

IIT Madras
ONLINE DEGREE

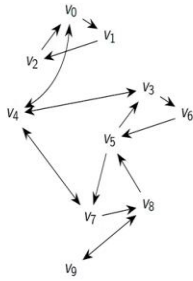
Programming Data Structures and Algorithms using Python
Professor Madhavan Mukund
Breadth First Search (BFS)

First strategy to systematically explore a graph is called Breadth First Search.

(Refer Slide Time: 0:15)

Reachability in a graph

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked



Madhavan Mukund

Breadth First Search

So, remember that we are looking at reachability. So, what reachability said was that we start from a vertex and marked it as reachable. And now we systematically mark all the neighbours of already known reachable marking. So, you start with the thing, you know it is reachable look at its neighbours, mark them as reachable, look at their neighbours mark them as reachable, stop when the target is marked, but you want to avoid going around and redoing something. So, you need to keep track of which guys are marked.

(Refer Slide Time: 0:37)

Reachability in a graph



- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Choose an appropriate representation
 - Adjacency matrix
 - Adjacency list
- Strategies for systematic exploration
 - Breadth first — propagate marks in "layers"
 - Depth first — explore a path till it dies out, then backtrack

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 4 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

| 0 | {1,4} |
|---|---------|
| 1 | {2} |
| 2 | {0} |
| 3 | {4,6} |
| 4 | {0,3,7} |

| 5 | {3,7} |
|---|-------|
| 6 | {5} |
| 7 | {4,8} |
| 8 | {5,9} |
| 9 | {8} |



Madhavan Mukund

Breadth First Search

DOCA

We also said that we have two representations possible, we have this adjacency matrix, and we have the adjacency list, and we have to choose the correct one. So, what we are going to look at this time is the first strategy, the breadth first search, and next we will look at the depth first search.

So, in a breadth first search, we propagate these things one level at a time. So, we start with the vertex which is reachable in 0 steps, then we see what all we can reach in 1 step. Then from the 1 step vertices, we see what all we can reach in 1 more step, so in 2 steps. From 2 what we can reach in 1 more step that is in 3 steps, and so on. So, we kind of that is why it is called breadth first, kind of growing this set of nodes, broader and broader.

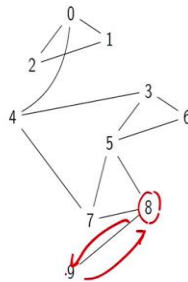
And in depth first search, on the other hand, I pick one neighbour, I say, I can go from here to this neighbour, then you do not go and look at other neighbours, you say if I can go to that neighbour, where can I go from there? So, I start exploring from that neighbour. And I keep exploring until I get stuck, I cannot go any further. Then I come back and at each point, I say I got stuck now, which are the other neighbours, which I did not explore. So, I go backwards and forwards and explore all the nodes. And this is called depth first search.

(Refer Slide Time: 1:43)

Breadth first search (BFS)



- Explore the graph level by level
 - First visit vertices one step away
 - Then two steps away
 - ...
- Each **visited** vertex has to be **explored**
 - Extend the search to its neighbours
 - Do this only once for each vertex!



Navigation icons: back, forward, search, etc.

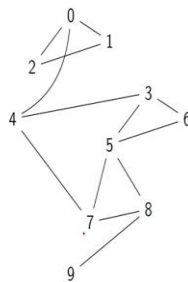
Madhavan Mukund

Breadth First Search

Breadth first search (BFS)



- Explore the graph level by level
 - First visit vertices one step away
 - Then two steps away
 - ...
- Each **visited** vertex has to be **explored**
 - Extend the search to its neighbours
 - Do this only once for each vertex!
- Maintain information about vertices
 - Which vertices have been visited already
 - Among these, which are yet to be explored



Navigation icons: back, forward, search, etc.

Madhavan Mukund

Breadth First Search

So, in breadth first search, we explore the graph level by level. So, we first visit vertices one step away, then two steps away, and so on. So, whenever we reach a vertex for the first time, it has to be explored in the sense that I have to look at its neighbours, which I have not seen so far, and explore them. And but, we must make sure that we do this only once. So, we do not want to come back from, for example, we go from, as we said, we go from 9 and we say we restate, then I do not want to look at 8, explore it, and then come back to 9.

