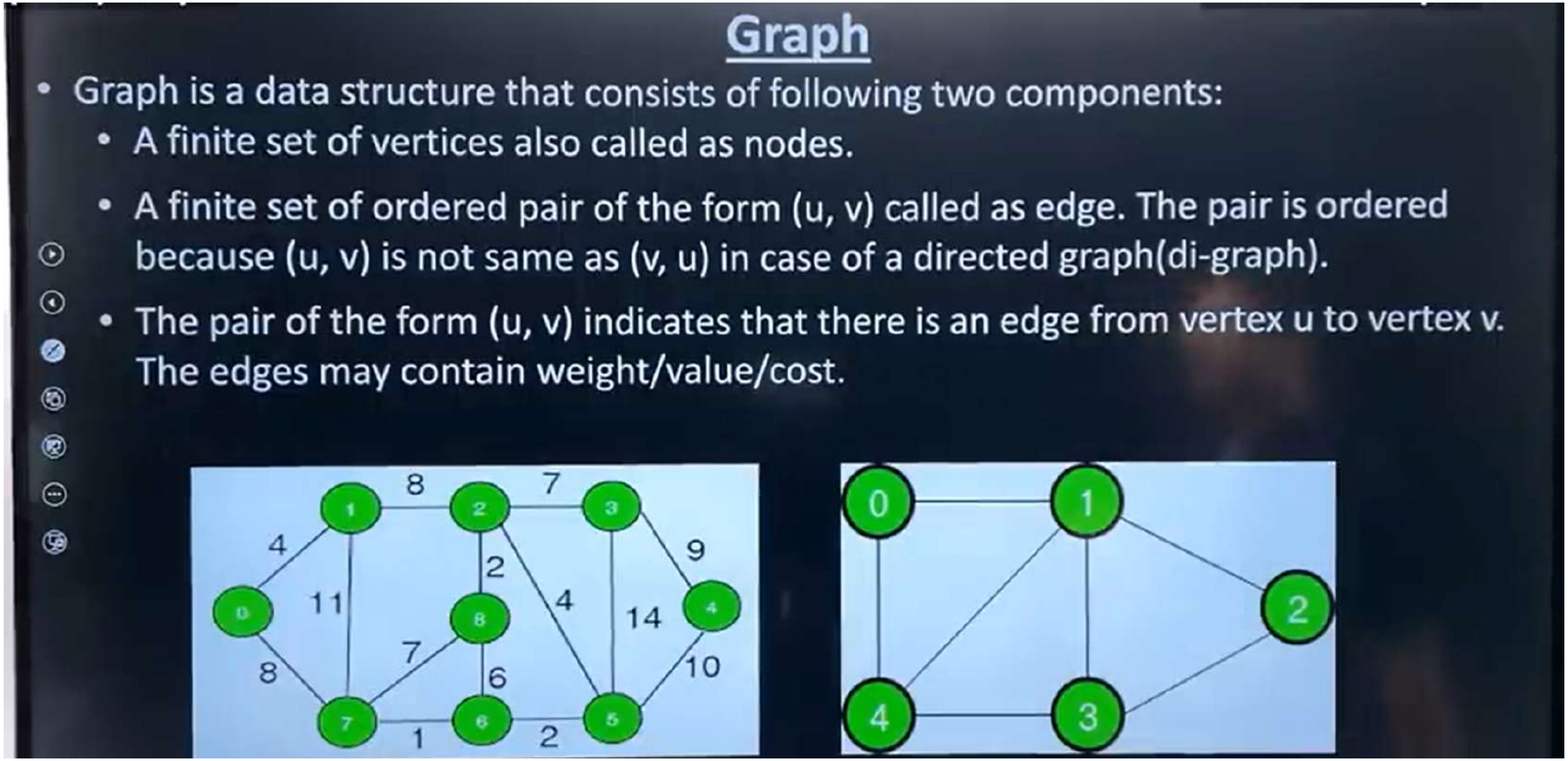
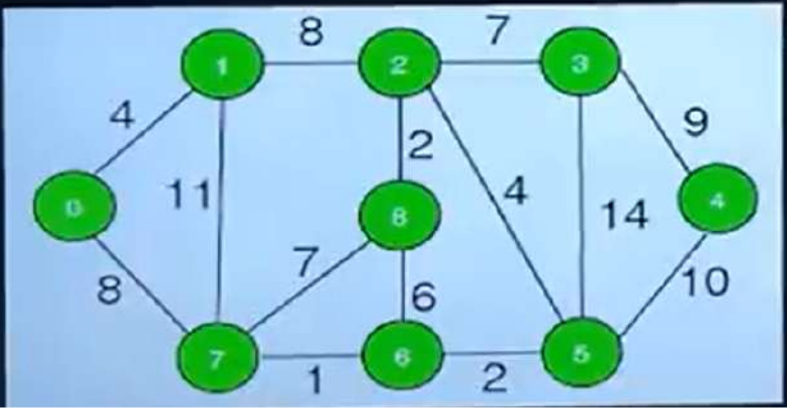
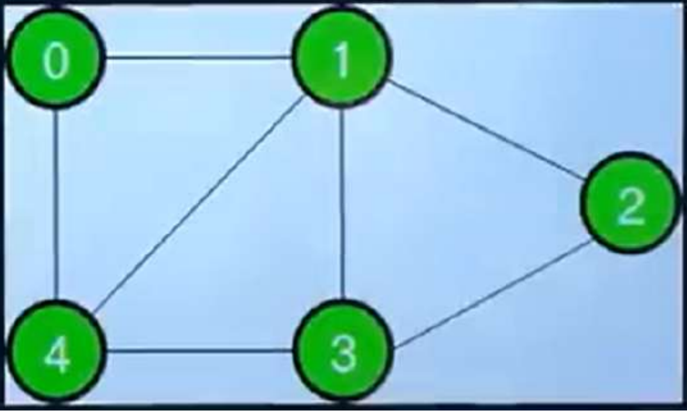
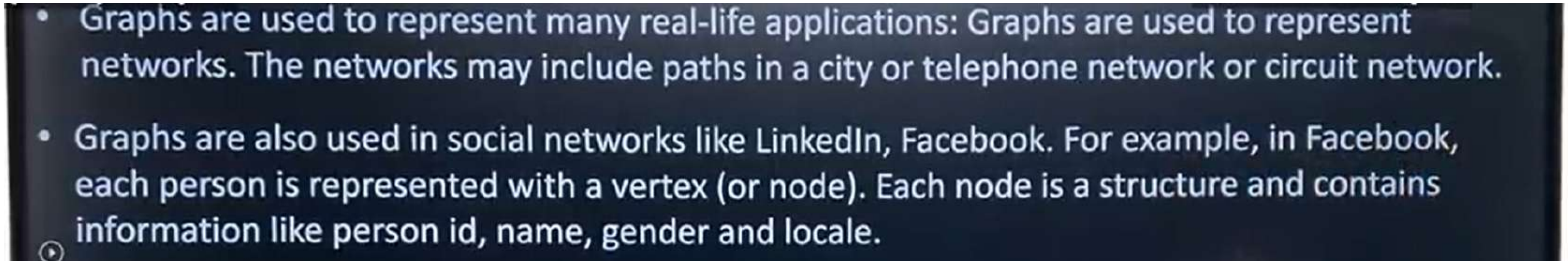


