

IIT Madras
ONLINE DEGREE

Mathematics for Data Science 1
Professor. Madhavan Mukund
Department of Computer Science
Indian Institute of Technology, Madras
Lecture No. 66
Complexity of BFS and DFS

(Refer Slide Time: 00:14)

Complexity of BFS and DFS

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

Mathematics for Data Science 1
Week 10

So, having seen some of the applications that we can achieve using BFS and DFS. Let us go back a little bit and look at the connection between BFS and DFS and these representations that we talked about Adjacency Matrices and Adjacency Lists.

(Refer Slide Time: 00:28)

BFS and DFS

Breadth first search

- Explore graph level by level
- Keep track of
 - $visited : V \rightarrow \{True, False\}$
 - Queue of unexplored vertices
- BFS from vertex j
 - Set $visited(j) = True$
 - Add j to the queue
- Explore vertex i at head of queue
 - For edge (i, j) , if $visited(j)$ is **False**,
 - Set $visited(j)$ to **True**
 - Append j to the queue
- Stop when queue is empty

Depth first search

- Start from i , visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Keep track of
 - $visited : V \rightarrow \{True, False\}$
 - Stack of suspended vertices

So, let us just formally remember what BFS does. So BFS explores a graph level by level. So, we maintain 2 pieces of information we maintain this flag called visited, which indicates whether a vertex have been visited or not. And we keep this queue of unexplored vertices. So initially, we mark all vertices as unvisited, we start from a vertex j . So, we start by setting that to be visited, set its value of visited to true, add j to the queue, and then we repeatedly process the queue.

And by processing the queue, what we mean is we take out the first element in the queue, look at its neighbors, and if there is any neighbor, which is unvisited, we mark it as visited, and push that neighbor back into the queue, so it will get processed later on. And finally, when the queue gets empty, BFS terminates.

On the other hand, with Breadth first search, we do a kind of aggressive or, impatient traversal. So, we start with i , and we visit 1 neighbor j . And now once we visited 1 neighbor, j , instead of going back to i and continuing with another neighbor of i we, we suspend i , and we go to neighbors of j . The first time we find a neighbor of j again, we will suspend j , and go to a neighbor, that neighbor k and continue.

And at this point, when we finish, and we cannot go further, we go back. And then we have traversed back all the suspended vertices until we find 1 which is not finished, and look for another neighbor, and so on. So here, we keep track of this visited information like in BFS, but we also

have the stack, which remembers the suspended vertices, so we can go back in the correct sequence Last In First Out.

(Refer Slide Time: 02:05)

Complexity BFS and DFS

Graph Properties:

- $G = (V, E)$
- $|V| = n$
- $|E| = m$
- If G is **connected**, m can vary from $n - 1$ to $n(n - 1)/2$
- In both BFS and DFS, each reachable vertex is visited exactly once
 - Visit and explore at most n vertices
- Each visited vertex is explored once
 - Check all outgoing edges
 - How long does this take?

Adjacency matrix

- To explore i , scan row i
- Look up n entries, regardless of number of actual edges from i
 - Degree of i
- Overall, n^2 steps

Adjacency list

- To explore i , scan list of neighbours of i
- Time to explore i is degree of i
- Degree varies across vertices
- Estimate overall time?

So, what I want to talk about is how much time this takes as a function of the size of the graph that we are trying to explore. So typically, in a graph, there are 2 parts, there is the vertex set, and there is the edge set, which is a subset of the pairs of vertices. So, edge relation is a subset of V cross V . Now, the vertex set is usually denoted as having size n . So, this tells you how many nodes or vertices there are in the graph.

Now the same set of vertices, you can draw many edges, or we can draw a few edges. So, actually, the parameter of the number of edges is independent, in a sense, as a measure of how complicated the graph is. So, this is usually denoted by a small m . So, m is usually the number of edges, n is the number of vertices. So, we saw that, if you have a tree, that is a minimally connected graph, then you will have n minus 1 edges. So, we can have interesting graphs in which the number of edges is roughly the same as the number of vertices, you have n vertices, you have n minus 1 edges.

On the other hand, if you connect everything to everything, then for every pair of vertices, you have an edge. And this gives us n into n minus 1 by 2, which is n choose 2 vertices. So, this is about n square so some something like n square, so we can have either order n vertices or order n

square vertices. So therefore, the number of edges, so the number of edges forms and somewhat independent parameter in our calculation.

