DS-221 Introduction to Scalable Systems
Assignment - 2

Submitted by - Ocima Kamboj
Serial Number - 06-02-01-10-51-18-1-15899

Matrix Operations

1 Analysis of Space Complexity of the Load function

1.1 2D Array Implementation

1.1.1 Analytical Space Complexity

For a matrix of size $N \times N$, the 2D array will store every element, and thus will take $O(N^2)$ space.

1.1.2 Emperical Space Complexity

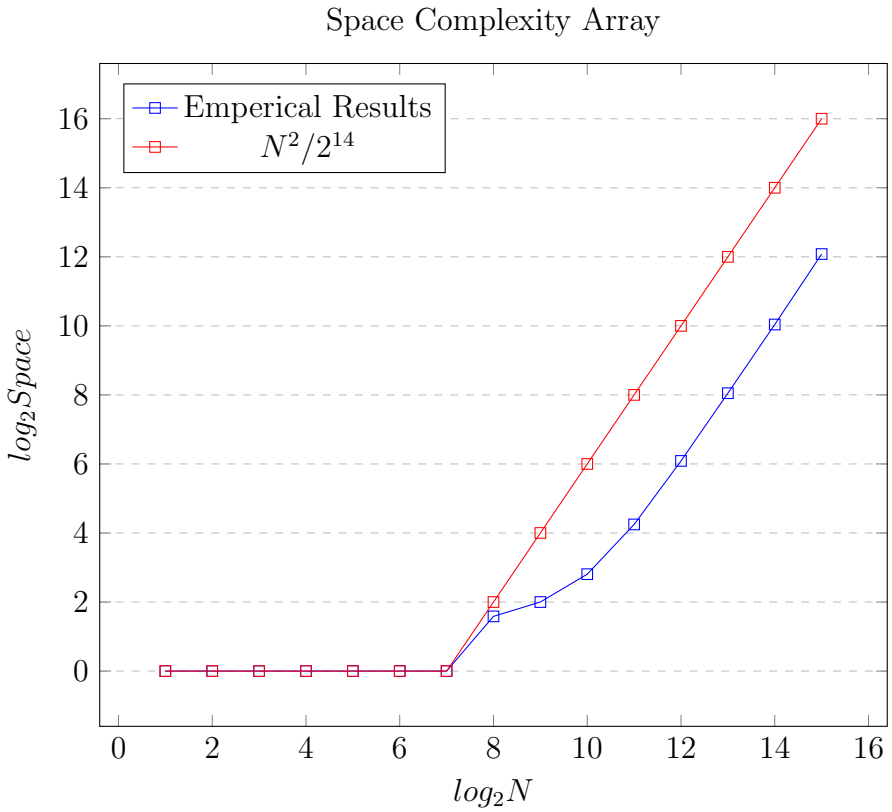
The space complexity was estimated using the following piece of code taken from stackoverflow.
(https://stackoverflow.com/questions/63166/how-to-determine-cpu-and-memory-consumption-from-inside-a-process) -

```
int parseLine(char* line){
    // This assumes that a digit will be found and the line ends in " Kb".
    int i = strlen(line);
    const char* p = line;
    while (*p <'0' || *p > '9') p++;
    line[i-3] = '\0';
    i = atoi(p);
    return i;
}

int getValue(){ //Note: this value is in KB!
    FILE* file = fopen("/proc/self/status", "r");
    int result = -1;
    char line[128];

    while (fgets(line, 128, file) != NULL){
        if (strncmp(line, "VmRSS:", 6) == 0){
            result = parseLine(line);
            break;
        }
    }
    fclose(file);
    return result;
```

}

The analysis was done for matrices of sizes $2^n \times 2^n$.

The following plot shows the results -



As we can see, the plotted line corresponding to our results always lies below the N^2 line. Therefore, the space complexity is $O(N^2)$, which matches with our analytical space complexity.

The biggest matrix size that was loaded on Turing was $2^{15} \times 2^{15}$, because of the 20GB limit. The biggest matrix size that was loaded on my machine was $2^{16} \times 2^{16}$. The next size went out of available memory on my hard disk.