Graph

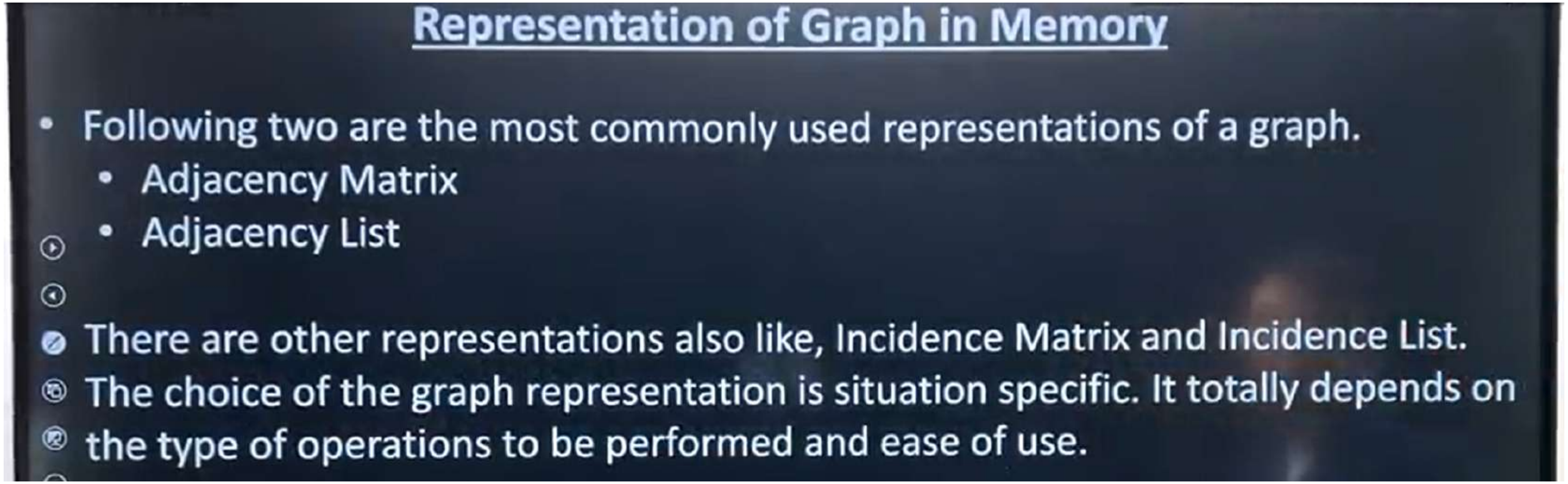
- Graph is a data structure that consists of following two components:
  - A finite set of vertices also called as nodes.
  - A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph).
  - The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.







- Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale.



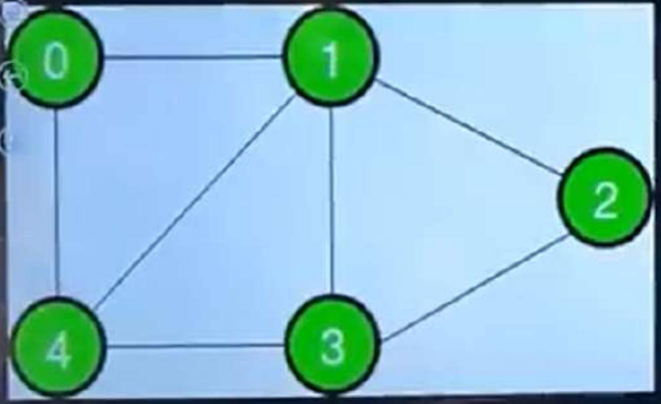
Representation of Graph in Memory

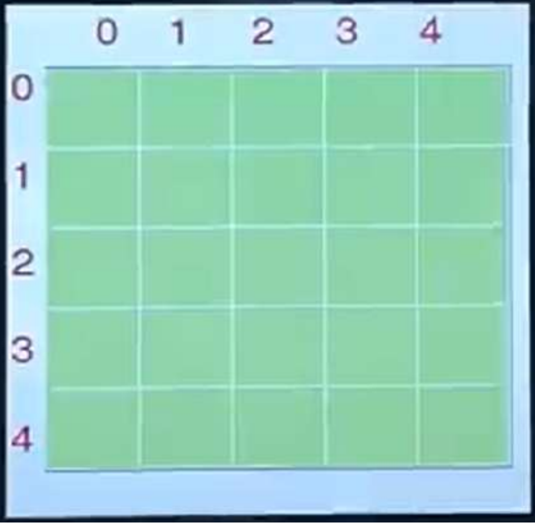
- Following two are the most commonly used representations of a graph.
    - Adjacency Matrix
    - Adjacency List

There are other representations also like, Incidence Matrix and Incidence List.

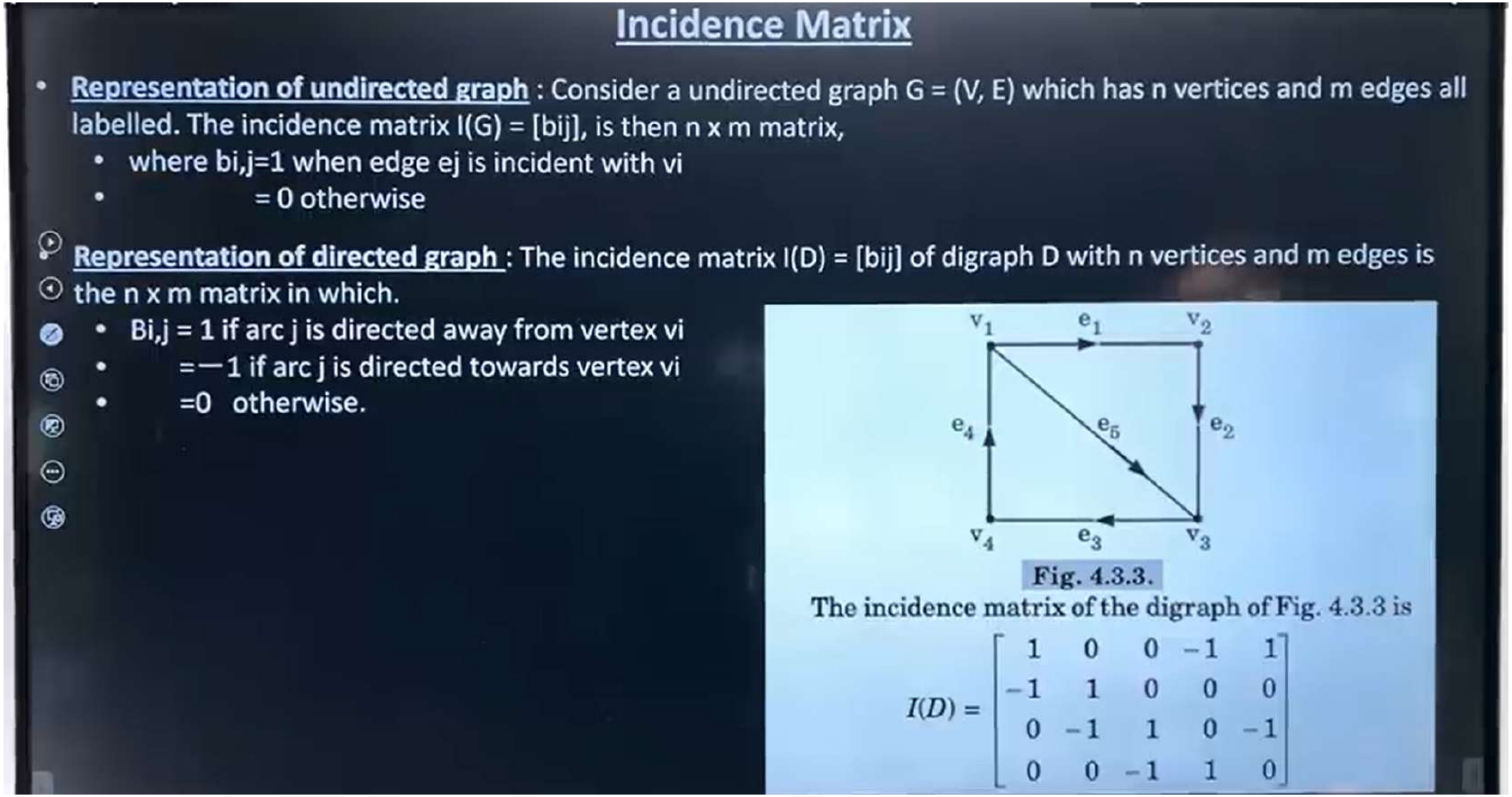
The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