So, now let us look at both BFS and DFS and see how they do this. So, the first thing is that they visit and explore every vertex exactly once, that is the purpose of that flag visited, it makes sure that we never go back to a vertex which is already visited and try to visit it a second time. So therefore, we will visit and explore each vertex exactly once. So, that whole thing happens in n times. There are n times when we visit and explore vertices. Now, what is exploring a vertex mean?

Exploring a vertex essentially means looking at all its neighbors. So, if we are looking at all the neighbors of a vertex, we are looking at all the edges which are outgoing from that vertex. So, the question is, how long does it take us to do this, when we are actually doing it computationally not doing it by looking at the picture and doing it by hand. So, we said that there are 2 representations that we have of the graph.

The first is this adjacency matrix in the adjacency matrix, the entries are 0/1. And the ij th entry indicates whether there is an edge from vertex i to vertex j . So, if there is an edge, it is 1 if there is no edge is 0. So, if we want to look at the outgoing edges of a vertex i , the only way we can do it in an adjacency matrix is to walk down the entire row for i . So, these are also look at $a_{i,1}, a_{i,2}, a_{i,3}$ and so on up to $a_{i,n-1}$ if the vertices are numbered 0 to $(n-1)$.

So, we will have to look up the entire row so, whether or not i has many neighbors, or few neighbors are no neighbors at all. I mean, of course, in an undirected graph, it must have at least if we had reached i during this thing, it would have an incoming thing. So, at least one neighbor, but if it is a disconnected vertex from where we are starting, like we saw a connected component, which has only a single vertex, then it may have nothing at all. But we would not know that until we see the entire row for i .

So, regardless of how many neighbors i actually has, we have to spend time proportional to n , to discover all these neighbors. So, this means that I have overall and processing n vertices. And for each vertex, I have to scan n entries in this matrix. So, $n \times n$, so I have to do something proportional to n^2 , in order to do Depth first search or Breadth first search.

On the other hand, if I use an Adjacency list, then I have for each vertex an explicit list of its neighbors. So, if it has a lot of neighbors, at least to be long, if it has very few neighbors, and this will be short, but I will spend no more and no less time than I need to scan this list. So, if I have k neighbors, I have to look at k entries. But I do not need to spend more, I do not need to spend n steps looking for k entries when k is small.

The now, the problem with this is that the degree of the vertex as we call it, right degree is the number of vertices which are number of edges, which are incident at a vertex, the degree of a vertex varies. So, maybe for vertex 1, I had a small degree vertex 2, I had a big degree and so on. So, if I want to count how many steps it takes across these n vertices, I have to add up the degrees, I have to look at how much time it takes degree 1, how much time it took for degree 2, and so on. So, how do I get a good way of estimating what this adds up to?

(Refer Slide Time: 06:48)

Calculating with degrees

Adjacency list

- To explore i , scan list of neighbours of i
- Time to explore i is degree of i
- Total time is the sum of the degrees

Sum of degrees

- Each edge (i, j) contributes to degree of both i and j
- Sum of degrees is $2m$

Diagram: A vertex i is shown with four edges connecting to other vertices. Below the vertex is a circled number 4, indicating its degree. Handwritten red text (i, j) is also present.

Footer: Madhavan Mukund, Complexity of BFS and DFS, Mathematics for Data Science

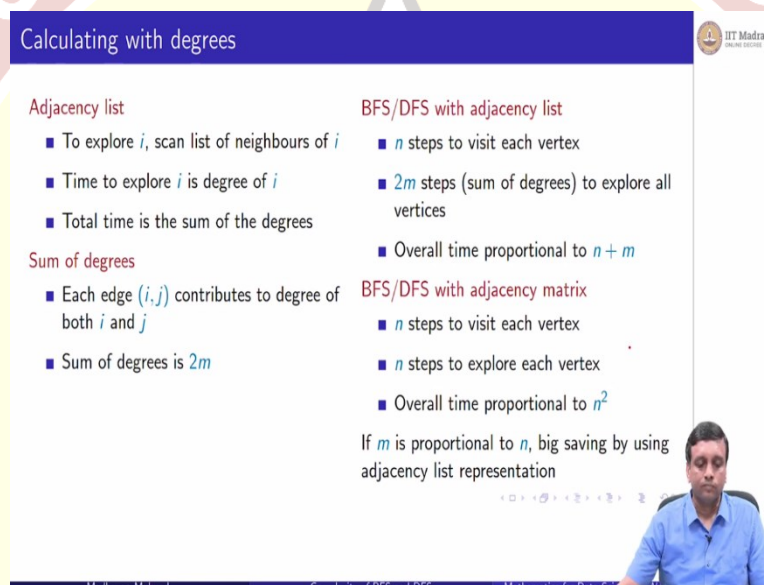
So, the question is, when I am processing an Adjacency list, I do work proportional to degree of each vertex added up. When I process vertex 0, I will look at all its neighbors. So, I will spend time proportional degree of 0 and I do vertex 1, the same thing with vertex i the same thing. So, we are really interested in identifying what the some of the degrees in an undirected graph represents. So, what is a degree?

