

IIT Madras
ONLINE DEGREE

Mathematics for Data Science 1

Professor Madhavan Mukund

Applications of BFS and DFS-1

(Refer Slide Time: 0:15)

Applications of BFS and DFS

Madhavan Mukund
<https://www.cmi.ac.in/~madhavan>

Mathematics for Data Science 1
Week 10

So, we have looked at breadth first search and depth first search as two ways to systematically explore a graph.

(Refer Slide Time: 0:21)

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 neighbours
 - 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?

Graph diagram showing vertices 0 through 9 and their connections.

So, what we are going to look at now is how to go beyond just reachability, so what we have done so far is starting with a vertex how to find out what all we can reach in that graph, and we

said that BFS and DFS are two systematic strategies to do this. So when we do BFS, we do it level by level, so we start with its vertex, we go to its nearest neighbors, then from those nearest neighbors we go to the next set of neighbors and so on.

So in this process, one of the things we said is that BFS will discover the shortest path, the shortest level for every reachable vertex because it is processing the graph layer by layer in some sense, everything which is reachable in one round, in two edges and three edges and so on. And of course the whole point of calling it breadth-first search is that we need a systematic way to do this, so we had this queue which kept track of how to make sure that we explore all the vertices in this level by level order.

So everything at level one is put into the queue and it will get processed before everything at level two and this will guarantee that everything is reached in the shortest number of levels. Now DFS was a very different strategy, it was in some sense an aggressive strategy, the moment it moved to a neighbor, it would suspend the current vertex and then it will start exploring the neighbor. I mean in a way it is a bit like how when we start looking up information on the internet, right?

So, we start looking for something and then before we finish reading the article we find an interesting link and we go to that link and we start reading that link, that has another link and we go to that link and so on. So eventually we have to remember to come back to where we were reading in the first place.

So DFS is like that, you start with a vertex, look at any vertex that is neighboring it which you can explore, go down that path and only when you run out of things to see you come back to the original vertex. So you keep this stack of suspended vertices in DFS. So the question that we are going to address in this lecture is what more we can do then just reachability with BFS and DFS. So, is BFS and DFS only for reachability or are there more interesting things that we can do?

(Refer Slide Time: 2:16)

Connectivity

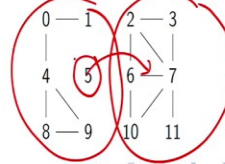


- An undirected graph is **connected** if every vertex is reachable from every other vertex

Connected Graph



Disconnected Graph



Madhavan Mukund

Applications of BFS and DFS

Mathematics for Data Science

Connectivity

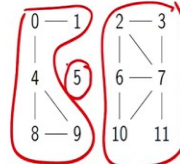


- An undirected graph is **connected** if every vertex is reachable from every other vertex

Connected Graph



Disconnected Graph



Madhavan Mukund

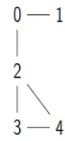
Applications of BFS and DFS

Mathematics for Data Science

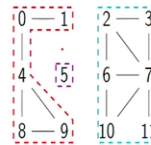
सिद्धिर्भवति कर्मजा

- 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

Mathematics for Data Science

So, first aspect of a graph that we will explore is that of connectivity. So we say that a graph, an undirected graph is connected if every vertex is reachable from every other vertex. So you can on the right two graphs, the first graph is clearly connected from any vertex of 0 1 2 3 4, you can go to every other vertex. But if you look at the bottom you have clearly two disconnected components.

So, you have this left hand side component and you have this right hand side component and there is no way you can go from here to there and in fact 5 on its own is also isolated from everything else. So that is not a component on its own, so really technically you have these components which are, so you have this component, everything here is reachable from itself. Here this component, everything inside this component is reachable from within that component and finally we have this component which consists of just one vertex

So, in a disconnected graph we can identify these components. So what we want to do is put these red border around this component and this blue border around the second component and then find also in particular these isolated vertices which are what you might think of is trivial components. So we said that technically there are no self-edges in a graph, we said that we do not assume there are edges from i to i .

So, when we say a single vertex is a component we are just saying that it cannot reach anything and so there is nothing else that it can be connected to which can come back to it. So these are

the trivial components. So our goal is to see how BFS or DFS can help us identify these components.

(Refer Slide Time: 3:54)

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

Connected Graph

Disconnected Graph

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

So, when we are doing one of these, either BFS or DFS, what we do is, just like we kept track of extra information like in BFS we kept track of the level number and the parent information and even in DFS we said we could keep track of parent information, so we will have one more component number that we keep track of. So we have a number which we are going to use to label these components, so we are going to call it component 0, component 1, component 2 and so on.