- **Adjacency Matrix:** Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.





	0	1	2	3	4
0					
1					
2					
3					
4					



Incidence Matrix

- **Representation of undirected graph** : Consider a undirected graph G = (V, E) which has n vertices and m edges all labelled. The incidence matrix I(G) = [bij], is then n x m matrix,
  - where bi,j=1 when edge ej is incident with vi
  - = 0 otherwise

Representation of directed graph : The incidence matrix I(D) = [bij] of digraph D with n vertices and m edges is the n x m matrix in which.

- Bi,j = 1 if arc j is directed away from vertex vi
- =−1 if arc j is directed towards vertex vi
- =0 otherwise.

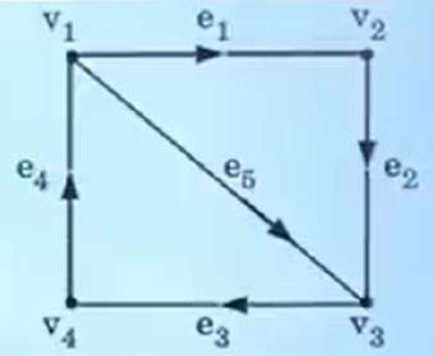
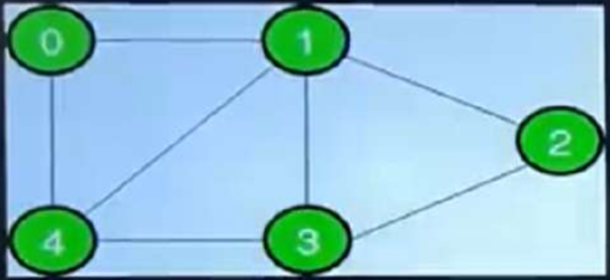


Fig. 4.3.3.

The incidence matrix of the digraph of Fig. 4.3.3 is

$$I(D) = \begin{bmatrix} 1 & 0 & 0 & -1 & 1 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 & 0 \end{bmatrix}$$

- **Adjacency List:** An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.