1.2 CSR Implementation

1.2.1 Analytical Space Complexity

Let the number of non-zero elements be nnz.
CSR implementation maintains three arrays, one of type $float$ and size nnz, second of type int and size nnz, and third of type int and size $N + 1$. Both int and $float$ take up four bytes of space.
Therefore, the space occupied will be -

$$nnz + nnz + N - 1$$

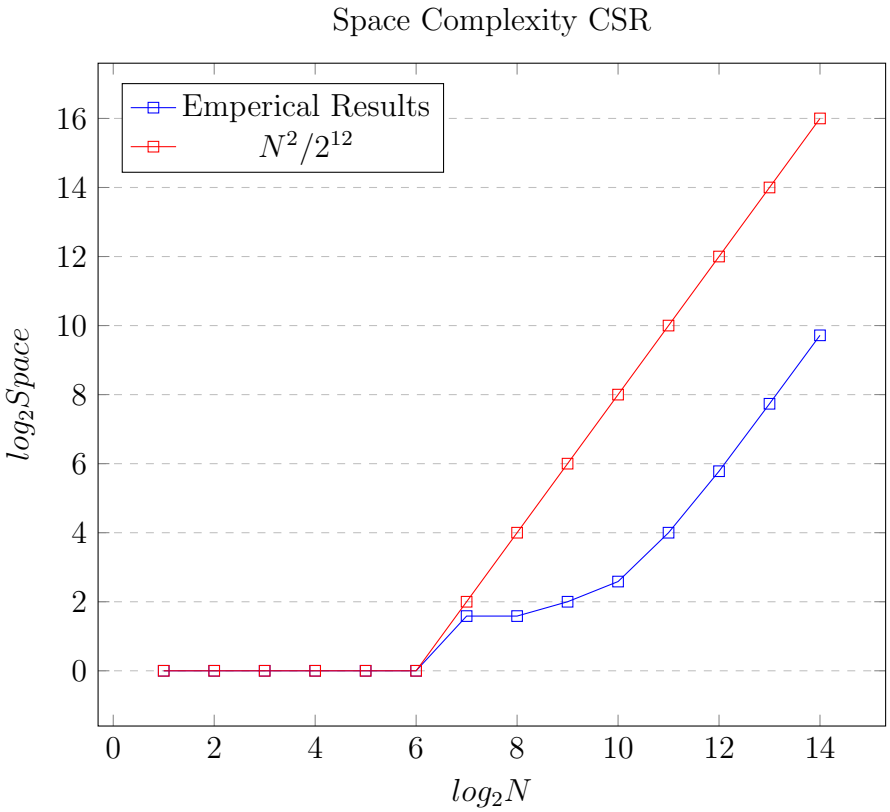
The sparse matrices generated for testing had 60% sparsity. Therefore, the number of non-zero elements is $0.4N^2$. Using this relation, the space occupied will be -

$$2(0.4N^2) + N - 1 = 0.8N^2 + N - 1$$

Therefore, the space complexity will be $O(N^2)$.

1.2.2 Emperical Space Complexity

The results are plotted below -



As we can see, the plotted line corresponding to our results always lies below the N^2 line. Therefore, the space complexity is $O(N^2)$, which matches with our analytical space complexity.

The biggest matrix size that was loaded on Turing was $2^{14} \times 2^{14}$, which took a runtime of about 3 hours.

1.3 Array Implementation vs. CSR

Although both the array and CSR implemention require $O(N^2)$ space, the space required by CSR is less.
The actual space required by 2D array of type $float$ in MB is -

$$\frac{4N^2}{10^6}$$

As established in the previous section, for a sparse matrix of sparsity 60%, the space occupied in MB would be -

$$\frac{4(0.8N^2 + N - 1)}{10^6} = \frac{3.2N^2 + 4N - 4}{10^6}$$

The difference in the space stored by the array implementation and the CSR implementation is -

$$\frac{0.8N^2 - 4N - 4}{10^6}$$

We'll refer to this expression as the analytical difference.