So, we must make sure that we do not visit a vertex twice. Or in other words, we do not explore its neighbours twice, because if we do that, then we could go around forever. So, for this, we need to maintain some information, so we need to know which vertices have been

visited. And among these, we need to know which ones have been explored. So, those which have been visited, are then to be explored.

So, there are two steps. First, I visit a vertex saying this is the first time I have come here, so I need to explore it. But I might be busy doing something else. So, I come back later and I say now I have not visited this, let me explore it.

(Refer Slide Time: 2:50)

Breadth first search (BFS) ...

■ Assume $V = \{0, 1, \dots, n-1\}$

■ $visited : V \rightarrow \{True, False\}$ tells us whether $v \in V$ has been visited

■ Initially, $visited(v) = False$ for all $v \in V$

Madhavan Mukund Breadth First Search

So as usual, let us assume that our vertices are 0 to n minus 1, then what we do is we set up this dictionary or function or whatever you want to call it, which marks each visit vertex as either visited or not visited. So, we have the keys or the position 0 to n minus 1. And each entry is either true or false. So initially, everything is false, nothing is visited.

(Refer Slide Time: 3:13)

Breadth first search (BFS) ...

■ Assume $V = \{0, 1, \dots, n-1\}$

■ $visited : V \rightarrow \{True, False\}$ tells us whether $v \in V$ has been visited

■ Initially, $visited(v) = False$ for all $v \in V$

■ Maintain a sequence of visited vertices yet to be explored

■ A queue — first in, first out

■ Initially empty

```
class Queue:
    def __init__(self):
        self.queue = []

    def addq(self, v):
        self.queue.append(v)

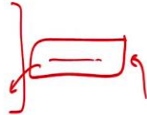
    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return v

    def isempty(self):
        return self.queue == []

    def __str__(self):
        return str(self.queue)
```

Madhavan Mukund Breadth First Search

- Assume $V = \{0, 1, \dots, n-1\}$
- $\text{visited} : V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
 - Initially, $\text{visited}(v) = \text{False}$ for all $v \in V$
- Maintain a sequence of visited vertices yet to be explored
 - A queue — first in, first out
 - Initially empty



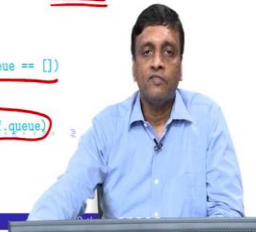
```
class Queue:
    def __init__(self):
        self.queue = []

    def addq(self, v):
        self.queue.append(v)

    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return v

    def isempty(self):
        return self.queue == []

    def __str__(self):
        return str(self.queue)
```



Now we have to maintain those vertices. So, this is telling us which vertices are visited. But it has not told us which of these have been explored. So, there is a two step process, I visit a vertex, and then I explore it that is I look at its neighbours. So, in order to do that, we need to maintain this information in some systematic way so that does the ones which I have yet to explore will eventually be explored, and only once.

So, the standard way to do this is to use a queue. So, what is the queue, a queue is just what you think of as a queue in real life, it is you join a queue from one end, and then when you move up to the queue. And when you reach the end, that way, you are going to get serviced, you move out of the queue.

So, supposing standing in a bank, or you are standing, waiting for some delivery in a fast food place, and you join at the end of the queue, and when it is your turn, you will be served at the counter. So, there is an entry end, and there is an exit and otherwise the sequence. So, this is a queue.

So, let us, so we can of course describe a queue in Python as a list. And you can say that when I want to add to the queue, I add on the right hand side of the list, I append. And I when I want to exit the queue, I remove the leftmost element. So, let us just for convenience, wrap this up as a class and an object so that we deal with it more abstractly.

So, we have this class queue and that class queue internally maintains a list, which is called queue the small queue. So, self dot queue is initialized to empty. So, when I set up a queue, it is empty. Then I have an add operation. So, I want to add v to the queue. What it does, as we

said is it appends to the right so it takes a queue which is stored internally and appends the value v to it.

Now, when I want to remove from the queue, delete from the queue, I essentially want to take the zeroeth element, return it and reset the queue to start from the first element. So, I am taking the zeroeth element out. So, deleting from a queue or removing from a queue reduces the length of the queue by 1. So, this customer is gone, this customer is out of the queue, otherwise, the next customer not going to get to the head of the queue.