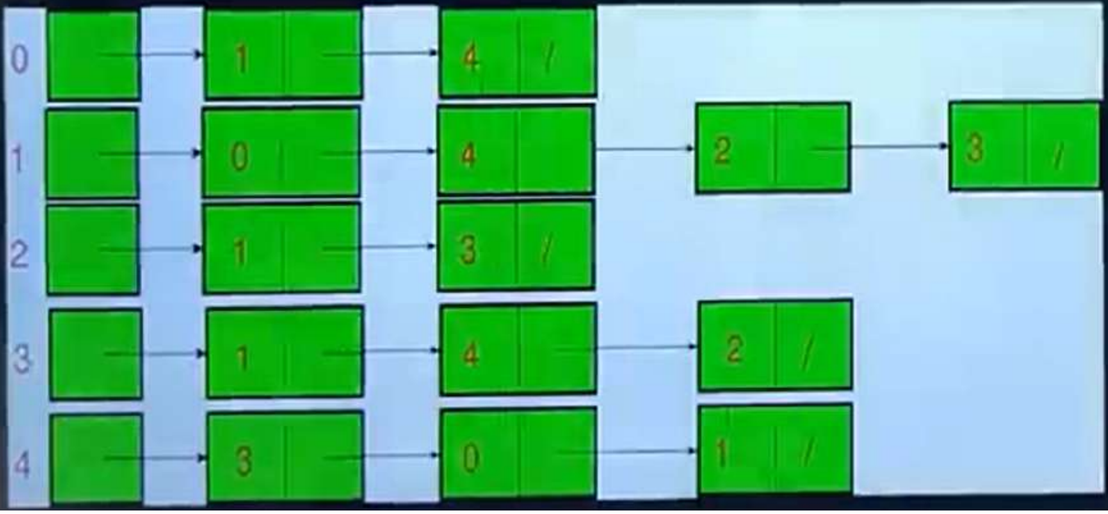


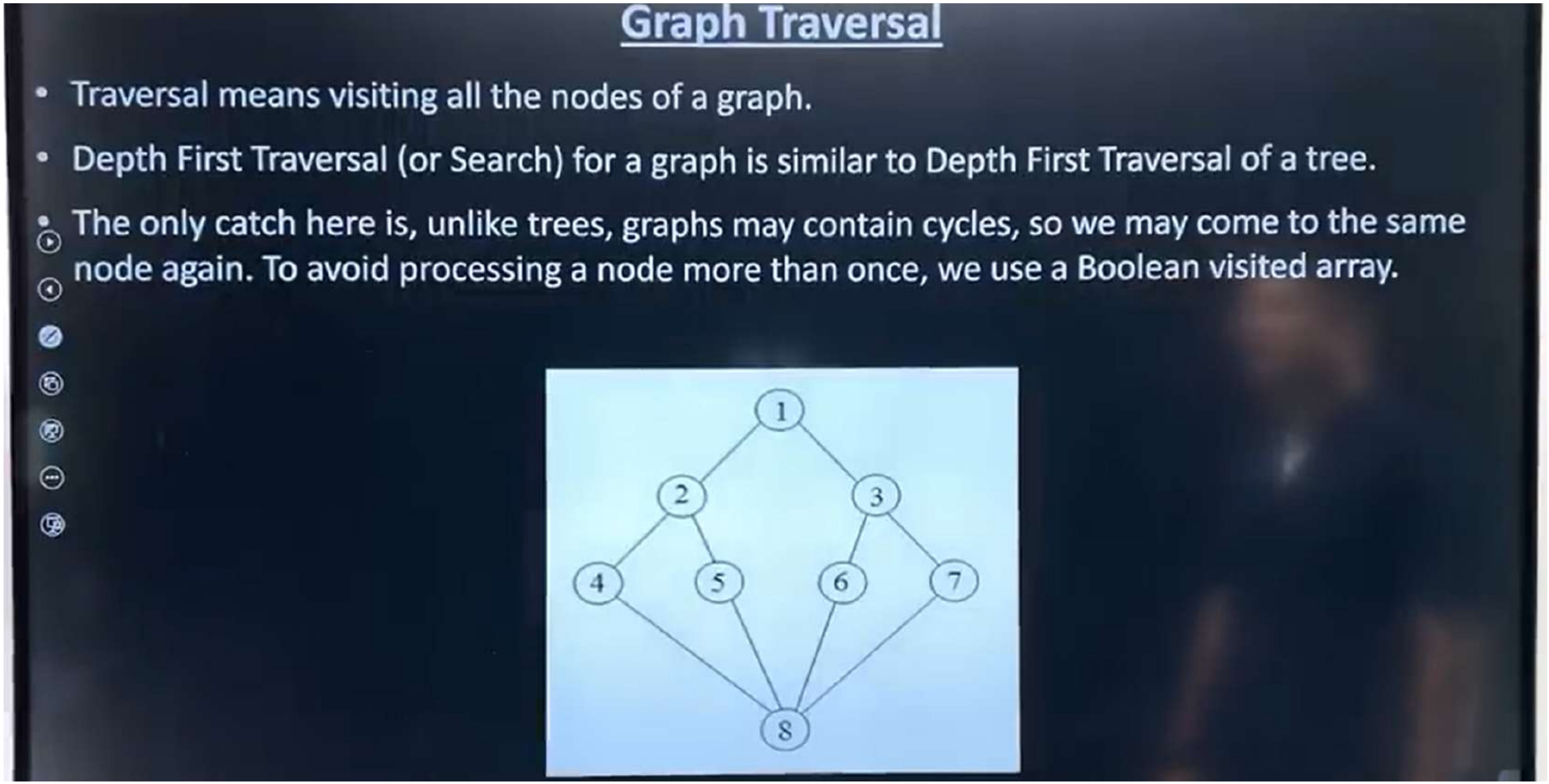
a. For non-weighted graph :

INFO	Adj-list

b. For weighted graph :

