Data Structures and Algorithms

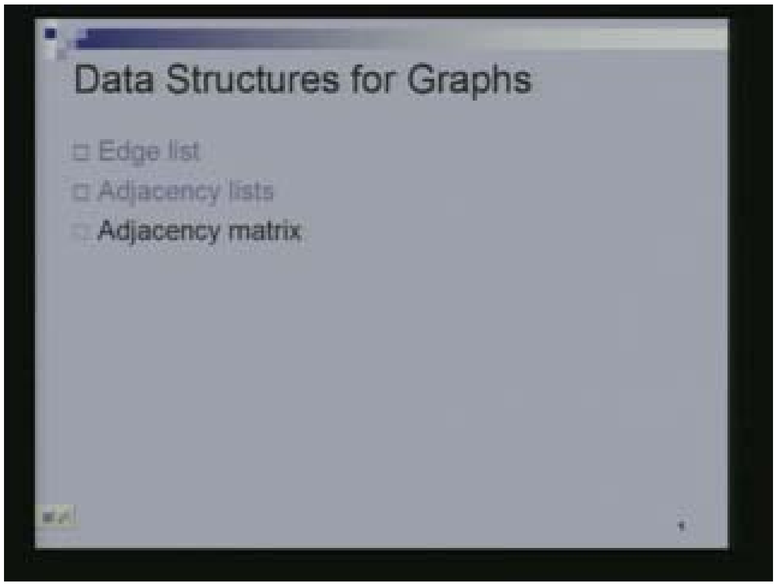
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 25

Data Structures for Graphs

Today's class we are going to be talking about data structures for graphs. If you recall in the last class we discussed various things about graphs. Various terms actually, what undirected graphs are, what directed graphs are, what is a path in a graph, what is a cycle, what are connected components so on and on. We are going to start using the terminology now. I am going to be discussing three different data structures for representing graphs.

(Refer Slide Time: 01:36)

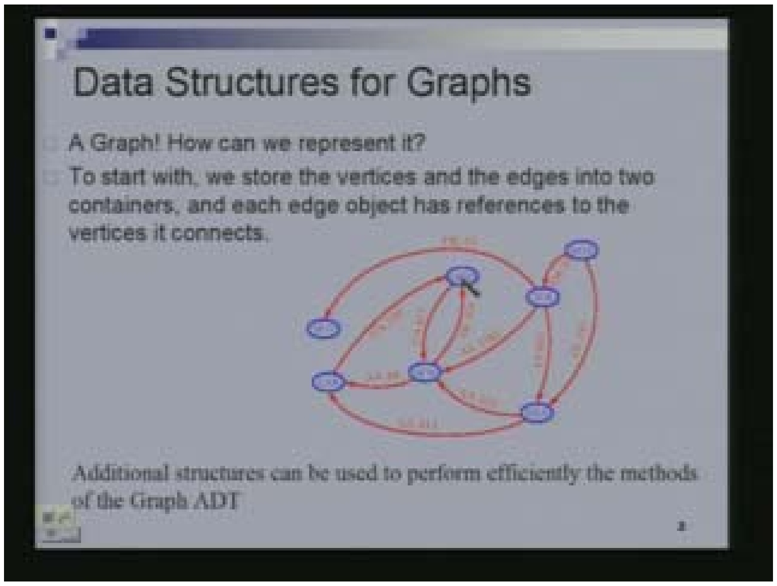


Data Structures for Graphs

- Edge list
- Adjacency lists
- Adjacency matrix

One would be the Edge list data structure, second adjacency list data structure and third would be adjacency matrix data structure. We will see what these are and how they can be augmented, how they can be combined to give better performances, faster running times.

(Refer Slide Time: 02:43)



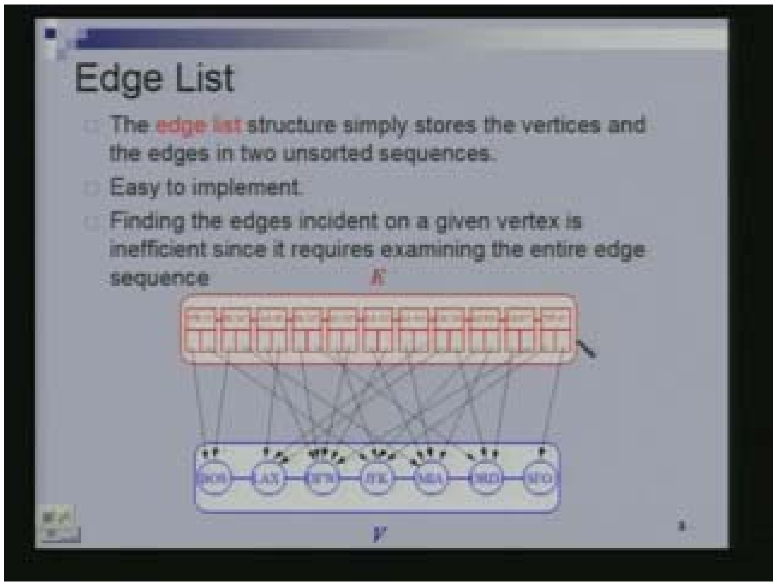
Data Structures for Graphs

- A Graph! How can we represent it?
- To start with, we store the vertices and the edges into two containers, and each edge object has references to the vertices it connects.

Additional structures can be used to perform efficiently the methods of the Graph ADT

The simplest data structure is what we call an Edge list data structure. Suppose this is my graph, it is a directed graph. This is just a graph of a various flights. These are airports the blue vertices and the red arcs are flight numbers from a airport to some other airport and suppose we want to represent this. One way is to have two lists, one of the vertices and one of the edges. We call each edge is a pair of vertices. In this case it will be an ordered pair of vertices so we could have two such lists. Let’s see what that corresponds to. That is we called the edge list data structure. The edge list data structure simply stores the vertices and the edges in two unsorted sequences. It’s very easy to implement and this is what it looks like.

(Refer Slide Time: 02:59)



Edge List

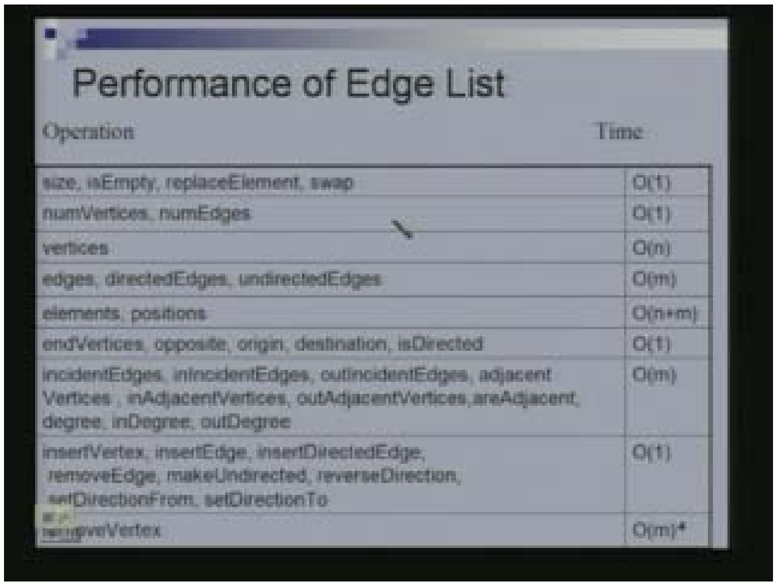
- The edge list structure simply stores the vertices and the edges in two unsorted sequences.
- Easy to implement.
- Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence

E
BOS
LAX
DFW
JFK
MIA
ORD
SFO
V

These are lists of vertices that you had and these were the various edges. Each edge recall corresponds to a certain flight between two airports. This is a flight let’s say NW 35. It goes from airport Boston to JFK and so this particular node has references pointers to these corresponding vertices here. For each edge I will keep two pointers, two references to the vertices between which that edge goes. This is called the edge list data structures. It is very easy to implement so there are many operations which can be done very quickly. For instance suppose there was one operation which was given an edge, find its two end points. So that can be done very quickly. You are given the certain edge and you want to find the two end points of that edge that can be done. Or there was an operation called opposite, given an edge and a vertex, you wanted to find out what was the other end point and so on and on. But there is one operation which is very inefficient and that is finding the adjacent vertices of a given vertex.

Suppose I give you a certain vertex. I give you vertex DFW and I say which are the vertices which are adjacent to this. How will I do thus? I will have to go through the list of edges and I have to find out all such edges. Suppose I wanted to find out vertices which are adjacent to DFW, I will have to look at this edge. This edge is not an end point of this, so I go to the next one. This is not an end point of this, I go to the next one. This is an end point of this so I will look at what the other end point is, that is LAX. So LAX becomes adjacent to DFW and so on and on. This is what I will have to do to find out the adjacent vertices of a given vertex.

