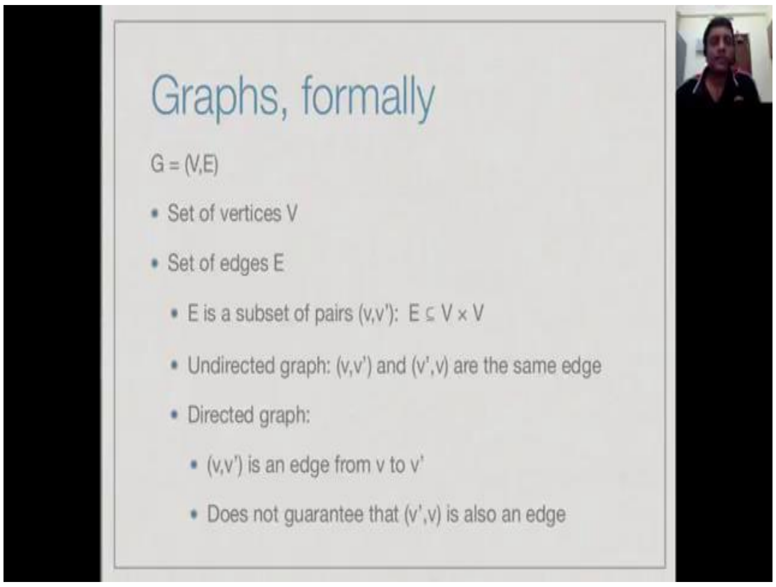
Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Week - 03
Module - 03
Lecture - 20

Breadth First Search (BFS)

(Refer Slide Time: 00:05)



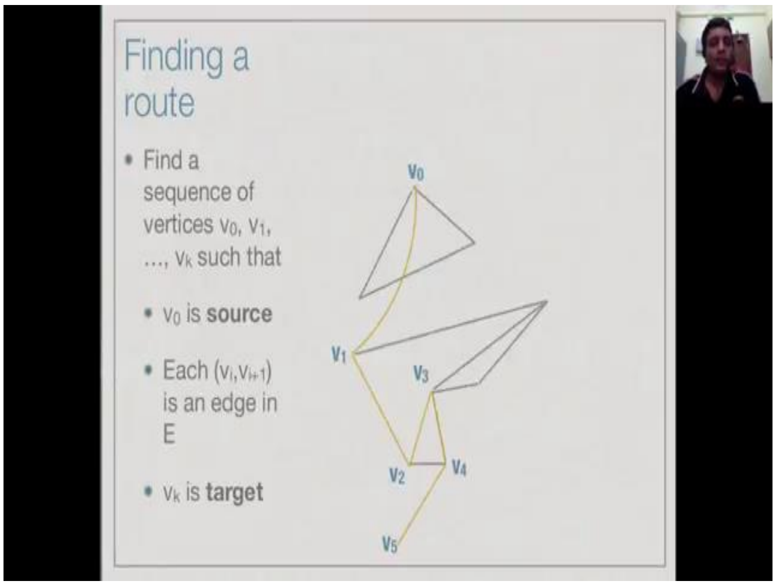
Graphs, formally

G = (V,E)

- Set of vertices V
- Set of edges E
  - E is a subset of pairs (v,v'): $E \subseteq V \times V$
  - Undirected graph: (v,v') and (v',v) are the same edge
  - Directed graph:
    - (v,v') is an edge from v to v'
    - Does not guarantee that (v',v) is also an edge

So, let us look at a formal algorithm to explore a graph. So, recall that a graph consists of a set of vertices and a set of edges, the edges are the connections between the vertices, they may be directed or undirected.

(Refer Slide Time: 00:15)

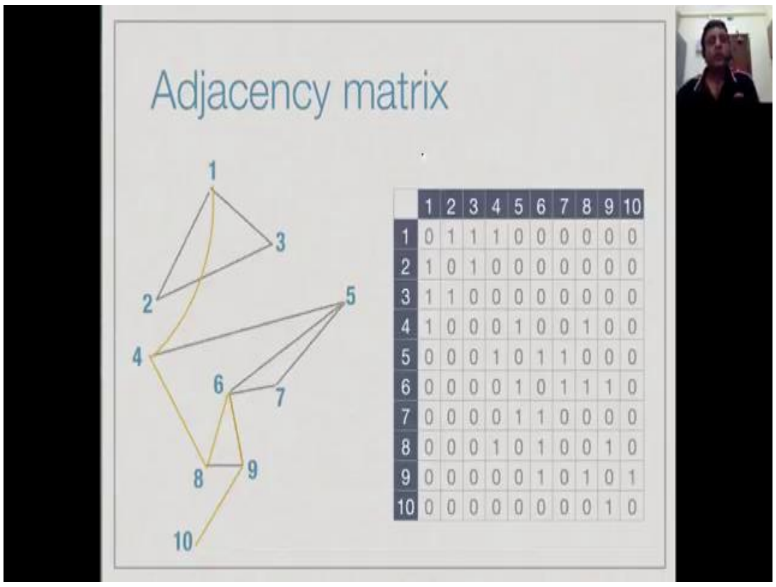


Finding a route

- Find a sequence of vertices $v_0, v_1, \ldots, v_k$ such that
  - v_0 is **source**
  - Each (v_i, v_{i+1}) is an edge in E
  - v_k is **target**

So, if we look at it undirected graph, the problem we are looking at is to find out, whether the source vertex is connected to a target vertex. And we said that this amount to finding a path from v 0 the source vertex to v k the target vertex, where each pair of vertices on the path is connected by an edge in the graph.