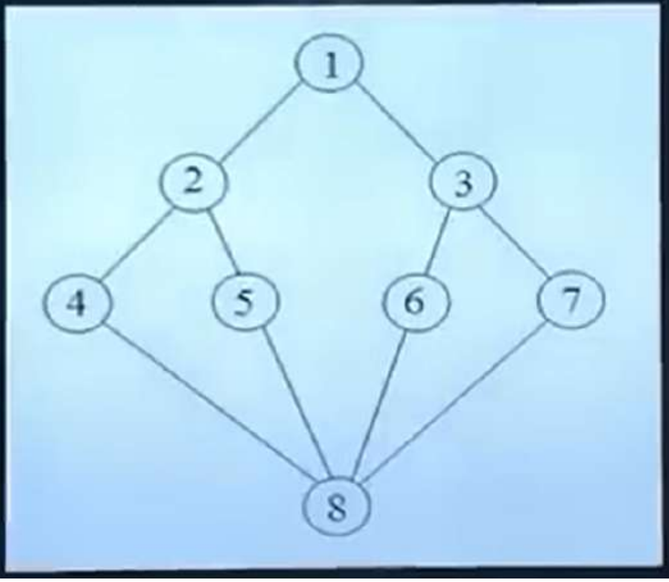
Weight	INFO	Adj-list

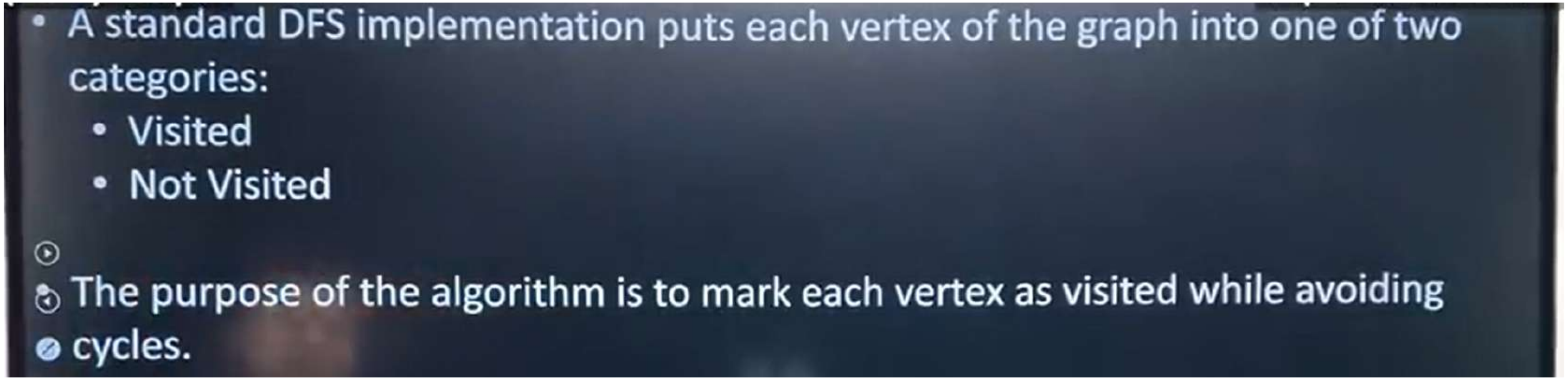




Graph Traversal

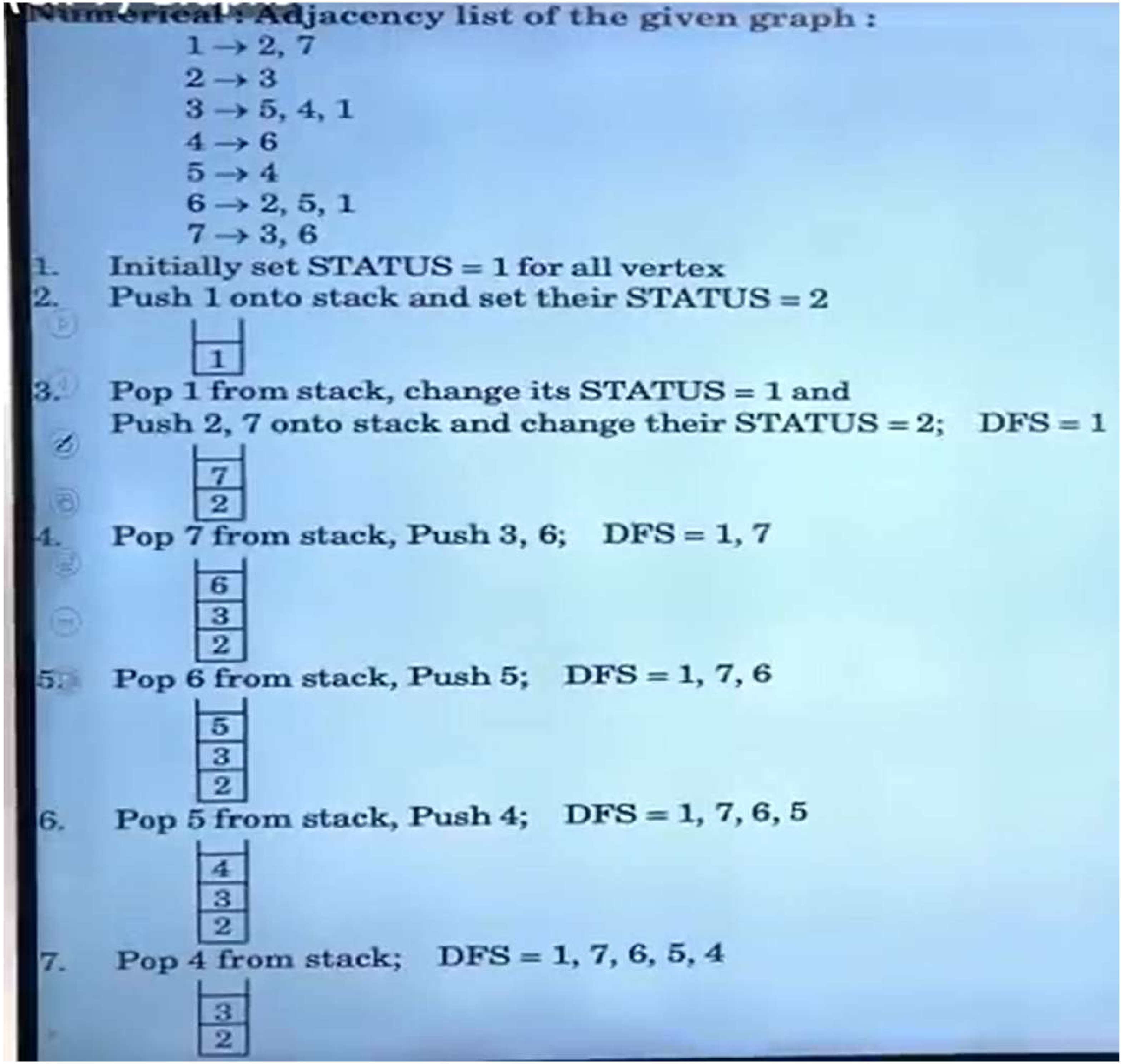
- Traversal means visiting all the nodes of a graph.
- Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree.
- The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array.





- A standard DFS implementation puts each vertex of the graph into one of two categories:
  - Visited
  - Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.



Numerical : Adjacency list of the given graph :

$1 \rightarrow 2, 7$

$2 \rightarrow 3$

$3 \rightarrow 5, 4, 1$

$4 \rightarrow 6$

$5 \rightarrow 4$

$6 \rightarrow 2, 5, 1$

$7 \rightarrow 3, 6$

1. Initially set STATUS = 1 for all vertex
2. Push 1 onto stack and set their STATUS = 2

$$\begin{array}{|c|}\hline 1\\ \hline\end{array}$$

3. Pop 1 from stack, change its STATUS = 1 and
   Push 2, 7 onto stack and change their STATUS = 2;   DFS = 1

$$\begin{array}{|c|}\hline 7\\ \hline 2\\ \hline\end{array}$$

4. Pop 7 from stack, Push 3, 6;   DFS = 1, 7

$$\begin{array}{|c|}\hline 6\\ \hline 3\\ \hline 2\\ \hline\end{array}$$

5. Pop 6 from stack, Push 5;   DFS = 1, 7, 6

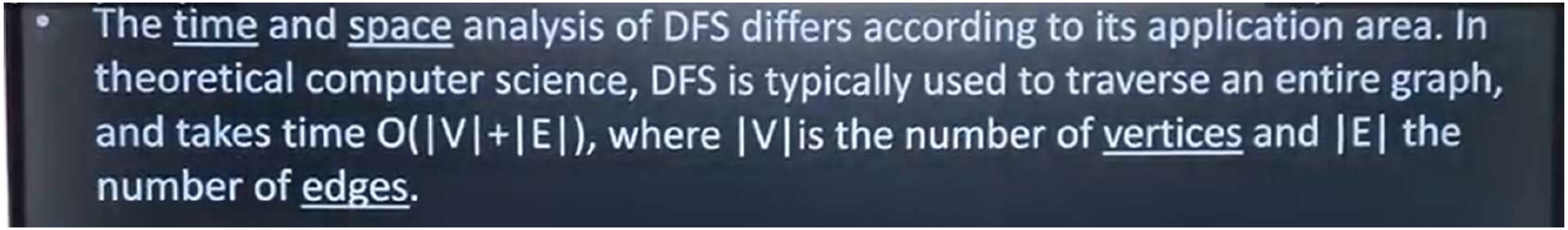
$$\begin{array}{|c|}\hline 5\\ \hline 3\\ \hline 2\\ \hline\end{array}$$

6. Pop 5 from stack, Push 4;   DFS = 1, 7, 6, 5

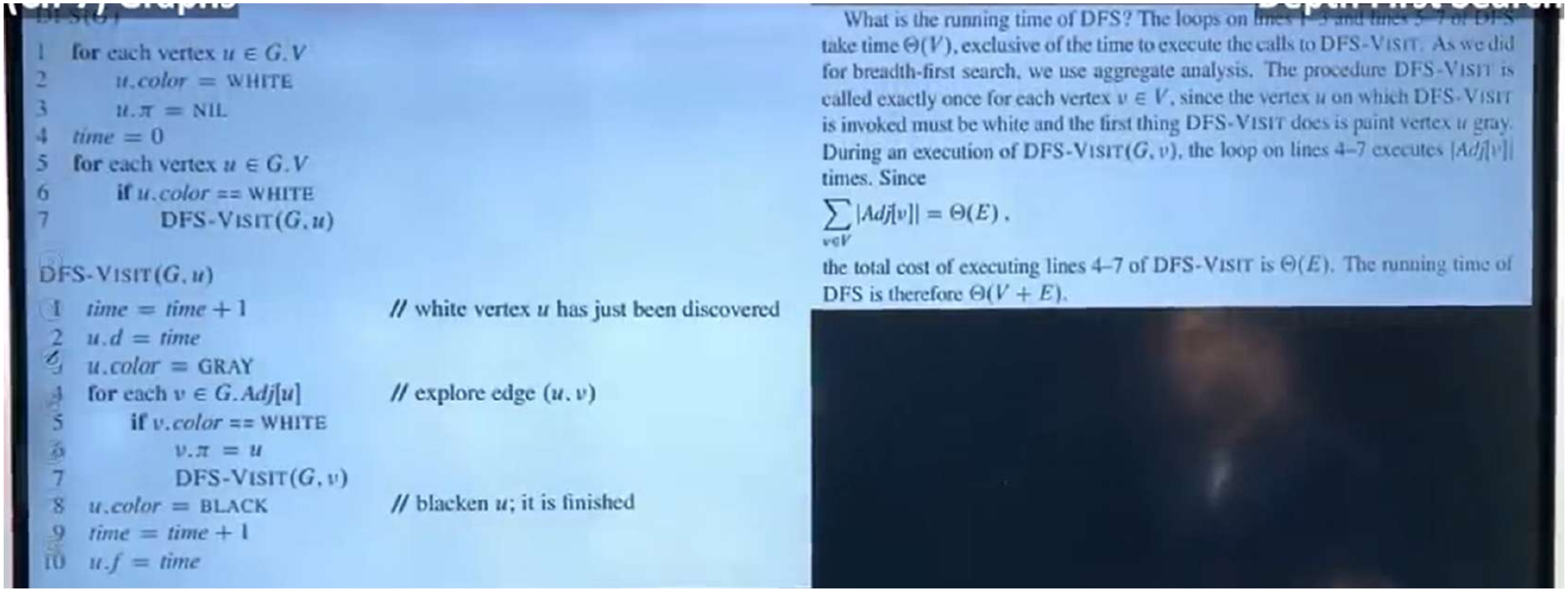
$$\begin{array}{|c|}\hline 4\\ \hline 3\\ \hline 2\\ \hline\end{array}$$

