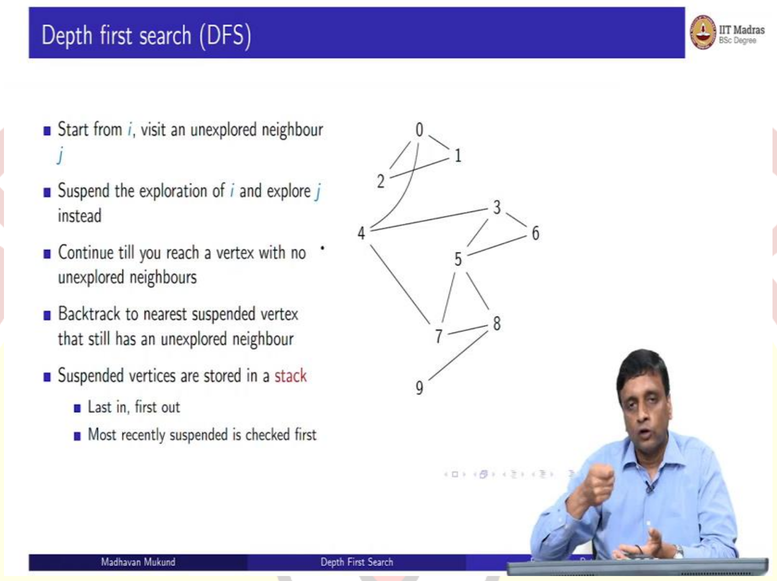
IIT Madras

ONLINE DEGREE

Programming Data Structures and Algorithms using Python
Professor Madhavan Mukund
Depth First Search (DFS)

We said that there were two fundamental strategies to search and find reachability in graphs. One was breadth first. And now we will look at depth first search.

(Refer Slide Time: 0:18)



Depth first search (DFS)

- Start from i, visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Suspended vertices are stored in a stack
  - Last in, first out
  - Most recently suspended is checked first

So, in depth first search, we start from some vertex, and we look at any one unexplored neighbour. And then from that explored neighbour, we will not come back and look at more unexplored neighbours with the original vertex, but rather we will go to j and then proceed from j. So, we kind of follow along path until we get stuck, and then we come back.

So, unlike breadth first search, where we explore all the neighbours of the first level, and all the neighbours second level, here, we will kind of follow exploration. So, it is sort of like, if you are reading a web page and the first link that you come across, instead of reading the rest of the text, you click on that link, and you go to a new web page. And now the first link that you see on that, you click on that and go to a new web page. So, you are kind of getting infinitely distracted from what you were doing earlier, but this in a finite graph will actually work.

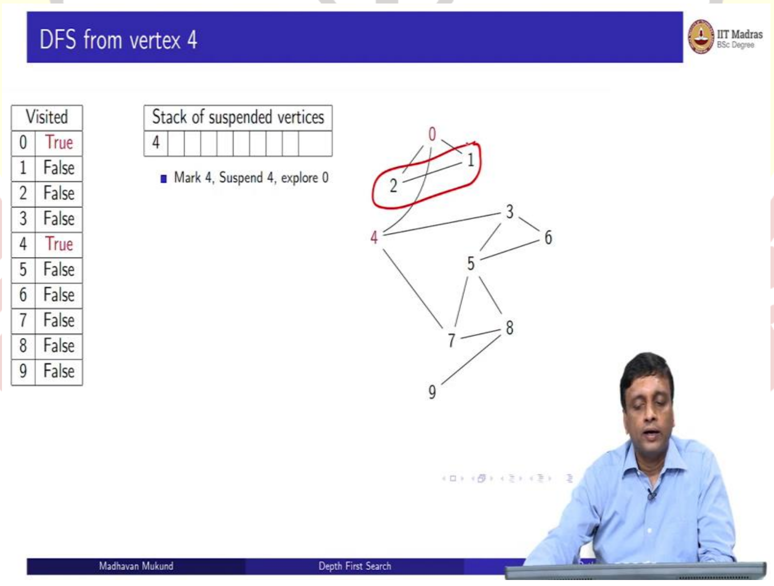
So, we will continue until we reach a vertex which has no unexplored neighbours. And when we hit that point, because we did not look at all the neighbours before, when we looked at a vertex, we explored the first neighbour that we found, maybe the second neighbour has not been explored yet, it could have been explored also, because we could have reached it in around about way after going through that first neighbour.

But if we find a neighbour, which is not explored, we will visit that so we kind of backtrack, but we backtrack systematically. So, we take the last place, we got stuck, come back to the previous step, backtrack there, then come back to this backtrack there and so on. So, in order to do the systematic backtracking, we will use a different data structure from what we did for the breadth first search and the breadth first search, in order to process vertices, we put them in a queue.

So, every time we saw a new visited vertex, which is unexplored, we put it in a queue, and they all came up for exploration in the order in which they went into the queue. So, earlier vertices got (pro) processed before later vertices. Whereas in depth first search, this is not going to happen in general, because the earlier vertices, which are not, which are neighbours of which could have been seen later, will be processed only later.

But we want to make sure that when we backtrack, we go back to the most recent vertices from which we made a choice. That is why we use a stack. So, a stack is a last in first out. So, just imagine a stack, like a stack of books, you put a book on top of the stack, the book that you take out will be the one you put last, if you want the second last book, you have to take out the top book and then take out, you can pull out something from in between.