(Refer Slide Time: 00:35)



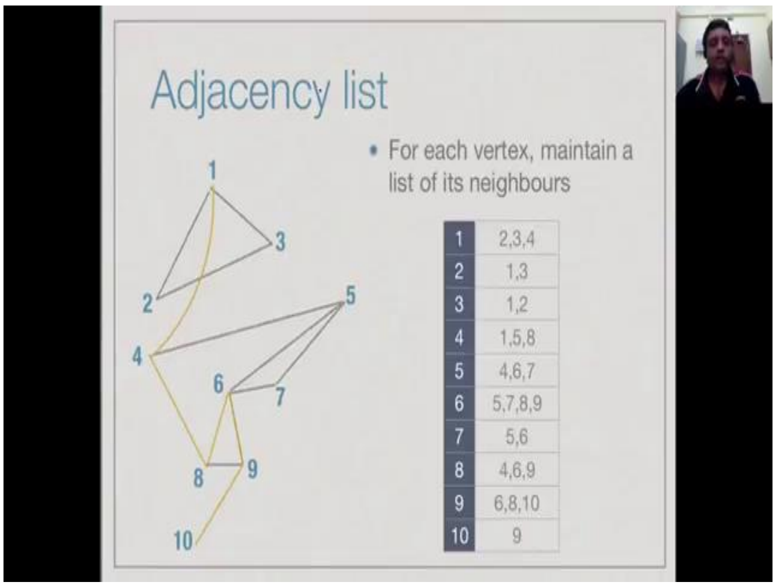
Adjacency matrix

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

So, we argued that we could easily do this for small graphs visually, but if you want to write an algorithm, we need a way to representing the graph. So, the first thing we decided was, that we would name the vertices 1 to n. So, we have n vertices, we will just call them 1, 2, 3, 4 up to n. Then we can represent to structure of the graph through an adjacency matrix.

In this adjacency matrix, the i j'th entry indicates the presents or absents of an edge from vertex i to vertex j. So, a i j is 1, there is an edge, a i j is 0, if there is no subject. In this matrix, if you want to find out, whether pair of vertices are directly connecting, we just have to prove their appropriate entry, we can do that in constant time. To find all the outgoing neighbors of a vertex, we have to scan the row for that vertex. So, that takes linear time in terms of the number of vertices. Now, one thing we observed is that, typically many graphs, this matrix is largely 0, most pairs are not connected.

(Refer Slide Time: 01:36)



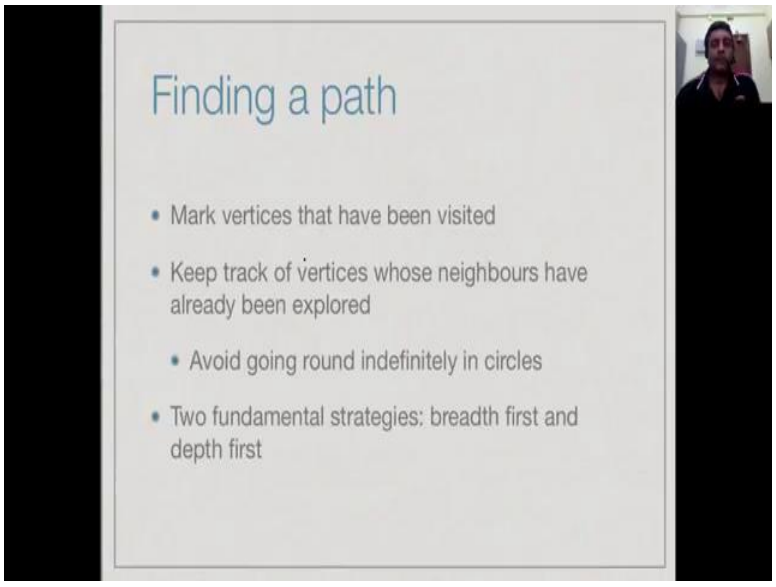
Adjacency list

- For each vertex, maintain a list of its neighbours

1	2,3,4
2	1,3
3	1,2
4	1,5,8
5	4,6,7
6	5,7,8,9
7	5,6
8	4,6,9
9	6,8,10
10	9

So, we can get a more compact representation by this listing out the neighbors of each vertex. So, instead of keeping a row of 1's and 0's, we just record those vertices, whose entries are 1. So, this keeps as a more efficient representation for graphs, where the number of edges is closer to the number of vertices. But, in this case in order to find out, whether a given pair i j is connected, we have to look at the list for i and see, if j appears in it.

(Refer Slide Time: 02:00)

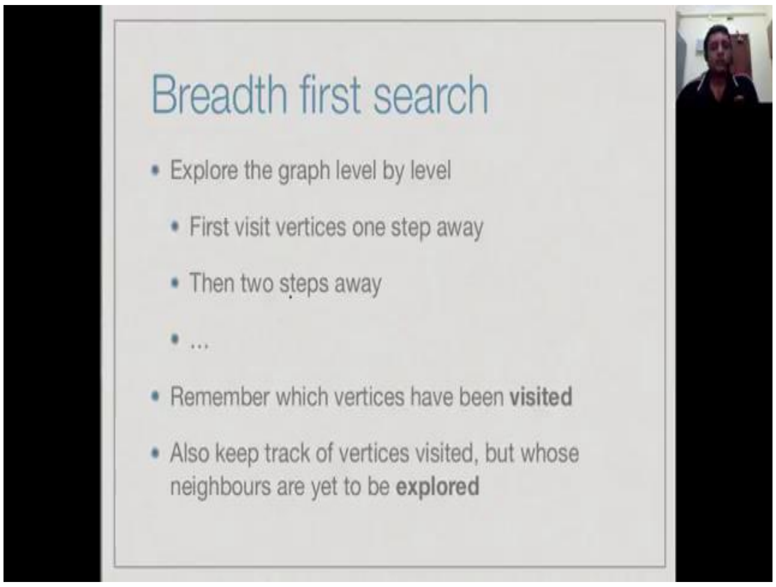


Finding a path

- Mark vertices that have been visited
- Keep track of vertices whose neighbours have already been explored
  - Avoid going round indefinitely in circles
- Two fundamental strategies: breadth first and depth first

Now, our strategy for finding a path connecting a source vertex in a target vertex, which is follows, we would start at the source vertex and keep exploring the graph systematically. Each time, we went to a vertex, you would mark it as visited, and then we have to make sure, that when you are exploring vertices, we did not re explore the same vertex twice. So, we said they would be two fundamental strategies breadth first and depth first. So, today we are going to look at breadth first search.

