



IIT Madras

ONLINE DEGREE

Programming Data Structures and Algorithms using Python

Professor Madhavan Mukund

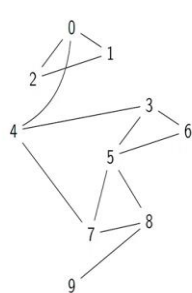
Applications of BFS and DFS

So, we have been looking at graph exploration using breadth first and depth first search and we have primarily focused on reachability and recovering a path and also in the case of breadth first search, we said you also recover the distance in terms of number of edges.

(Refer Slide Time: 00:24)

BFS and DFS

- BFS and DFS systematically compute reachability in graphs
- BFS works level by level
 - Discovers shortest paths in terms of number of edges
- DFS explores a vertex as soon as it is visited
 - Suspend a vertex while exploring its neighbours
 - DFS numbering describes the order in which vertices are explored
- Beyond reachability, what can we find out about a graph using BFS/DFS?



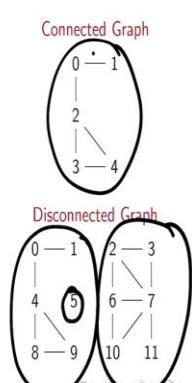
Madhavan Mukund Applications of BFS and DFS

So, both of them systematically compute reachability and since breadth first search works level by level, it also discovers shortest paths. So, the question now we are going to ask is beyond reachability, what can we do with breadth first and depth first search.

(Refer Slide Time: 00:39)

Connectivity

- An undirected graph is **connected** if every vertex is reachable from every other vertex



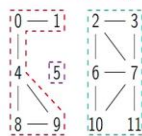
Madhavan Mukund Applications of BFS and DFS

- An undirected graph is **connected** if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components
 - Maximal subsets of vertices that are connected
 - Isolated vertices are trivial components

Connected Graph



Disconnected Graph



Madhavan Mukund

Applications of BFS and DFS

So, one of the important things that we want to know about a graph remember that a graph is not something that we can see. So, if we see this graph visually, we can see that there is something here which is not connected to the rest of the graph, we can also see that the graph as a whole consists of multiple parts which are not connected to each other.

So, we can see that this graph is connected, disconnected whereas this graph is connected. So in a connected graph, we are talking say here about undirected graphs, in a connected directed undirected graph, you can get from everywhere to everywhere, in a disconnected directed graph, there are parts which cannot be reached from other parts. So now, the question really is how do we discover this algorithmically.

So, one way is to find out what are the connected parts, so these are called connected components. So, if the graph is connected, then there will be only one connected component which collects all the vertices, if the graph is disconnected, we will find more than one connected component.


So, connected component is a maximal set of vertices that are connected, that is everything in that set of vertices connected and I cannot add any more vertices to make it connected. So, for example, in the bottom graph this 0, 1, 4, 8, 9, so I cannot add any more vertices and keep it connected, any vertex in that graph below which I add to it will be disconnected from these. So, these 5 vertices form one connected component.

These 6 vertices on the right 2, 3, 6, 7, 10 and 11 are also one connected component and we could also have a trivial connected component which has only one vertex because it has no edges. So, 5 is a connected component, because 5 is not connected to anything else. So it is

connected to in a vacuous way because it is, it is connected to itself even though there is no explicit edge, but there are no edges anyways. So, what we want to know is how to find this.

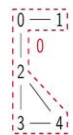
(Refer Slide Time: 02:31)

Identifying connected components

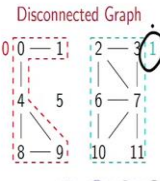


- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j
 - Increment component number to 1
 - Run BFS/DFS from node j
 - Assign each visited node component number 1

Connected Graph




Disconnected Graph




Madhavan Mukund

Applications of BFS and DFS

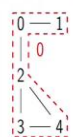


Identifying connected components

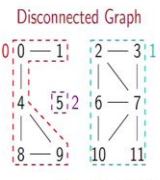


- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j
 - Increment component number to 1
 - Run BFS/DFS from node j
 - Assign each visited node component number 1
- Repeat until all nodes are visited

Connected Graph




Disconnected Graph



Madhavan Mukund

Applications of BFS and DFS



सिद्धिर्भवति कर्मजा

Identifying connected components



- Assign each vertex a **component number**
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j
 - Increment component number to 1
 - Run BFS/DFS from node j
 - Assign each visited node component number 1
- Repeat until all nodes are visited

```
def Components(AList):
    component = {}
    for i in AList.keys():
        component[i] = -1

    (compid, seen) = (0, 0)

    while seen <= max(AList.keys()):
        startv = min([i for i in AList.keys()
                     if component[i] == -1])
        visited = BFSList(AList, startv)
        for i in visited.keys():
            if visited[i]:
                seen = seen + 1
                component[i] = compid
                compid = compid + 1
        return(component)
```

visited
to file



So, the way to do this is to identify these components through reachability. So, we assign each vertex a component number. So, we start from, so now we are really trying to find the number of components, so we are not using BFS or DFS in the usual sense, where we start from one vertex and want to know what all we can reach.

We are asking something about the graph as a whole, it does not really matter where we start, so you might as well start with 0. So, we start with vertex 0 and what it will do is it will explore everything that is reachable from vertex 0. So, we will call all of this to be the 0th component. So, we start our component numbering from 0, so we say everything that we can reach from vertex 0 is an component 0.

