Algorithms Lab

Assignment 4: Breadth First Search

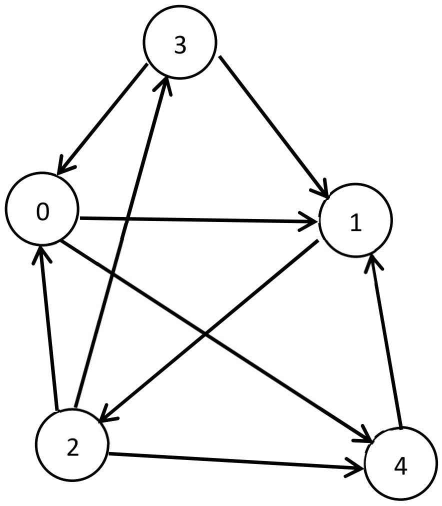
Aditya Badayalya

510819056

IT-Hy

Breadth First Search:

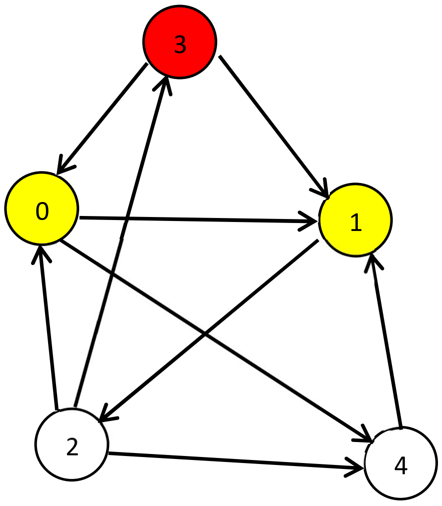
Let us consider the following directed graph:



For Breadth First traversal of the above graph from the vertex ‘3’ the following steps take place

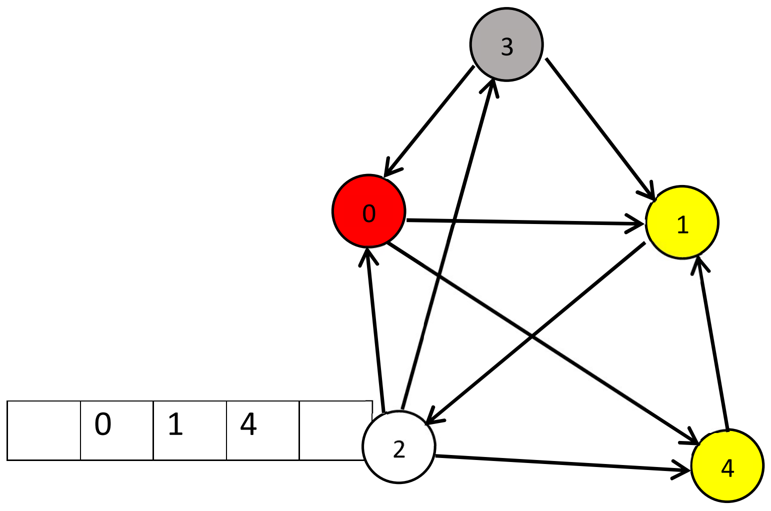
The Vertex ‘3’ is pushed into the traversal queue and after that its neighbors are pushed into the queue.