7. Pop 4 from stack;   DFS = 1, 7, 6, 5, 4

$$\begin{array}{|c|}\hline 3\\ \hline 2\\ \hline\end{array}$$



- The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time O(|V|+|E|), where |V|is the number of vertices and |E| the number of edges.



DFS(G)

```
1  for each vertex $u \in G.V$
2      $u.color$ = WHITE
3      $u.\pi$ = NIL
4  $time = 0$
5  for each vertex $u \in G.V$
6      if $u.color$ == WHITE
7          DFS-VISIT$(G, u)$
```

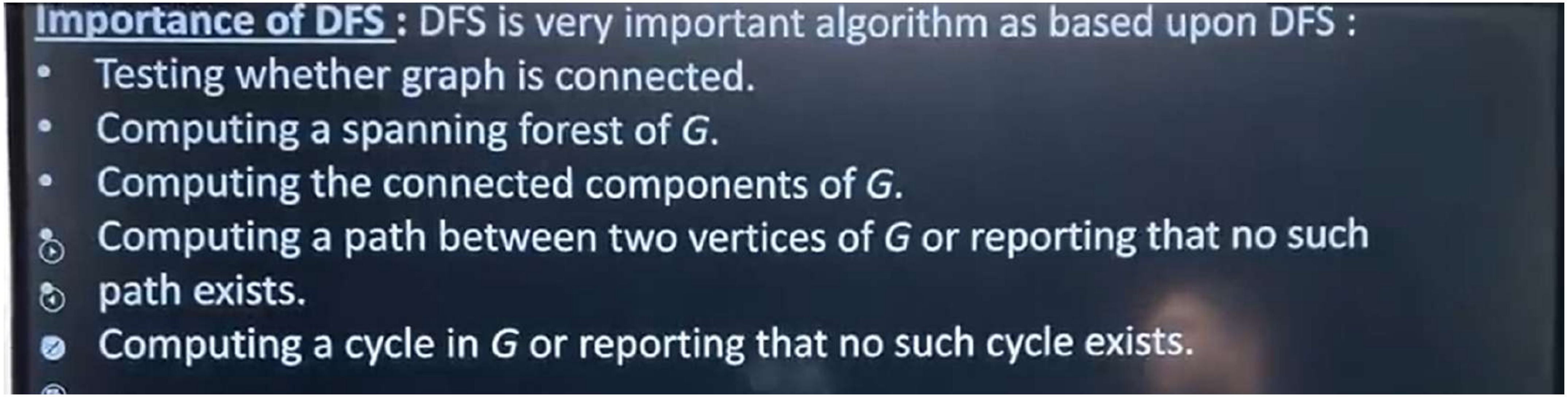
DFS-VISIT(G, u)

```
1  $time = time + 1$                  // white vertex $u$ has just been discovered
2  $u.d = time$
3  $u.color$ = GRAY
4  for each $v \in G.Adj[u]$          // explore edge $(u, v)$
5      if $v.color$ == WHITE
6          $v.\pi = u$
7          DFS-VISIT$(G, v)$
8  $u.color$ = BLACK                  // blacken $u$; it is finished
9  $time = time + 1$
10 $u.f = time$
```

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G, v), the loop on lines 4–7 executes $|Adj[v]|$ times. Since

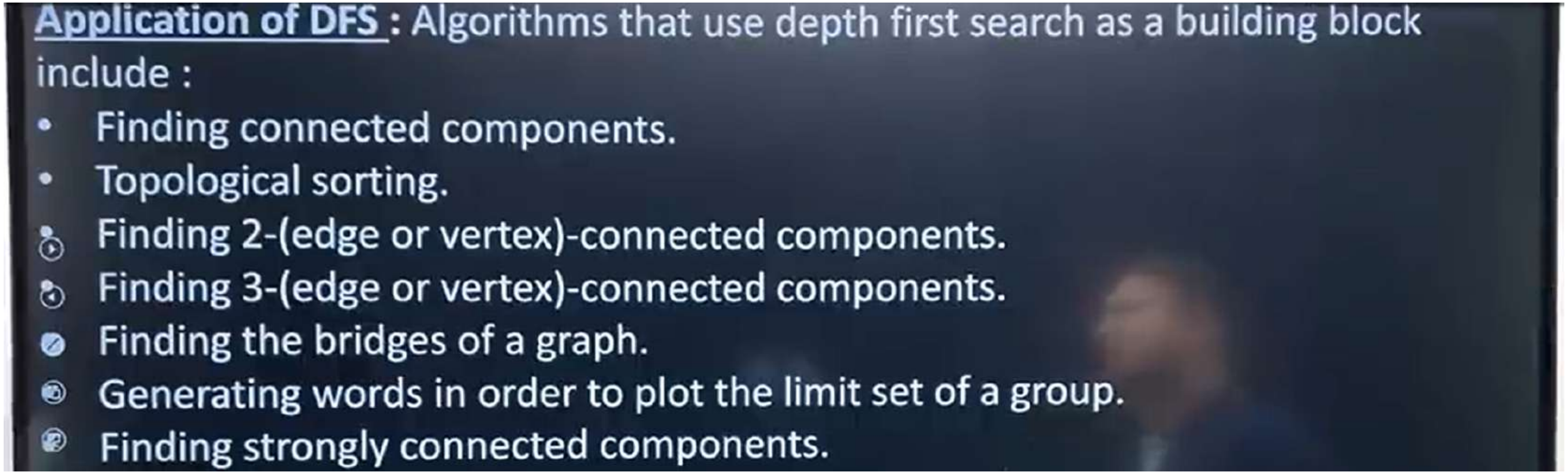
$$\sum_{v \in V} |Adj[v]| = \Theta(E) ,$$

the total cost of executing lines 4–7 of DFS-VISIT is $\Theta(E)$. The running time of DFS is therefore $\Theta(V + E)$.