(Refer Slide Time: 2:36)

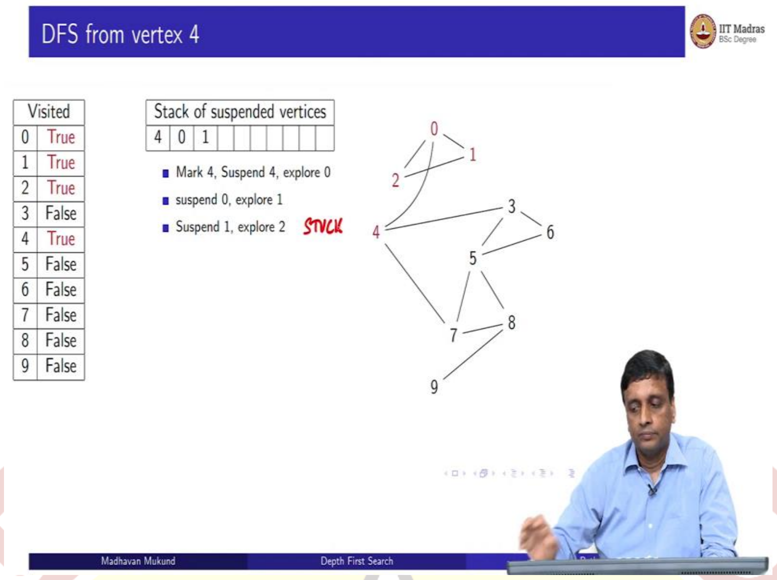


DFS from vertex 4

Visited	
0	True
1	False
2	False
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									
4									

- Mark 4, Suspend 4, explore 0



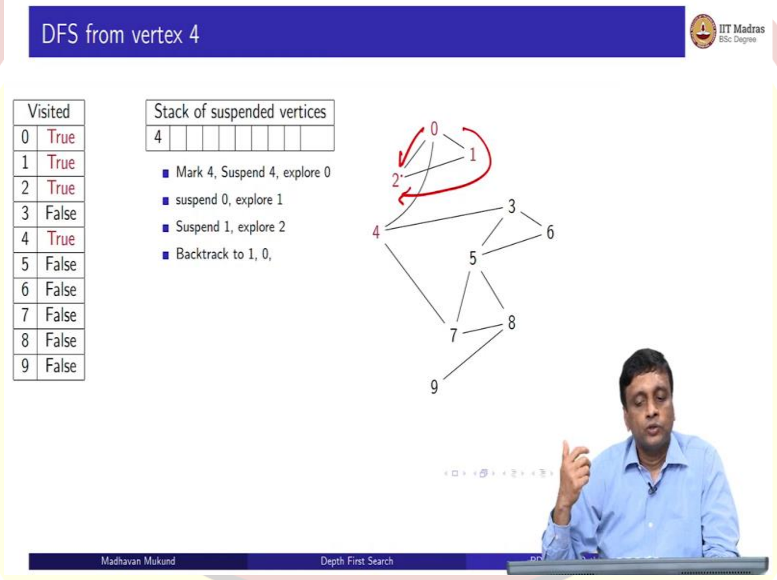
DFS from vertex 4

Visited	
0	True
1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								
4	0	1						

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2 STUCK

Madhavan Mukund Depth First Search



DFS from vertex 4

Visited	
0	True
1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								
4								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0,

Madhavan Mukund Depth First Search

So, if we run depth first search, for example, from vertex 4 here, then we can take any one of its neighbours, so it has 0, and 3, and 7. So, if we take the smallest one, for instance, 0, so we mark it, and then we explore, say, the strategy is that we will explore the smallest outgoing vertex from a given vertex, so 4 now. So, now I basically put 4 on the stack saying that I am not finished with 4. The thing that I had done with 4 was incomplete. But I am now processing 0.

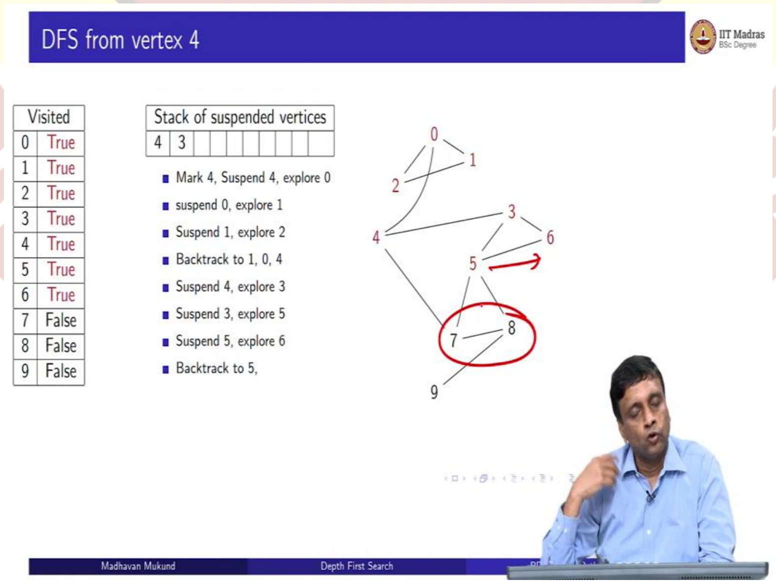
So, now I look at the neighbours of 0, so the neighbours of 0 are 1 and 2. And if I look at them in this order, then I will find that 1 is not yet visited. So, I will suspend 0 and look at 1 instead. So, now 0 has been put on top of 4 on the stack. So, the stack is growing from left to right. So, the stack is growing in this direction.