So, in the graph on the top which is connected this spans everything, because all the 5 vertices are actually reachable from 0 including 0 itself, whereas in the bottom, it does not span everything. So, we reach 5 out of the vertices, 5 out of the 12 vertices, the remaining 7 vertices are not yet reached.

So, now we assign all these things which you can reach in the first round of BFS or DFS, we assign them the value 0 that is these are all in the 0th component. Now, I look for something which has not been visited. So, I pick the smallest unvisited node. So, in the first graph, there is no such node, in the second graph I pick 2 for instance and I starting with the smallest unvisited graph, unvisited node I run again DFS or BFS and it will now explore all of its reachable vertices, but I have to now record that this belongs to a different component.

So, before I start this, I will increment the component number. So, I will say now whatever I can reach from 2 belongs to component number 1 not 0, because 0 was whatever I could

reach in the first DFS. So, I ran BFS or DFS from component 2 in this case and I will get all the 6 vertices and they will now all have a component number 1 attached to them.

Now, I see if there are still, so I repeat this I see if there are still unvisited vertices, I pick the smallest one namely in this case 5 and then I will explore everything I can reach from 5 which is just 5 itself and I will give it a new component number 2 and now after these 3 BFSs or 3 DFSs, I have now visited all the vertices, so there is nothing left to do. So, this is how I can discover the components in my graph and if there are more than one component in my graph, it means the graph is disconnected, if there is only one component means it is connected.

So, here is a BFS version of this connected thing. So, I want to find the components in a graph which is given by an adjacency list. So, what I do is I initialise that, so this is going to be a dictionary which tells me the component number of each vertex, so component of I is going to tell me which component vertex I belongs to. So, I initialise the component of every vertex to minus 1.

So, if a vertex has component minus 1, it means that it has not been visited yet. So, this is like saying that it is level was minus 1 and BFS. So, this is an implicit way of saying something has not been visited. So, I have a current component ID, which I will compid, and I will mean this is one way to see whether we have finished or not, I will keep track of how many vertices I have actually visited.

So, this is the number seen. So, seen is just a number of vertices which I have been visited. So, when I have seen n vertices, I visited them. So, if I look at the keys of my adjacency list, they run from 0 to $n - 1$. So, so long as I have not reached $n - 1$. So, so long as the number of vertices I have seen is less than $n - 1$, there is still some work to do. So, what is this work, I must find the smallest vertex which has not been seen.

So, I take all those vertices whose component is minus 1 and I take the minimum, so this is a kind of short form for saying, I construct a list of all i whose component value is minus 1 and I take the minimum of that, which is a built in list function in Python and I say that that is going to be my start vertex for my next round of BFS. I am going to start a BFS from here.

So, now I will visit everything from here. Now, notice that when I visit from here, I am not visiting the things in the other components, but I do not care, I just want to know which ones I visited from this start vertex and for each of the nodes that I visit from, so now I am, I have

got visited is set to true because I have kind of explored those things which have been visited from here.

So, if visited is true, then I set this seen to be seen plus 1 and I said the component to be the current component and then eventually I update this. So, the point to note is that this BFS list, is going to start with visited to false for everything. So, each time I start a BFS list with start v, it is as though I start a fresh BFS on that list.

So, the earlier BFS values are not there. So, each time the visited dictionary that I get back from BFS is only true for those things which are reachable from that particular node. So, that is why I am not going to be overwriting the component of anything that I have seen from an earliest starting node.

So, at this point, I go back and I check, have I finished seeing everything, in the process of this has seen reached n minus 1? If not, there is still something to be seen. I look for the smallest unseen thing run another BFS and so on and finally, this component dictionary is the one that I want, it tells me the component ID for every vertex in my graph and by finding out how many values there are in the dictionary and know how many components there are. So, this is a BFS version of this component function.

(Refer Slide Time: 08:53)

Detecting cycles

■ A cycle is a path (technically, a walk) that starts and ends at the same vertex

■ 4 - 8 - 9 - 4 is a cycle

0 — 1
|
2
| \ 4
3

0 — 1 2 — 3
| | \ 4
4 6 — 7
| | \ 8
8 — 9 10 — 11

Madhavan Mukund Applications of BFS and DFS

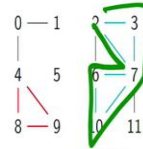
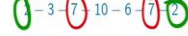
Detecting cycles



- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex

- $4 - 8 - 9 - 4$ is a cycle

- Cycle may repeat a vertex:



Madhavan Mukund

Applications of BFS and DFS

Detecting cycles



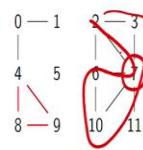
- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex

- $4 - 8 - 9 - 4$ is a cycle

- Cycle may repeat a vertex:

$2 - 3 - 7 - 10 - 6 - 7 - 2$

- Cycle should not repeat edges: $i - j - i$ is not a cycle, e.g., $2 - 4 - 2$



Madhavan Mukund

Applications of BFS and DFS

So, what else can we do with BFS and DFS? Well, one thing we can do is to detect cycles. So, a cycle is a path or technically a walk that starts and ends in the same vertex. We said, remember that a path is something which does not repeat vertices. So, let us look at this, so for example, 4, 8, 9, 4, if I go around counter clockwise, so this is a path which starts at 4, it ends at 4.