(Refer Slide Time: 04:54)



Performance of Edge List

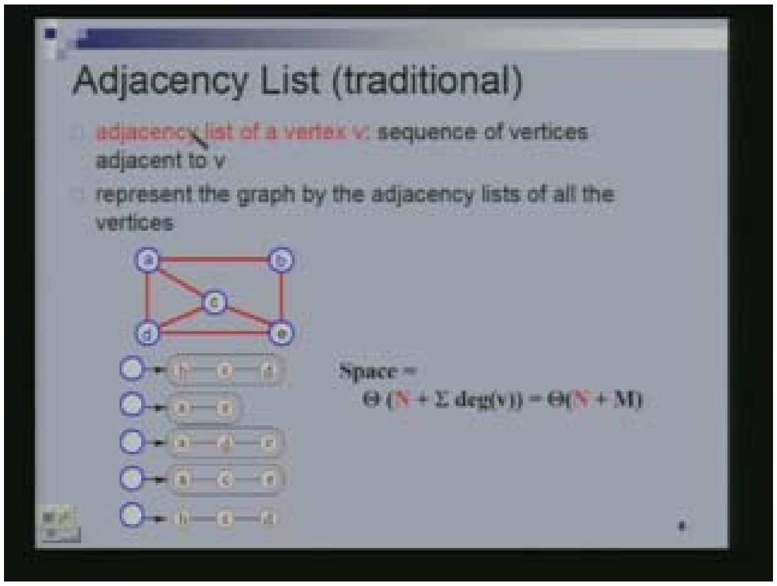
Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected	O(1)
incidentEdges, inIncidentEdges, outIncidentEdges, adjacent Vertices , inAdjacentVertices, outAdjacentVertices,areAdjacent, degree, inDegree, outDegree	O(m)
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	O(1)
removeVertex	O(m)*

If I look at the various operations, I can see what times they take. So let’s see. Size, isEmpty, replaceElement, swap these are all container operations. These are when I am giving you the position, so size for instance is constant time, I can keep track of the number of edges and vertices in the tool list. isEmpty is again constant time, if the size is zero than its empty. replaceElement, if I give you a particular position corresponding to an edge and I say put some other edge at that location, that just takes constant time.

Similarly swap all of these take constant time. Number of vertices, number of edges takes constant time. What does the method vertices do? It enumerates, it is an iterator over all the vertices. Since I would have to run through all the vertices, that would take time proportional to the number of vertices. Similarly these are iterator's over the edges, so they will take time proportional to the number of edges. Let's look at some more interesting thing. Suppose I say insertVertex, I want to insert a vertex. Then how much time should that take? It should take constant or order n. It should take constant time because these are unsorted lists.

Similarly insertEdge, insertDirectedEdge all of these can be done in constant time. Let's look at remove vertex. This last operation you will not able to see it very clearly perhaps because it is getting overlapped here but it is removevertex. So suppose I wanted to remove a vertex. How much time would it take? If I remove a vertex, I also remove the edges which are incident to that vertex clearly. Because otherwise where would the end points of the edges be referring to. I have to essentially get to that vertex and I also have to traverse to the list of edges. The list of edges is, number of edges is order m. I have to traverse through the entire list to find out whether which are the vertices which are adjacent, which are the edges which are incident to this vertex and also remove those edges. Which is why this operation is going to take order m time. [Student:] here when I say removeVertex, I am assuming that you are given the particular vertex you want to remove. Let's say you are given the position in the list. [Hindi]. In this manner you can look at this slide more carefully and understand the times.

(Refer Slide Time: 07:10)



Adjacency List (traditional)

- adjacency list of a vertex v: sequence of vertices adjacent to v
- represent the graph by the adjacency lists of all the vertices

Space = $\Theta(N + \Sigma \deg(v)) = \Theta(N + M)$

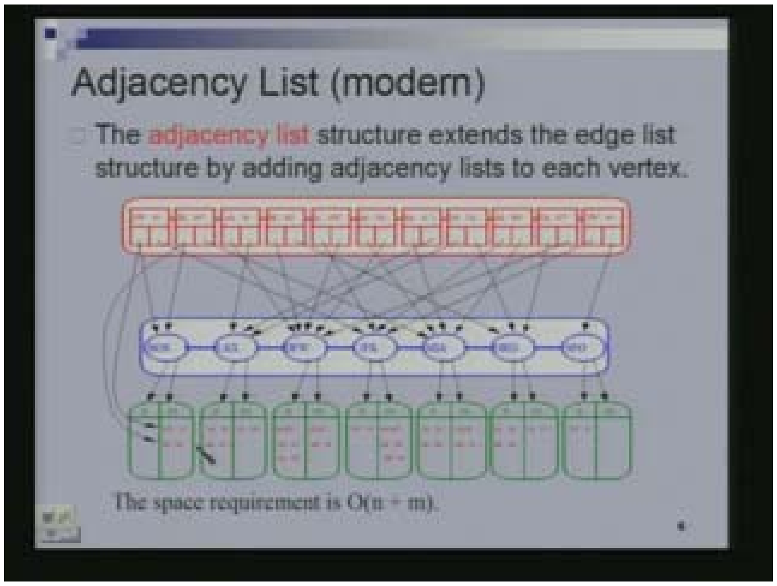
This is not the only way of representing a graph. There could be other ways. We are now going to look at what's called the adjacency list data structure. This is your graph, here I am taking an example of an undirected graph but all these data structures, all the three data structures that I am going talking of today can be used to represent both directed and undirected graphs.

If so in this undirected graph how do I represent it? I have an array of vertices, an array corresponding to vertices let’s say. This location corresponds to vertex a, this corresponds to the vertex b, this location corresponds to c d and e that’s how it should be. Now what are the vertices which are adjacent to a? They are b c and d, so I will have a link list starting from this location which will have elements b c and d in it. This corresponded to location vertex b. So which are the vertexes adjacent to b? a and e so this links contain only a and e in it. This was corresponding to d, vertices adjacent to d are a c e so that’s why we have a c e in this. These lists are also unsorted lists. So adjacency list of a vertex, so what we are keeping track of is the adjacency list of each vertex.

How much space does this data structure require? I will have an array of size n and what will be the length of the link list at each location. The degree of that vertex. The total space required in this part is sum of the degrees which we argued is two times m. The total space required is N plus M, so it should be of the order of theta of N plus M. [Student: so having implementing this as a link list [Hindi] we could keep them in an array, that’s the next data structure. We will see what are the pros and cons for that. [Student: array implementation of the linked list] pardon [student: array implementation of the linked list] No, what he is saying is, we will come to what he is saying in the next slide, when I show you the adjacency matrix implementation.

What’s the advantage of this? How is this better than the previous data structure? [Student: moving a vertex] Adjacent vertices, to find out what are the adjacent vertices of a given vertex that can be done much more quickly here. How much time does it take now? [Student: order] order degree [Student: order degree] If I want to list out all the adjacent vertices. If I give you two vertices and I ask you are these two vertices adjacent? How much time would you take? [Student: order degree] order degree still. Degree of one of the vertices let’s say, the smaller one. [Student: the smaller one] [Student: the larger one] this is an array. This one is an array. This is an array of just references of pointers, nothing else. There is nothing else stored in this array. You can store more information if you want, any information associated with the vertex you can store at this storage location in the array. [Student:]It’s just an array, you can. [Student:] it’s an array typically indexed by… Suppose so you typically number your vertices 1 2 3 4 and so on and then that would correspond to this location.

(Refer Slide Time: 10:59)



Adjacency List (modern)

- The adjacency list structure extends the edge list structure by adding adjacency lists to each vertex.

The space requirement is O(n + m).

We can combine this with the edge list data structure and we will get something more complicated like this. What is this? [Hindi] is just your edge list. Just I showed you and now with each of these vertices, I have the adjacency list associated with each of these vertices. I have both the in adjacency list and the out adjacency list. From each of these elements, there are two pointers. One is pointing to a list of incoming edges and the other is pointing to a list of outgoing edges. We have combined this and that. The adjacency list and the edge list data structure, somehow we have combined them. Let's see in what regard is this better than this? [Student:] here for instance the operation, suppose I said, given a particular edge. Here we actually have no mechanism of storing edges really. We are not really storing edge information. We are storing information only with regard to vertices, given a vertex what are the adjacent vertices. [Student: so edge information regarding the edges easily obtained what information regarding the edges do we want that we what].