So, now I have suspended 4 in the process of exploring neighbours of 4, I have suspended 0. And now I am looking at neighbours of 1. So, when I look at the neighbours of 1, I find that it has 2 neighbours 0 and 2, but only 1 of them 2 is unexplored. So, I will suspend the 1 and explore 2 instead. But 2 has neighbours, which are already explored. So, the exploration of 2 does not get me anywhere. So, this is a situation where I get stuck. So, I get stuck here.

So, since I get stuck, I have to go back and say, I have nothing I can do here. So, I go back to 1 and I ask was there anything left pending in 1 which I could do now that I know that 2 is a dead end? So, I go back to 1. So, I pull 1 off the stack and I say, now let me look at its neighbours again and see if there is anything left to do. And I find there is nothing left to do because there are only to neighbours 0 and 2, 0 anyway, I had ignored because 0 was marked right at the beginning and I just marked 2. So, now I say this 1 is also a dead end for me. So, I backtrack to 0.

Now I look at what was left with 0 so when I went from 0 to 1; 2 was unexplored, but notice that I came through this roundabout route and explored 2. So, now if I look at 0 to 2 directly, I find that it is already been visited by some other exploration which I initiated earlier. So, though 0 has never explored 2 itself by the time 0 looks at 2; 2 is visited. So, this is unlike breadth first search, where 2 would be visited only from 0, it will not be visited somewhere else. So, 0 visits cannot visit 2 so now I backtrack back to 4.

(Refer Slide Time: 5:17)



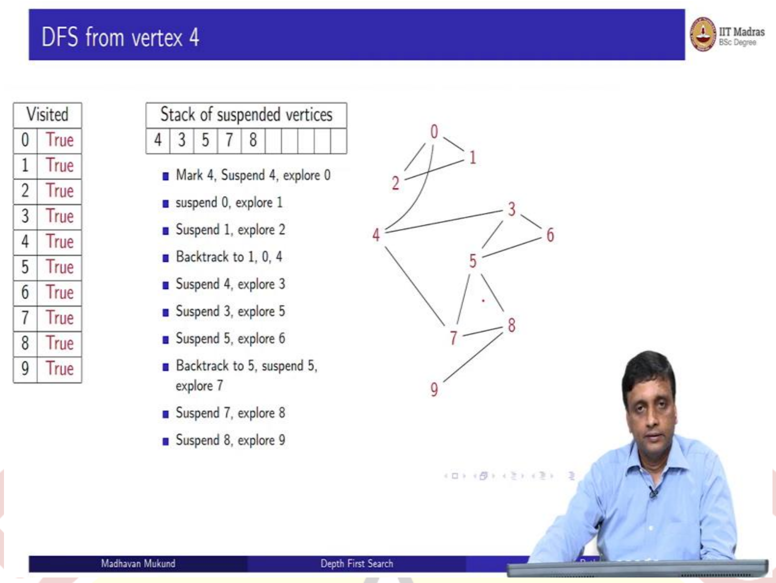
DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	False
8	False
9	False

Stack of suspended vertices									
4	3								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5,

Madhavan Mukund Depth First Search



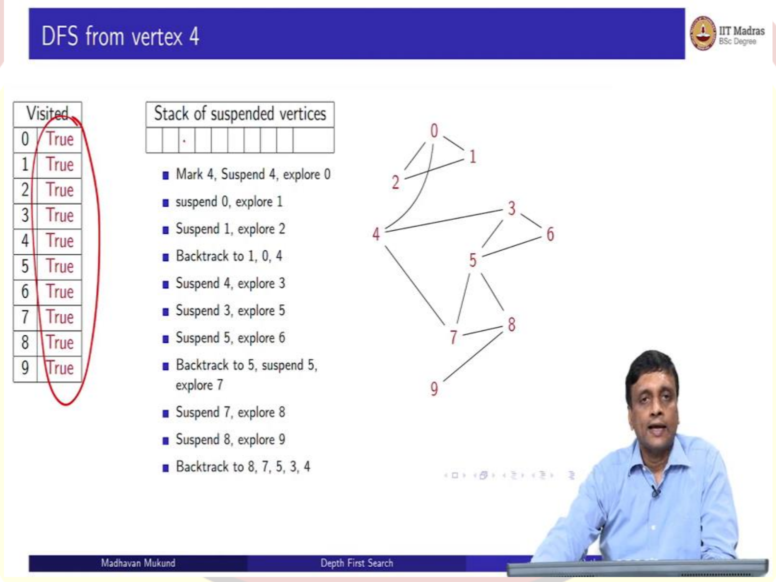
DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices									
4	3	5	7	8					

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9

Madhavan Mukund Depth First Search



DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7, 5, 3, 4

Madhavan Mukund Depth First Search

So, now I basically come back to the first vertex, I started with saying that whatever I did in the direction 0 has now completed and there is nothing more to search below 0. So, I look at my next neighbour, which is 3. So, I say let me suspend 4 again, and explore 3. Now 3 will suspend itself and explore 5 because that is a smaller of the two unvisited neighbours, 5 will suspend itself and explore 6 because that is the smaller of the two unvisited neighbours.