So, by definition, the cycle will start and end at the same vertex. So, in that sense, it will not be a path because it will repeat a vertex, so it will be a walk. Another cycle is this one. So, in this cycle, you might also repeat an intermediate vertex. So, we start with 2 go to 3, 7, then we go down here, come back here, come back here and come back here.

So obviously, the first and the last vertex in a cycle must repeat. But in this case, also there is cycle vertex in between that repeats. So, this is allowed. But what we do not want is that we

go round and round the same edge again and again. So, we do not want cycles to repeat stages. So, in particular, we will not say that, for instance this is a cycle.

Otherwise every edge will become a cycle and then it will be a bit problematic, because that is not what we want to really look at. So, we want to really look at non-trivial cycles, which at least go around 3 vertices. So, if I go from i to j and back, that is not a cycle. So, I should never get the same edge in both directions in a cycle. I can repeat vertices, of course, I will repeat the start and end I can also repeat intermediate things, but if I just pass through like this, you know, so like this ray pass through this vertex twice, if I pass through this vertex twice, but I go through it through different edges.

(Refer Slide Time: 10:44)

Detecting cycles

■ A cycle is a path (technically, a walk) that starts and ends at the same vertex

- $4-8-9-4$ is a cycle
- Cycle may repeat a vertex:
 $2-3-7-10-6-7-2$
- Cycle should not repeat edges: $i-j-i$ is not a cycle, e.g., $2-4-2$
- Simple cycle — only repeated vertices are start and end

So, if I do not have this situation, then it is called a simple cycle, because this cycle that I drew earlier decomposes into 2 cycles. So, there is this cycle and there is this cycle. So, this are 2 simple cycles and if I combine them by attaching them at the 7, I get a complex cycle which involves both of them.

So, in many situations, we are interested in acyclic graph. So, a graph is acyclic which has no cycles. So, the graph at the bottom is actually cyclic as we saw, because there are cycles in the graph. Whereas this graph on the top is a cyclic because if I start anywhere, I cannot come back following the edges in the graph.

(Refer Slide Time: 11:24)

BFS tree

- Edges explored by BFS form a tree
 - Technically, one tree per component
 - Collection of trees is a forest

Acyclic Graph

```

graph TD
    0 --- 1
    0 --- 2
    2 --- 3
    2 --- 4
  
```

Graph with cycles

```

graph TD
    0 --- 1
    0 --- 2
    2 --- 3
    2 --- 4
    4 --- 5
    5 --- 6
    6 --- 7
    7 --- 8
    8 --- 9
    9 --- 10
    10 --- 11
    11 --- 12
    12 --- 13
    13 --- 14
    14 --- 15
    15 --- 16
    16 --- 17
    17 --- 18
    18 --- 19
    19 --- 20
    20 --- 21
    21 --- 22
    22 --- 23
    23 --- 24
    24 --- 25
    25 --- 26
    26 --- 27
    27 --- 28
    28 --- 29
    29 --- 30
    30 --- 31
    31 --- 32
    32 --- 33
    33 --- 34
    34 --- 35
    35 --- 36
    36 --- 37
    37 --- 38
    38 --- 39
    39 --- 40
    40 --- 41
    41 --- 42
    42 --- 43
    43 --- 44
    44 --- 45
    45 --- 46
    46 --- 47
    47 --- 48
    48 --- 49
    49 --- 50
    50 --- 51
    51 --- 52
    52 --- 53
    53 --- 54
    54 --- 55
    55 --- 56
    56 --- 57
    57 --- 58
    58 --- 59
    59 --- 60
    60 --- 61
    61 --- 62
    62 --- 63
    63 --- 64
    64 --- 65
    65 --- 66
    66 --- 67
    67 --- 68
    68 --- 69
    69 --- 70
    70 --- 71
    71 --- 72
    72 --- 73
    73 --- 74
    74 --- 75
    75 --- 76
    76 --- 77
    77 --- 78
    78 --- 79
    79 --- 80
    80 --- 81
    81 --- 82
    82 --- 83
    83 --- 84
    84 --- 85
    85 --- 86
    86 --- 87
    87 --- 88
    88 --- 89
    89 --- 90
    90 --- 91
    91 --- 92
    92 --- 93
    93 --- 94
    94 --- 95
    95 --- 96
    96 --- 97
    97 --- 98
    98 --- 99
    99 --- 100
  
```

BFS tree

- Edges explored by BFS form a tree
 - Technically, one tree per component
 - Collection of trees is a forest
- Any non-tree edge creates a cycle
 - Detect cycles by searching for non-tree edges