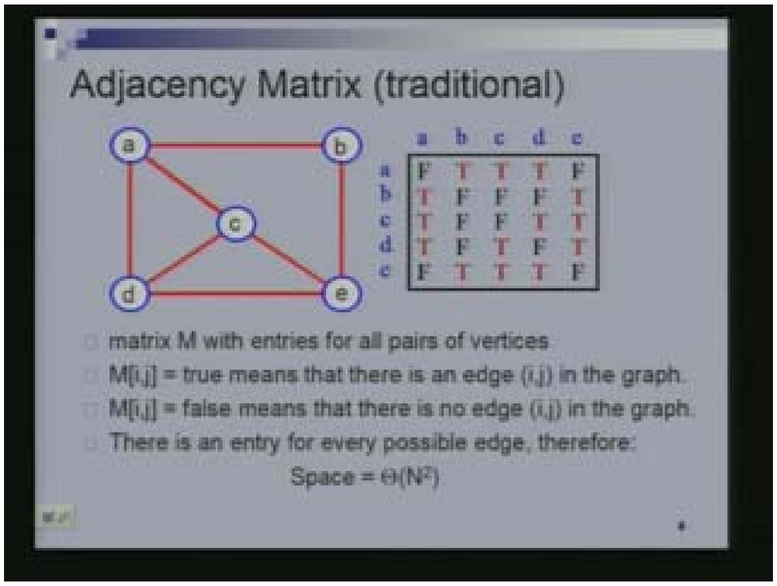
Suppose I had the same picture as before. I have this graph. I ask you flight UA 120 which airport from, what is the starting airport, what is the ending? Yes, so we have information associated with edges and that is somehow not getting represented in this. [Student: we can store name of] pardon [student: we can store] you can store there is no harm. You can store, so with this whichever was that airport you could do that, but now if you have to retrieve that information. Suppose you have to answer that question, given a particular flight number, what are the starting and the ending airports. You will have to go through this entire data structure to be able to figure that out. While here you could get that information very quickly. [Student: there is also an]. That's not connected to the top. [Student: as good as having a double arrow for an edge list so this thing is as good as having a double arrow for the edge list] double arrow, what do you mean by double arrow? [Student: doubly length like in the above portion of this slide an arrow points from an edge to its vertex] vertices and from here it's pointing from the vertex to the edges exactly. Let’s look at what are the time requirements for this particular data structure let’s see.

Suppose I want to find out the incident edges. Edges incident to a certain vertex so I can get to that vertex, so given a particular vertex I can find both the in edges and the out edges in time proportional to the degree, the in degree and the out degree respectively. So that’s the incident edges. Given two vertices, are those two vertices adjacent? How much time do I need for this? So given one vertex I just need to run through the… it’s given a particular vertex DFW and let’s say MIA. Are this two adjacent? Is there a flight from DFW to MIA, what will I do? I will go in the out list of this and see. No, I will have to do more, I will come to this. I will come to the out list and then I have to go from here to this list here. This is just numbers or whatever, this would be referenced to this information.
It depends upon how you organize it. This could be organized as out list of edges or it could be organized as out adjacent vertices. If it were organized as out adjacent vertices it would have been easy. But if like here, I am organizing it as edge list, so you will have to now go to the corresponding edge and see what the other end point of that edge is. All of that is constant time.

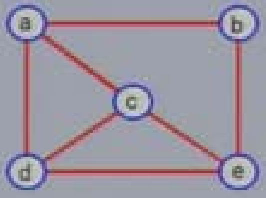
There are many ways of organizing a graph. I am just giving you a very high level idea and then for your particular application, depending upon what operation you are doing more often, you will have to choose the appropriate organization. [Student: organize the data it will be in minimum of degree of u and common degree of worst case will be maximum] Worst case would be maximum but suppose I also kept degree information associated with each vertices. That’s not very hard to do. Just one integer variable which will keep track of them, then I can make this. It depends upon where you want to optimize. If you are doing this kind of operation very often then it makes sense keep degree and try and reduce running time. If you are not doing this operation then there is no reason why you should keep track of the degrees of the vertices.

Third representation is what’s called the adjacency matrix representation and this is very simple representation. Here you just have an n cross n matrix and there would be basically just binary entries bits 1 0, there is a one which is a true, if there is an edge between those two vertices. There is a one here because there is an edge between a and b. What can you say about this matrix? What property does it have? [Student: symmetric] it’s symmetric. If it is an undirected graph it would be symmetric. If its directed graph it need not be symmetric. If in a directed graph you can have it that this would be one, if there is an edge from b to a or you can have it the other way round depending upon what, you can keep any way you like. M [i, j] is true that means there is an edge i, j in the graph. M [i, j] false means there is no edge in the graph and the space requirement is N square. It’s again a very simple implementation, it’s also quite efficient in a certain sense or let’s see.

(Refer Slide Time: 17:37)



Adjacency Matrix (traditional)



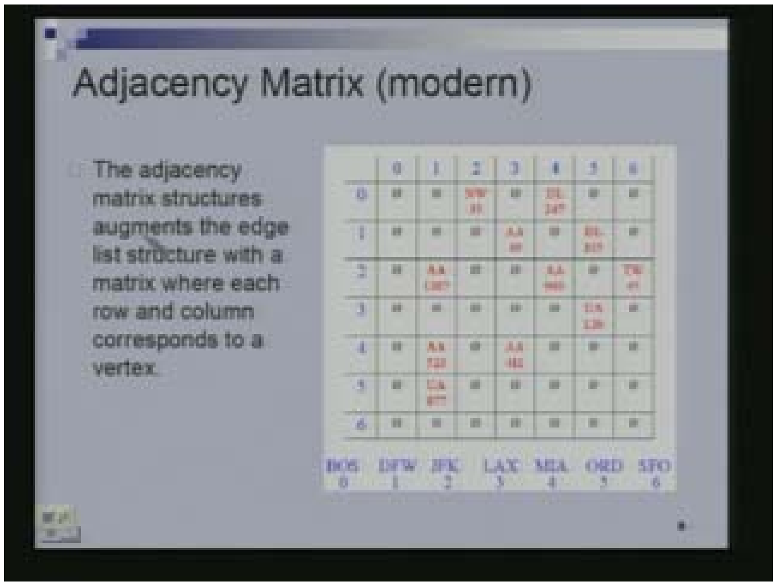
$$
\begin{array}{c|ccccc}
 & a & b & c & d & e \\
\hline
a & F & T & T & T & F \\
b & T & F & F & F & T \\
c & T & F & F & T & T \\
d & T & F & T & F & T \\
e & F & T & T & T & F
\end{array}
$$

- matrix M with entries for all pairs of vertices
- M[i,j] = true means that there is an edge (i,j) in the graph.
- M[i,j] = false means that there is no edge (i,j) in the graph.
- There is an entry for every possible edge, therefore:

  Space = $\Theta(N^2)$

[Student: adding a vertex means] Adding a vertex would mean creating a new row and a new column. It is order n time, order n so that will be order n time. I could have for instance… Again there is a possibility of, instead of having 1's and 0's, you could keep track of the edge information here. You could keep… The adjacency matrix structure augments the edge list structure with a matrix, so you could also have the edge list together with the adjacency matrix both of them together. In the edge list recall that for each edge you would have information about what are the two end points.

(Refer Slide Time: 18:21)



Adjacency Matrix (modern)

- The adjacency matrix structures augments the edge list structure with a matrix where each row and column corresponds to a vertex.

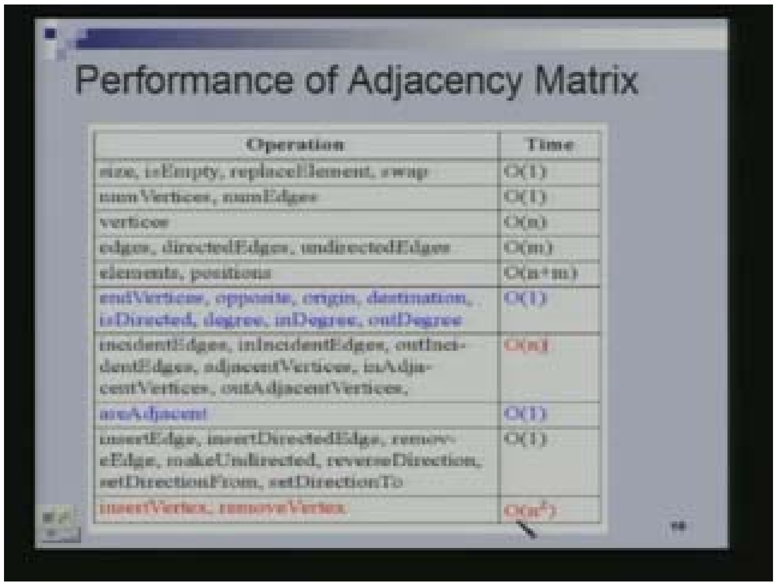
	0	1	2	3	4	5	6
0	ø	ø	NW 35	ø	DL 247	ø	ø
1	ø	ø	ø	AA 49	ø	DL 335	ø
2	ø	AA 1387	ø	ø	AA 903	ø	TW 45
3	ø	ø	ø	ø	ø	UA 120	ø
4	ø	AA 523	ø	AA 411	ø	ø	ø
5	ø	UA 877	ø	ø	ø	ø	ø
6	ø	ø	ø	ø	ø	ø	ø