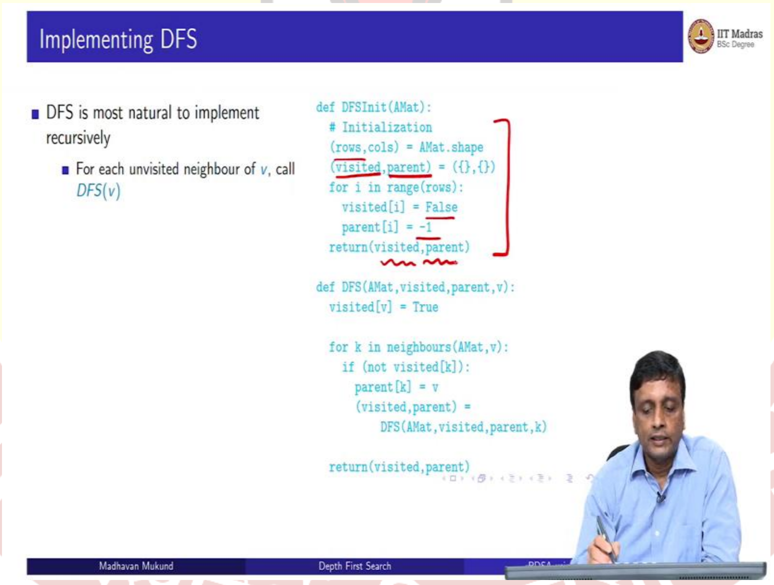
But 6 has no new neighbours to visit, because 6 has neighbours 3 and 5, which are both marked as visited. So, 6 will say backtrack to 5. So, now I have something interesting because I went that way but I still have these two pending. So, backtracking to 5 allows me to do more things, I do not have to go all the way back to 4, like we did last time. So, 5 will say, let me now explore 7 instead.

So, I spend 5 one more time and this time I explore 7. From 7, I can explore 8, because 4 and 5 are known, but 8 is not. So, I suspend 7, explorer 8. From 8, I explored 9. Now 9 is a dead end. So, having explored 9, I will come back and backtrack to 8. But there is nothing more to see from 8, so I backtrack to 7. There is nothing more to see from 7, so I backtrack 5. There is nothing more to see from 5, so I backtrack to 3. There is nothing more to see from 3, so I backtrack to 4.

And now the stack is empty. So, there is nothing pending. I have seen everything. And as you can see, because it is the same graph. And hopefully, you get to get the same result everything is actually reachable from 4. So, this is how depth first search works with a stack.

Now, when we implemented breadth first search, we actually had to construct that queue. If you remember, we created a class called queue, and we created a queue object and kept track of it. So, you might imagine that when you do depth first search, you have to do the same thing, you have to create a stack object and keep track of it.

(Refer Slide Time: 7:15)

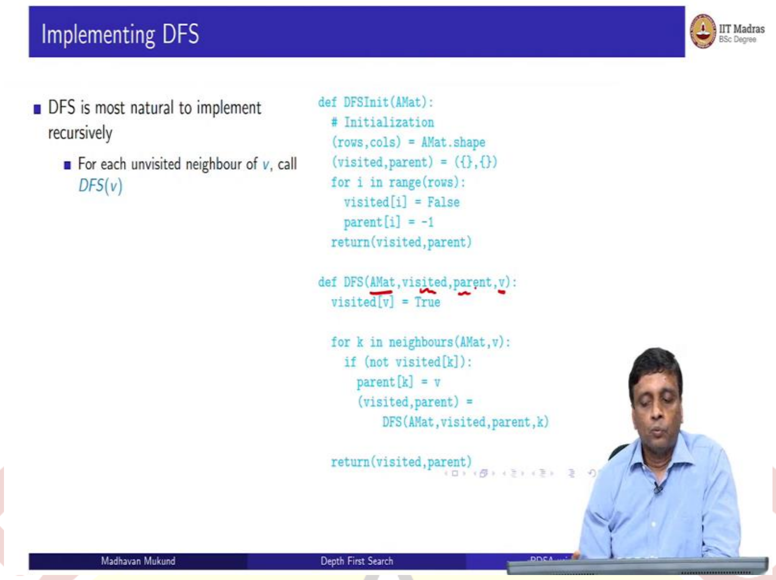


Implementing DFS

- DFS is most natural to implement recursively
  - For each unvisited neighbour of *v*, call *DFS*(*v*)

```
def DFSInit(AMat):
  # Initialization
  (rows,cols) = AMat.shape
  (visited,parent) = ({},{})
  for i in range(rows):
    visited[i] = False
    parent[i] = -1
  return(visited,parent)

def DFS(AMat,visited,parent,v):
  visited[v] = True

  for k in neighbours(AMat,v):
    if (not visited[k]):
      parent[k] = v
      (visited,parent) =
          DFS(AMat,visited,parent,k)

  return(visited,parent)
```

Madhavan Mukund Depth First Search


Implementing DFS

- DFS is most natural to implement recursively
  - For each unvisited neighbour of v, call $DFS(v)$