(Refer Slide Time: 02:28)



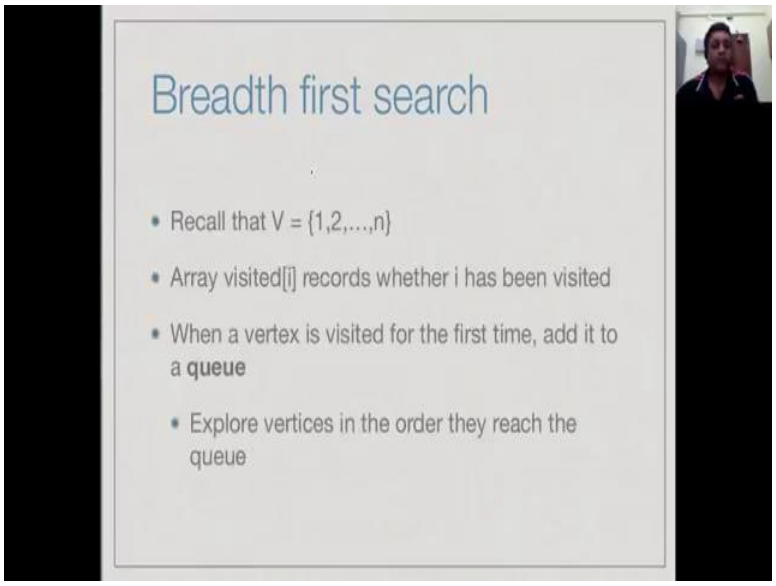
Breadth first search

- Explore the graph level by level
  - First visit vertices one step away
  - Then two steps away
  - ...
- Remember which vertices have been **visited**
- Also keep track of vertices visited, but whose neighbours are yet to be **explored**

So, the idea breadth first search is to explore the graph level by level. So, we start at the source vertex and we now explore all the vertices, which are one step away, connected by a direct edge to the source vertex. Then, we leak out all vertices which are newly connected one step away from level 1 vertices. So, these are two steps away from the source vertex, then for three steps and so on.

Now, while we are doing this set duration, we have to keep track of two quantities, we have to first note of course, whether a given vertex has been visited. Has it been explored, is it connected to the source vertex, also for each such vertex, we have to explore it is neighbors. So, for each vertex, we need to know, a has it been visited and we may not be able to explored it as soon as it is visited. So, we need to remember that we have explored it or not explored it and those which are not explore, we have to make sure we explore them once.

(Refer Slide Time: 03:26)



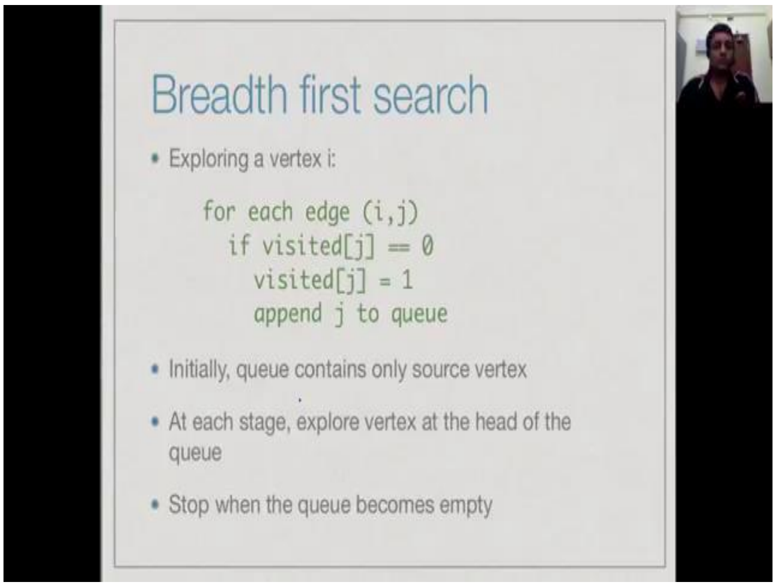
Breadth first search

- Recall that V = {1,2,...,n}
- Array visited[i] records whether i has been visited
- When a vertex is visited for the first time, add it to a **queue**
  - Explore vertices in the order they reach the queue

So, the way to do this is to use some data structures to keep track of these quantities. As we said the set of vertices is numbered 1 to n. So, we can keep an array visited with entries 1 to n, which tells us whether do not vertex i has been visited. Now, when I will be visit a vertex, we need to subsequently explore it. Since, we may not explore it immediately, we may have to keep it pending for a while. So, we have to keep a collection of these vertices, which have been visited, but not yet been explored.

There are many ways to do this, but one natural way is to explore them in the order in which they will visited. So, we want to explore in the order in which they will visited, a natural data structure to keep the list of unexplored, but visited vertices is a queue. So, we put each visited vertex into the queue, the first time we come to it, and then we process all the vertices in the queue, until all vertices have been explored.

(Refer Slide Time: 04:30)



Breadth first search

- Exploring a vertex i:

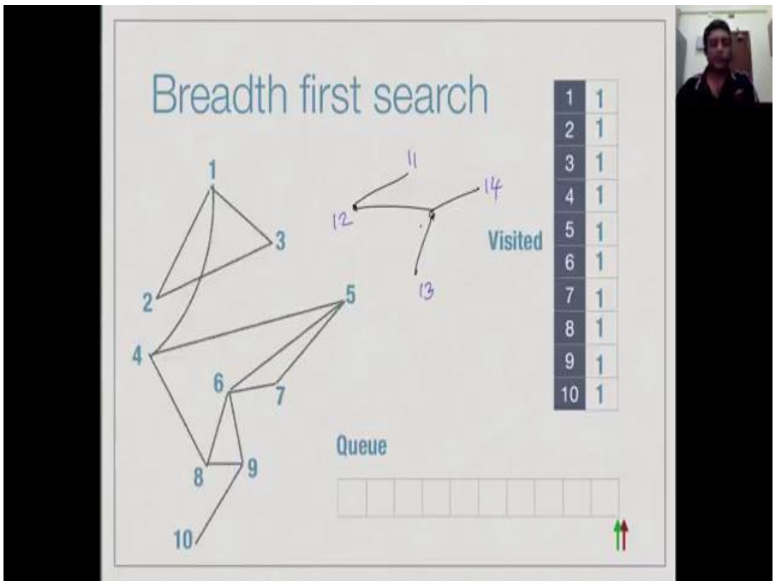
```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```

- Initially, queue contains only source vertex
- At each stage, explore vertex at the head of the queue
- Stop when the queue becomes empty

So, else some high level Pseudo code for breadth first search, when we reach a vertex i which we have visited for the first time, we will eventually explore it. What is it mean to explore a vertex, we look at the outgoing edges i comma j, if j are already been visited, we are nothing to do. On the other hand, if j is not been visited, then we will mark it as visited and add it the queue to be explore later.

So, if begin by marking the source vertex and adding it to the queue and at each stage, we will explored the vertex of the head of the queue. If at some stage, there are no vertices in the queue, it means that every vertex, we have visited have been explored and so there is no new information left and we can stop the exploration.

(Refer Slide Time: 05:10)



Breadth first search

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

So, before giving the actual code for breadth first search, let us workout the example that we have and see, how this data structures are updated. So, we have these 10 vertices 1 to 10 and we have looking for path starting it 1. So, in the queue, we have two pointers as usual, we have a pointer which we indicate in green, which is the head of the queue, in a pointer red, which is tail of the queue.

So, we initialize the queue to have the starting vertex 1 and we also mark this vertex has been visited. Now, we have to explore 1, so what we do is we remove 1 from the queue and systematically check each of it is outgoing edges and if those target vertices are not already visited, we add that. So, two is the first in order of 1 to 10, first neighbor of 1, it is not been visited, so we add it to the queue and mark to the visited in the visited array.

Likewise, we mark 3 is visited, and then we mark 4 is visited, at this state, we finished exploring vertex 1. So, now, we go back to the queue and see, if there is anything yet to be explore, which have been visited and there is we pick up the first search namely 2. So, I remove 2 from the queue and we explore it is outgoing neighbors. Now, the outgoing neighbors of 2 in this case are 1 and 3. But, since 1 and 3 are both visited, we have no work to do and we go on and go back to the queue.

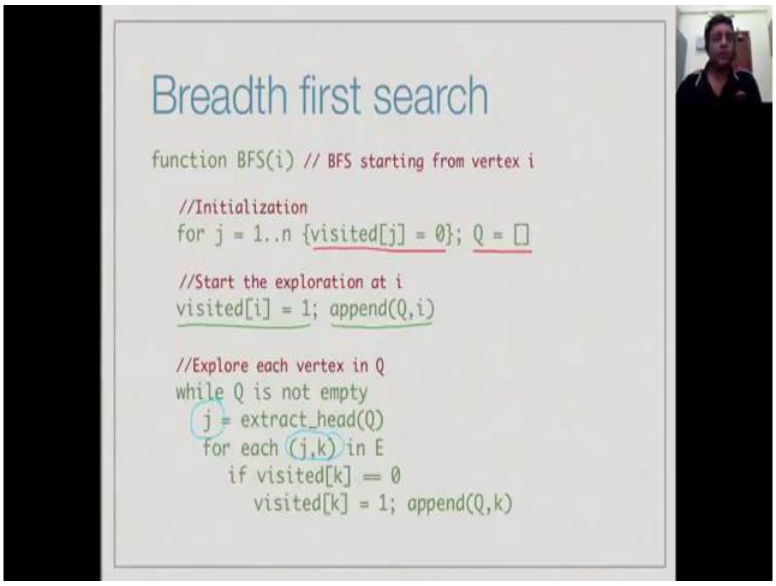
For in the next stage, we pick up the vertex 3 and we look at it is outgoing neighbors. Again, it has outgoing neighbors 1 and 2, both the feature already visited, so there is nothing to be done. We come back to the queue and pickup 4 and try to explore it. So, 4 has neighbor 1, which is already been visited, the next neighbor is 5, since 5 is new we added to the queue and we mark it as visited, then it has a neighbor 8. So, we add that also to the queue and mark it as visited.

Now, we are finished exploring 4. So, we go back to the queue and look for the next thing to be explore, which in this case is 5. So, 5 as got neighbors 4, 6 and 7; 4 is already been visited, but 6 is new and so is 7. Now, we go back and explore 8, so 8 has neighbors 4, 6 and 9; 4 and 6 are old, but 9 is new, we add 9 to the queue. Now, we go back and explore 6, because 6 is the vertex of the head of the queue.

So, now, we look at neighbors of 6, neighbors of 6 are 5, 7, 8 and 9. So, when we remove 6 from the queue, we are nothing to do, we go back and find that, there is nothing to be done, because all the neighbors of 6 have already been visited. Then, we pick up 7; once again we find that all the neighbors of 7 have already been visited. Finally, we pickup 9 and we pickup 9, we find that it has neighbor 6, 8 and 10, 10 is new, so 10 gets mark and add it to the queue.

Finally, we go to 10, we find it has only one neighbor 9, which is already visited, at this stage the queue becomes empty and the algorithm terminates. Now, in this case, we have marked all the vertices are visited, which indicates that every vertex in this lab was actually reachable from that. If we had add more vertices, for example, if we had add a component in this graph; that is nothing to stop us from having many components to be could had a graph, it more vertices 11, 12, 13 and 14. Then this process starting from 1 would have left 11, 12, 13 and 14 and marked as unvisited.

(Refer Slide Time: 08:54)



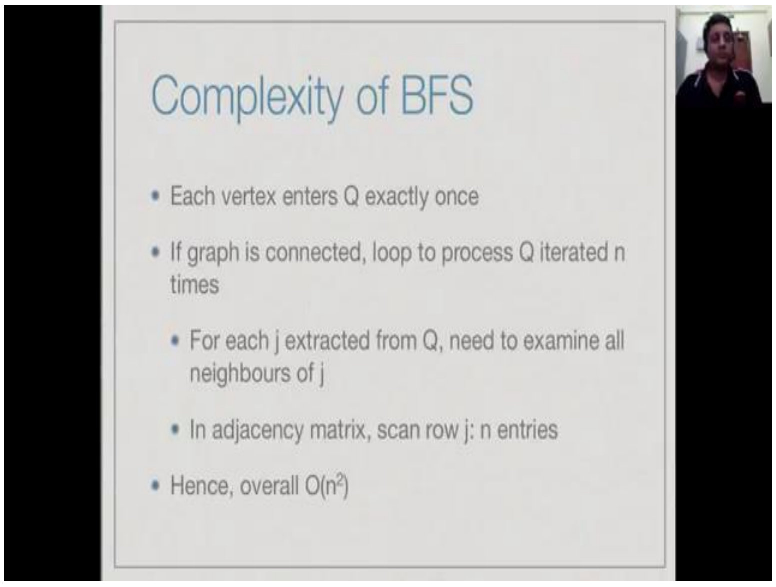
Breadth first search