A degree indicates a number, which is the number of things going out. So, if I have i , then if I have so many things going out, then I get a contribution of 4 from i to the sum of the degrees, because

it has 4 outgoing things. But each of these things will go and terminate also in j . So, if I look at this edge i,j , it adds 1 to the degree of i , it also adds 1 to the degree of j . So, each edge contributes to degree of both i and j .

And every number that I get in the degree must come from some edge. So, since there are m edges, and each edge contributes to the degree count of both the starting point and the ending point, the sum of the degrees must actually be $2 \times m$. For each, each vertex each edge, 1 to m , it contributes 1 plus 1. So, it is 2 times.

(Refer Slide Time: 08:16)



Calculating with degrees

Adjacency list

- To explore i , scan list of neighbours of i
- Time to explore i is degree of i
- Total time is the sum of the degrees

Sum of degrees

- Each edge (i,j) contributes to degree of both i and j
- Sum of degrees is $2m$

BFS/DFS with adjacency list

- n steps to visit each vertex
- $2m$ steps (sum of degrees) to explore all vertices
- Overall time proportional to $n + m$

BFS/DFS with adjacency matrix

- n steps to visit each vertex
- n steps to explore each vertex
- Overall time proportional to n^2

If m is proportional to n , big saving by using adjacency list representation

So, now we can see that if you have BFS or DFS with an adjacency list, then you make n steps, you take n steps to visit each vertex. And then across the n vertices, you take the, the sum of the degrees or $2m$ steps to explore all of them. So, this says that the time that overall that you spend is proportional to n plus m , you have to visit all the vertices because for instance, if the entire graph is disconnected, there are no edges at all, m is 0, but it does not mean that you will finish your DFS or BFS in 0 time, you have to visit all the n things.

So, you have to spend n steps looking at all the vertices and across the vertices, the contribution of that vertex to your work is the degree of that vertex. So, across the vertices is the summation of the degrees, that is where this m term comes. Whereas as we saw, if you did this with an adjacency matrix, it does not matter really what the degrees are, and what m is, you will end up having to

spend n steps for every vertex, because you cannot find out the neighbors of a vertex without looking at the entire row. So, you end up taking time n square.

So, the whole distinction is between n square and n plus m . So, we said that m could be small. So, m could be like n , like in a tree, we have only n minus 1 vertices, or m could be large, it could, in principle be n square. If it is n square, then these 2 are roughly the same. But if it is small, then we have a difference between something which is proportional to n , and something which is proportional to n square.

And that is why adjacency lists can be beneficial for doing our thing. So, if m is proportional to n , then we have a big saving by using adjacency list representations. So, though adjacency matrix is fine in terms of understanding what is going on in terms of mechanically computing something, it can actually cost us not just because it is a large thing, and we are keeping a lot of 0s in it. But also, because in order to extract this interesting information about what are the neighbors, we have no better way than to walk down the entire row.

(Refer Slide Time: 10:19)

The slide is titled "More about degrees" and features a list of properties of vertex degrees. To the right of the list, there are handwritten calculations in red ink. At the bottom right, a small video inset shows a lecturer in a blue shirt.

More about degrees

- Degree of a vertex i is
 - Number of 1's in row i of adjacency matrix
 - Number of 1's in column i of adjacency matrix
 - Length of adjacency list for i
- Sum of degrees is $2m$
 - Sum is an even number
 - For each vertex with odd degree, must be another vertex of odd degree to make the sum even

Handwritten notes:

$3 + 17 \rightarrow 20$
odd + odd \rightarrow even

$3 + 16 \rightarrow 19$
odd + even \rightarrow odd

Lecturer: Madhavan Mukund

Page footer: Madhavan Mukund, Complexity of BFS and DFS, Mathematics for Data Science

So, here are some more interesting things that is useful to remember about degrees. So, remember that the degree of a vertex can be obtained from an adjacency matrix by just looking at its row. So, if you just count the number of 1s in the adjacency matrix in a row, you get the degree of the vertex. And if it is an undirected graph, you can also get it from the columns, because it is the

number of incoming edges and the number of outgoing is the same, because it is a symmetric kind of edge graph.