BOS	DFW	JFK	LAX	MIA	ORD	SFO
0	1	2	3	4	5	6

There could be referring to the corresponding locations here, instead of pointers they could just be integers not they are telling which row they corresponds to and here instead of 1’s and 0’s, for the ones you could also have the corresponding edge. Augmenting an array, again so this is an operation which is not done very often. Quite often the graphs that you work with are static graphs. That is you don’t add new vertices in the graph, you don’t add, remove new vertices from the graph. If you want to have a data structure which implements that then perhaps this is not the right data structure. [Student: array implementation] you could do that, of course all of those thing could be done. I am not saying it cannot be done here at all but then this would perhaps not be the best data structure to use.

If this was a frequent operation you were doing, adding vertices. [Student: even that will take order n time only because just when 2n minus 1 basically n square] Let’s look at what are the times required for the various operations here? Given two vertices to determine if they are adjacent or not, it is just a constant time operation. But now given a particular vertex to find out all the vertices which are adjacent to it, how much time does it take? [Student:] Row or column which is order n. It is not order degree now, it is order n this is the difference.

(Refer Slide Time: 20:21)

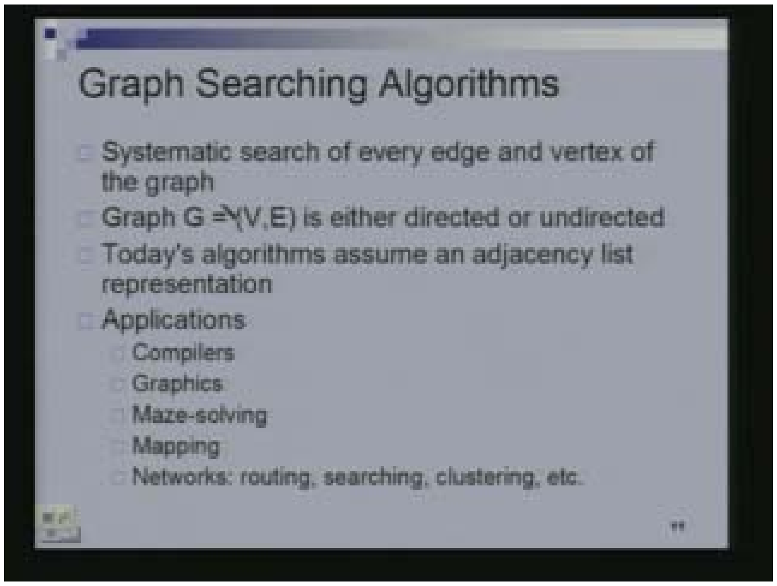


Performance of Adjacency Matrix

Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	O(1)
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices,	O(n)
areAdjacent	O(1)
insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	O(1)
insertVertex, removeVertex	O(n^2)

Why because now you will have to take that particular row or column and look at all the entries and see which are ones and which are zeros. That will give you order n. So incident edges, inIncidentEdges, outIncidentEdges all of them will take order n. insertVertex, remove a vertex I have put down order n squared here because I am assuming that you have to copy them into a new array. It’s not very easy to take a two dimensional array and extend it by one row and one column. You understand why? Because the problem is that two-dimensional arrays are stored as one dimensional arrays. You know after all in a particular row major or column major form, If you want to extend it now how does that happen? Because you have to then move all the information. That's why extending a two dimensional array is not an easy task. You essentially have to copy all the information into a new array. That's all I will have to say about data structures for representing graphs. There are three different things you have seen adjacency list, adjacency matrix and the simple edge list. You can depending upon what operations are critical, which are the ones that you are doing more often combine them in a suitable manner. If space is not such an issue then you can keep the adjacency matrix data structure because it's quite simple but it requires a lot of space. It requires n squared space. The standard implementation which is preferred very often is the adjacency list. If you can't think of anything then just use the adjacency list data structure to implement a graph. Now I am going to go onto graph searching algorithms. This is something that you have done in a certain sense which is why I am going to be taking it up right in this class.

(Refer Slide Time: 22:27)

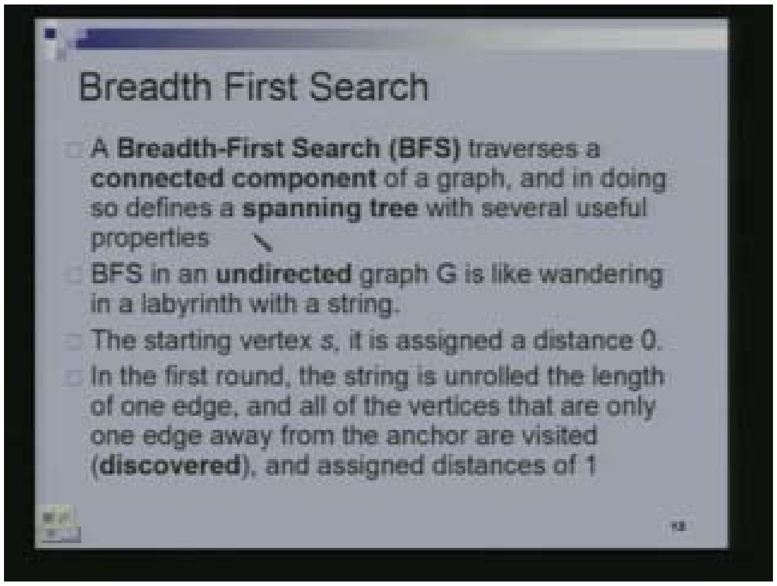


Graph Searching Algorithms

- Systematic search of every edge and vertex of the graph
- Graph G =(V,E) is either directed or undirected
- Today's algorithms assume an adjacency list representation
- Applications
  - Compilers
  - Graphics
  - Maze-solving
  - Mapping
  - Networks: routing, searching, clustering, etc.

What is the graph search algorithm? It is basically a mechanism of visiting all the vertices of the graph in some systematic manner. By systematic I mean, you know in some organized manner so that you don't miss out on any vertex. A graph could be either a directed or an undirected graphs and we are going to be assuming adjacency list algorithm implementation of the graph for the algorithm that we would be discussing. Graph searching algorithms are the most common algorithm that you typically perform on graphs and it appears on a whole lot of settings.

(Refer Slide Time: 23:03)



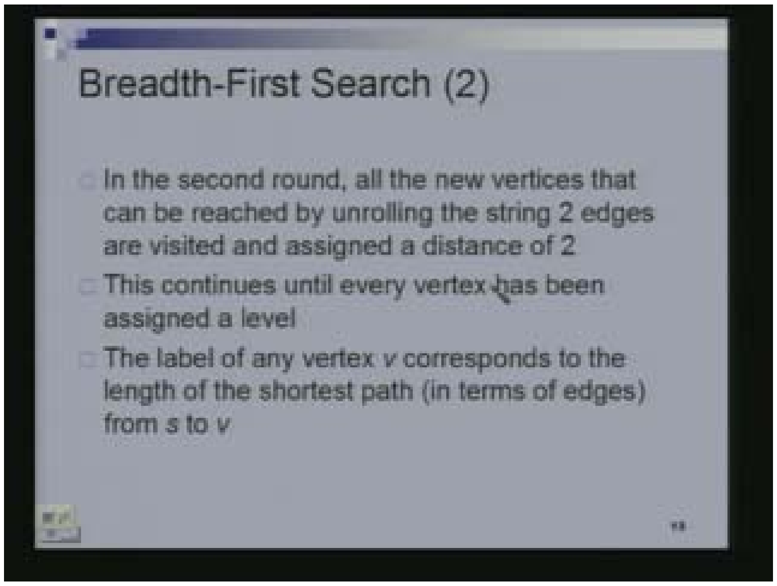
Breadth First Search

- A **Breadth-First Search (BFS)** traverses a **connected component** of a graph, and in doing so defines a **spanning tree** with several useful properties
- BFS in an **undirected** graph G is like wandering in a labyrinth with a string.
- The starting vertex *s*, it is assigned a distance 0.
- In the first round, the string is unrolled the length of one edge, and all of the vertices that are only one edge away from the anchor are visited (**discovered**), and assigned distances of 1

The algorithm that I am going to be discussing now is what is called the breadth first search algorithm or BFS for short. What does BFS do? It will visit all the vertices of a connected component in a graph and it will define for us what we will call a breadth first search tree which will be a spanning tree on this particular connected component. We are going to be discussing breadth first search on undirected graph only today. Breadth first search makes more sense in undirected graphs and the idea is roughly the following. You start from a vertex and this starting vertex let’s call it as s, it is assigned a initial distance of zero.

