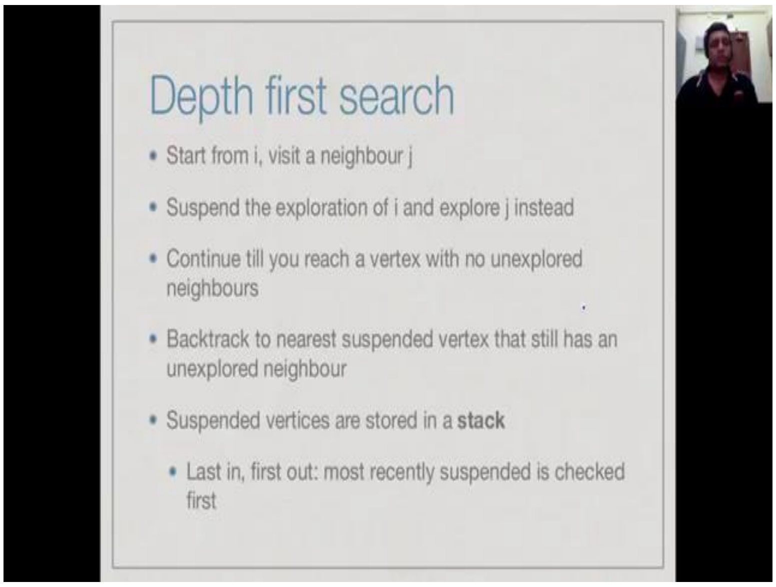
Design and Analysis of Algorithms

Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 03
Module - 04
Lecture - 21
Depth first search (DFS)

(Refer Slide Time: 00:09)



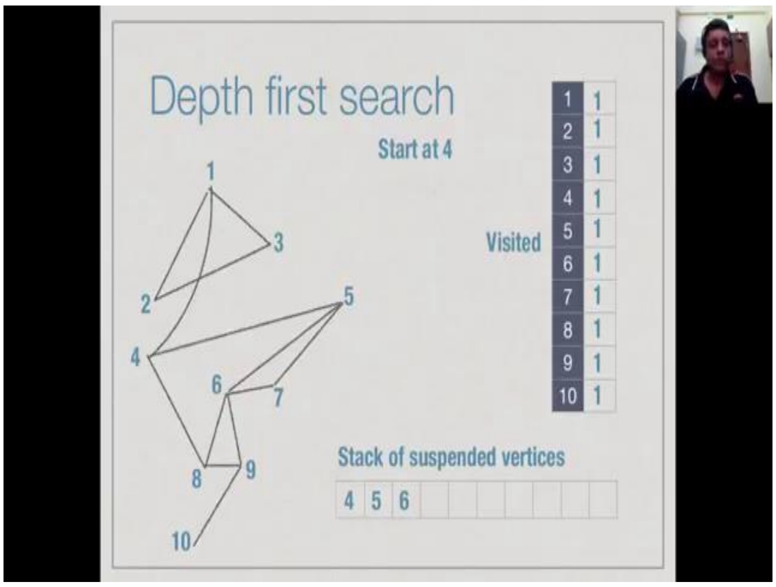
Depth first search

- Start from i, visit a 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, we have seen how to explore connectivity using breadth first search. So, now let us look at the other strategy which is commonly use called depth first search. So, an depth first search instead of exploring all vertices level by level. Each time we explore a new vertex, we immediately explore it is steps. So, we start now vertex i and visit the first neighbour j from i which is not yet explore. When we suspend the explanation why and explore j instead, and you keep doing this until you can no further. So, when you gets stuck when there is nothing new to explore you walk back, because you have already left some vertices earlier unexplored, because is suspended. So, you go back to the nearest suspended vertex such still has an unexplored neighbor, and then you explore the next unexplored neighbour of that vertex.

So, the wait think about it is that all the vertices which are pending which has suspended are now in a stack. So, you build up to stack as you go to deeper and deeper into the graph, and then whenever you get stack at a dead end, walk back up and you process things earlier in the stack.