Acyclic Graph

```

graph TD
    0 --- 1
    0 --- 2
    2 --- 3
    2 --- 4
  
```

Graph with cycles

```

graph TD
    0 --- 1
    0 --- 2
    2 --- 3
    2 --- 4
    4 --- 5
    5 --- 6
    6 --- 7
    7 --- 8
    8 --- 9
    9 --- 10
    10 --- 11
    11 --- 12
    12 --- 13
    13 --- 14
    14 --- 15
    15 --- 16
    16 --- 17
    17 --- 18
    18 --- 19
    19 --- 20
    20 --- 21
    21 --- 22
    22 --- 23
    23 --- 24
    24 --- 25
    25 --- 26
    26 --- 27
    27 --- 28
    28 --- 29
    29 --- 30
    30 --- 31
    31 --- 32
    32 --- 33
    33 --- 34
    34 --- 35
    35 --- 36
    36 --- 37
    37 --- 38
    38 --- 39
    39 --- 40
    40 --- 41
    41 --- 42
    42 --- 43
    43 --- 44
    44 --- 45
    45 --- 46
    46 --- 47
    47 --- 48
    48 --- 49
    49 --- 50
    50 --- 51
    51 --- 52
    52 --- 53
    53 --- 54
    54 --- 55
    55 --- 56
    56 --- 57
    57 --- 58
    58 --- 59
    59 --- 60
    60 --- 61
    61 --- 62
    62 --- 63
    63 --- 64
    64 --- 65
    65 --- 66
    66 --- 67
    67 --- 68
    68 --- 69
    69 --- 70
    70 --- 71
    71 --- 72
    72 --- 73
    73 --- 74
    74 --- 75
    75 --- 76
    76 --- 77
    77 --- 78
    78 --- 79
    79 --- 80
    80 --- 81
    81 --- 82
    82 --- 83
    83 --- 84
    84 --- 85
    85 --- 86
    86 --- 87
    87 --- 88
    88 --- 89
    89 --- 90
    90 --- 91
    91 --- 92
    92 --- 93
    93 --- 94
    94 --- 95
    95 --- 96
    96 --- 97
    97 --- 98
    98 --- 99
    99 --- 100
  
```

So, the way to explore this using BFS is to observe that if I start with BFS and I record the edges that are used for BFS, remember when I use BFS when I find an unvisited neighbour, I

add it to the queue and I record this parent thing. So, in some sense, I am recording this edge from j to k was used in BFS.

So, these used edges actually will form a tree because I will never come back to a node which I have already seen and so, BFS guarantees that I will never add an edge to a vertex which has already been visited. So, I will never that, so if there is a cycle it must be of that form, I have I have already seen it from an earlier node and then through some long path I tried to come back and visit it again.

So, this cannot happen, because BFS does not allow you to visit the same vertex twice. So, BFS will always visit edges that form a tree. Now, of course technically, it may not be a tree because a tree is connected graph on n with $n-1$ vertices and this may not be a tree in that sense, but it might be a collection of trees like on the bottom here, these red edges are what maybe BFS does and it forms a collection of trees and a collection of trees from the English term is called a forest.

So, the point we want to observe is that if there is an edge which is not used by BFS, then if I add that edge, it must correspond to an edge which I discarded because I, the target of the edge was already visited and that means that there must have been a cycle. So, any non-tree edge must cause a cycle. So, if I look at these blue, green edges, the bottom, all of these actually correspond to cycles.

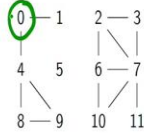
So, this is a cycle, this is a cycle, this is a cycle, this is a cycle. So, any of these if I add through the cycle. Now here, for instance, there are no such edges, because the tree itself exhausts all the edges and there is no other way of reaching any of the vertices except for following the BFS. So, if I have a non-tree edge in a BFS tree, that is the cycle.

(Refer Slide Time: 13:31)

DFS tree



- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)



L pre
0 0

Navigation icons

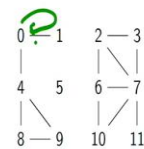
Madhavan Mukund

Applications of BFS and DFS

DFS tree



- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)



L pre
0 0
1 2
post

Navigation icons

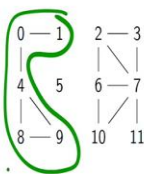
Madhavan Mukund

Applications of BFS and DFS

DFS tree



- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)



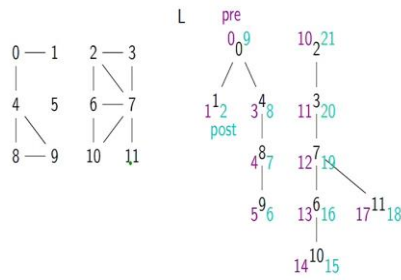
L pre
0 0
1 2
post
4 8
3 7
5 6

Navigation icons

Madhavan Mukund

Applications of BFS and DFS

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)



Madhavan Mukund

Applications of BFS and DFS

Now, we can do the same thing with the DFS tree. So, let us see what happens to DFS tree. So, in a DFS tree, I will do something more complicated. So, I will maintain a counter which tells me when I explored a vertex and when I finished exploring a vertex, so in the BFS tree, I just kept track of which edges I explored. In a DFS tree, I will actually keep track of a more complicated thing I will say, I will count in some sense that time that I started exploration and finished exploring the vertex. So, here is what we will do, so we will assign a pre and a post. So, let us look at an example.

So, supposing I start with 0. So, I start exploring it at step 0. So, I assign the vertex 0 the pre-number 0 and now I this counter will keep incrementing. So, from 0 I will go to 1. So, I enter 1 at step 1. So, this is the pre-number for 1, but 1 is a dead end. So, I will finish with 1 and now I will say I finished with 1 at step 2. So, each vertex has a purple number, a pre-number and a green number a post number. So, the pre-number is when I started it and the post number is when I finished it and the post number will be after the pre-number and I keep incrementing.

So, now I will come back to 0. So, I have gone here and I have come back to 0 but there is I have not finished with 0 yet. There is more work to do. So, I will now enter 4 at step 3. Because I have not finished with 0, I will not mark it as done. Instead, I will say that I increment a counter which I left it 2. I will increment to 3 and I will start with vertex 4, 4 will now explore its smallest neighbour which is 8.