```
function BFS(i) // BFS starting from vertex i

  //Initialization
  for j = 1..n {visited[j] = 0}; Q = []

  //Start the exploration at i
  visited[i] = 1; append(Q,i)

  //Explore each vertex in Q
  while Q is not empty
    j = extract_head(Q)
    for each (j,k) in E
      if visited[k] == 0
        visited[k] = 1; append(Q,k)
```

So, here is some more formal code for BFS, so BFS is a function which takes as it is argument, the vertex from where we want to explore the graph. So, as we said the vertices are called 1 to n, we have this array visited, which is initialize to 0 and we have a queue to mark the vertices still to be explored, this is initialized to empty. So, we begin by marking that visited of i equal to 1. So, vertex i is mark that visited and we added to the queue.

Now, so long as a queue is not empty, we proceed in the following loop, we extract the head of the queue, the vertex to be it is it explored next. For each of it is outgoing edges, if it is something which has not been visited before, we mark it as visited and we add it to the queue. So, this is exactly the code for the algorithm, we just execute by hand and it is quiet straight forward. There are two data structures and array to keep track of the visited vertices and a queue to keep track of vertices which need to be explored file.

(Refer Slide Time: 10:00)



Complexity of BFS

- Each vertex enters Q exactly once
- If graph is connected, loop to process Q iterated n times
  - For each j extracted from Q, need to examine all neighbours of j
  - In adjacency matrix, scan row j: n entries
- Hence, overall $O(n^2)$

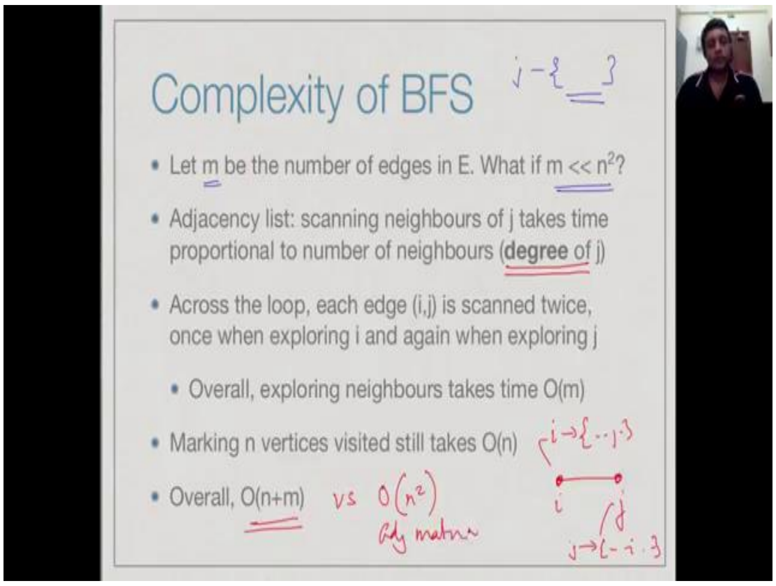
So, how do we analyze the complexity of an algorithm ((Refer Time: 10:05)), one way to do it is to look at the loop. So, remember that, when we have an algorithm like this ((Refer Time: 10:10)) which has the loop, the loop is usually the place where we have to look for it. So, we have one iteration, where we have the loop of 1 to n; where we assign, this is it j equal to 0. So, this is something which takes order n time.

And now, we have a loop here, where we keep extracting things from the queue and we do things here. So, we have to kind of understand, how many times this loop is going to execute and how much time this inner loop going to execute. So, each vertex enters the queue exactly once. So, it we enter at exactly once, ((Refer Time: 10:47)) then it says that this outer loop here will take order n time.

Because, assuming the graph is connected, every vertex will be visited once and when that vertex will visited to enter the queue and it will come out in the head of the queue exactly once. Now, for each vertex, we have to scan all it is neighbors. So, what we said was that, this to be order n, if we have an adjacency matrix. Because you need to go from the breadth first entry for j and we have to look at the entire row for this, so this is order n.

So, therefore, the graph is connected first of all every vertex will get into the queue. So, we will do the queue loop exactly n times. And for each j that is extracted, we need to examine all it is neighbors. So, that will be a scan of size n and therefore, overall it is order n square, this two nested loops are size n.

(Refer Slide Time: 11:43)



Complexity of BFS

- Let $\underline{m}$ be the number of edges in E. What if $\underline{m << n^2}$?
- Adjacency list: scanning neighbours of j takes time proportional to number of neighbours (**degree** of j)
- Across the loop, each edge (i,j) is scanned twice, once when exploring i and again when exploring j
    - Overall, exploring neighbours takes time O(m)
- Marking n vertices visited still takes O(n)
- Overall, $\underline{O(n+m)}$ vs $O(n^2)$

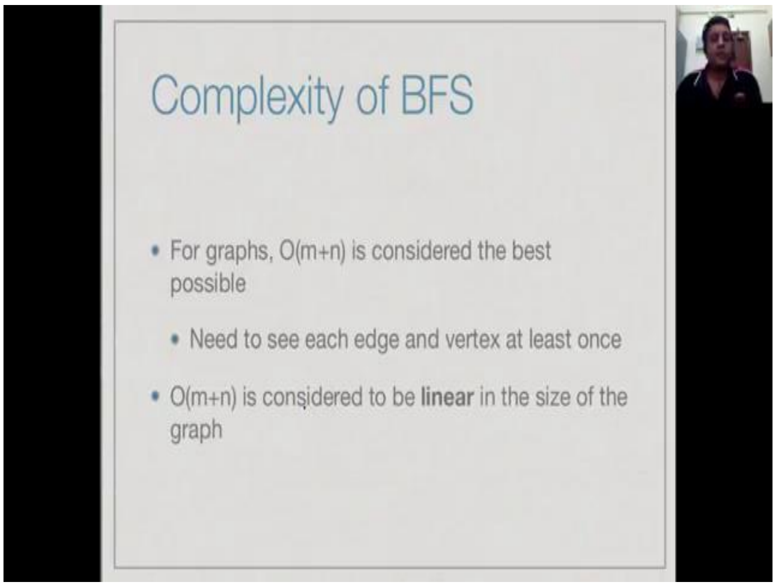