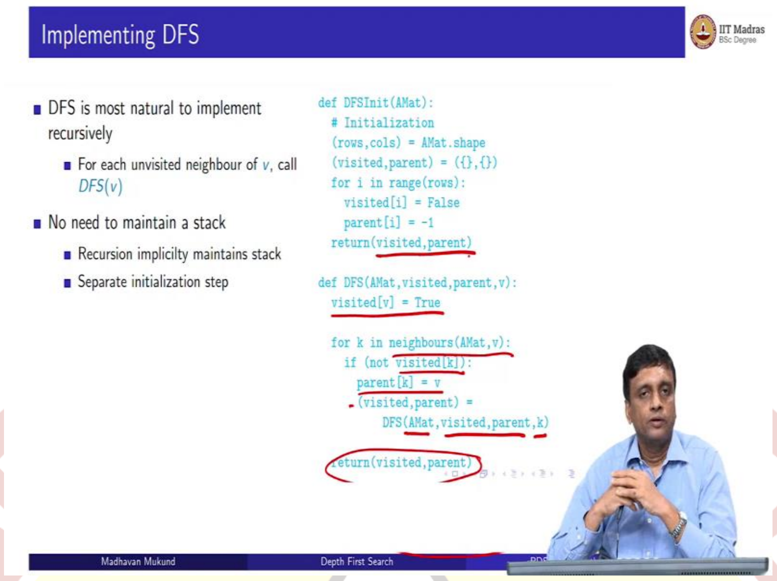
```
def DFSInit(AMat):
  # Initialization
  (rows,cols) = AMat.shape
  (visited,parent) = ({},{})
  for i in range(rows):
    visited[i] = False
    parent[i] = -1
  return(visited,parent)

def DFS(AMat,visited,parent,v):
  visited[v] = True

  for k in neighbours(AMat,v):
    if (not visited[k]):
      parent[k] = v
      (visited,parent) =
          DFS(AMat,visited,parent,k)

  return(visited,parent)
```

However, it turns out that you can actually implement depth first search using recursion, and recursion implicitly keeps a stack. So, the thing about depth first search is if you want to do it, recursively, then you must separate out the initialization from the recursive call, because otherwise, every time you initial you call depth first search it initializes all the visited to be false in all that it is going to be something that will not work.

So, there is an initialization phase where you as always set in this case say let us do it in one shot, this is visited and parent. So, we set visited to be the empty dictionary, the parent to be the empty dictionary. And then for having extracted the number of so this is with an adjacency matrix, having extracted the number of rows, we can set all the vertices to not be visited and all of them to have parent minus 1. So, this is an initialization phase.

So, what it does is it returns an initialized version of parent and visited. So, we now have to take this initialized version of parent and visit and pass it through to DFS. So, the actual DFS is here. So, DFS now the way I have written it takes 4 parameters, it takes the matrix of the graph, it takes the vertex to start and it also takes the current status of visited and parent. So, what does it do?

(Refer Slide Time: 8:44)



Implementing DFS

- DFS is most natural to implement recursively
  - For each unvisited neighbour of *v*, call *DFS*(*v*)
- No need to maintain a stack
  - Recursion implicilty maintains stack
  - Separate initialization step

```
def DFSInit(AMat):
  # Initialization
  (rows,cols) = AMat.shape
  (visited,parent) = ({},{})
  for i in range(rows):
    visited[i] = False
    parent[i] = -1
  return(visited,parent)

def DFS(AMat,visited,parent,v):
  visited[v] = True

  for k in neighbours(AMat,v):
    if (not visited[k]):
      parent[k] = v
      (visited,parent) =
          DFS(AMat,visited,parent,k)

  return(visited,parent)
```

Madhavan Mukund | Depth First Search

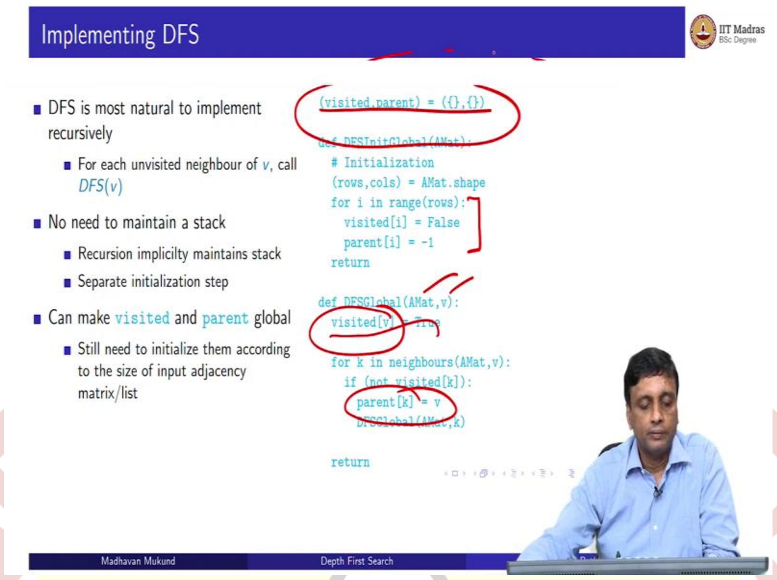
So, it basically calls itself, so it does the obvious thing that it visits the vertex, which it has been told to visit, so it sets visited of V to be true. And for every neighbour, if it is not visited, it sets the parents so all this is familiar to us, this is what we do, whether you are doing breadth first search or depth first search. But now this is the new thing, which is I do not proceed with the next neighbour.

So, I am looking at all the neighbours but I am not if I see one neighbour, which is not visited, I kind of suspend myself and this is done now implicitly through recursion, I restart DFS from the new vertex k which I just saw, and I passed the current value visited and parent is going to return back to me an updated value of visited and parent. So finally, DFS when it concludes, gives back so basically each time I visit something it update visited and parent gives it back to me. So, this is the recursive nature of DFS.