And if you have an adjacency list, then the degree is just the length of the list associated with i , it is all the neighbors of i . Now, we already calculated that the sum of the degrees is $2n$. So, 2 times anything must be an even number. So, the sum is an even number, because it is 2 times the number of edges. So, the number of edges themselves will be odd, but the sum of the degrees is going to be 2 times m , and therefore the sum of the degrees is an even number.

So now, you should remember that, if you have an odd number plus an odd number, then I will get an even number. So, for example, if I do 3 plus 17, then I get 20. But if I do an odd number plus an even number, then I get an odd number. So, you can think of the fact that if I am pairing up an odd number in 2's, then I get 1 leftover element, that is why it is odd. And if I am pairing up an even number in 2s, I get no leftover elements.

So if I take them together and pair it off in 2s, then I have 1 leftover element, that is why it is odd. Whereas if I have 2 odd numbers, then each of them contributes 1 leftover element, I can take those leftover elements and pair them up and I get a new pair, and so it is even, so, odd plus odd is even or plus even is odd. So, this means that if I spot an odd degree vertex in my graph, it cannot be the only 1 there must be another 1 right because otherwise the sum of the degrees will become odd everything else is even plus 1 odd number will be an odd number.


So, if there is an odd number here, there must be an odd number somewhere else, so I can pair them off. Similarly, if I find a third odd vertex there must be a fourth odd vertex. So, for every odd degree vertex, there must be another odd degree vertex. So in other words, the number of degrees or the odd vertex or degree vertices must themselves be even, if I have an odd number of vertices with odd degree, then the overall sum of the degrees will be odd which is not possible.

So remember also that the degree of a vertex can be any number between 0 and n minus 1. So, 0 happens when this vertex is actually disconnected from the entire graph, n minus 1 happens when it is connected to every 1 of the remaining n minus 1 vertices. So, it is not connected to itself obviously, we said no self loops, but it could be connected to everything else. So, the special case where every vertex is connected to every other vertex is what is called a Complete graph.

(Refer Slide Time: 13:15)

More about degrees

- Degree of a vertex i is
 - Number of 1's in row i of adjacency matrix
 - Number of 1's in column i of adjacency matrix
 - Length of adjacency list for i
- Sum of degrees is $2m$
 - Sum is an even number
 - For each vertex with odd degree, must be another vertex of odd degree to make the sum even
 - Number of vertices of odd degree is even
- Degree of a vertex between 0 and $n-1$
 - Degree 0 — disconnected vertex
 - Degree $n-1$ — connected to all other vertices
- Complete graph — every vertex has degree $n-1$



Madhavan Mukund Complexity of BFS and DFS Mathematics for Data Science

So, for instance, on a 3 vertices graph, a complete graph is a triangle, if a 4 vertex graph, then I must connect everything to everything. So, I have a square and then I also have the diagonals. Similarly, if I have a 5 vertex graph, then I will have a pentagon with all these diagonal things. So, this is what is called a complete graph. So, in a complete graph, the degree of every vertex is n minus 1 and I will actually have n into n minus 1 by 2 because every pair is connected. So, I have n choose 2 edges.

(Refer Slide Time: 13:44)

More about degrees

- Degree of a vertex i is
 - Number of 1's in row i of adjacency matrix
 - Number of 1's in column i of adjacency matrix
 - Length of adjacency list for i
- Sum of degrees is $2m$
 - Sum is an even number
 - For each vertex with odd degree, must be another vertex of odd degree to make the sum even
 - Number of vertices of odd degree is even
- Degree of a vertex between 0 and $n-1$
 - Degree 0 — disconnected vertex
 - Degree $n-1$ — connected to all other vertices
- Complete graph — every vertex has degree $n-1$
- If all degrees are bounded by k , at most $kn/2$ edges
- For directed graphs, indegree and outdegree
- Sum of indegrees = m = Sum of outdegrees

Madhavan Mukund Complexity of BFS and DFS Mathematics for Data Science

Now many graphs that we encounter in practical problems like the 1s we discussed about graph coloring or vertex cover, so on. In many reasonable situations, you can actually say that a particular node will have no more than a certain number of neighbors for example, remember the graph coloring problem for the security cameras? So, clearly, if I give you a particular building, then I know that a given intersection will not have more than a certain number of corridors fixing I cannot have a large number of corridors at some point.