So, we have a component number and we attach component numbers to vertices. So initially we want to do it for the whole graph, so we might as well start at vertex 0, we want to find out how many components are there in this graph. So we start at vertex 0, with either BFS or DFS it does not matter, and then this new quantity that we are going to assign to vertices which is the component number, we initialize this quantity to 0.

So now, when we do a BFS or a DFS from vertex 0, we will reach some vertices and all of these vertices will be connected because they are all reachable from 0 and therefore they are reachable from each other, in fact can reach it from 0, if I can reach say 0 to i , and I can reach 0 to j , then I can go from j to i by going back to 0 and then because these are undirected edges, you can always go backwards.

So, if I can go from 0 to some vertex i and I can go to 0 to some vertex j , then there is a connection from i to j also. So everything that you get in a single component, in a single scan of BFS or DFS is going to form a connected component. So what you do is that, while you are performing the scan you remember this component number is 0 and you just assign component number 0 to all of these. This is an extra piece of information that you keep just like we keep visited v , we keep component of v and we just keep assigning component of v equal to 0 for all these vertices.

Now, if you are in the connected graph like in the first case, all the vertices would have been covered. But if you are in a disconnected graph like in the bottom case, there are some vertices after you have reached everything that you can reach from 0, there are some vertices which are not yet marked as visited. So this means that they are not in component 0, they are in another component, so you have to find that.

So, you have to pick any one of them, you have to pick any one of the vertices which are not visited and start a new breadth-first or depth first search from there. So in particular let us pick the smallest one, so supposing we identify the two in the second graph, in the first graph there is nothing to do because everything is already being visited, but here we identify that vertex 2 is a candidate to start of a new BFS or DFS because it was not visited when I started from 0.

But now this is going to be a different component. So we cannot call the component that we are going to discover starting from 2, the same as the component we already discovered. So we have

to increment the component number, so now we are looking at vertices which will be called component 1. So we perform this breadth-first search or depth first search, we will find that all those 6 vertices on the right hand side are reachable from 2. And as we are done with the first component while we are doing this we will attach the current component number which is 1 to all of these.

So, now after two DFSs or two BFSs I have identified component 0 and component 1, but at this point I still have an unvisited vertex, so I will repeat this. So in general there may be many components, so each time I finish one component I will look at the remaining unvisited vertices and start yet another DFS or BFS from safer instance the smallest numbered vertex in that set.

So, we just keep repeating this, in this case we only have to do it one more time because we have only one such vertex, vertex 5 and because we cannot go anywhere from 5 this BFS rapidly stops and we get a component 2. So remember each time we repeat we also increment the component number, so we increment the component number and start a fresh scan. So this is how we can build in a component discovery algorithm to BFS or DFS.

So, while we are doing BFS and DFS we can also discover what are the connected components in the graph and label them. So we now know at the end of this, that for example 9 and 4 in the bottom graph are both in component 0, so they are in the same component, whereas 9 and 10 have different component number 0 and 1, so 9 and 10 are not in the same component. So it is important that we can at the end of this look at two vertices and decide are they connected to each other or not without having to start another breadth-first search from those vertices and then try to again connect them.

(Refer Slide Time: 8:27)

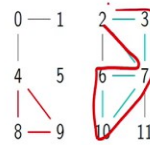
Detecting cycles



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



Madhavan Mukund

Applications of BFS and DFS

Mathematics for Data Science

Detecting cycles



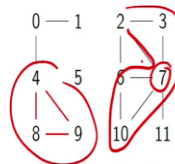
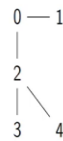
- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex

- $4-8-9-4$ 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



Madhavan Mukund

Applications of BFS and DFS

Mathematics for Data Science

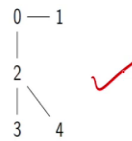
सिद्धिर्भवति कर्मजा

Detecting cycles

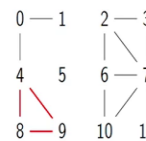


- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex
 - $4-8-9-4$ is a cycle
 - Cycle may repeat a vertex:
 $2-3-7-10-6-7-2$
 - Cycle should not repeat edges: $i-j-i$ is **not** a cycle, e.g., $2-4-2$
 - **Simple cycle** — only repeated vertices are start and end
- A graph is acyclic if it has no cycles

Acyclic Graph



Graph with cycles



So, related to breadth-first search and depth first search is also the idea of a cycle. So a cycle as you could imagine is something which is circular, so a cycle in a graph is a path which starts at some vertex and then comes back to that same vertex. So remember that we said technically that a path does not repeat vertices, so technically this is a walk because we start at a vertex, traverse some edges and come back to the same vertex.