Now, here, because we have essentially made visited and parent kind of internal to DFS, we have to keep passing it around. We have to call DFS with visited and parent and get it back. There is a little bit cumbersome and this is not how you would normally see it presented if you see an implementation in a language other than Python.

So, usually in presentations of DFS, they will assume that this visited and parent, in this case dictionaries, or lists or arrays or whatever you are using to store that are actually global values which you can access from inside DFS. And so you can update them from inside DFS without having to pass them around. So, we can also do that.

(Refer Slide Time: 10:25)



Implementing DFS

- DFS is most natural to implement recursively
  - For each unvisited neighbour of *v*, call *DFS*(*v*)
- No need to maintain a stack
  - Recursion implicilty maintains stack
  - Separate initialization step
- Can make `visited` and `parent` global
  - Still need to initialize them according to the size of input adjacency matrix/list

```
(visited,parent) = ({},{})

def DFSInitGlobal(AMat):
  # Initialization
  (rows,cols) = AMat.shape
  for i in range(rows):
    visited[i] = False
    parent[i] = -1
  return

def DFSGlobal(AMat,v):
  visited[v] = True

  for k in neighbours(AMat,v):
    if (not visited[k]):
      parent[k] = v
      DFSGlobal(AMat,k)

  return
```

Madhavan Mukund   Depth First Search

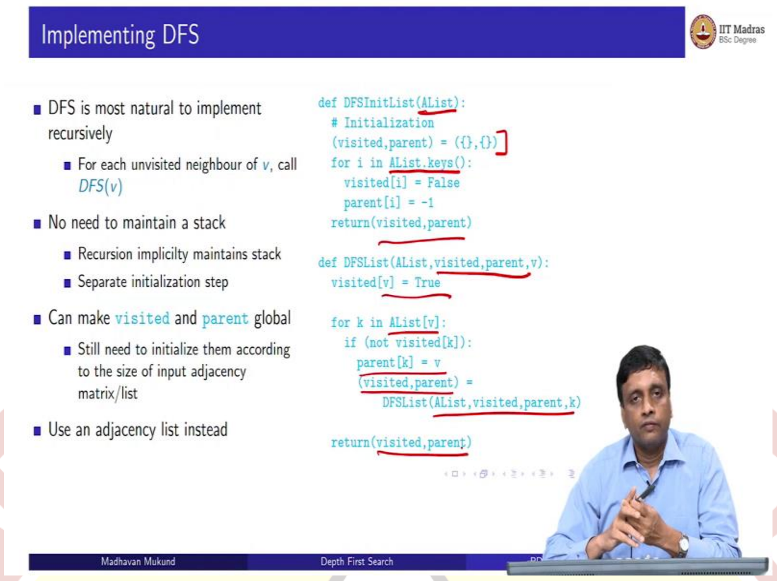
So, we can make visited and parent global. So, we declared outside all our functions, these two empty dictionaries called visited and parent. And now because Python has this convention that all mutable values, a mutable value is either a list or a dictionary. A mutable value can be globally referenced from inside a function. So, this is one of the decisions that Python has made, which is basically to take care of the situation. So, you do not have to keep passing lists and dictionaries in and out of functions.

I still have to initialize it, but the initialization happens now in this case, without having to pass the dictionary. So, remember, in the earlier thing, the initialization actually returned back the initialized visited and parent dictionary here, it is just going to initialize it. So, it is going to do the same thing, it is going to run through a loop. But now it is initializing it outside. So, this visited and parent dictionary are outside the function is just setting them up. So, I will have to first call this function to initialize it.

And now when I do the DFS itself, I only have to pass the functions, the parameters, which I would expect to pass, which is the matrix and the vertex to start with. So, I set visited of this vertex to be true. And then for every neighbour, if it is not visited, I will call DFS with that. Earlier, I had to call it with the updated value of visited and parent and get it back.

So, this is a simpler, easier to understand version. And this recursive function is implicitly manipulating this visited and parent which are sitting outside. So, that is something so these updates here, setting visited v equal to true and setting parent k equal to V, these are happening to this global data structure, which is outside, those are something to keep in mind.

(Refer Slide Time: 12:11)



Implementing DFS

- DFS is most natural to implement recursively
  - For each unvisited neighbour of v, call $DFS(v)$
- No need to maintain a stack
  - Recursion implicilty maintains stack
  - Separate initialization step
- Can make `visited` and `parent` global
  - Still need to initialize them according to the size of input adjacency matrix/list
- Use an adjacency list instead