We are going to proceed in rounds. In the first round you are going to, so think of yourself as in a maze, in some kind of a maze. You have a string with you, you are going to use this to help you search the maze. You have tide one end of the string at one location in the maze. Now you unroll the string by let’s say just one unit and you see where all can you reach by unrolling this string by just one unit. Those in some sense will be, what we will call the vertices at a distance of one from the starting vertex. After you visited all such vertices then you will unroll the string by one more unit and see which all vertices, new vertices you can visit as a consequence of that and so on. You will understand all of this when we start discussing the algorithm in more detail. We will do this, so we will unroll the string by one more unit find out all that we can visit now.

(Refer Slide Time: 24:59)



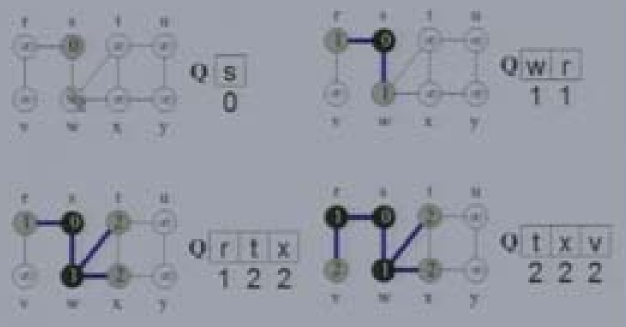
Breadth-First Search (2)

- In the second round, all the new vertices that can be reached by unrolling the string 2 edges are visited and assigned a distance of 2
- This continues until every vertex has been assigned a level
- The label of any vertex *v* corresponds to the length of the shortest path (in terms of edges) from *s* to *v*

Unroll the string by one more unit, find all the vertices that can be visited now and so on and on. Each vertex we are going to give it a label which will be that when it was first visited what was the length of my string then? If it was visited in the first round then I am going to give it a label of one. If it was visited in the second round I am going to give it a label of two and so on and on. What this label will signify eventually would be the distance of the vertex from the root, from the starting vertex or the root as I call.

(Refer Slide Time: 25:34)

BFS Example



All of these will be clear with some example. Suppose this was my graph, very simple graph and s was my starting vertex. I give it a label of 0. I am going to have a Q, so this is the only data structure I need to implement in this algorithm a Q. Recall this is very similar to your minor question. You have a Q and on which you have s. At any point what you are going to do is look at the front element in the Q and look at all the neighbours of that front element. The neighbours of this front element are w and r, so you are going to put them into the Q now. I am going to remove an element from the queue, remove the front element from the queue, find its neighbours and put them into the queue, insert them into the queue. When I insert a vertex into the queue I color it gray, after I remove a vertex from the queue I color it black.

Initially this is the only vertex in the queue so it's grey. All the vertices which are in the queue will always have a color of grey. In some sense the grey vertices are vertices which have been discovered till now but I have not gone beyond that, grey signifies that. Black means that I have also gone beyond those vertices and white means undiscovered, I have not reached those vertices at all. This is the order in which the thing is. This is the first picture, the second, the third and the fourth.

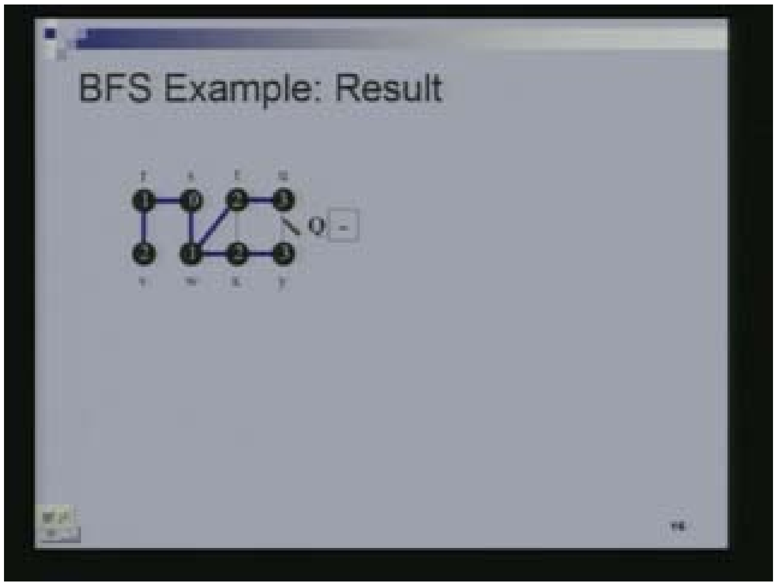
Let's understand this, I had s in the Q, I removed s, colored the vertex black, took its two neighbors and put them into the queue. I assigned them a label of one node then the label of s, so the label of s was 0 so I gave them both of them a label of a 1. Now let's see what the procedure here would be. I remove the front element of the queue which is w. I will color it black, I will look at its neighbors. How many neighbors does w have? 3 neighbors s, t and x. Amongst these s already is black so I don't touch it at all. t and x are white so I will put them into the queue and color them grey. From white I color grey, grey color gets colored to black.

When an element gets inserted in the queue, its gets colored grey. When it gets knocked off from the queue, it gets colored black, as simple as that. t and x get to put into the queue. What label do they get? They get a label of one more than the label of w. Why one more than the label of w? It was because of w, the t and x came into the queue. When I had knocked off w and then looked at its adjacent vertices, I have found t and x so they get label of one more than that. So they get a label of two.

This is the Q at this point. Now I look at vertex r which is the front of the queue. What will I do? First color it black, look at its adjacent vertices which are white and put them into the queue. Color it black look at its adjacent vertices which are white, this is the only vertex which is white, put that into the queue with the label equal to one more than the label of r. That's what this is and this get a grey color. Once again you see all the vertices which are grey are sitting in the queue. At any point this is the invariant you have. If a vertex is grey, it is in the queue. If a vertex has not yet been visited it is white, if a vertex has been visited and removed from the queue its black. Everyone understands this. Next vertex we are going to touch is t. We are going to remove t from here, going to look at its adjacent vertices. How many adjacent vertices it has? 3 but the only vertex which is white is u. It is only u which will get entered into the queue and nothing else and u will get colored grey. What will be its label? [Student: three] 3. u will get colored grey, its label is 3 and its get added to the queue.

Now I knock x out of the queue, it gets colored black. I look at its neighbours which are white. This is the only one which is white, this gets a label of a three, it gets colored grey and it’s added to the queue. So y is colored grey, gets a label three and is added to the queue. This is what the queue looks now like. Now I remove the front element that’s two, look at its right neighbours, it has no white neighbour nothing needs to be done. It gets colored black and we remove it from the queue, so the queue is now u and y only. So I remove u from the queue, I color it black, look at its white neighbours, it does not have any white neighbour so nothing to be done. Then I look finally at y, y is at the front of the queue. I look at its white neighbours, no white neighbour nothing needs to be done. This gets removed from the queue and the queue becomes empty. The procedure stops when the queue become empty, these are the labels on the vertices.