So, the first element of the queue is the one that is going to be processed next. And when you process it, you remove that thing from the queue. Now, obviously, there is a problem here, if there is nothing in the queue? If I try to process an element, and the queue is empty, it should not work. So, what we are going to do is not worry too much about it. But we are just going to say that, by default, we assume that there is nothing to process.

And if the queue is not empty, so I put in a check before, I do not want this call this thing to give me an error. So, this slice is not going to give me an error. But if I try to access the zeroeth element of an empty list, Python is going to say it is an index error. So, I am going to ensure that it is not empty. So, it is not empty, is basically saying that the internal queue is not the empty list.

So, I write a separate function, which basically returns true or false. And if it returns true, in other words, it is not empty, then I will process it. If it is empty, then I will skip it. And finally, just for inspection for debugging, you might want to check what the queue looks like. So, we will have this function which converts this queue object to a string by just returning the string representation of the internal list. So, if I try to print a queue, it will print a list. That is what it says.

So, this is now my class representing a queue, which I will be using in breadth first search in order to maintain this extra information about which vertices have been visited, but whose exploration is still pending.

(Refer Slide Time: 6:50)

Breadth first search (BFS) ...



- Assume $V = \{0, 1, \dots, n-1\}$
- `visited` : $V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
 - Initially, `visited(v) = False` for all $v \in V$
- Maintain a sequence of visited vertices yet to be explored
 - A `queue` — first in, first out
 - Initially empty

```
q = Queue()
for i in range(3):
    q.addq(i)
    print(q)
    print(q.isempty())
for j in range(3):
    print(q.delq(), q)
    print(q.isempty())
```

0, 1, 2

0, 1

[0]

[0, 1]

[0, 1, 2]

False

0 [1, 2]

1 [2]

2 []

True




So, this to get an idea how the queue data structure works. So, supposing I start with an empty queue, and then for 0, 1, 2, I add them. And then for 0, 1, 2, I remove them. So, what I do is I print the queue at every stage. This is why I said for debugging purposes. So, after I add a 0, my queue looks like this. After I add a 1, my queue looks like this, after added 2 my queue looks like this. And now I check whether the queue is empty. And as expected, it says it is not empty, it is false. So, this is the first half of the program.

And now in the second half, I remove from the queue 3 times. So, I print the value that is removed and the remaining q. So, initially, the value because this is my q, so the leftmost item 0 is removed. And what remains is 1 2, I do it again, 1 is removed. And what remains is 2. I do it 1 more time, the final item comes out and add the empty list. And now if I query whether the queue is empty, indeed, it is empty and returns me true. So, this is just a simple way to illustrate how this queue data structure works in this object situation.

सिद्धिर्भवति कर्मजा

(Refer Slide Time: 8:01)


Breadth first search (BFS) ... 

- Assume $V = \{0, 1, \dots, n-1\}$
- $visited : V \rightarrow \{True, False\}$ tells us whether $v \in V$ has been visited
 - Initially, $visited(v) = False$ for all $v \in V$
- Maintain a sequence of visited vertices yet to be explored
 - A queue — first in, first out
 - Initially empty
- Exploring a vertex i
 - For each edge (i, j) , if $visited(j)$ is False,
 - Set $visited(j)$ to True
 - Append j to the queue

```
q = Queue()
for i in range(3):
    q.addq(i)
    print(q)
    print(q.isempty())
for j in range(3):
    print(q.delq(), q)
    print(q.isempty())
```

```
[0]
[0, 1]
[0, 1, 2]
False
0 [1, 2]
1 [2]
2 []
True
```

Madhavan Mukund Breadth First Search



So, with this queue in place, what are we going to do? When we explore a vertex, we look at all its neighbours. So, there are two things remember whether a vertex is visited and whether it is explored. So, when we look at a neighbour, we first check, whether it is been visited, if it is been visited, then we have already done something or something is pending, we do not touch it again.

So, if visited of j is false, then we want to visit it. So, we say, if it has not been visited before, I will now visit it, so I will mark it as visited. And I will put this thing into the queue. Putting into the queue means at some later stage, I am going to look at its neighbours I am going to explore it.

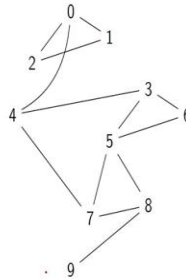
So, this is the basic step in breadth first search, I start with the vertex I mark it is visited, put it in the queue, then I pull out something from the queue. I check whether its neighbours are visited, put them into the queue. And I keep doing this until I processed everything that has been visited.