So, 8 will now be explore at step 4, 8, will explore 9, so 9 will be explored in step 5. But these are all slippery numbers because I have not finished any of these, I just going forward.

When I come backwards, I close. So, now 9 is a dead end, because from 9 I can only come back to 4 which is already marked or 8, which is already marked.

So, now I say 9 is finished. So, I mark 9 as closed and say its post number is step 6, I come back to 8, 8 is done. So, its post number is 7, 4 is done, its post number is 8. Now, I come back to 0, I have no further things to do 0. So, its post number is 9. So, I finished exploring this component.

Now remember, what happens next is I look for the smallest unmarked vertex, so the smallest unmarked vertex is 2. So, I start with 2, but I continue my numbering. So, I want to record the fact that I reached 2 only after finishing everything that I saw from 0. So, when I finished 0, I was at step 9.

So, I start 2 at step 10. So, from 2, I go to 3, which is exclude smallest neighbour, which is the step 11, from 3, I go to 7 which is its smallest neighbour, from 7, I go to 6 which is its smallest neighbour at each point, I keep incrementing this counter and setting it as the starting point. So, I started 2 at 10, I started 3 at 11, started 7 at 12, I started 6 at 13, from 6 I go to 10, I started 10, at 14.

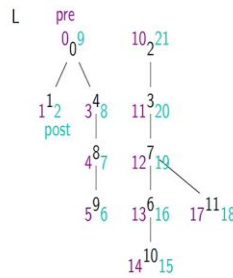
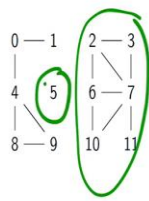
Now, from 10, I cannot go anywhere, because 6 and 7 are done. So, I finished 10 at 15, I come back to 6, 6 has only 2 and 7. So, 6 is also done at 16, I come back to 7, now 7 is not done, because 7 went this way and came down here and then down come back. So, now I can still go to 11, I can still go to 11. So, from 7 I can increment the counter to 17 and go to 11. But 11 is a dead end, so I finish 11 and 18, now I come back to 7, 7 is a dead end, so I finish 7 at 19. Now I finish 3 at 20 and I finish 2 at 21.

(Refer Slide Time: 17:22)

DFS tree



- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)



Navigation icons: back, forward, search, etc.

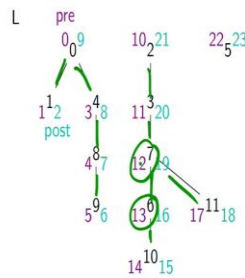
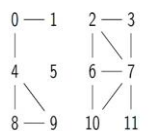
Madhavan Mukund

Applications of BFS and DFS

DFS tree



- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)



Navigation icons: back, forward, search, etc.

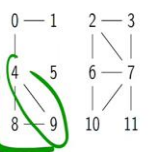
Madhavan Mukund

Applications of BFS and DFS

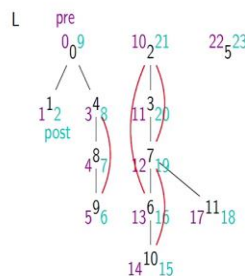
DFS tree



- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)



- As before, non-tree edges generate cycles



Navigation icons: back, forward, search, etc.

Madhavan Mukund

Applications of BFS and DFS

So, now starting at step 10 and going up to step 21 I finished these 6 vertices, now I have to do, again look for the smallest unmarked vertex which is actually 5. So, I have to look at it and continue in the sequence. So, 5 will now start at step 22. So, I start at step 22 and then I come back at step 23. Because that is it and now there are no unmarked vertices and I will stop. So, this way when I am doing my depth first search, not only am I keeping track of this tree, so I have this tree here, this forest, but I am also keeping track of this order in which I went in and came out. So, it said that I did 6 after I 7 and so on.

So, as before any edge which is there in the graph, but which is not in my tree creates a cycle. So, for instance I have this edge from 4 to 9 for instance. So, the way I did my depth first search I came this way and then I backtrack that is because 9 to 4 would have created a cycle. So, in this if I look at my picture on the right, it creates a non-tree, so any non-tree generates a cycle.

(Refer Slide Time: 18:34)

DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)

- As before, non-tree edges generate cycles
- To compute *pre* and *post* pass counter via recursive DFS calls

```

L
(visited, pre, post) = ({}, {}, {})

def DFSInitPrePost(AList):
    # Initialization
    for i in AList.keys():
        visited[i] = False
        pre[i], post[i] = (-1, -1)
    return

def DFSPrePost(AList, v, count):
    visited[v] = True
    pre[v] = count
    count = count + 1
    for k in AList[v]:
        if (not visited[k]):
            count = DFSPrePost(AList, k, count)
    post[v] = count
    count = count + 1
    return(count)

```

Madhavan Mukund

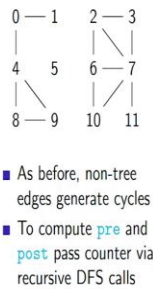
Applications of BFS and DFS

सिद्धिर्भवति कर्मजा

DFS tree



- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (*pre*) and exit number (*post*)



- As before, non-tree edges generate cycles
- To compute *pre* and *post* pass counter via recursive DFS calls

```
(visited, pre, post) = ({}, {}, {})
```

```
L
def DFSInitPrePost(AList):
    # Initialization
    for i in AList.keys():
        visited[i] = False
        pre[i], post[i] = (-1, -1)
    return

def DFSPrePost(AList, v, count):
    visited[v] = True
    pre[v] = count
    count = count + 1
    for k in AList[v]:
        if (not visited[k]):
            parent[k] = v
            count = DFSPrePost(AList, k, count)
    post[v] = count
    count = count + 1
    return(count)
```



So, here is an implementation of this DFS with this pre and post. So, we will use this global implementation, remember in DFS, we had this implementation we passed around these dictionaries and the implementation where we did not pass around. So, here I will dispense with the parent information, I am just in keeping the pre and the post.