(Refer Slide Time: 01:08)



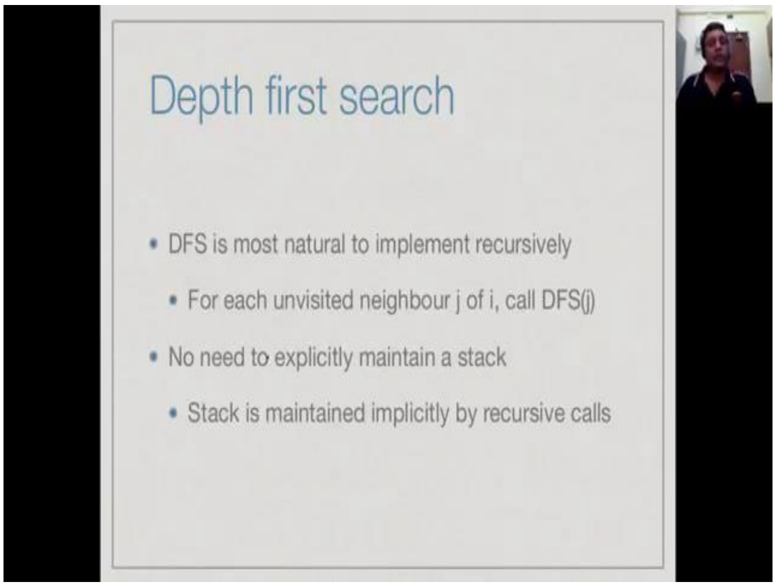
So, as before let us execute this algorithm by hand on our graph, you see how it works before we write the pseudo code. So, this time we are starting at 4. So, first step is to mark 4 as visited. So, we star mark 4 as visited and we identify that it has 3 neighbours before neighbours 1, 3, 5, and 6. So, taking them an order we start with 1; 1 is not visited. So, we mark it as visited. And we put 4 on the stack, saying a suspending the execution of 4, because now we are going to explore 1 in step. So, now we go to 1, then we see that 1 has neighbours 2, 3, and 4 among these of first is 2. So, if could 2 on the stack, since it could not is 2, remark 2 is visited. So, which you are not visited earlier, in we put 1 on the stack, since now we are suspending 1.

So, now we look at the neighbours of 2. So, neighbours of 2 are 1 and 3, 1 is already been visited, 3 has not. So, we will now mark 3 as visited and suspend 2. Now when we come to 3, it has 2 neighbours 1 and 2, but both of them are already visited. So, thus nothing to become. So, we must go and we must go back to the last vertex which we suspended namely 2, and explore it is neighbours. To a go back to 2, so we had already declared that 1 was not to be done, we have already process 3. So, we have to look at 4, but 4 is also already visited. So, I have done with 2. So, now we must go back and go the store the next vertex in the stack which was left number that was 1. Now we go back to 1, and we find that from 1 we had already visited 2, 3 which we are not visited to 1 was visited via to such already mark to this is done. And 4 also is where we came from to 1. So, this is done. So, 1 is also useless.

So, now we go back and go back to the first vertex in started with 4, and see the something more to be done. So, having come back to 4, we find that, it has more neighbours, beyond the explore 1 the first step. So, now we continue we find that there are ((Refer Time: 03:18)) 5, 6, and 8. So, we pick up 5 and see is not visited, we mark 5 ((Refer Time: 03:27)) again suspend 5. So, from 5, now we have 4, 6, and 7. So, we will now pick up 6 and suspend 5. From 6, we have 7, 5, 7, and 8; the 5 is already done, but 7 is not done. So, we will pick up 7 and suspend 6. Now from 7, we find that it has neighbours 5 and 6, both of which are already mark. So, 7 is ((Refer Time: 03:56)). So, we come back and look at 6 again, and see to anything move to be done, there is where is an 8. So, now we add 8, and again suspend 8. Now we are at 8, and now we are to ask what can be explore from 8, from 8 4 is already done – has 4, 6, and 9. So, 4 is already done, 6 is already done, but 9 is need. So, we add 9 and suspend 8, then we move to 9.

So, from 9 we find it have a new neighbour 10. So, we add 10 then we are done, because 10 has nothing to do. So, we go back and process 9; 9 has no more new things to say, because 9 we other neighbours are 6 and 8 to be already done, you go back to 8, 8 has nothing more to says will go back to 6, 6 has nothing more to says will go back to 5, 5 has nothing more to says will go back to 4, now when we started this whole expiration you get it 4 to 5. So, now we look and see the 4 to 6 is already done, it is enough 2 to 6 4 to 8 is already done. So, this nothing more we can do from 4 and so this expiration is done.