So, if you look at the bottom graph for instance, 4 then 8 then 9 and then back to 4. So this is a cycle. Here is a more complicated cycle, 2 to 3 to 7, so I start at 2, then I go to 3 and then I come to 7, but then I go to 10 and then I come back to 6 and then I come back to 7 and then I come back to 2. So in this case the walk not only repeats the starting and the ending point 2 but it also repeats 7 along the way. So this is also called a cycle.

But though you can repeat vertices, you cannot repeat edges, so you cannot claim that this is a cycle, I went to 2 and then I came back, this is not a cycle. So a cycle cannot go back and forth along the same edge, otherwise every edge will become a cycle and we do not really intend that. So finally what we are typically interested in is what are called simple cycles. So simple cycle is like this one.

So, this one is a simple cycle because it went from start to end and came back to start rather, so the only vertex that was seen twice was a starting vertex when we closed the cycle. Whereas, this one was not simple because we went down and then we came back up and then we visited this 7

twice, so it is actually two simple cycles which have been joined at 7. So we can do 2 3 7 and then we can do 7 10 and 6 7.

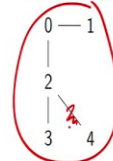
So, if a graph does not have any cycles, then it is called acyclic. So this graph is acyclic because there are no cycles in it. Whereas, the graph on the bottom is not acyclic, so we do not really call it acyclic graph, we only interest, we are interested in whether it is acyclic or it is not acyclic. So these graph has cycles and what we would like to do now is say that if a graph has cycles how do we find these cycles. That is our goal.

(Refer Slide Time: 10:51)

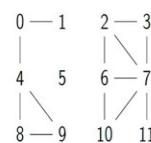
BFS tree

- A tree is a minimally connected graph

Acyclic Graph



Graph with cycles



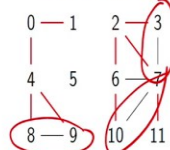
BFS tree

- A tree is a minimally connected graph
- Edges explored by BFS form a **tree**
 - Technically, one tree per component
 - Collection of trees is a **forest**

Acyclic Graph

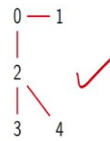


Graph with cycles

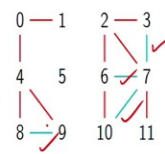


- A tree is a minimally connected graph
- Edges explored by BFS form a **tree**
 - Technically, one tree per component
 - Collection of trees is a **forest**
- Facts about trees
 - A tree on n vertices has $n - 1$ edges
 - A tree is acyclic
- Any non-tree edge creates a cycle
 - Detect cycles by searching for non-tree edges

Acyclic Graph



Graph with cycles



So, we start by looking at what is called as tree. So tree is a minimally connected graph. So here look at this example here, so this acyclic graph that we drew, so it is connected because we saw that it is one connected component. It is also acyclic and in particular now if we drop any edge like for example if we drop the 2 4 edge, then this graph will become disconnected. So if I want to minimally connected, I need to draw at least these many edges in this case there are five vertices, 0 to 4 and I have drawn four edges.

So, it turns out that when we explore for instance the tree using BFS, then the edges that we use to visit new vertices, so I start at a vertex and I visit a neighbor if the neighbor is not visited, so then I will count that edge as being visited by BFS. So the edges that BFS visits actually form a tree. So if you look at this acyclic graph on the top, it actually visits all the edges because the graph itself is a tree and there are not fewer edges which form a tree in that case.

Now, if you have a graph like the bottom one which has multiple components then technically you have to start BFS each time from a new component. So it is not one tree because the tree as a whole would connect the entire graph, but the entire graph is not connected. So each component gets connected by a tree. So here for instance BFS does not visit this edge, so this edge is outside the BFS tree.

Similarly, here, this edge and this edge are outside the BFS tree. Now from English since we say that a forest is a collection of trees, in this kind of a thing also we talk about multiple trees are forming a forest. So technically what BFS does is it discovers a collection of trees or a forest

inside the graph. So some useful facts about trees, so the first fact is that if you have n vertices then a tree on n vertices will have n minus 1 edges.

Now, it does not specify which edges, so you can connect them in many different ways, but whichever way you connect them you will have to use n minus 1 vertices. So just as an example supposing we have four vertices, so one way to connect this in a tree is to connect it like this in a single path. So this is one way to do it. And of course we can choose different paths, so we could connect it this way or we could connect it this way and so on.