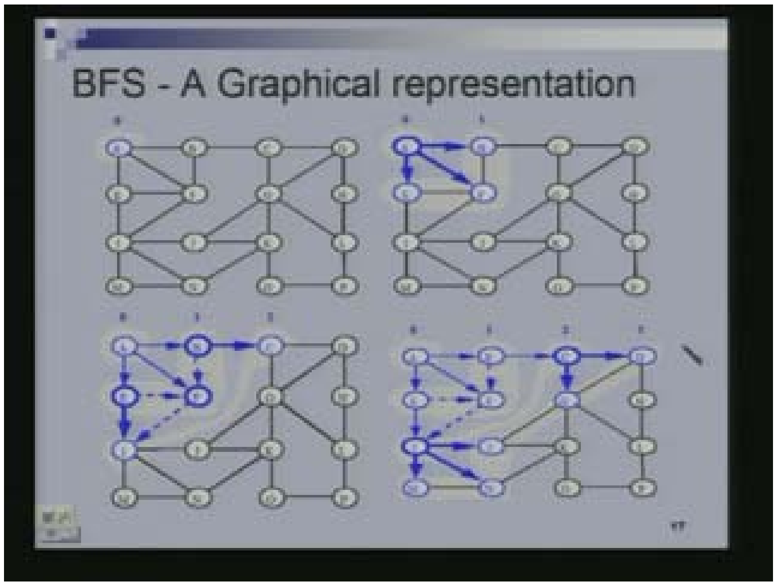
(Refer Slide Time: 31:29)



BFS Example: Result

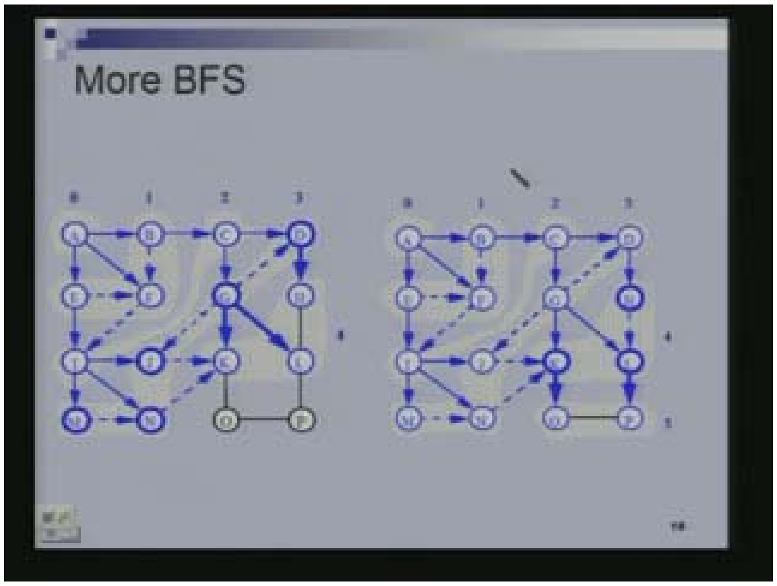
Now what do these labels signify? One signifies that it was discussed in the first round, two that it was discussed in the second and three that it was discovered in the third round. This one is also the length of the shortest path from s. [Hindi] if I look at this vertex u, there are many paths from s to u. I am interested in the path which has the least number of edges on it, the smallest number of edges on it and the path with the smallest number of edges is this path with three edges on it and so this is label three. We will see why this is getting done in this manner shortly.

(Refer Slide Time: 33:41)


BFS - A Graphical representation

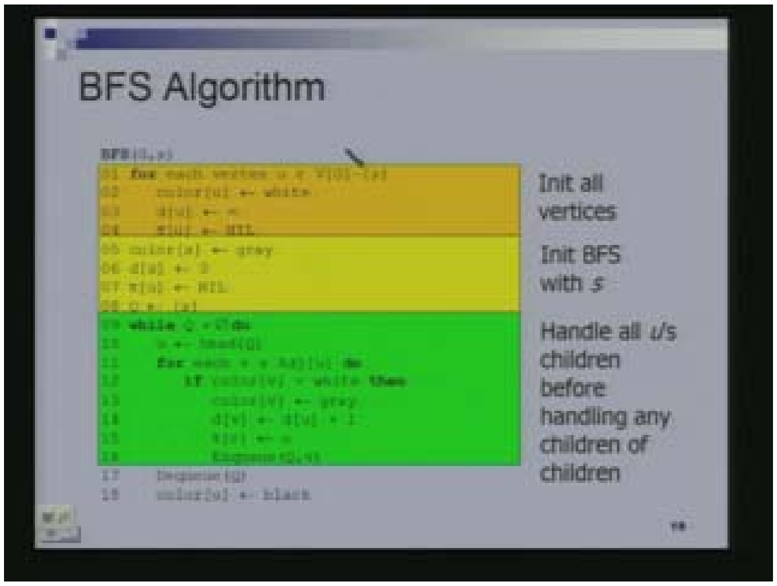
So one more way of thinking of this so that you understand this completely, I started from this. In the first round I am visiting the adjacent vertices of this. These are the vertices which are getting a label of one. These are also called level one vertices, these are the vertices which are getting a label of a one. In the next step, all though I am going one vertex at a time but now the vertices which are going to get a label of two will be vertices which are adjacent to these. Which are these vertices? These are I and c, so these are the two vertices which will get a label of a two now. The vertices which get a label of a three are the ones which are adjacent to the vertices which are at two and they are basically m, j, g and d. This is getting a label of a three.

(Refer Slide Time: 34:02)


More BFS

The vertices which are getting the label of a four would be the one which are adjacent to the vertices which are at a label three, which are these vertices. So these get a label four and these would finally get a label of a five. You can think of a breadth first search as dividing your vertices or partitioning your set of vertices into levels or sets. There is one vertex at level zero, some vertices at level one, some vertices at level two, some vertices at level three and so on. What will be the number of levels going to be? The number of levels would be the maximum distance of any vertex from s.

(Refer Slide Time: 34:56)



BFS Algorithm

```
BFS(G,s)
for each vertex u ∈ V[G]-{s}
    color[u] ← white
    d[u] ← ∞
    π[u] ← NIL
color[s] ← grey
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ ∅ do
    u ← head(Q)
    for each v ∈ Adj[u] do
        if color[v] = white then
            color[v] ← grey
            d[v] ← d[u] + 1
            π[v] ← u
            Enqueue(Q,v)
    Dequeue(Q)
    color[u] ← black
```

Init all vertices

Init BFS with *s*

Handle all *u*'s children before handling any children of children

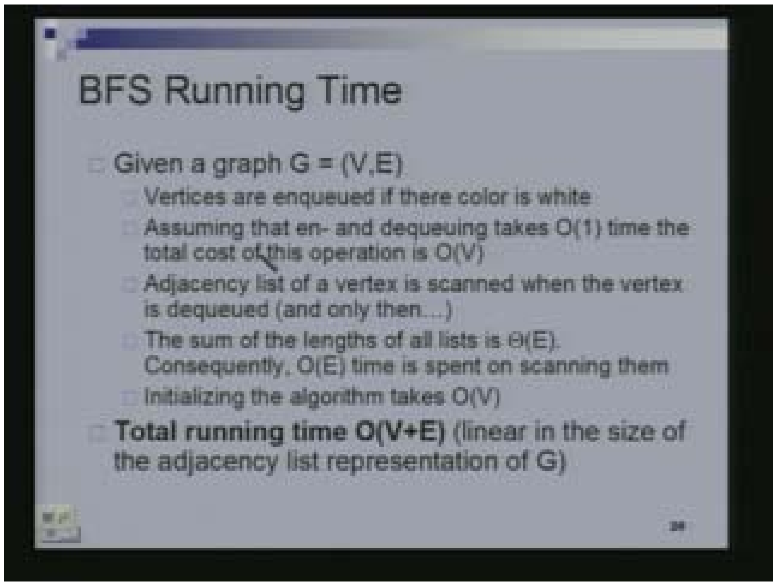
Now this is what the algorithm is, so let's run through the algorithm and then we will look at the other aspects of the algorithm. Initially every vertex is given a color of a white. So du is the label on that vertex, on a vertex u du is the label so it's initially infinite which means I have not put any labels on it and pi u, I will come to what pi u is. Pi u signifies the predecessor vertex. The vertex because of which you got its label. What I mean by this is so for instance let's look at this vertex c here. This got a label of 2 which was the vertex because of which it got the label 2 b, so pi of c would be b. What is the pi of k? You can tell me, this vertex got its label from either this or this. I do not know which, it could be any. We will just pick one of them arbitrarily. This is the initializing all vertices and then how do we begin?

We color the vertex s which is our starting vertex grey, we give it a label of a 0. Its pi of u is null because it doesn't get its label from anyone else but from itself and we add it to our Q. We insert it into our queue and this is the entire process. This green should have extended all the way here. What we are doing is while the Q is not empty, we keep repeating something. Let's say we remove the element from the head of the queue, so u is the element from the head. We are not removing it yet, so u is the element at the head of the queue. We are looking at all the adjacent vertices of u. For all v which are adjacent to u, if the color of v is white only then we do process it. If it is already grey or black, we don't do anything with it.

If the color of v is white then what do we do? We add it to the queue, we color it grey and we give it a suitable label. What is the label we give it? d of u plus 1. Whatever was the label of u, we add one to that and we give back to this. Since this vertex v is getting its label from u, pi of v becomes u. We add the vertex v, such vertex into the queue and once you have done it for all the vertices, we do this dequeue operation which is we remove u from the queue. This could have been done here also, it could have been done at the end, it could also have been done here that's okay and u is colored black to signify that it has been removed from the queue. We keep doing this till the queue has an element in it.