So, this is clearly a situation, where if we have very few edges, we can improve the complexity by using an adjacency list representation is in the adjacency matrix representation. So, the number of edges is m and m is much smaller than n squared. So, if m it say proportional to n itself, then what we can do is in the list associated with j, which is have to look at the neighbors of j. We do not have to look at all the vertices all of order n entries in your adjacency matrix, this is called a degree.

So, the degree of a vertex is a number of vertices connected to it. So, if we do this, then what we will find is that, inside the iteration of the inner loop, each step of scanning for the neighbors takes time proportional degree of j. Now, the degree of j of course, varies from one node to another, so how do we count this correctly. So, if you do not count it paired loop, but count it across the entire processing of the entire cycle vertices, what we will see is that for every edge in the graph, the form i comma j.

There will be a time and i'st process and I will see j a list and there will be a time and j'st processed and I will see i in is this. In other words, if I take all the adjacency list of all the vertices, each edge will be represented twice and an edge i j will appeared as an entry j and the list for i and an entry i and the list for j. So, while, it is not easy to count per iteration, I count across all iterations, I will explore each edge exactly twice.

So, 2 m is ordered m, so overall exploring the neighbors across all n iteration takes times O m. So, what this means is that, the first iteration of visiting the marking everything visited takes time more m, when we have a O n loop, across which we do O m steps totally across the n steps, so it is overall n plus m. So, the complexity of the breadth first search, if you are using an adjacency list, it drops to O n plus m versus order n squared for the adjacency matrix. So, if we have fewer edges then it makes a lot of sense to use adjacency list for the breadth first search.

(Refer Slide Time: 14:08)

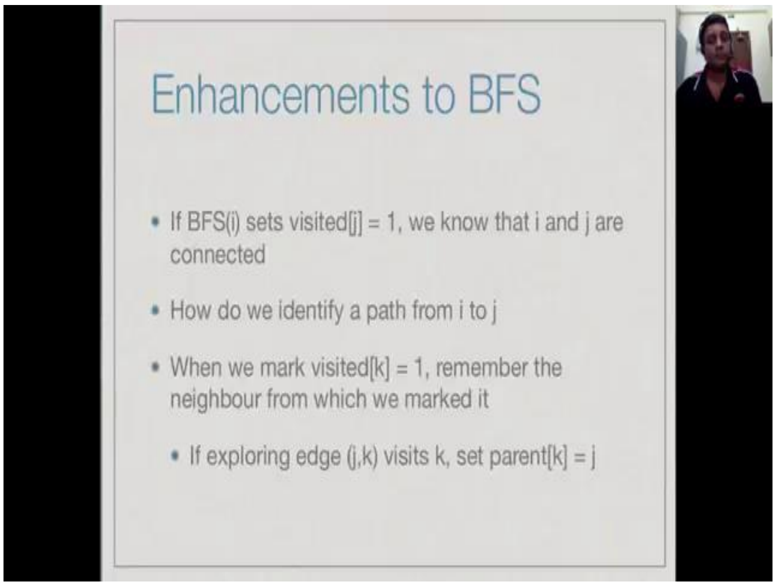


Complexity of BFS

- For graphs, O(m+n) is considered the best possible
  - Need to see each edge and vertex at least once
- O(m+n) is considered to be **linear** in the size of the graph

Now, in graphs the input size is typically taken to the m and n. In other words, both parameters are somewhat independent of each other; because we could have the set of vertices could very few edges with very many edges. So, usually m and n together are taken to be the input size. So, order m plus n is actually a linear algorithm. So, this is for best possible, because we have to read the input not at to process it typically. So, in linear time, were able to do breadth first search.

(Refer Slide Time: 14:40)



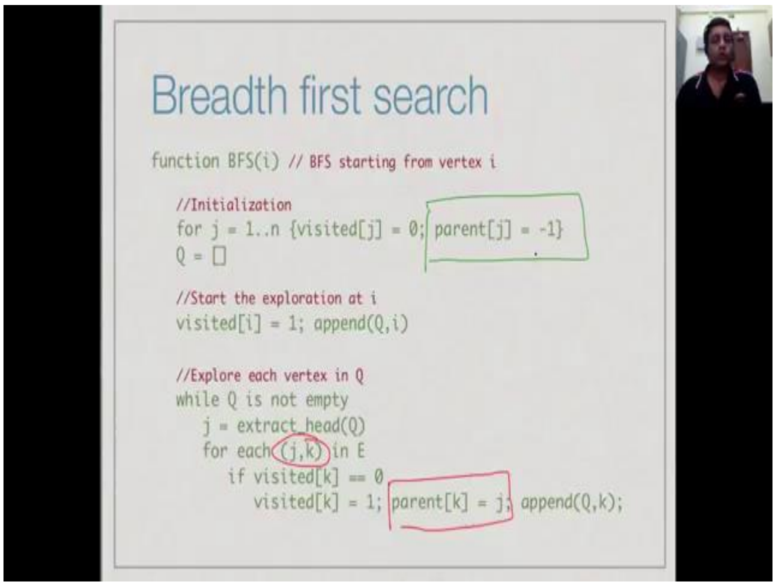
Enhancements to BFS

- If BFS(i) sets visited[j] = 1, we know that i and j are connected
- How do we identify a path from i to j
- When we mark visited[k] = 1, remember the neighbour from which we marked it
    - If exploring edge (j,k) visits k, set parent[k] = j

So, we can do something more in breadth first search, all we have identified is which vertices are connected to the usual vertex one of the source vertex in general from where you start breadth first search. So, how do we identify, how to go from the source vertex for given vertex. So, if we started i and BFS i sets visited j equal to 1, we know that i and j are connected, but we do not know the path.

So, what we can easily do is to remember, where we marked each vertex from. So, when we mark a vertex k as visited, it is marked because it is a neighbor of some already visited neighbor j. So, when we can say that k was marked i j, we will use the word parent. So, we will say that the parent of k is j, when but following the parent links; we can re construct the path backwards from k to the original source vertex.

(Refer Slide Time: 15:38)

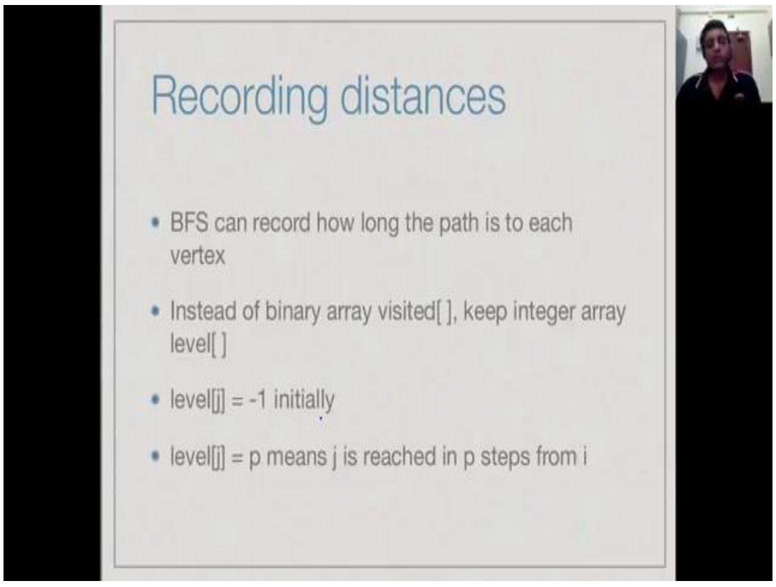


Breadth first search