(Refer Slide Time: 8:59)

Breadth first search (BFS) ...



- Initially
 - $visited(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty
- Start BFS from vertex j
 - Set $visited(j) = \text{True}$
 - Add j to the queue
- Remove and explore vertex i at head of queue
 - For each edge (i, j) , if $visited(j)$ is False,
 - Set $visited(j)$ to True
 - Append j to the queue
- Stop when queue is empty



Navigation icons

Madhavan Mukund

Breadth First Search

So, initially visited is false for every v , and the queue is empty. So, if I start my breadth first search at a particular vertex j , then I set that vertex to be visited, I set its visited value to true and I added it to the queue. And now I keep removing from the queue. So, now I have something in the queue.

So, I would have j and j needs to be explored. So, right at the beginning, I have started off with one vertex which needs to be explored. So, I remove it from the head of the queue. I look at all its neighbours, and all the unvisited neighbours get pushed in the queue. Then at the next step, I will take one of those neighbours and do the same thing and keep going until I have processed everything which I have visited. So, I stop when the queue is empty.

(Refer Slide Time: 9:38)

Breadth first search (BFS) ...



- Initially
 - $visited(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty
- Start BFS from vertex j
 - Set $visited(j) = \text{True}$
 - Add j to the queue
- Remove and explore vertex i at head of queue
 - For each edge (i, j) , if $visited(j)$ is False,
 - Set $visited(j)$ to True
 - Append j to the queue
- Stop when queue is empty

```
def BFS(AMat, v):  
    (rows, cols) = AMat.shape  
    visited = {}  
    for i in range(rows):  
        visited[i] = False  
    q = Queue()  
    visited[v] = True  
    q.addq(v)  
  
    while(not q.isEmpty()):  
        j = q.delq()  
        for k in neighbours(AMat, j):  
            if (not visited[k]):  
                visited[k] = True  
                q.addq(k)  
  
    return(visited)
```

Navigation icons

Madhavan Mukund

Breadth First Search

- Initially
 - $visited(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty
- Start BFS from vertex j
 - Set $visited(j) = \text{True}$
 - Add j to the queue
- Remove and explore vertex i at head of queue
 - For each edge (i, j) , if $visited(j)$ is False,
 - Set $visited(j) = \text{True}$
 - Append j to the queue
- Stop when queue is empty

```
def BFS(AMat, v):
    (rows, cols) = AMat.shape
    visited = {}
    for i in range(rows):
        visited[i] = False
    q = Queue()

    visited[v] = True
    q.addq(v)

    while(not q.isEmpty()):
        j = q.delq()
        for k in neighbours(AMat, j):
            if (not visited[k]):
                visited[k] = True
                q.addq(k)

    return(visited)
```



So, here is my Python code for this breadth first search. So, I am taking an adjacency matrix. This is the graph that I am trying to explore. And this is my starting vertex v . So, I need to now keep track of these data structures. I need to keep track of the queue and I need to keep track of this visited value. So, the first thing I do as before is I find out how many vertices there are by querying the shape attribute that NumPy tells me.

So, remember that this is, logically it should be a square matrix. So, rows should be equal to columns, I just want one of them. So, what I do now is I initialize this visited thing which I will keep as a dictionary to be false. So, for every vertex for every i in range of rows, that is for everything from 0 to n minus 1, I initialize the visited of that i to be false. And I create an empty queue. So, I am here right now. So, I have basically, I have initially set visited v to be false for all the vertices in the graph. And I have taken the queue to be empty.

So, now I have done this. So, I have now finished my initialization. So, now I come to this one. So, I want to start from j , which is, confusingly called V over there. So, I set visited v equal to true. So, I initialize my thing by saying the vertex I start my BFS from has been visited, and I put it into the queue.

And now finally, I have this loop here. So, it says, so long as the queue is not empty, take the first vertex in the queue, initially, is going to be the vertex V that I started with. For every neighbour, remember, we had this neighbour function, which looked at all the elements in the row j , and pulled out the ones. So, that gives me a list of neighbours. So, for every K , which is a neighbour of J , if it has not been visited, then make it visited. So, basically, do this, make it visited and add it to the queue.