The following table compares the result for different matrix sizes -

Space-Array	**Space-CSR**	**Difference**	**Analytical Difference**
1	1	0	0
1	1	0	0
1	1	0	0
1	1	0	0
1	1	0	0
1	1	0	0
1	3	-2	0
3	3	0	0
4	4	0	0
7	6	1	1
19	16	3	3
68	55	13	13
265	213	52	54
1052	842	210	215

Table 1: Array and CSR Space complexity comparision

The emperical differences lie very close to the analytical differences, thus showing the space efficiency of CSR implementation.

1.4 Loading Times

The loading times for CSR implementation were observed to be more than those of Array implementation. This is happening because of the complex way in which rows are appended in a CSR implementation. During *append* method, the arrays A, corresponding to non zero elements, and JA, c haveponded to column indexes of non zero elements, to be redeforresined with bigger size to make room for more non-zero elements, and the previously existing elements have to be copied to the new defined arrays. This process takes time, which leads to more loading times for CSR implementation.

2 Analysis of Time Complexity for ADD function

2.1 2D Array Implementation

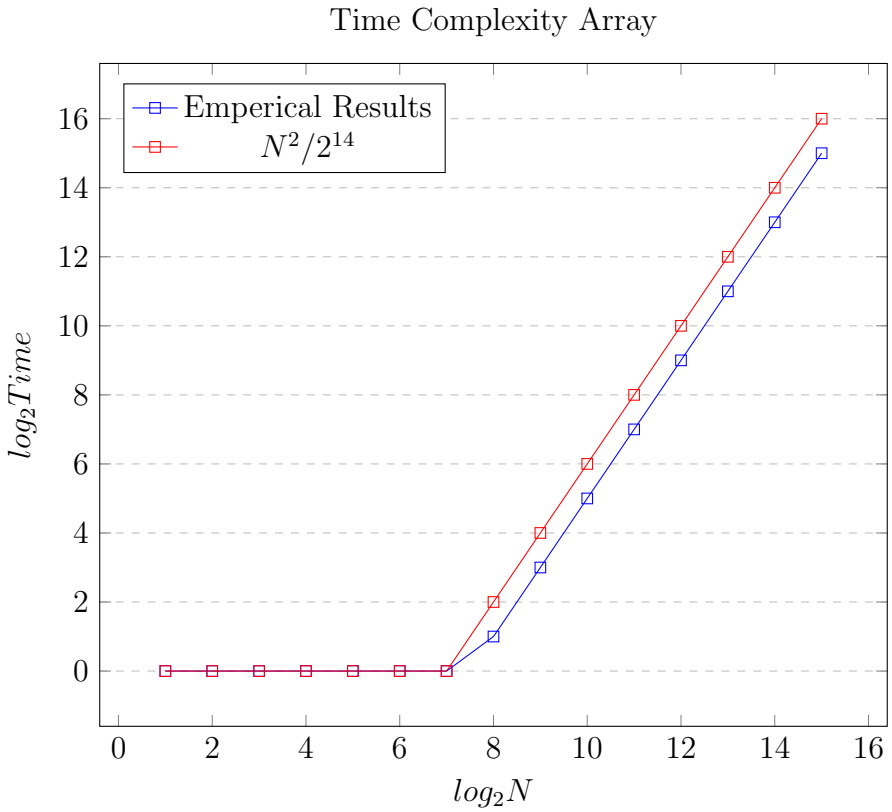
To calculate the (i, j) element of matrix-3, which is the sum of matrix-1 and matrix-2, we perform $get(i, j)$ on matrix-1 and matrix-2, add them, and store it in an array which is later appended to matrix-3.

2.1.1 Analytical Time Complexity

The $get(i, j)$ method for Array Implementation is constant time, that is $O(1)$. This is done N^2 times, over a double loop of i and j. Thus the overall time complexity is $O(N^2)$.

2.1.2 Emperical Time Complexity

The plots are shown below -



As we can see, the plotted line corresponding to our results always lies below the N^2 line. Therefore, the time complexity is $O(N^2)$, which matches with our analytical time complexity.