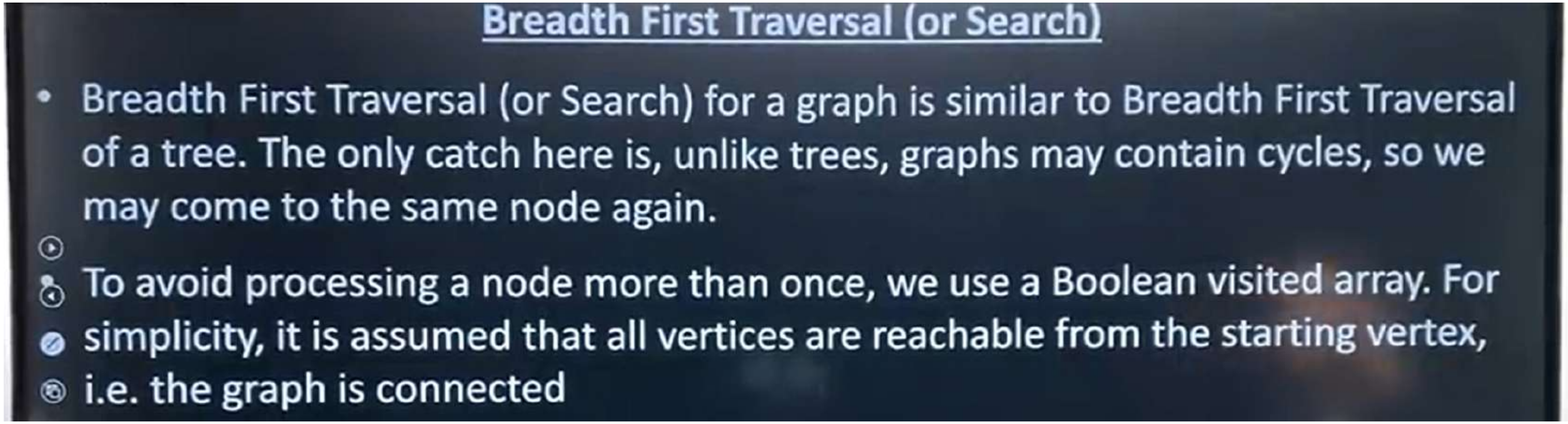
Importance of DFS : DFS is very important algorithm as based upon DFS :

- Testing whether graph is connected.
- Computing a spanning forest of *G*.
- Computing the connected components of *G*.
- Computing a path between two vertices of *G* or reporting that no such path exists.
- Computing a cycle in *G* or reporting that no such cycle exists.



Application of DFS : Algorithms that use depth first search as a building block include :

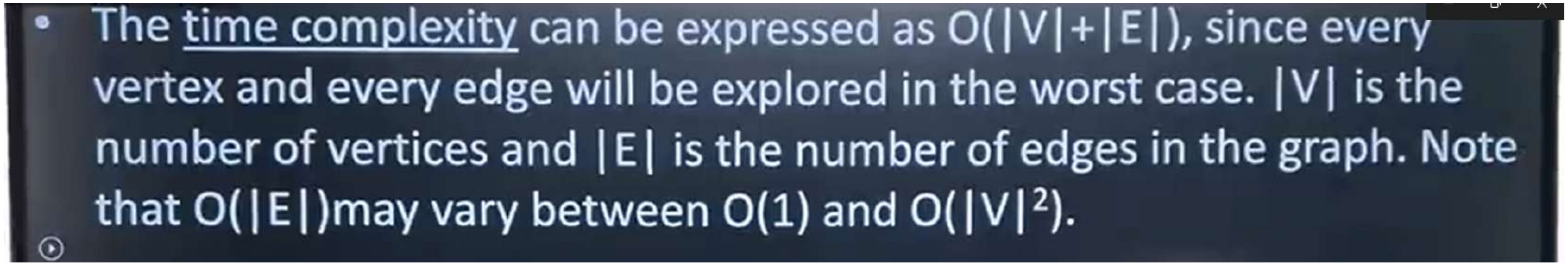
- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding 3-(edge or vertex)-connected components.
- Finding the bridges of a graph.
- Generating words in order to plot the limit set of a group.
- Finding strongly connected components.



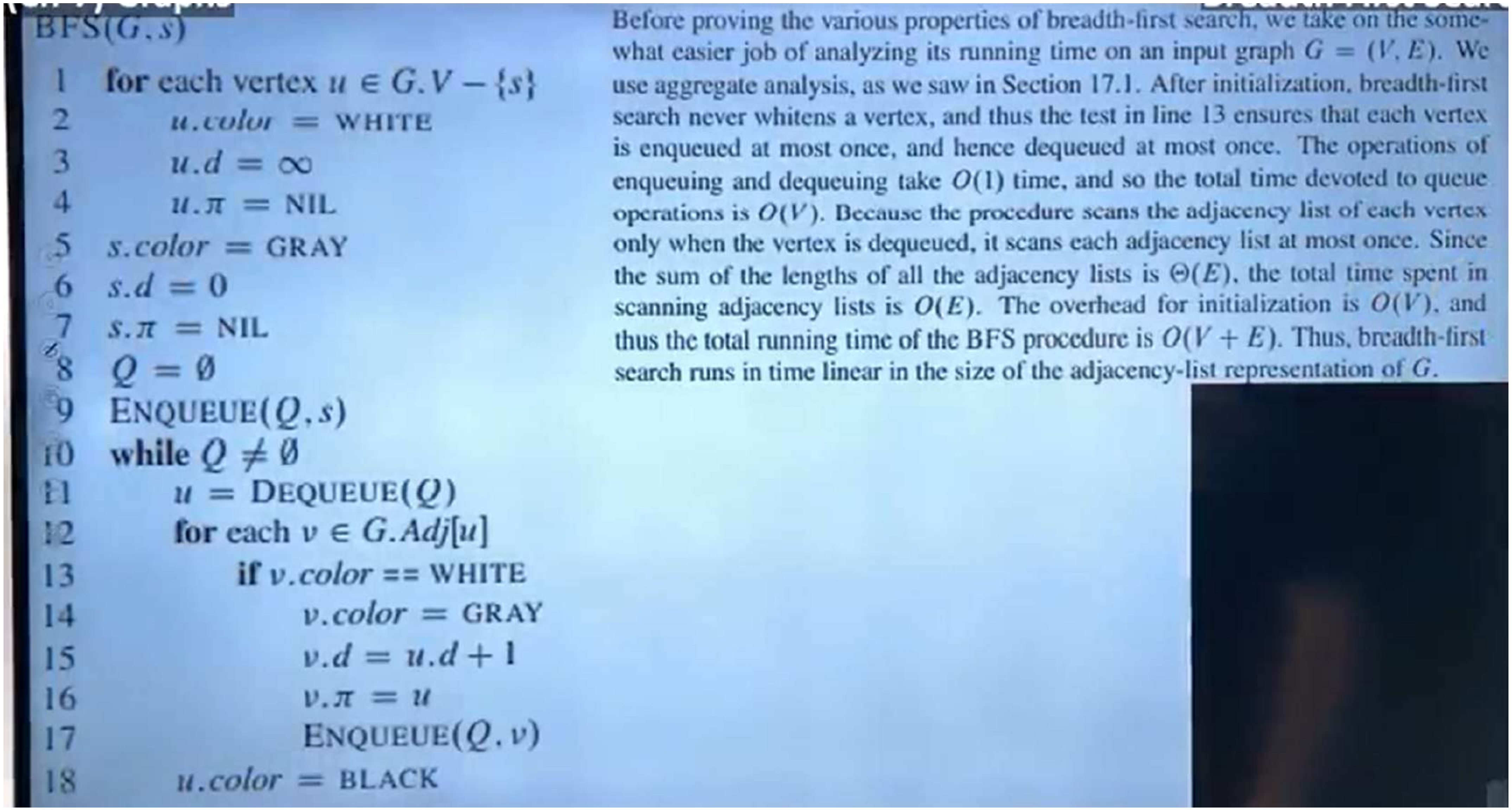
Breadth First Traversal (or Search)

- Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again.