Does everyone understand what I am saying? [Student: so for initializing also we have to do some operations like breadth first search] What do you mean for initializing? [Student: making the color of every node to be white] Making the color of every node to be a white, what can we do? What is color? Color is something like an array. For every vertex so this is an array of size… [Student: values into it is less than the value we would assign to it] but we can also assign it, you know it is we are just creating an array, so let's say white is zero. So just assign, put zero to all the entries in the array. We just giving a color to… each of these color d and pi will have to be separate arrays indexed by the vertices, we are just assigning that.

(Refer Slide Time: 39:09)



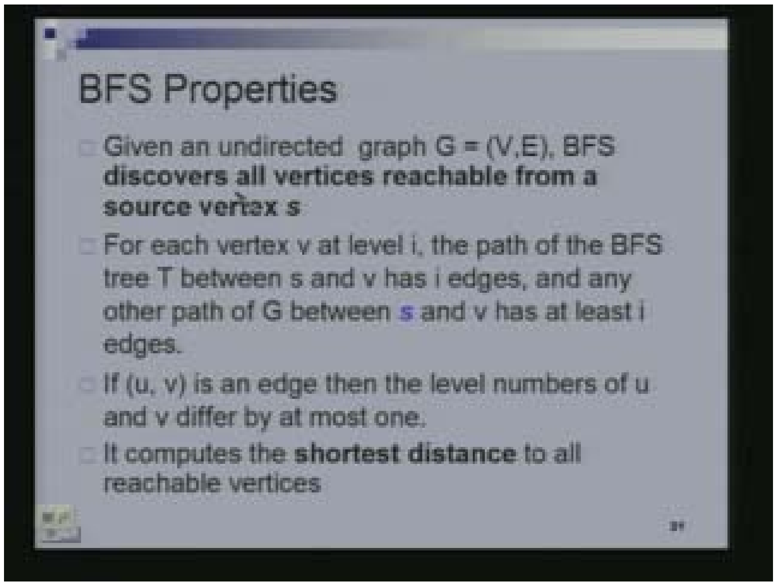
BFS Running Time

- Given a graph G = (V,E)
  - Vertices are enqueued if there color is white
  - Assuming that en- and dequeuing takes O(1) time the total cost of this operation is O(V)
  - Adjacency list of a vertex is scanned when the vertex is dequeued (and only then…)
  - The sum of the lengths of all lists is Θ(E). Consequently, O(E) time is spent on scanning them
  - Initializing the algorithm takes O(V)
- **Total running time O(V+E)** (linear in the size of the adjacency list representation of G)

How much time does the breadth first search procedure take? What are we doing? Basically all the time is being spent in this loop. Yes, because this is just, how much time do I spend here? One for each order n for each vertex, I am spending constant amount of time. How much time do I spent in this part? Constant time and here, all the time is getting spent in this loop. How many time is this loop executed? Order n times. How many times is this part of the loop executed? This is two loops, one within the other. For each vertex v adjacent to u, I am doing. So we are doing as much as the degree times. If I look at these statements they are being, what is the total time I am spending on these statements? Order degree for each vertex and summed over all the degrees of the vertices which is order m, twice the number of edges. Let's look at each statement and see what is the total time, what is the maximum time that could be spent on each statement. How many times the statement is executed, how many times the statement 10 executed? [Student: order n] Order n. How many times is statement 12 executed? [Student: order m] Order m because 12 is executed degree many times for each of the vertices. So order m, so 12 is executed order m times in the worst case. Similarly if 12 is executed order m times so 13, 14, 15 and 16 could also be executed no more than order m times.

Actually you can say something about 16, how many times is 16 executed? [Student: order] order n and not order m because you Enqueue a vertex only once. Once you Enqueue it, it becomes grey, once it get removed from the queue it becomes black. You don't ever touch it again. Once it becomes black you don't ever put it back into the queue. You only put a white vertex into the queue. In fact this statement 16 here is executed order n times and so this is also executed order n times and this is also executed order n times. It is only this if statement which is really executed order m times. Yes, you understand why if this is executed order n times, this is also executed? Because they are one after the other with no condition in between them. In any case the total time spent on the entire thing is order m plus n.

(Refer Slide Time: 42:18)



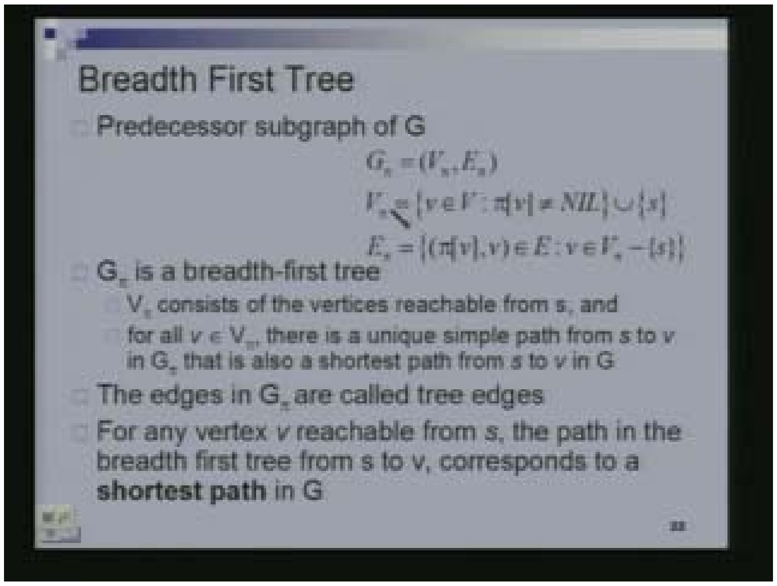
BFS Properties

- Given an undirected graph G = (V,E), BFS **discovers all vertices reachable from a source vertex *s***
- For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G between s and v has at least i edges.
- If (u, v) is an edge then the level numbers of u and v differ by at most one.
- It computes the **shortest distance** to all reachable vertices

Let's look at the couple of properties of BFS. BFS what it is doing is it starts from a certain vertex, a source vertex s and it is visiting all the vertices which can be reached from s. It will visit all such vertices which can be reached from s. What do I mean by that? All such vertices to which there is a path from s, all those vertices will get visited which means that all those vertices which are in the connected component of s, connected component containing s will get visited.

If the graph was in more than one connected component, if the graph had more than one connected component then if s is in a certain component, I will only visit those vertices. The vertices in the other connected component I will never be able to reach them at all. The first thing to keep in mind is that it will discover all the vertices which are reachable from a source vertex. If a vertex v is at level I then there is a path between s and v with I edges on it. I have not told you what a BFS tree is? So let's first understand what a BFS tree is. So my slide order is a bit wrong here. What is a BFS tree that we have generated as a consequence? So recall that for each vertex, I have kept track of one edge which gave that vertex its label.

(Refer Slide Time: 44:16)



Breadth First Tree

- Predecessor subgraph of G

$$G_\pi = (V_\pi, E_\pi)$$

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$$

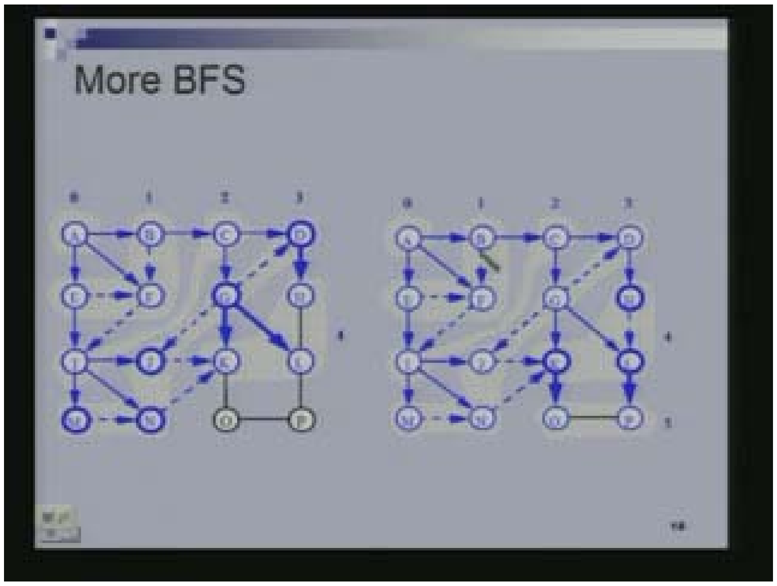
$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$$

- G_π is a breadth-first tree
  - V_π consists of the vertices reachable from s, and
  - for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G
- The edges in G_π are called tree edges
- For any vertex *v* reachable from *s*, the path in the breadth first tree from s to v, corresponds to a **shortest path** in G

22

So let me consider the following graph of the graph G. What is the sub graph? The set of vertices or all the vertices which are reachable from s, all those vertices which have a pi of v which is not null and s was given a pi of v which was null. So pi is also known as the predecessor. Each of the vertices which have a predecessor is the set of vertices V_{pi}. Every vertex is given a predecessor, every vertex which was visited given a predecessor. It is basically the set of all vertices which are visited. What are the edges in the sub graph? The edges in the sub graph are the edges from the predecessor vertex to this particular vertex for every vertex. Let me illustrate this. Let's look at this picture here. This picture if I ignore the dotted edges, if I just keep the dark edges with me, the solid edges they are my… this is the sub graph that I am talking about. Note that each of these vertices has a predecessor except for this starting vertex, this has no predecessor.