(Refer Slide Time: 05:00)

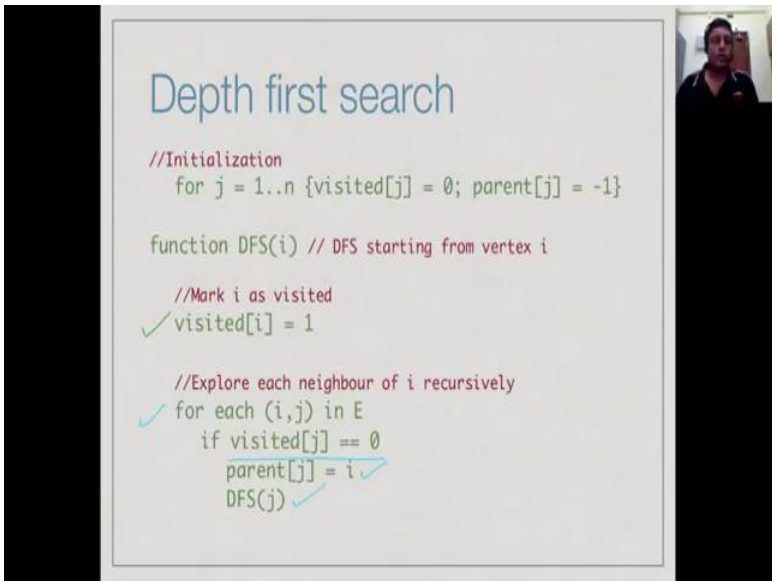


Depth first search

- DFS is most natural to implement recursively
  - For each unvisited neighbour j of i, call DFS(j)
- No need to explicitly maintain a stack
  - Stack is maintained implicitly by recursive calls

So, what we did explicitly with a stack can be done more cleverly, we just implemented recursive. So, whenever we visit a new vertex j, we call DFS of j an suspend DFS of the current vertex i. So, we do not have explicitly maintain this stack that we are simulating, because it will be implicitly maintain by the recursive function calls.

(Refer Slide Time: 05:27)