So, I have this visited dictionary and I have now these two things which record pre of i and post of i, so all of them are empty. So, I initialise them by saying that for every vertex visited is false. So, there should be a v here and pre and post are both set to minus 1 and now, the way it works is that I start DFS, so I have to keep this counter going, remember this counter is a running counter, which starts at 0 and as I go on it keeps getting incremented.

So, this running counter is initially 0. So, we will have to call this thing 0 which I have not shown, but now when I come to visit a vertex I set pre to be the current value of the counter and then I increment the counter, then having incremented the counter I do the usual thing which is I look at all the neighbours and if they are not visited I set, so this parent should not be there. So, let me just remove this, so this parent is not there.

So, I do not need to keep the parent, but what I will do is I will recursively call this DFS for k, with the new value of counter. Remember I have currently I have just incremented the counter, so I am now going to start this. So, when I reach that counter, its pre value will be the current count plus 1 and when I finish all these recursive calls and I am about to exit, I will set my post value to be the current value of the counter and I will again, increment the counter and leave so that the next day will get the, so every vertex which is explored, can work with the current value of the counter that is returned by this DFS, then assign it as pre or

post, depending on whether its call is coming from pre or post and then incremented to pass it on to the next round.

So, this is the only work that we have to do is to maintain this extra pre and post dictionary as a global dictionary and update it before and after each vertex, each DFS call and pass this counter around and increment it every time we assign it. So, we assign pre-increment, call it with a new value of count, I have to get back the value of count. So, I have to keep this count running through all these DFS calls, because the same count is being used in this whole sequence.

So, this is showing you for one particular search. Now, if you want to do the earlier thing, you have to overlap it with the component thing and do this counting on one component, then do it on the next component, do it on the next component and keep the count value circulating through all of these, so that I have not shown in this.

(Refer Slide Time: 21:32)

Directed cycles

■ In a directed graph, a cycle must follow same direction

- $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
- $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not

Madhavan Mukund Applications of BFS and DFS

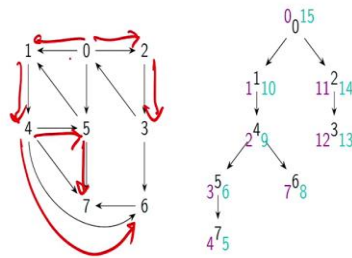
Directed cycles



- In a directed graph, a cycle must follow same direction

- $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle

- $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not



Navigation icons: back, forward, search, etc.

Madhavan Mukund

Applications of BFS and DFS

Directed cycles

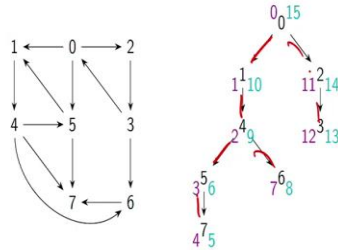


- In a directed graph, a cycle must follow same direction

- $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle

- $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not

- Tree edges



Navigation icons: back, forward, search, etc.

Madhavan Mukund

Applications of BFS and DFS

Directed cycles



- In a directed graph, a cycle must follow same direction

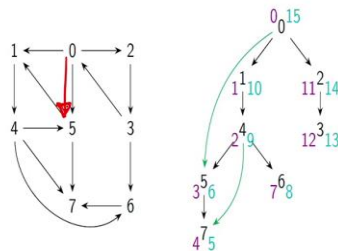
- $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle

- $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not

- Tree edges

- Different types of non-tree edges

- Forward edges



Navigation icons: back, forward, search, etc.

Madhavan Mukund

Applications of BFS and DFS

So, what can we do with these numbers? I mean, in the undirected case, we just said this non-tree edges will give us the cycles. But in a directed case, the natural nature of a cycle is a little bit more complicated. So, for instance, if you look at this, if we look at 0 to 3 and 0, then this is a cycle because it goes around following the arrows.

Whereas what looks like a cycle in the picture, if I say 0 5 1 0, this is not a cycle, because this last arrow is going the wrong direction, I can go from 0 to 5 I can go to 5 to 1, but I cannot come back from 1 to 0. So, it is not so straightforward. So, a cycle must follow the edges. So, if I do the same DFS, remember the DFS will work whether the graph is directed or undirected. So, if I do the same DFS on this particular graph, then these are the pre and post numbers.

As usual, the purple numbers are the pre-numbers and the greenish numbers are the post numbers. So, I start with 0, then I will go from 0, I will explore 1 from 1, I will explore 4, from 4 I will explore 5, from 5 I will get stuck. So, from 4 I will then explore. So, from 5 I will explore 7 and 7 has no outgoing edges. So, I get stuck, then I come back and then I come back to 4 and I can explore 6.

So, this is the left-hand side. Then after that I am done with that side. So, then I will go from 0 to 2 and from 2, I will come to 3 and that is the right answer. So, this is how this DFS actually was explored and these are the numbers. So now if I look at this, of course, I have the tree edges. So, I have these edges, which are edges from the original graph, which I have used in my DFS. So, all these mark edges are tree edges.