But a different way to connect it which is not a path is to connect it all to one vertex like this, so this is also a tree. But notice that all these ways of connecting these four vertices, so that each one is connected to the other but in a minimal way, all of them have three edges. So whenever you have n vertices, you will have exactly n minus 1 edges and of course the tree is acyclic. Because if you just think of it as a minimally connected graph, then if it has a cycle for instance if I had another vertex, another edge from here to here, then I have two ways of going from all the vertices on the cycle.

Two ways of going from 2 to 1, two ways of going from 0 to 4 and so on. So if I delete one, for example now if I delete this vertex or this edge, then I still have a connected graph. So if I had a cycle, then it cannot be minimal because some edge along the cycle can be removed and I will still be able to reach everything by going around the cycle the other way. So we are now going to prove this fact, but it is useful to know that these are all equivalent ways of thinking of a tree, so tree is a minimally connected undirected graph, a tree is necessarily acyclic.

So, an acyclic connected undirected graph is a tree. And on the other hand if it is connected and it has n minus 1 edges, then it is a tree and if it is a tree, it has n minus 1 edges. So just remember all these, because these are all different ways of thinking about a tree. So now coming back to our question, our question was how do we detect whether the graph we have is acyclic or not. So what we saw is that we have acyclic as like the first one, then the BFS tree that we get covers all the edges.


So, if we do not cover all the edges on the graph through our tree, if there are non-tree edges, then those non-tree edges if we add them must form a cycle and that is exactly what we get, so we can detect a cycle by searching for non-tree edges. So as we are going along just like we

mark vertices as visited, we can also if we want mark edges as visited or we can keep track of them by looking back afterward we have done it to see which edges we use, say the parent thing also gives us the visited.

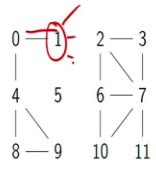
So, if I say parent of i is j , that means I went from j to i to visit i , so therefore ji was a tree edge. So whichever way we can recover the tree edges at the end of our BFS and then any non-tree edge if it exists must form a cycle. So here we have these non-tree edges, so $6\ 7$; $3\ 7$; $7\ 10$ and $8\ 9$; which are not part of the BFS tree that we constructed. So since there is at least one such edge there must be a cycle in particular in this case both these components have a cycle.

(Refer Slide Time: 15:53)


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*)



pre




Madhavan Mukund

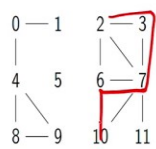
Applications of BFS and DFS

Mathematics for Data Science

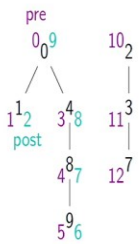
DFS tree



- 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*)



pre



Madhavan Mukund

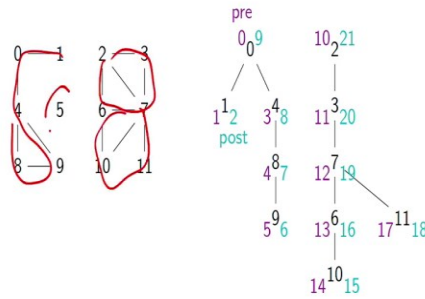
Applications of BFS and DFS

Mathematics for Data Science

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*)



Madhavan Mukund

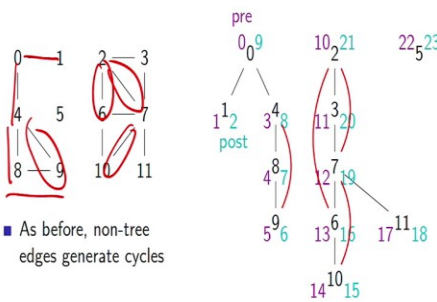
Applications of BFS and DFS

Mathematics for Data Science | V

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



Madhavan Mukund

Applications of BFS and DFS

Mathematics for Data Science | V

So, we could do this using DFS also in the same way, but we mentioned last time that DFS also comes with a different type of strategy to keep track of it called DFS numbering. So let us build a DFS tree for this same graph and let us describe formally how this numbering works through an example. So we initially maintained a separate counter for numbering when we enter and exit a vertex and we will start with 0.

So, we increment this counter every time we enter a vertex, every time we start exploring it and every time we leave the vertex, every time we finish exploring it. So this will become clearer as we go along, so in this process, this counter value is assigned to the vertex as an incoming

number and then when we leave the updated counter value is assigned as an outgoing number. So we have a pre number and a post number for every vertex.

So, let us try and do a DFS for this graph here, starting at vertex 0. So we initialize our counter to 0, so we are starting, so the black indicates the vertex number and this purple number is the pre number, a number of our DFS counter when we entered vertex 0 was 0. Now we explore for instance 0 to 1. We have to explore the unvisited vertices which are neighboring 0, so we can do 1.