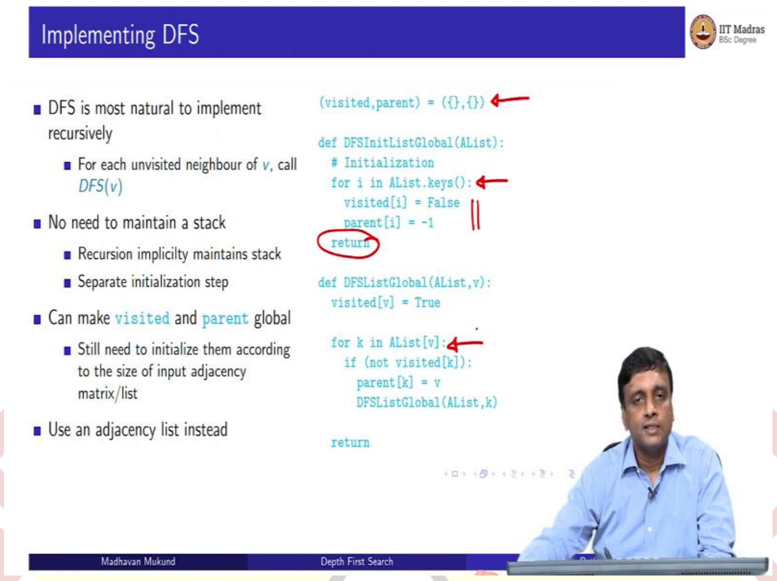
```
def DFSInitList(AList):
  # Initialization
  (visited,parent) = ({},{})
  for i in AList.keys():
    visited[i] = False
    parent[i] = -1
  return(visited,parent)

def DFSList(AList,visited,parent,v):
  visited[v] = True

  for k in AList[v]:
    if (not visited[k]):
      parent[k] = v
      (visited,parent) =
         DFSList(AList,visited,parent,k)

  return(visited,parent)
```

Madhavan Mukund Depth First Search

Now, you can do the same thing with an adjacency list. So first, let us look at the version where we have this thing passing around. So as before, except that we have a list, you will initialize these dictionaries to be empty. So, this is the version where visited and parent are maintained inside the function, so we have to keep passing them around. So, I will initialize them using the list of keys of the adjacency list. But otherwise is the same thing I updated to false and minus 1 and I return the initialized arrays.

And now when I call will DFS list, I have to pass visited and parents. So again, I do the same thing I set visited to true. And instead of checking the neighbouring function, which looks at all the rows in the matrix, I just check the adjacency list of the current vertex. And for every such thing, if it is not visited, I will set its parent to be the current vertex. And I will call recursively DFS with the current value of visited and parent and get back an updated value of visited parent. And finally, DFS when it returns has to return this updated value. So, this is the non-global version of DFS using an adjacency list.

(Refer Slide Time: 13:22)



Implementing DFS

- DFS is most natural to implement recursively
  - For each unvisited neighbour of v, call $DFS(v)$
- No need to maintain a stack
  - Recursion implicilty maintains stack
  - Separate initialization step
- Can make `visited` and `parent` global
  - Still need to initialize them according to the size of input adjacency matrix/list
- Use an adjacency list instead

```
(visited,parent) = ({},{})

def DFSInitListGlobal(AList):
  # Initialization
  for i in AList.keys():
    visited[i] = False
    parent[i] = -1
  return

def DFSListGlobal(AList,v):
  visited[v] = True

  for k in AList[v]:
    if (not visited[k]):
      parent[k] = v
      DFSListGlobal(AList,k)

  return
```

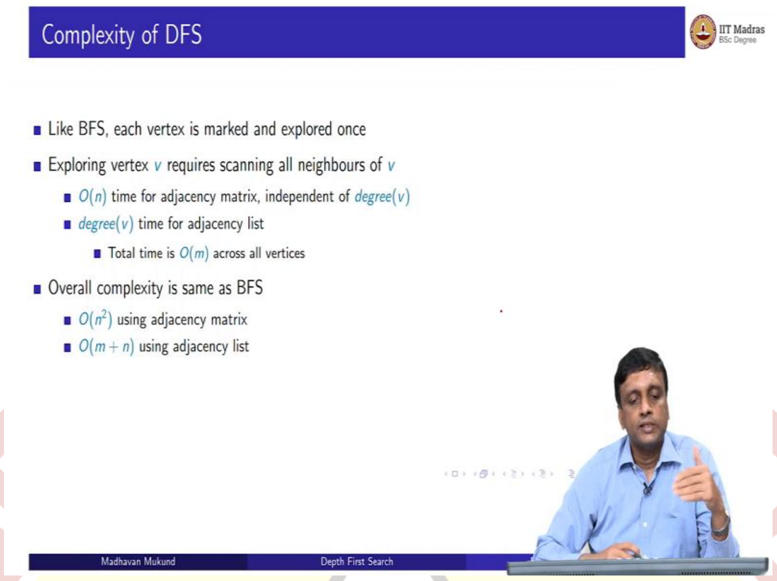
Madhavan Mukund Depth First Search

And here is the global version of DFS using an adjacency list. So, as before, now, we make visited and parent dictionaries which are outside. So, when we initialize, we only initialize but we do not return the initialized values because they are being globally updated. And the only change is that the loop for initialization runs over the keys of the adjacency list rather than the columns of the matrix.

And then the DFS itself is the same thing. The only difference is that instead of looking for the neighbours of k, v, which looks at all the columns in the matrix, I just look at the list of neighbours of V. And if it is not visited, I set the parent and I call it again. So, these are now 4 different implementations of DFS, which I have shown you, with and without, with an adjacency matrix, with an adjacency list, and maintaining visited and parent globally and locally.

So, all 4 of them work. And it is a question of which one you find most convenient. Generally, if these are large values, large dictionaries this is generally desirable, not to pass them around. So, global things make it easier to write because you do not have to pass these dictionaries and remember to call them and get the updated value. So usually, for such things, the global version is preferred.