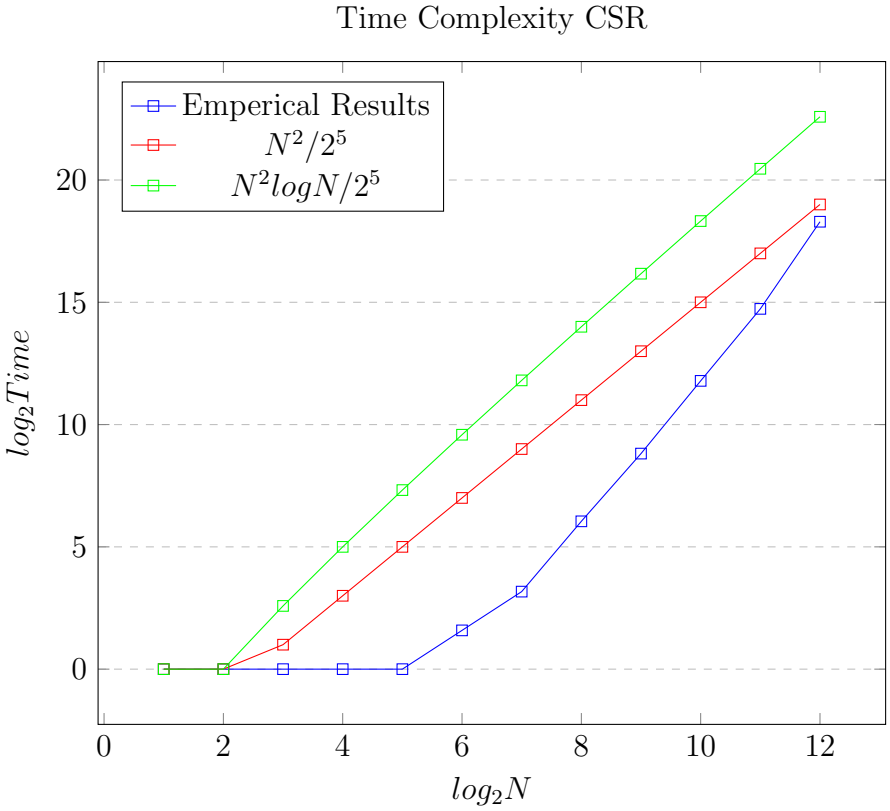
2.2 CSR Implementation

2.2.1 Analytical Time Complexity

As before, the add method performs two $get(i, j)$ operations, but the time complexity for $get(i, j)$ in CSR Implementation is not $O(1)$.
The arrays $A[nnz]$ and $JA[nnz]$ store the non-zero elements and their column indexes. The array $IA[N + 1]$ stores the cumulative no. of non zero elements in each row.
Thus, to find an element, we first find the start index $IA[i]$ and end index $IA[i+1]$, between which to search in $JA[nnz]$, and then scan through $JA[nnz]$ between the said limits to see whether the index j is present or not. This searching is done through Binary Search. Therefore, in the worst case, each get operation would take $O(logN)$ time.
Because this is done N^2 amount of time, the overall time complexity is $O(N^2logN)$.

2.2.2 Emperical Time Complexity

The results are plotted below -



As can be seen from the plot, the line corresponding the emperical results would pass the N^2 line for big values of N, implying complexity is more than $O(N^2)$.
The same also seems to be true for the N^2logN line, which would imply that time complexity is bigger than $O(N^2logN)$.
This disparity seems to be arising due to the way the $append(float * row\_vals)$ method is being implemented.

If matrix-3 is the sum of matrix-1 and matrix-2, then after the sum of elements of matrix-1 and matrix-2 of one complete row have been found out, then that row is appended to matrix-3. During this method, the arrays A and JA have to be redefined with bigger size to make room for more non-zero elements, and the previously existing elements have to be copied to the new defined arrays. This process takes time, which is disturbing our study of time complexity for CSR implementation.