3				

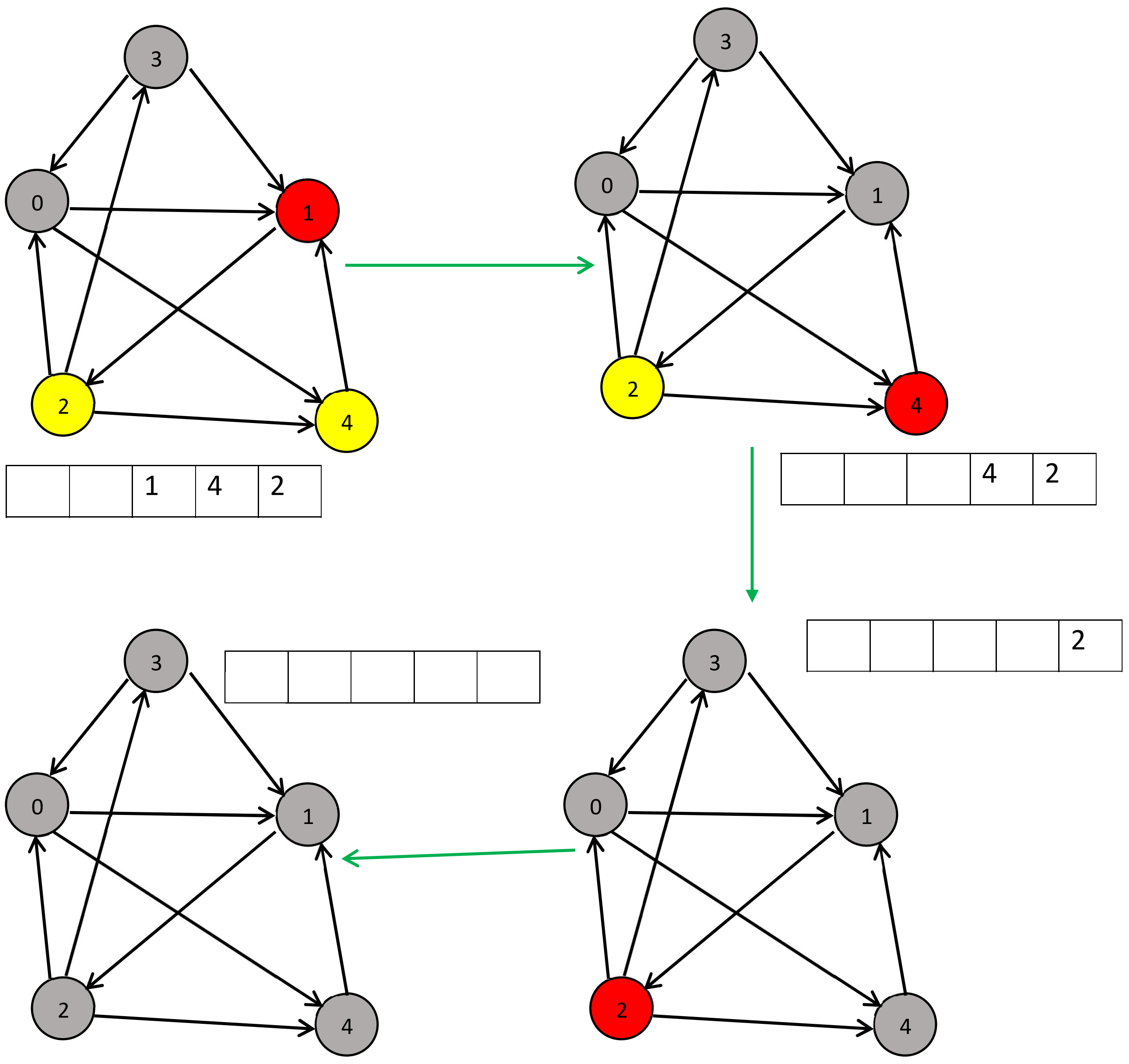


3	0	1		

After pushing all the neighbors of the vertex ‘3’ in the queue, it is marked as visited and the next vertex that is front of the traversal queue is brought into focus and same process in performed on this vertex.



The above process continues until the traversal queue is empty



The Breadth First Traversal that is obtained from the above process for the graph is as follows: 3->0->1->4->2

The Breadth First Traversal of a graph is much similar to the level traversal of a tree, except at times there are some graphs that contain cycles and thus if not taken care of properly, we would be stuck in an infinite loop. To avoid this, a Boolean array of vertices is considered where if the vertex is already marked visited then no further operations are performed on the vertex and the cycle is tackled.

The Breadth First Traversal of a graph also helps in the searching of any element in the graph and is rather quicker than going through the entire edge list. The resulting search algorithm is called as Breadth First Search

Following is the Breadth First Traversal Algorithm:

```
void BFS(int s){
    void BFS(int s){
        bool *visited = new bool[V];
        for(int i=0;i<V;i++){
            visited[i]=false;
        }
        list<int>queue;
        visited[s]=true;
        queue.push_back(s);
        list<int>::iterator i;
        while(!queue.empty()){
            s=queue.front();
            cout<<s<<' ';
            queue.pop_front();
            for(i = adj[s].begin();i!=adj[s].end();i++){
                if(!visited[*i]){
                    visited[*i]=true;
                    queue.push_back(*i);
                }
            }
        }
    }
```

Here the argument that is passed to the function is the node from which the traversal begins and ‘adj’ is the adjacency list of the graph where the details regarding the edges are stored.

From the initial look the algorithm seems to be of the complexity O(n^2) with nested loops being into play but on a closer inspection, it can be observed that the maximum number of elements that would be in the queue are equal to the number of vertices in the graph and for the inner for loop, the number of times it would be executed would be equal to the total number of edges as it is just iterating over the adjacency lists which is nothing but the details of the edges.

So, the overall complexity of the algorithm can be determined as O(V+E) where 'V' is the number of vertices and 'E' is the number of edges of the graph.

There are limitations to the above algorithm for instance, not every vertex can be accessed from the starting vertex that if provided to the algorithm as the vertex might be standalone or the graph maybe disconnected. Thus, to traverse the graph entirely, the above algorithm has to be modified to execute itself from every vertex resulting in complete traversal of the graph.