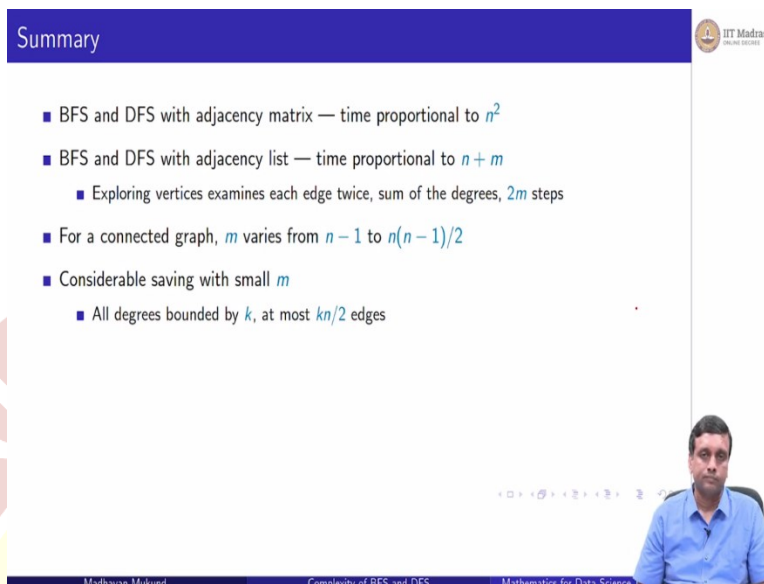
So, there will be some upper bound saying that no corridor, no intersection has more than 5 corridors which meet there or if we are looking at some, some other problem, which is say for instance, placing ambulances at an intersection, then you want to know how many roads meet at that intersection. Or if you are looking at say, the timetabling problem you want to know how many different courses can be scheduled in the same slot. Now, obviously, if you have a fixed number of courses in your curriculum, not more than that many can be there.

So, very often the degree is actually independently bounded by some number, you do not have arbitrarily large degree. So, the number the degree bounded by an external constraint. And if you have now a constraint on the degree, it says that each degree is k , then the total sum of the degrees can be at most k times the number of vertices k times n . And since this is twice the number of edges, the number of edges must be $k \times n/2$.

So in other words, if you have a bounded degree graph, where every edge has a bounded degree, which is independent of the size of the graph, then the total number of edges cannot be more than linear, cannot be more than some function of n . So therefore, you should be working with adjacency lists. And finally, if you have directed graphs, we said that it is no longer enough to talk about degree because there are edges coming in, and there are edges going out, which are quite independent of each other. So, we must talk about the in degree and the out degree.

So, in this case, because each edge contributes to 1 in degree and out degree, the sum of the in degrees is the number of edges and the sum of the out degrees is also the number of edges. So, together the in degrees plus out degrees is $2m$, but they get partitioned into 2 quantities which add up to m each.

(Refer Slide Time: 16:06)



Summary

- BFS and DFS with adjacency matrix — time proportional to n^2
- BFS and DFS with adjacency list — time proportional to $n + m$
 - Exploring vertices examines each edge twice, sum of the degrees, $2m$ steps
- For a connected graph, m varies from $n - 1$ to $n(n - 1)/2$
- Considerable saving with small m
 - All degrees bounded by k , at most $kn/2$ edges

Madhavan Mukund Complexity of BFS and DFS Mathematics for Data Science IIT Madras

So, what we have seen is that if we do an analysis of how the BFS and DFS work with respect to exploring the neighbors of a vertex, if we use an adjacency matrix, it turns out that regardless of how many neighbors a vertex has, we must scan the entire row and therefore the time taken by BFS and DFS becomes proportional to n times n , I have to process n vertices and for each vertex, I have to scan n elements in that row.

On the other hand, if we use an adjacency list, then the time taken to process a vertex is exactly the number of neighbors it has. So, it is across the vertices is the sum of the degrees and this gives us an overall timing, which is proportional to n plus m . And we also saw that there is a large variation in m . So, m can be linear, like in a tree, or it could be quadratic, like in a complete graph. So, it is important to be able to distinguish and use the appropriate representation in particular use adjacency list whenever we can.

Another situation where we get small number of edges is when we have a bounded degree. So if we, if we have some other constraints on our problem, it says that the number of edges coming out of a given node cannot be more than a certain number independent of the total number of edges. Then we have necessarily a graph which has only a linear number of edges. So again, an adjacency list representation would work best.