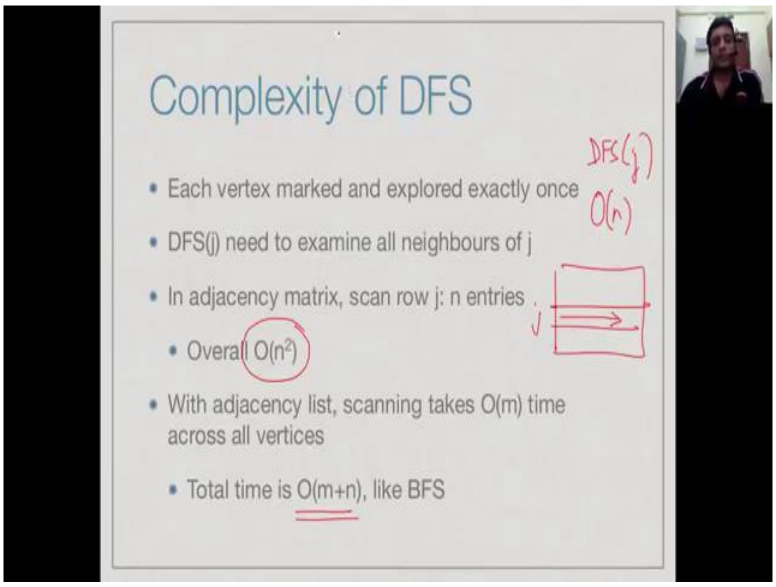
Depth first search

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

function DFS(i) // DFS starting from vertex i

  //Mark i as visited
  visited[i] = 1

  //Explore each neighbour of i recursively
  for each (i,j) in E
    if visited[j] == 0
      parent[j] = i
      DFS(j)
```

So, DFS is therefore a very simple algorithm to implement. So, initially we say for every vertex visited is 0, and remember like in DFS breadth first search we want to keep track of where we came from, so we will say that everything as an undefined there. Now we initialize we call DFS of the start vertex. So, when we call DFS from a vertex, we mark it has been visited. So, the first thing we do this to mark it has been visiting. And now for every other vertex that it is connected to we do the usual think, we check whether or not that vertex we already visited; if it is not visited, we want visit it. So, how do we visited will we mark its parent is being i.

And then we suspend this DFS and call that DFS. So, this is what we are doing explicitly with a stack, but it much easier to do recursive. So, DFS is a very, very simple recursive algorithm, starts in I look at every unexplored neighbour and recursively in would DFS on that unexplored it.

(Refer Slide Time: 06:30)



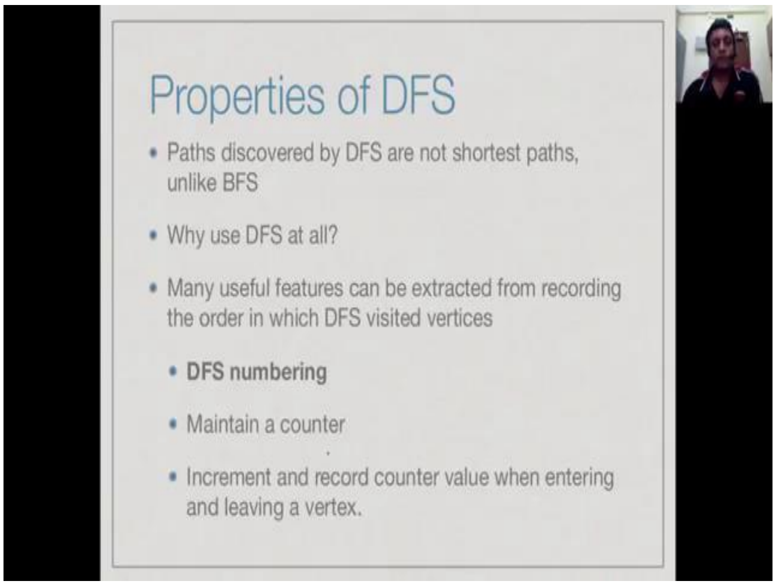
Complexity of DFS

- Each vertex marked and explored exactly once
- DFS(j) need to examine all neighbours of j
- In adjacency matrix, scan row j: n entries
    - Overall $O(n^2)$
- With adjacency list, scanning takes O(m) time across all vertices
    - Total time is O(m+n), like BFS

So, what is the complexity of depth first search? Well, each vertex is marked and explored exactly once. So, we do DFS of j, once for every j's. So, these are order n ((Refer Time: 06:42)), actually exactly n calls if everything is reach a row. Now when we call DFS of j for a particular j, we need to examine all the neighbours of j. As we saw before if we have adjacency matrix; that means, we have to look at the row j, and we have to look at every entry in this row. So, this takes order n time. So, we have order n calls and each call takes order n times. So, over all we have order n square time.