- To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex, i.e. the graph is connected

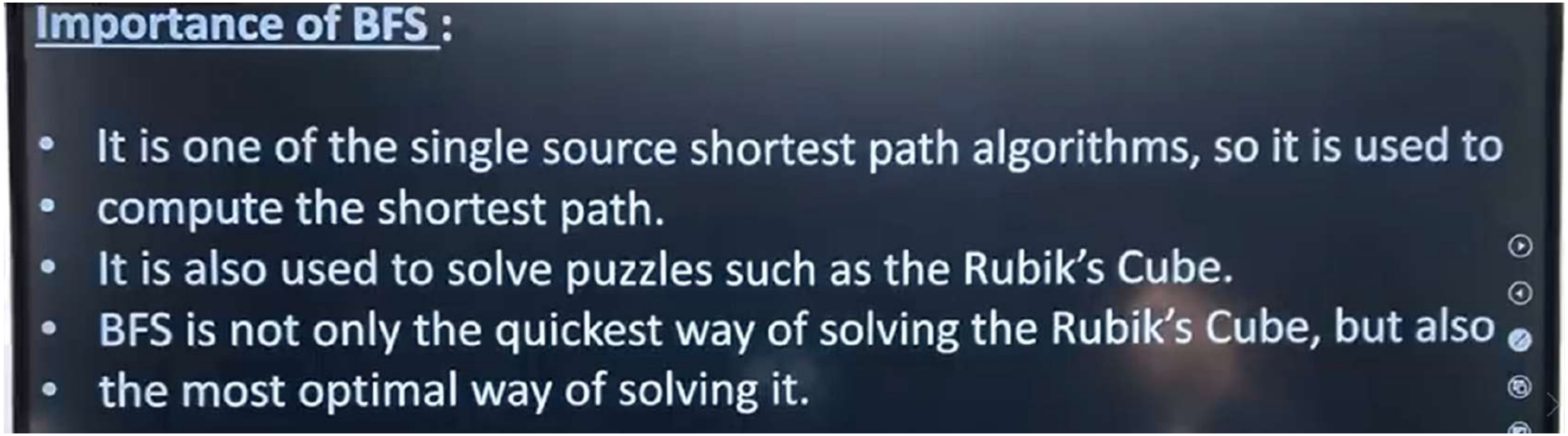


- The time complexity can be expressed as O(|V|+|E|), since every vertex and every edge will be explored in the worst case. |V| is the number of vertices and |E| is the number of edges in the graph. Note that O(|E|)may vary between O(1) and O(|V|^2).



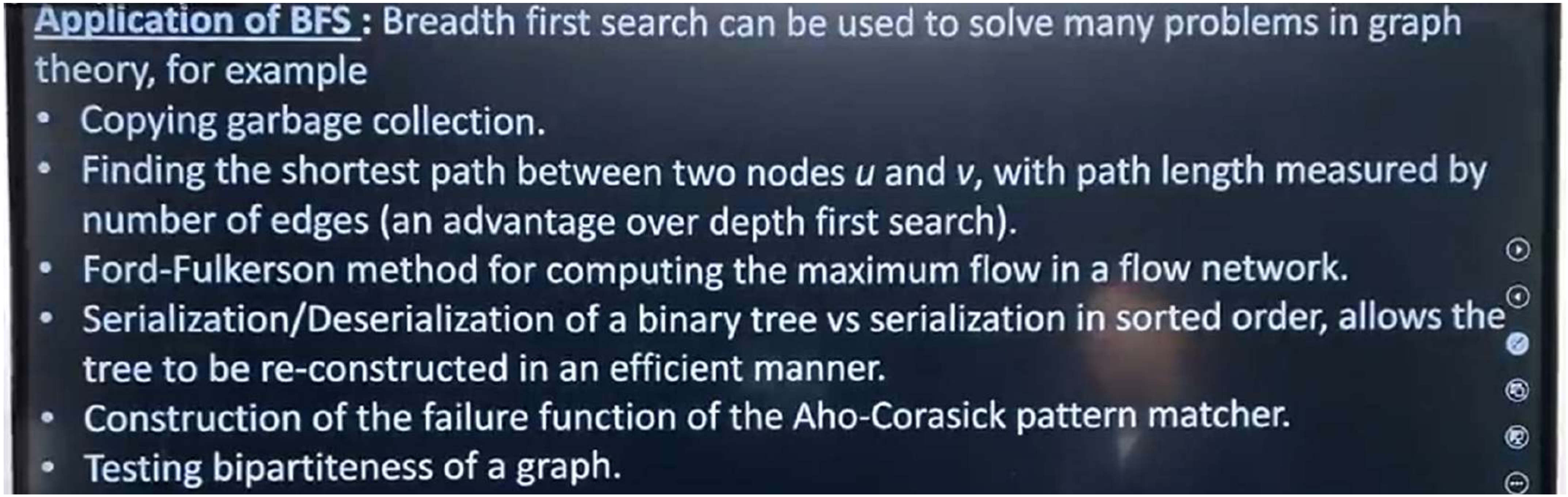
```
BFS($G, s$)
1  for each vertex $u \in G.V - \{s\}$
2      $u.color = \text{WHITE}$
3      $u.d = \infty$
4      $u.\pi = \text{NIL}$
5  $s.color = \text{GRAY}$
6  $s.d = 0$
7  $s.\pi = \text{NIL}$
8  $Q = \emptyset$
9  ENQUEUE($Q, s$)
10 while $Q \neq \emptyset$
11     $u = $ DEQUEUE($Q$)
12     for each $v \in G.Adj[u]$
13         if $v.color == \text{WHITE}$
14             $v.color = \text{GRAY}$
15             $v.d = u.d + 1$
16             $v.\pi = u$
17             ENQUEUE($Q, v$)
18     $u.color = \text{BLACK}$
```

Before proving the various properties of breadth-first search, we take on the somewhat easier job of analyzing its running time on an input graph $G = (V, E)$. We use aggregate analysis, as we saw in Section 17.1. After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.



Importance of BFS :

- It is one of the single source shortest path algorithms, so it is used to
- compute the shortest path.
- It is also used to solve puzzles such as the Rubik's Cube.
- BFS is not only the quickest way of solving the Rubik's Cube, but also
- the most optimal way of solving it.



Application of BFS : Breadth first search can be used to solve many problems in graph theory, for example

- Copying garbage collection.
- Finding the shortest path between two nodes *u* and *v*, with path length measured by number of edges (an advantage over depth first search).
- Ford-Fulkerson method for computing the maximum flow in a flow network.
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.
- Construction of the failure function of the Aho-Corasick pattern matcher.
- Testing bipartiteness of a graph.