3 Analysis of Time Complexity for Multiply function

3.1 2D Array Implementation

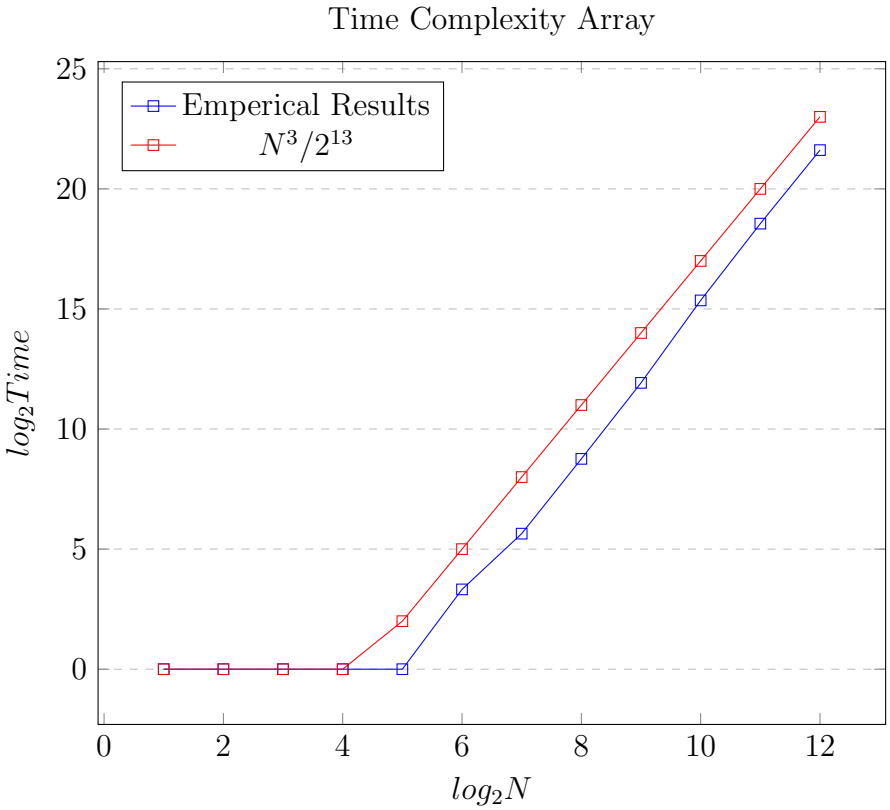
To calculate the (i, j) element of matrix-3, which is the product of matrix-1 and matrix-2, we perform $get(i, k)$ on matrix-1 and $get(k, j)$ on matrix-2, multiply them, add the product to the already existing element at the (i, j) position in matrix-3 (stored in an array which is later appended to matrix-3). We do this for i, j and k varying from 0 to $N - 1$.

3.1.1 Analytical Time Complexity

As stated earlier, the $get(i, j)$ for Array Implementation is $O(1)$. As this is done N^3 number of times, the over time complexity is $O(N^3)$.

3.1.2 Emperical Time Complexity

The results are plotted below -



As we can see, the plotted line corresponding to our results always lies below the N^3 line. Therefore, the time complexity is $O(N^3)$, which matches with our analytical time complexity.