On the other hand, if you use and adjacency list then when we look at the neighbours of j, we only have to look at the exact vertices connected to and as we said before if we count across all the calls each edge will be accounted for twice; once from i to j, and once from j to i. So, the total numbers of calls, the total number of steps to scan the neighbours we will be order of the number of edges. So, the overall time will be linear m plus n like breadth first search. So, both depth first search, and breadth first search are linear in the size of the input, if we use adjacency list.

(Refer Slide Time: 07:45)



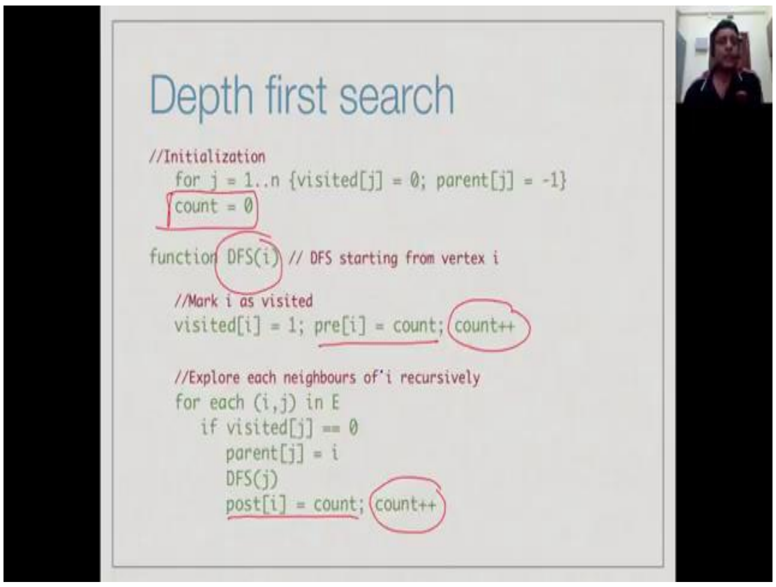
Properties of DFS

- Paths discovered by DFS are not shortest paths, unlike BFS
- Why use DFS at all?
- Many useful features can be extracted from recording the order in which DFS visited vertices
  - **DFS numbering**
  - Maintain a counter
  - Increment and record counter value when entering and leaving a vertex.

So, one big differences between depth first search and breadth first search is that the paths that breadth first search discovers are not shortest paths. So, if we have a graph as we saw before a triangle like this will we have 1, 2, and 3; then what depth first search will do to find a path from 1, 2, 3 which goes via 2, if we take this smallest neighbor it is time to explore. So, when actually come to the 1 3 part, I will find it 3 is already visit in ((Refer Time: 08.14)) right. So, it appears therefore the depth first search may not between something very useful, but actually this recursive way of exploring gives us a lot of information.

So, many useful features about the graph can actually be recorded by can be extracted by recording the order in which DFS which is vertices. So, for this we argument DFS, we something called numbering. So, we maintain a counter which we increment every time we enter a vertex, when DFS starts in a vertex and when it leaps. So, we associate with each vertex in the graph 2 values; the value of the counter that was there when I entered when I did DFS of j for the first time, and the value of the counter when DFS of the j execute.

(Refer Slide Time: 09:00)

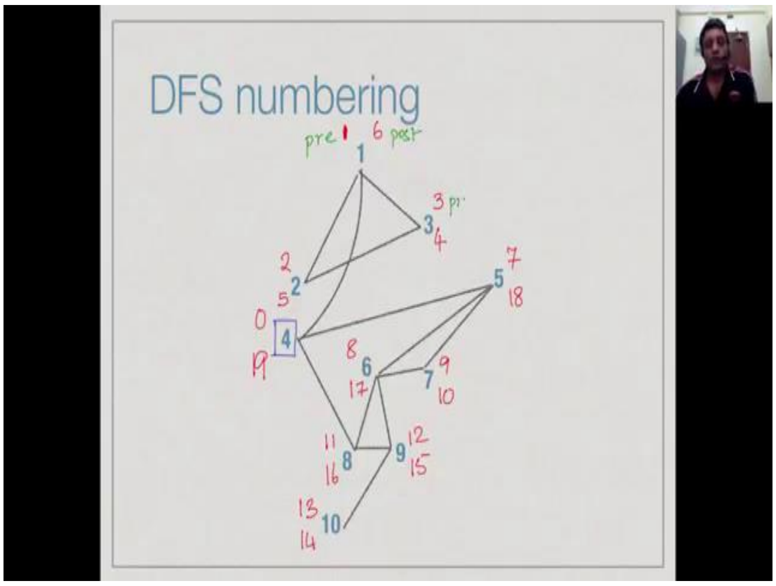


Depth first search

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

function DFS(i) // DFS starting from vertex i

   //Mark i as visited
   visited[i] = 1; pre[i] = count; count++

   //Explore each neighbours of i recursively
   for each (i,j) in E
      if visited[j] == 0
         parent[j] = i
         DFS(j)
         post[i] = count; count++
```