But I have many different types of non-tree edges. So first of all, I have edges which follow along the paths in this tree, so I have edges that go forward go down a path in the tree. So, 0 is connected to 5 in the tree via 1 and 4. But there is also a direct edge from 0 to 5 in my original graph, which is not there in my DFS tree. So, this is a forward edge, so it goes in the tree, from a node to something below it in the tree.

(Refer Slide Time: 23:46)

Directed cycles



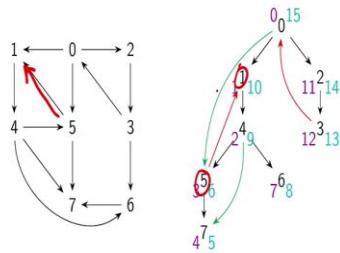
- In a directed graph, a cycle must follow same direction

- $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
- $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not

- Tree edges

- Different types of non-tree edges

- Forward edges
- Back edges



Madhavan Mukund Applications of BFS and DFS

Directed cycles



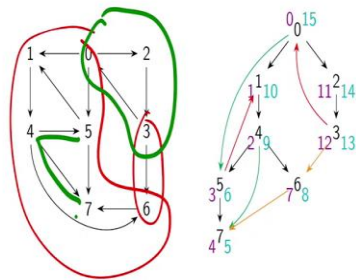
- In a directed graph, a cycle must follow same direction

- $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
- $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not

- Tree edges

- Different types of non-tree edges

- Forward edges
- Back edges
- Cross edges



Madhavan Mukund Applications of BFS and DFS

Directed cycles



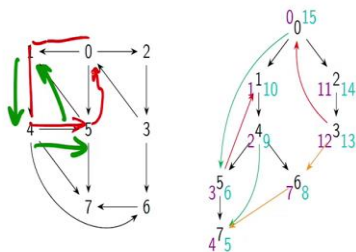
- In a directed graph, a cycle must follow same direction

- $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
- $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not

- Tree edges

- Different types of non-tree edges

- Forward edges
- Back edges
- Cross edges



Madhavan Mukund Applications of BFS and DFS

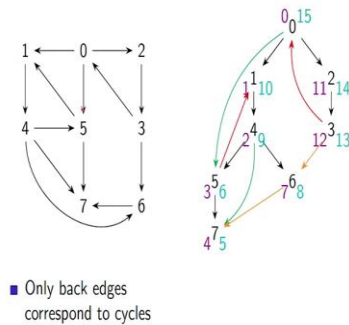
- In a directed graph, a cycle must follow same direction

- $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
- $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not

- Tree edges

- Different types of non-tree edges

- Forward edges
- Back edges
- Cross edges



- Only back edges correspond to cycles

Navigation icons: back, forward, search, etc.



Madhavan Mukund

Applications of BFS and DFS

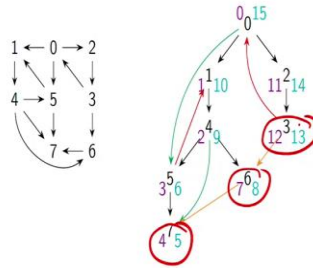
And of course, symmetrically, there is a backward edge, something which comes from below to above. So, we have an edge, for example, from 5 to 1, so 5 to 1 is not there in my tree, but it is an edge from a node below to a node above in the tree. So, it is connected by path in the opposite direction, this is a backward edge and finally there will be edges which go across branches in the tree.

So, if I look at 3 to 6 for example. So, this part, if you remember, was explored from this side and this part was explored from the side. So, these were part of two different explorations. So, the edge from 3 to 6 is an edge from the right-hand side to the left-hand side. Similarly, the edge from 6 to 7 is also something which we will explore because these were explored in two different settings. So therefore, 6 to 7 is also different edge. So, these are called cross edges. So, they do not go from ancestor to child or child to ancestor but they go across.

So, we have these non-tree edges come in these three flavours either they go down the tree, they are forward edges or they go up the tree, they are back edges or they across the tree. They are cross edges. Now, which of these are cycles? So, remember we said that if you look at this edge, for instance, 8, 5, 2, 1.

So, the thing is that we have an edge from 1 to 4 we have an edge from 4 to 5 and we have an edge from 5 to 1, which goes back and this is indeed a cycle, what about the 0 to 5 edge. So, we have an edge from 0 to 1 and then we have an edge from 1 to 4 and then we have an edge from 4 to 5. But now 0 to 5 in the wrong direction, it does not complete a cycle, I need an edge from 5 to 0. So, 0 to 5 does not make, I mean 1 to 5 does not make a cycle here. So, it turns out that only the back edges will give you cycles, the cross edges and the forward edges will not give you the cycle.

- Use pre/post numbers
- Tree edge/forward edge (u, v)
Interval $[pre(u), post(u)]$ contains $[pre(v), post(v)]$
- Back edge (u, v)
Interval $[pre(v), post(v)]$ contains $[pre(u), post(u)]$
- Cross edge (u, v)
Intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are disjoint



Navigation icons: back, forward, search, etc.