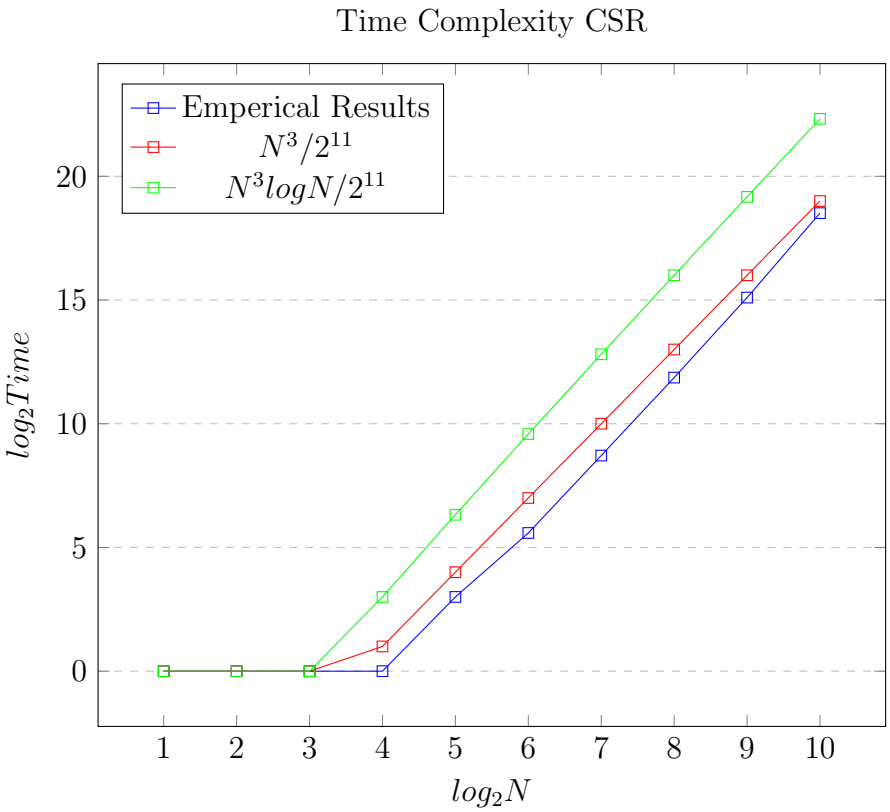
3.2 CSR Implementation

3.2.1 Analytical Time Complexity

As established in Section-2.2.1, the $get(i, j)$ operation takes $O(logN)$ time. As this is done N^3 number of times, the overall time complexity is $O(N^3 logN)$.

3.2.2 Emperical Time Complexity

The results are plotted below -



As can be seen from the plot, the line corresponding the emperical results would pass the N^3 line for big values of N, implying complexity is more than $O(N^3)$.
The plotted line corresponding to our results always lies below the $N^3 log(N)$ line. Therefore, the time complexity is $O(N^3 log(N))$, which matches with our analytical time complexity.
The disparity that was arising in the CSR implemention of ADD due to the append function is not arising here. This might be because the N^3 term is dominating, suppressing the effects of copying and deletion of arrays.

4 Analysis of BFS using CSR Implementation

For this section, we'll refer to the number of vertices as V. As the graph is stored in an adjacency matrix format, the size of the graph (as the matrix) will be V^2.
The BFS is carried out in the following way - the source vertex is stored in a Queue and marked as 'visited'. Then we scan through the row corresponding to the source vertex in the adjacency matrix to find its neighbours, add them to the end of the queue, and mark them as visited. When all the neighbours of the source vertex have been found, we remove it from the queue, and repeat the process for the remaining vertices.