So, this is how we would do this in our algorithm. So, we start by initializing this counter to 0, now whenever I invoke DFS of i, the first thing I do is assign the current counter value to something call the 3 number of i. So, this is the number of the counter before the DFS of i actually started, and then i increment the count. Likewise when I am about to exit, I would mark post of i equal to count, and again i increment the count. So, this is that of... So, in between remember a lot of recursive call (Refer Time: 09:33)). So, this point is not going to the same as the count with i incremented is going to be a lot of ((Refer Time: 09:38)) happening in between, and it terms out this the order in which these 2 numbers pre i and post i, if you across vertices I can actually recover a lot of information.

(Refer Slide Time: 09:48)



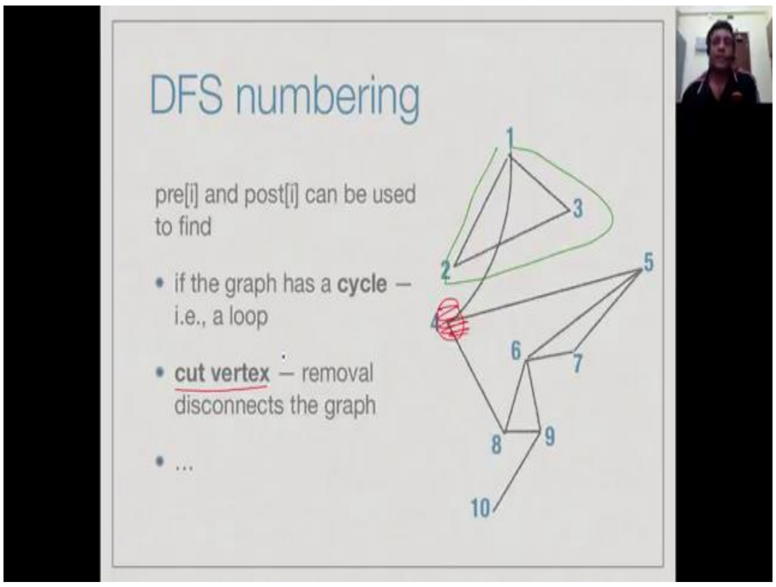
DFS numbering

So, let us look at this example that we did before. So, supposing as before we start at 4. So, we will say that its pre number is say 0, because will start there and then we increment and then we go to 1. So, when we enter 1, the number is the count is 1. So, its pre number is 1, and then we go to 2, its pre number is 2; then we go to 3, that is pre number is 3, and then we immediately if 3, because we do not have any new neighbors to exit. So, it is post number is now 4. So, I am writing to pre number about and the post number below. When I come back to 2 and I have find that this nothing more to be done it twos are exit from twos. So, its post number becomes 5. So, notice that here, I enter at step 2 and left step 5 in between I did 3 and 4 somewhere else. Likewise I come back to 1, and I ((Refer Time: 10:36)) it has nothing new to say. So, now I leave 1 at step 6, then I come back to 4, but I am not finish to 4, because I can do 5.

So, I enter 5 at step 7, from 5 I enter 6 at as step 8, from 6 I enter 7 at step 9, now from 7 I cannot to anywhere else, so I leap 7 and I come back to 6, when from 6 I look further and I have find 8. So, enter 8. So, I was a 10 at that point. So, now I enter 8 step the 11, from 8 I enter 9 at step 12, from 9 I enter step 10 at step 13, when I leaf 10 and I leave 9 at 15, I leave 8 at 16, I leave 7 17, I leave 5 at 18, and I come back to 4 and I have finally finish, thank you.

So, I have with each, each node now these 2 values of 3 value, and 4 step. So, I would pre value and a post value, and it turns out the, this pre and post value can be very helpful.

(Refer Slide Time: 11:46)



DFS numbering

pre[i] and post[i] can be used to find

- if the graph has a **cycle** – i.e., a loop
- **cut vertex** – removal disconnects the graph
- ...

So, we can find out thinks like whether graphs are the cycle. So, this is a cycle. So, whether a graph has a loop like this or we can find out whether there are go to see is like this. So, this is the special vertex, because if I remove this 4 from the graph, in the graph which are earlier connected everything to reach everything no longer connected. So, thus no way to now to get from 1 2 3 to this ((Refer Time: 12:06)). So, are they such cut vertices. So, these are various things that can be computed using these DFS numbers, we will see some of these from computations in later lectures, but this makes DFS actually a much more useful exploration strategy on breadth first search the BFS does give us shortest paths, but on the other hand DFS gives as a wealth of information which is implicitly hidden in this recursive ((Refer Time: 12:31)) exploration which will can explode to find out various structural properties of the graph for free more or less while we are doing DFS, we can find out many, many interesting things about it all.