(Refer Slide Time: 45:41)


More BFS

What is predecessor of this? It was this, the predecessor of this was this and the predecessor of this was this. So these are 3 edges that I am including in my sub graph. The predecessor of this is this, the predecessor of this is this one, this was level 2. At level 3 when I had vertices, this has the predecessor as this, this had a predecessor let's say this. This has its predecessor this, N has I as its predecessor, M has I as its predecessor. (Refer Slide Time: 46:00) Is this clear to everyone? Because this vertex was discovered because of I. When I took I out from my queue and looked at its adjacent vertices which were not yet visited which are colored white then I found M N and j, so M N and j have as their predecessor vertex I and D and G have as their predecessor vertex c, k could have G or J as its predecessor.

Let's say we decided G as its predecessor and H as D as its predecessor and L has G as its predecessor. These are the predecessors and then finally P has L and O has K as its predecessor. [Student: G and J are at the same level] G and J are at the same level, yes. Why is not? [Student: not necessary] It is. They both, why are they at the same level? Because they both get the same label, same level number [student: G is not necessary] they would, it is necessary because they are both adjacent to vertices which have label two, G is adjacent to a vertex which is… [Student: is but I am saying for deciding for an element K which has two predecessors] yeah [Student: you will see which which predecessor has the shortest label] No, both will have the same label then. [Student: then you will not assign] both will have the same label.

If there was such a, you cannot have a vertex which has two predecessors at different levels. [Student: but this we only call ordered at which order can be different that we can] This is the predecessor information and these solid lines now form a spanning tree. Why do they form a spanning tree? How many solid lines are here? How many solid lines do I have? [Student: n - 1] n - 1, yes. Why n - 1 and not n?

There is no edge entering A because A did not have any predecessor. Every other vertex has a predecessor, for every other edge there is one solid line, exactly one. So exactly n minus 1 edges. Using these solid lines I can go from A to any other vertex. Yes, if there is a solid line entering here that means that I can come to this vertex from that vertex and there is a solid line entering here, so I can come to this vertex, from there is a predecessor of that and there is a solid line entering here, so I can come to this vertex from some other and so on. So eventually I hit the root. Basically starting from s, I can get to every other vertex. This is a tree, it is a connected graph. These solid lines form a connected graph with exactly n minus 1 edges. It has to be a spanning tree. Recall that we said that if a connected graph has no cycles in it then it has n minus 1 edges. If a connected graph has n minus 1 edges then it has no cycles in it and it is a tree. This is what we have. This is called the breadth first search tree, the BFS tree.

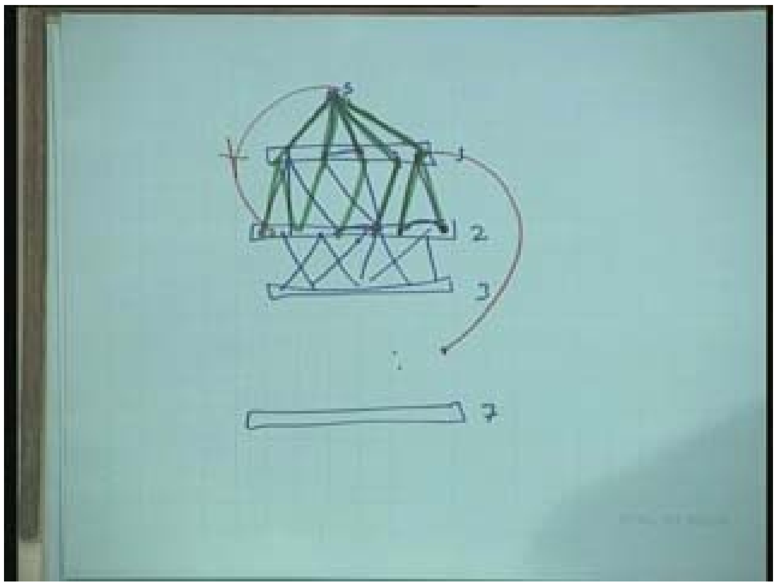
This spanning tree that we have which is the solid lines is the breadth first search tree. I think in both of the previous examples I had this. This is the breadth first search tree here, the blue lines they form the breadth first search tree. Once again for each vertex, I have just darkened the line. I have not drawn the arrows here but I have just darkened the line because of which was corresponded to the predecessor, because of which this vertex got its label. This vertex got its label 2 because of this vertex this got its label 2 because of this vertex and so I have darkened these lines and this forms your spanning tree so this is the breadth first search tree.

Let's quickly go on. This is the breadth first search tree. I will switch over to this screen to show you something. I did not. So what has really happened is that we started from this vertex s, these were my vertices at level 1, these were my vertices at level 2 and so on. Let's say the largest level was 7. Now this will answer some of your questions also. When I was at s, all these vertices which are here at level 1 are adjacent to vertices in s. That's why they are at level 1. The vertices in level 2 are all adjacent to vertices of level 1. That's why they got the level 2, number label 2. They got a label 2 because they were adjacent to some vertex, may be more than one but they were adjacent to some vertex in level 1.

Could these vertices have been adjacent to s? They would have been in level 1 because when I looked at all the adjacent vertices of s, I would have discovered this vertex and put it level 1 instead. So such an edge cannot appear. This is the nice thing about structure that you get. I am not showing the tree edges here, I am just showing all the edges of the graph now. All the edges of the graph just go between adjacent levels. They cannot skip a level. I cannot have an edge which skips a level, it cannot go like this. This cannot happen. Why? Because when I was this vertex, would then have been in this level instead. I made a small mistake. I said all the edges go between adjacent levels. They could also go within the same level, yes. Why could they? I could easily have this. This vertex was adjacent, these two vertices were adjacent to s but they were also adjacent to each other, no harm even later. This is what the graph looks like now and this is the important property of breadth first search that you have to keep in mind. Certain edges we would call them tree edges, so this is my BFS tree, this edge. So my BFS tree would look like this now.

Let's say these edges and then from each one of them, I have let's say something like this. I am basically covering all these edges. For each of these vertices is getting its level number because of certain vertex at the previous level and it's this edge having through did in my BFS tree. This is what my BFS tree [student: from the edges that through from elements of the each other if we have two elements of the same level in the] [Student: we we wont count those] no, if I have an edge between two vertices of the same level, such an edge, this edge was not part of BFS tree. Why? What are the edges which are in the BFS tree? [Student: solid] the predecessor. This vertex did not get its level number because of this vertex. It got its level number because of this vertex. It is this edge which would be part of the BFS tree and not this.

(Refer Slide Time: 54:24)



Let me [student: multiple edges leading to a node we take only one] we take only one. We have that, so just to show you all those things I just said to you, I had organized it like levels but it is the same thing happening here. These are the levels zeroth level, level 1, level 2, level 3, level 4, level 5. As you can see all edges are going either between adjacent levels or within the same level. This vertex could have got its level number from either this vertex or it's this vertex, so I picked one arbitrarily and this I included in my BFS tree. The BFS tree is not necessarily unique but the level number of each vertex will be unique. Why would it be unique? The level number of a vertex would be the length of the shortest path from s to that vertex. Why shortest path? We have not proved yet.

If there is a path from A to a certain vertex of length 6 then this certain vertex lets say whatever z, will get a label of at most 6. If the shortest path was of length 4 then this vertex cannot get a label of more than a 4 because on that path. If there is a path of length 4, what does that mean? There is s, there is a first vertex, the second vertex, the third vertex and then this vertex z.

Then the first vertex would get a label level which means that the first vertex is adjacent from s. It will get a level number of 1, the next vertex will get a level number of 2, the third vertex will get a level number of 3 and this vertex will get a level number of 4. That is why each vertex gets a level number equal to the length of its shortest path from s and that is unique. We said in choosing the predecessor edges, you could choose any one but the level numbers would be unique for each vertex because it corresponds to the length of the shortest path from the route. With that we are going to end today's discussion on breadth first search. We are going to be using breadth first search for finding the connected components in a graph and we will see that in the next class.