So, now we increment our counter and say we entered vertex 1 with counter 1. So now when we come to vertex 1, we have no further vertices from 1 which are unexplored, the only neighbor of 1 which I can go to is 0 but 0 is already been visited, so now I am going to leave 1. But before I leave 1, I will increment the counter, so I will leave 1, I will assign it the post number of 2, and now in my stack I am coming back to 0. So I am back at 0.

So, 0 is not finished because 0 has another neighbor which is unexplored which is 4. So I will increment the counter and enter vertex 4 with the pre number 3. So notice that this number is increasing 0, then 0 plus 1 is 1, then 1 plus 1 is 2, then I went back but did I, I did not leave 0, I am still processing 0. So I will assign a post number to 0 only when I have finished all the neighbors of 0, in this case I am not finished, I have gone down another path.

So, 0 1 2 and now I assign the pre number 3 to this vertex. So now I am at 4, so 4 has two unexplored neighbors other than 0, because 0 is already explored. So I go to the smaller one say 8, so again I increment my number from 3 to 4 and I enter 8. So I enter 8 with the pre number 4 and now from 8 I can go to 9, because 9 is not yet been visited. So from 8 I enter 9 with pre number 5, so each time I am just incrementing this one counter, the pre and post is the same counter is being incremented whether I go in or go out.

So, at 9 I get stuck, because I have only two neighbors 4 and 8, both of which have been visited. So 9 now terminates, saying I have finished processing 9, so I increment to 6 and I get out of 9 and come back to 8. At 8 again I have nothing more to do, so I increment to 7, get out of 8 and come back to 4. At 4 I have done 8, but I cannot do 9, because 9 was visited through 8, so 4 is also done, so I will assign my post number to 8 and get out of 4.

And now I come back to 0. Now at 0 I had two neighbors 1 and 4 both are done and I am now finished with 0, so I will exit 0, so this is now my first component that I have discovered, starting from 0 and coming back to 0 and I entered and exited each vertex by updating that counter. Now I go to an unmarked vertex 2 and I continue the same numbering, so I enter 2 with vertex, with pre number 10, from 2 I go to 3 with pre number 11, from 3 I go to 7 with, so I am following this path.

So I have the smallest neighbor, then the smallest neighbor, then the smallest neighbor and then I will get stuck. So 2 will go to 3, 3 will go to 7, 7 will go to 6, 6 will go to 10 and then at 10 I am stuck, because I cannot go back to any vertex which is not visited, so I will exit 10 with counter 15. Then I will come back to 6, again 6 has neighbors 2 7 and 10 all of which have been seen, so I will exit 6.

I will come back to 7, now 7 has another neighbor 11 which I have not yet explored. So when I come back to 7, I am not done, instead I start exploring vertex 11 with counter 17. Now 11 obviously has nowhere to go after that, so I exit 11 with 18. Now I am done with 7, so 7 becomes exited with 19, 3 is exited to 20 and finally I come back to 2 and the other vertices which are neighboring 2 namely 7 and 6, have already been explored through 3, so there is nothing more to be done at 2, so I exit vertex 2 with 21.

So, at this point I have visited all these vertices and I have visited all these vertices. So this vertex 5 remains, so I have to start a third DFS as we did before. So I started with a new counter value 22, because I finished the last one with 21 and then I immediately exit because 5 has nothing to do. So this is, at this point we have not use in these numbers, we will see soon why we are going to use these numbers, but at this point it is just to show that when we are doing BFS we construct in a tree, when we are doing DFS also we construct a collection of trees is one for each component.

And it is now useful to actually describe the order in which this tree was drawn. How did we add edges to the tree and when did we back track up the tree? That is what we are keeping track of for this pre and post number. So ignoring the numbering, now there are some edges which did not come into the tree. So these are these red edges here, so we have an edge for example from 4 to 9, which did not come into the tree because our tree went 0 to 1, 0 to 4 and then 4 to 8 and then 8 to 9.

So, since I covered 9 through 8, I never got to explore 9 from 4, so 4 to 9 is a non-tree edge. Similarly, 2 to 7 and 2 to 6 are non-tree edges, because I did not explore them and so is 7 to 10. Because I went from 7 to 6 to 10, so I never explored the edge 7 to 10. So these 4 edges are non-tree edges, and each of them as you can see creates a cycle, so exactly like in the BFS, in an undirected graph, in a DFS also all the non-tree edges create a cycle.