(Refer Slide Time: 14:39)



Complexity of DFS

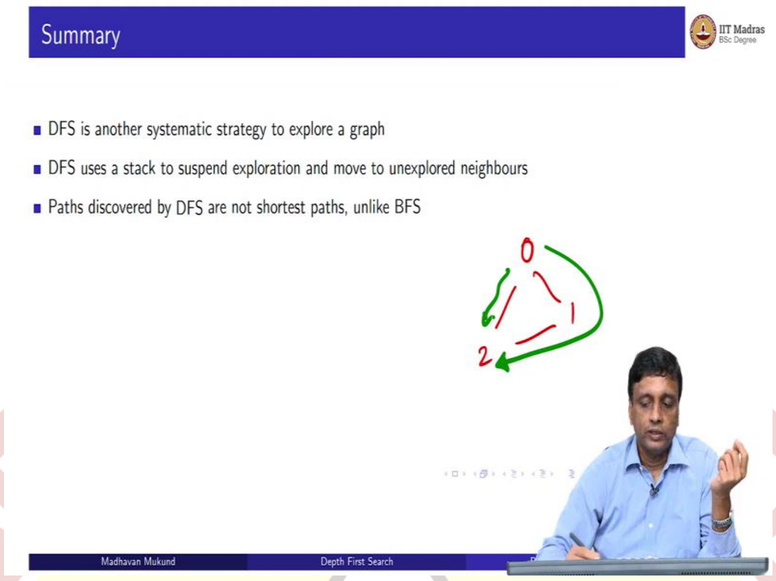
- Like BFS, each vertex is marked and explored once
- Exploring vertex v requires scanning all neighbours of v
  - $O(n)$ time for adjacency matrix, independent of $degree(v)$
  - $degree(v)$ time for adjacency list
    - Total time is $O(m)$ across all vertices
- Overall complexity is same as BFS
  - $O(n^2)$ using adjacency matrix
  - $O(m+n)$ using adjacency list

IIT Madras
BSc Degree
Madhavan Mukund
Depth First Search

So, what about the complexity? Well, the complexity despite all this recursion is very similar to BFS. So, every vertex is visited once so it is marked as visited once and it is explored once. So, you call DFS from that vertex only once. So, at that sense, we do order n processing in terms of vertices, we never see a vertex twice. And as before, in order to explore a vertex, to decide which of its neighbours to process, we have to check which are all the neighbours.

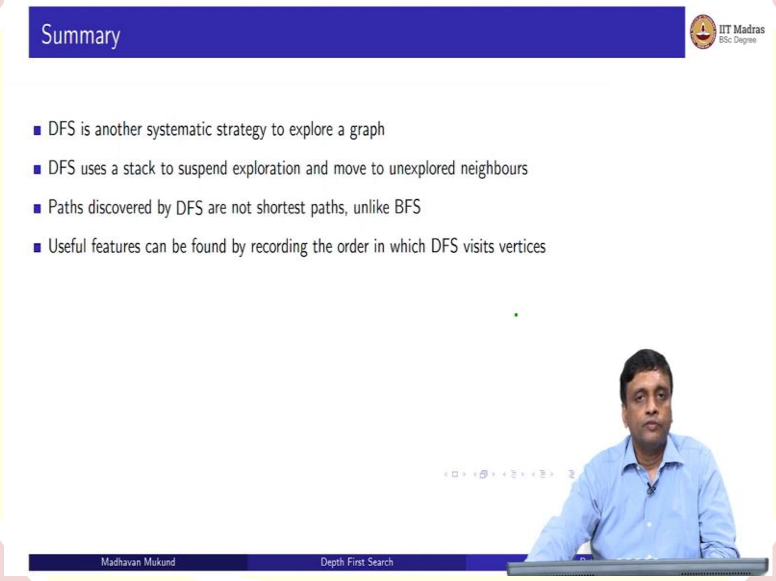
And an adjacency matrix that takes order n time. And in an adjacency list is take degree v time. So, as exactly as in the BFS, if we do this thing with an adjacency matrix, we get an order n squared algorithm. If we do this adjacency list, we get an order m plus n algorithm. So, there is no difference in the worst-case running time, of course, there will be a cost due to recursion, which we are not going to worry about right now. But otherwise, in terms of the number of times you process these vertices and edges, we have this usual scaling of going from n squared to m plus n.

(Refer Slide Time: 15:49)



Summary

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbours
- Paths discovered by DFS are not shortest paths, unlike BFS



Summary

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbours
- Paths discovered by DFS are not shortest paths, unlike BFS
- Useful features can be found by recording the order in which DFS visits vertices

So, DFS is a different strategy from BFS. And it uses a stack rather than a queue and the stack is implicit when we do it recursively. Now, BFS we said gives us some extra bonus, which is that you get shortest paths, so we did two things would BFS we first recovered the path using parent and then we use this level function to discover the distance.

Now, in the BFS, we only did the parent part, we did not do the level part. And that is because as we saw, even in that example, we saw that, like if I had 0, 1, and 2 connected like this, it could be that DFS finds this path because it suspends 0 then suspends 1. Whereas of course, there is a shorter path. So, DFS is not generally going to find the shortest path. So, even though the parent thing tells us something about one way of getting there, it is not really the fastest way so DFS does not have that advantage.

So, you might ask, I mean, you have to do all this work, you have to do recursion you have to do all this. And at the end, you do not even get this information what is the point of DFS? It turns out that for other things, DFS is very useful. So, actually, if on the balance of things DFS is a more informative search than breadth first search.

So, we will look at examples in a subsequent lecture as to why DFS gives you very useful information about the structure of a graph and therefore very often DFS is the primary way of exploring a graph rather than BFS.