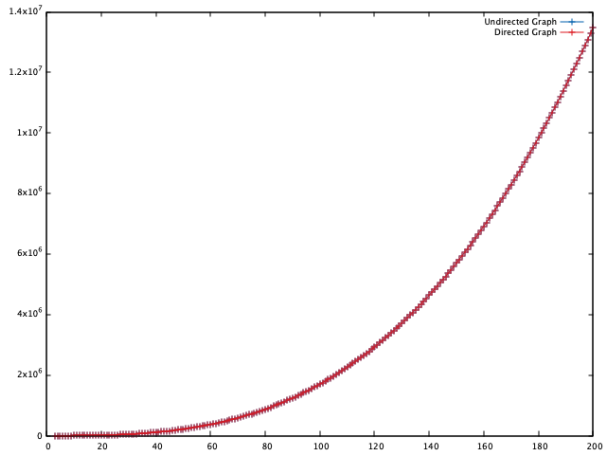
$$time_{best} <= t_{average} <= t_{worst}$$

But from above
$t_{best} = t_{worst}$
So
$=> t_{best} = t_{avg} = t_{worst}$

V. EXPERIMENTAL STUDY

The best way to study an algorithm is by graphs and profiling.

1) Graphs-Time-Complexity: We have seen that $t_{best} = t_{avg} = t_{worst}$ So, the graphs for best case, average, worst case time complexity remain the same.We can clearly see that for both un-directed and directed graph matrices the complexity of compression or decompression remains the same.



2) Profiling:

$NumberOfComputations-Directed/UndirectedGraph$

n	t_{avg}	t_{best}	t_{worst}
3	48	48	48
5	262	262	262
20	14532	14532	14532
50	215872	215872	215872
100	1699742	1699742	1699742
150	5704986	5704986	5704986
200	13485008	13485008	13485008

- We can seen that $t_{best} = t_{avg} = t_{worst}$
- We can also see that time increases as the value of n increases and $time_{worst} \propto n^2$,$time_{avg} \propto n^2$,$time_{best} \propto n^2$.
- We see that time complexity for both directed and undirected graph remains the same.

VI. DISCUSSION

A. Understanding the Use Of File Handling and generation of random numbers

In this context the file handling is used to generate the random test cases through which analysis and experimentation of algorithm becomes easier for handling different test cases. We have used $srand()$ and $rand()$ to generate the random values to fill the matrix with the help of $< time.h >$ header file.

B. About Binary to Decimal compression technique

We have seen the explanation for Binary to Decimal compression algorithm to solve this question $O(n) = n^2$. This algorithm converts the matrix consisting of 0s and 1s to decimal numbers and stores them in an array. It saves the space. Space complexity of storing adjacency matrix is NXN where N is the number of vertices whereas storing the compressed form requires array of size $N^2/64$. We see that space complexity for this has been reduced.

VII. CONCLUSION

The main idea of the problem is to represent the graph in a compressed form. We have used Binary to Decimal based compression technique for this. Binary to Decimal compression gives good space complexity but has higher time complexity. We read about other compression techniques such as run length encoding, incidence list etc but most of them had same time complexity.

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.
[2] Introduction to Algorithms English By Thomas H. Cormen , By Charles E. Leiserson , Ronald L. Rivest ,Clifford Stein(3rd edition)