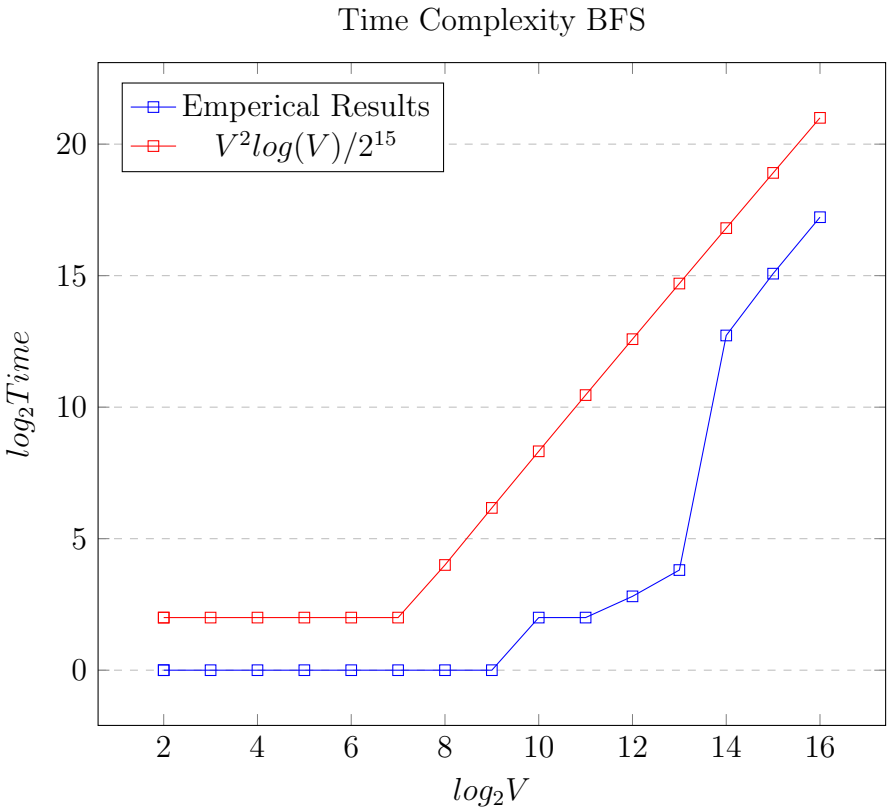
4.1 Time Complexity

4.1.1 Analytical Time Complexity

To find the neighbours of each vertex during BFS traversal, we look through the whole row corresponding to it in the adjacency matrix using *get* method, which is called V number of times. In CSR implementation, each get operation takes $O(logV)$ time. This bound can be made tighter by imposing conditions on the maximum number of immediate neighbours of each vertex. But, for now, we continue the analysis with this loose bound. Thus, the overall operation takes $O(VlogV)$ amount of time.
Let the total number of vertices that are reachable from the source vertex be R. This is the maximum number of times that we'll perform the search operation for the neighbours of one particular element. In the worst case scenario, all the vertices are reachable from the source vertex, thus performing the finding of neighbours V times, making the time complexity $O(V^2logV)$.

4.1.2 Emperical Time Complexity

The results are plotted below -



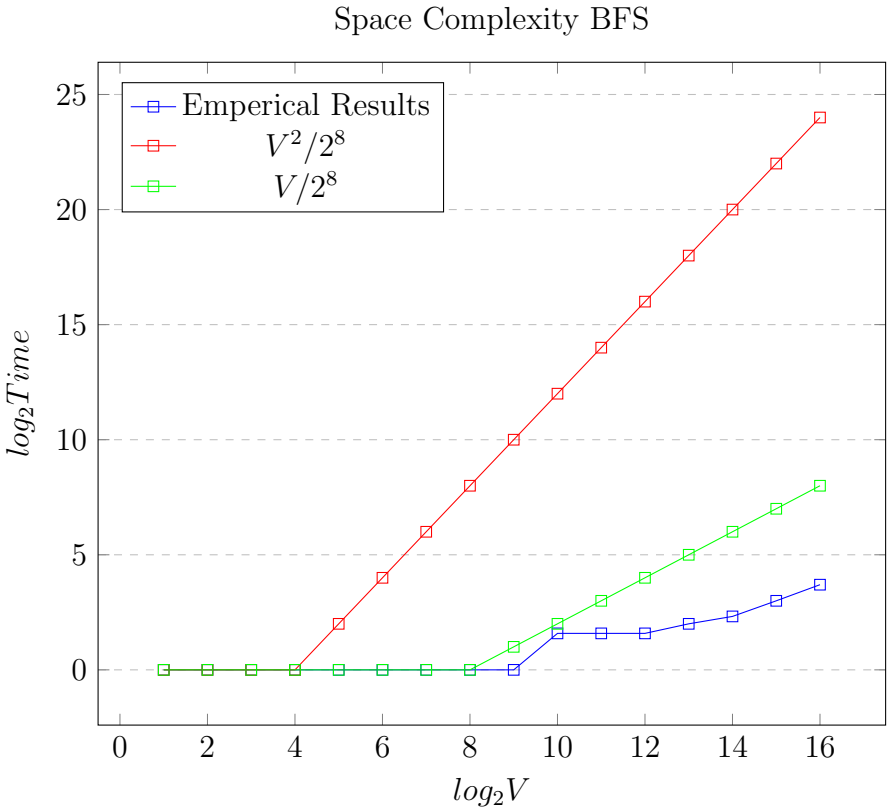
The line corresponding to the emperical results lies below the line corresponding to the analytical results, thus showing that the time complexity is $O(V^2logV)$.

4.2 Space Complexity

4.2.1 Analytical Space Complexity

The adjacency matrix format for a graph is a matrix of size V. As established in Section-1.2.1, the space complexity for storing the adjacency matrix will be $O(V^2)$.
In addition, we also maintain a list of all the traversed vertices. If the number of reachable vertices from the source vertex is R, the space complexity for this list would be $O(R)$. In the worst case, every vertex will be reachable from the source vertex, thus making the space complexity for this to be $O(V)$.

The results are plotted below -



The space complexity of our BFS algorithm seems to better than $O(V)$.
The largest graph size that was loaded is $V = 2^{16}$.

5 Change in code in Makefile

The Makefile was edited to compile using C++11 standard.