```
function BFS(i) // BFS starting from vertex i

   //Initialization
   for j = 1..n {visited[j] = 0; parent[j] = -1}
   Q = []

   //Start the exploration at i
   visited[i] = 1; append(Q,i)

   //Explore each vertex in Q
   while Q is not empty
      j = extract_head(Q)
      for each (j,k) in E
         if visited[k] == 0
            visited[k] = 1; parent[k] = j; append(Q,k);
```

So, this is very easy to do an app code, so we just have to add one line, so we two line. So, initially we say that for every vertex the parent is undefined. So, we had said a value like minus 1, because we note that the name of vertices are 1 to n, and then when we mark a vertex, since we are exploring an edge j k, the parent k is going to be j. So, we assign the parent of k equal to j, the rest of the code is identical to the earlier breadth first search, you just add it this new array parent to keep track of how we mark the vertices. So, that we can re constructs the path.

(Refer Slide Time: 16:16)



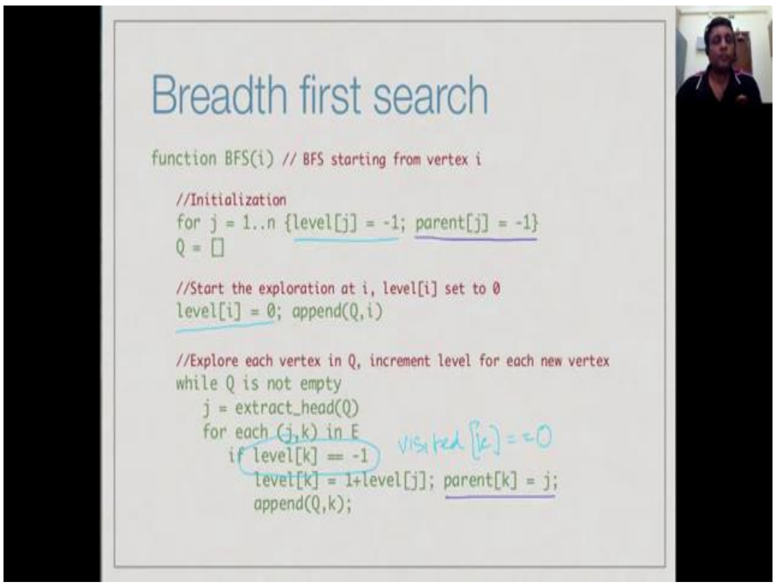
Recording distances

- BFS can record how long the path is to each vertex
- Instead of binary array visited[ ], keep integer array level[ ]
- level[j] = -1 initially
- level[j] = p means j is reached in p steps from i

So, we can go backward, so because k was mark by j, j would be mark by some other thing and so on. So, by this following these links back, we can go back from any k to this path vertex and therefore, we can reconstruct the path. The other thing that we can do in breadth first search is keep track of the level. So, remember, we said the breadth first search actually explores the graph level by level.

Looks at all the vertices to the one step away from the source, then those vertices are one step away from these what, so there at level 2, two steps away from the source and so on. So, instead of just saying that the vertex is visited, we can ask at what level it was visit. So, initially we say that all the levels are undefined, and then each time we visit a new vertex, it is 1 level more than the vertex from which it was visit. So, eventually we say that, if level j is assign some number p, it must be t steps away from the original vertex.

(Refer Slide Time: 17:15)



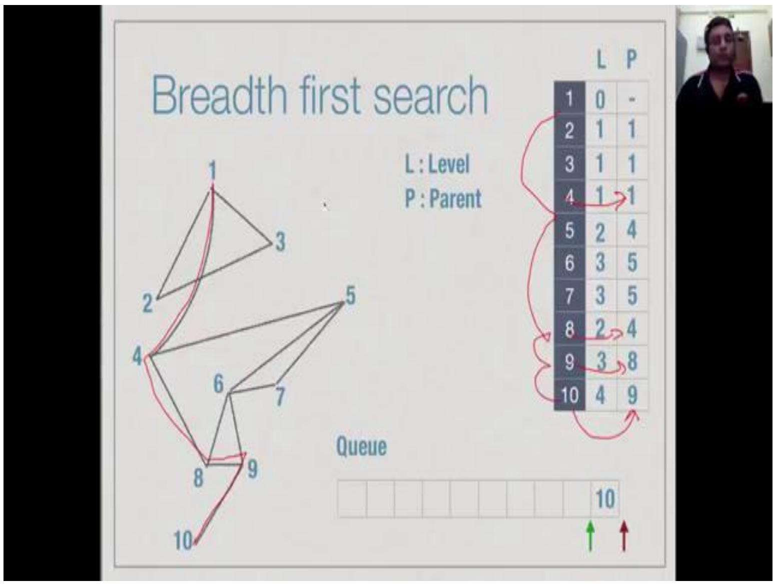
Breadth first search