Madhavan Mukund

Applications of BFS and DFS

So, we can find out whether an edge is a back edge or a forward edge or a tree edge by using this pre and post number. So, if a tree edge is a forward edge, so let us look at this edge for instance. So, it is going from some u to some v , then what this means is that v was processed under u . So, the starting of v happened after u started and the ending of v happened before u finished. So, if I look at the interval 0 to 15, which is the entire time that I was processing u and I look at the time that I was processing v , so this is the u interval and this is the v interval, the v interval is included in the u interval.

So, if I see an edge from u to v and the interval associated with the target is subsumed by the interval of the top, it means that this entire node happened below this node. So, that means it is a forward edge. The backward edges the reverse, this basically tells me this ancestor child relationship, a node is an ancestor of another node, if its interval spans the interval of the lower node, so a back edge will go from something which is sitting inside.


So, if I want to look at, for example the edge from 5 to 1, I will say that 3 to 6 you are sitting inside 1 to 10. So, if the starting interval is sitting inside the ending interval, then it is a back edge if the starting interval is subsuming the other intervals, so it is a forward edge and finally, if the 2 edges, 2 intervals are kind of overlapping or actually if they are disjoint, so if I look at this, I have 4, 5 and I have 7 8 there is no, they will not overlap, they will just be disjoint.

So, I have 7 8 and I have 12 13. So, they do not. So basically, while I was doing 6 I was not doing 7, while I was doing 7 I was not doing 6, while I was doing 6 I was not doing 3, while I was. So, this interval basically tells me I started doing it and I ended up doing it and anything which has a number in between happened while I was suspended or working on this node.

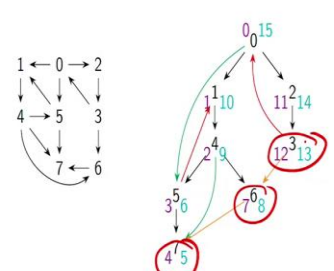
So, forward edge will say the entire node below was processed while I was working, the back edge will say that I was done, I was done before that and so on and the cross edge will say we happen to disjoint them. So, by looking at these pre and post numbers, I can do this classification. So, therefore calculating this pre and post numbers gives us extra information.


(Refer Slide Time: 28:47)

Classifying non-tree edges in directed graphs




- Use pre/post numbers
- Tree edge/forward edge (u, v)
Interval $[pre(u), post(u)]$ contains $[pre(v), post(v)]$
- Back edge (u, v)
Interval $[pre(v), post(v)]$ contains $[pre(u), post(u)]$
- Cross edge (u, v)
Intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are disjoint






Madhavan Mukund

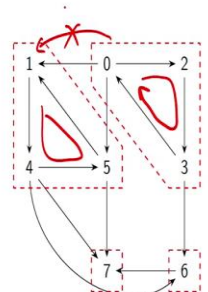
Applications of BFS and DFS




Connectivity in directed graphs




- Take directions into account
- Vertices i and j are strongly connected if there is a path from i to j and a path from j to i
- Directed graphs can be decomposed into strongly connected components (SCCs)
 - Within an SCC, each pair of vertices is strongly connected





Madhavan Mukund

Applications of BFS and DFS



We can also do other stuff with this. So, for instance, this notion of connectivity, which we talked about. So, connectivity, we said was something which tells us whether everything is reachable from everything. But in case of a directed graph, reachability also has to take into account the direction.

So, if I look at direction, then I will say that i and j are strongly connected. If I can go from i to j and I can go from j to i it is not enough to go in just one direction. So, these are what are

called strongly connected components. So, in the graph that we have on the right, the red marked portions are strongly connected, because I can go around this from anywhere to anywhere I can go around this from anywhere, anywhere.

Though I can go from 0 to 1, I cannot come back, there is no way to come back from this component to that. So, this does not help me, the fact that it is connected in one direction. So, strongly connected components correspond to subsets of vertices where I can go from anywhere to anywhere and it turns out that we can also use this DFS numbering to compute strongly connected components, but we would not do that now in this course.

(Refer Slide Time: 29:48)

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
 - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
 - Only back edges generate cycles
 - Classify non-tree edges using DFS numbering
- Directed graphs decompose into strongly connected components
 - DFS numbering can be used to compute SCC decomposition
- DFS numbering can also be used to identify other features such as articulation points (cut vertices) and bridges (cut edges)

Madhavan Mukund Applications of BFS and DFS

So, to conclude, BFS and DFS are first can do reachability they can talk about parents and paths and BFS can also tell you about distance. But in addition to that, we can find connected components. We can also find these cycles by looking for non-tree edges and by using this numbering scheme for DFS, we can also find forward back and cross edges because only back edges generate cycles.

So, DFS numbering is a very powerful way of recording the progress of DFS and uncovering the structure of the cloud of the graph and then connectivity also is more complicated for directed graphs, it is not enough to just be part of the same component as in terms of one directional edges you need bi directional connect, we need to go from i to j and j to i . So, the DFS numbering can also be used for strongly connected components.

There are also other structural features. So, you might ask can I if I did, if I remove this vertex will the graph fall apart? If I remove this edge will the graph fall apart? So, these are

important things. So, these are like if you have a telephone or internet cable and if one cable is snapped, when the whole network can be disconnected or if this router breaks, will our network become disconnected.

So, these are important points to find out in a graph and these DFS numbering can also help to find these. So, DFS numbering is actually a very powerful tool and that is why as I mentioned before, though, it seems to be more complicated as a concept because it uses recursion. It seems more complicated as a concept than BFS. Actually, DFS gives you a lot more information about the graph and therefore it is really the preferred tool normally for exploring a graph.