```
function BFS(i) // BFS starting from vertex i

    //Initialization
    for j = 1..n {level[j] = -1; parent[j] = -1}
    Q = []

    //Start the exploration at i, level[i] set to 0
    level[i] = 0; append(Q,i)

    //Explore each vertex in Q, increment level for each new vertex
    while Q is not empty
        j = extract_head(Q)
        for each (j,k) in E
            if level[k] == -1          visited[k] = =0
                level[k] = 1+level[j]; parent[k] = j;
                append(Q,k);
```

So, this can also be easily added to a code along with a parent code. So, last time, we added this assignment for the parent. Now, we also add the assignment for level, so we say initially the level of each vertex is undefined. So, it is some non sensible value like minus 1. When, we say that the level of the start vertex is 0, and now whenever we visit a new vertex, if it is level, if it is visited, then it must have a level.

So, this level is minus 1, so it is level is minus 1, it means is not visit, so level minus 1 means not visited. Same as in an earlier thing, same as visited k is 0 and if so we had adjust the level of k to be one more than the level of j from which it was visited and of course, we said the parent was before.

(Refer Slide Time: 18:17)



Breadth first search

L : Level

P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9	3	8
10	4	9

Queue

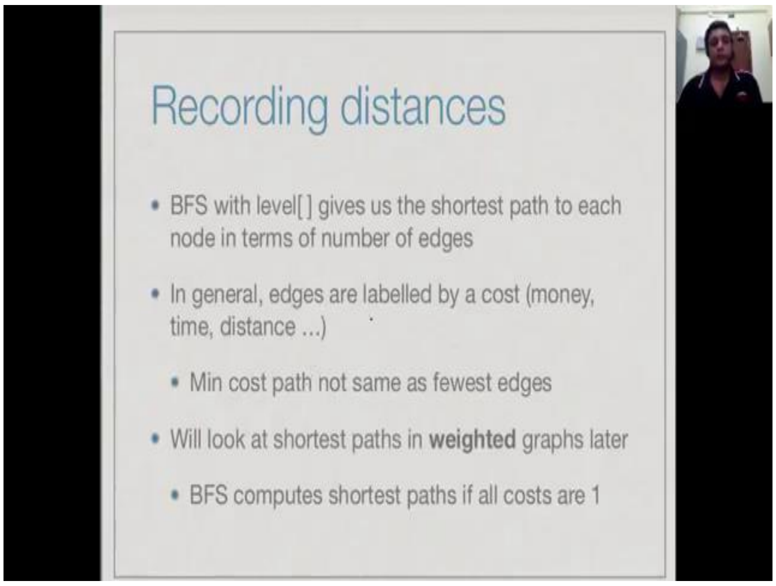
10

So, we can see all this works in our earlier example, as usual we start with 1 and we mark it is level as 0 and it does not have a pair. Now, when we add 2, 3 and 4 all of them will be at level 1 and all of them will have that is parent 1, because we have an explore from 1. Then, as before two does not add anything new to our list, p does not add anything to a list.

Now, 4 will add 5 and int, so far these 2, the level will be 2 and the parent will be 4, indicating that I bought to 5 and 8 from 4 in my exploration. Then, from 5, I will get 6 and 7. So, these are now at level 3, because 5 was itself at level 2, therefore, 6 and 7 are at level 3 and their parents are 5, from 8, I will go to 9, a 9 is also at level 3, because 8 was already at level 2, so 9 is level 3 and it is parent to the 8.

And finally, we will find that 10 is at level 4 and it is parent is 9. So, now, we can trace pair. So, we can say that, I got from 10, I came from 9, so 9 came from 8, so 8 came from 4, 4 came from 1. So, the path is actually 10 to 9 to 8 to 4 to 1 and you can see indeed, we have a path from 10 to 9 to 8 to 4 to 1. So, this is how you can re construct the path information and also the distance information at the same time, when you are computing breadth first search.

(Refer Slide Time: 19:57)



Recording distances

- BFS with level[ ] gives us the shortest path to each node in terms of number of edges
- In general, edges are labelled by a cost (money, time, distance ...)
  - Min cost path not same as fewest edges
- Will look at shortest paths in **weighted** graphs later
  - BFS computes shortest paths if all costs are 1

So, breadth first search if we add this level predicate, it actually gives us the shortest path to each node in terms of number of edges. Now, in general, we will see that, if you do not have the uniform cost, we have different cost on edges, then the shortest path need not be just the shortest in term to number of edges. We have to add the cost associated each path these are called weighted graphs. But, in an un weighted graph that is a graph we just has simple edges the way we have it now. Breadth first search has the added advantage of computing the shortest path in terms of number of edges from the source vertex to every reachable vertex from it.