So, notice what is happening. So, basically, this vertex is now being put into the queue for later processing. So, every vertex that I see for the first time gets put into the queue, so it will eventually be explored. But I will never put it a second time because it will be marked as visited. So, every time every vertex that is actually visited by it at some level of the search will be put into the queue. But once it is put in the queue, and I will put a second time because visited has been set to true, and I only do this for visited, where visited is false.

And finally, at the end of this whole thing, my visited dictionary is set to true for every vertex I actually saw. So, if I look at that, I should get my answer. So, I just return that dictionary as the outcome of this process. And this should tell me, which are all the vertices, which are reachable from the starting vertex V.

(Refer Slide Time: 12:36)

BFS from vertex 7

| Visited | |
|---------|------|
| 0 | True |
| 1 | True |
| 2 | True |
| 3 | True |
| 4 | True |
| 5 | True |
| 6 | True |
| 7 | True |
| 8 | True |
| 9 | True |

| To explore queue |
|------------------|
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2

Madhavan Mukund Breadth First Search

So, let us see how it works on this graph, for instance. So initially, I have everything visited is false. And supposing I want to run my BFS from vertex 7. So, when I do that I initially set visited of 7 to be true, and I put 7 into the queue. So, my queue looks like this. Now, I start processing the queue. So, I pull out the 7, and it has neighbours, 4, 5, and 8, all of which are previously unvisited. So, I marked them all visited, and I removed the 7 and put 4, 5 and 8 into the queue.

So, now I have to process these in this particular order. So, because 4 got put into the queue, first, I will now pull out 4 and look at its neighbours. So, the neighbours of 4, if you see are 0, 3, and 7, of which 7 is already marked, so from 4, it will add two new things 0 and 3 to be

true, and put them in the queue. And now 4 is done. Then I pick up the 5, because that is the next thing here, so 5 is my next element in this queue here. So, 5 is here. So, I pull out the 5.

And now 5 has neighbours 3, 6, 7, and 8 of which 7 and 8 were already seen before, so only 6 gets added. So, we keep doing this, now we pull out 8, and we look at its neighbour, so the new neighbour of 8. So, the neighbours of 8 are 5, 7 and 9, of which only 9 is new. So, I will pull out 8 and put a 9 to the queue. Then I will pull out 9. And nothing I am sorry, I will put out 0, because I am doing it from the head of the few obviously. So, after 8 I pull out 0, so 0 has neighbours 1 and 4, but 4 is already been marked. So, it will only add 1 and 2.

And now I will explore 3, the next thing and there is nothing new to add. I will explore 6, there is nothing new to add. I will explore 9 there is nothing new to add, explore 1, there is nothing new to add. I will explore 2. So, now I have explored everything that I put into the queue and now the queue is empty and I stop. And this visited array or dictionary tells me that every vertex was visited in the process. So, this is how breadth first search works.

(Refer Slide Time: 14:48)

Complexity of BFS

$G = (V, E)$

- $|V| = n$
- $|E| = m$
- If G is connected, m can vary from $n - 1$ to $n(n - 1)/2$

In BFS, each reachable vertex is processed exactly once

- Visit the vertex: add to queue
- Explore the vertex: remove from queue
- Visit and explore at most n vertices

Exploring a vertex

- Check all outgoing edges
- How long does this take?

Adjacency matrix

- To explore i , scan $neighbours(i)$
- Look up n entries in row i , regardless of $degree(i)$

Adjacency list

- List $neighbours(i)$ is directly available
- Time to explore i is $degree(i)$
- Degree varies across vertices

Sum of degrees

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

Madhavan Mukund | Breadth First Search

So, as an algorithm, how much time does it take? So, let us assume that we have n vertices and m edges. So, if G is a connected graph as we said everything is connected, then it has at least n minus 1 edges, but it could have at most n into n minus square 1 by 2, so n squared edges. Now, in BFS each reachable vertex is processed exactly once this is what we said once you see it for the first time you mark visited, you never see it a second time.

So, there are two steps you visit the vertex, which case you added to the queue and you explore the vertex, you remove it from a queue. So, each vertex enters the queue once and

leaves the queue once. So, you visit and explore at most n vertices. I mean, you may not explore all of them, if not all of them are reachable if you have parts of the graph which are unreachable will not reach them. But you certainly cannot explore more than n steps, you have only n vertices and each vertex can enter and leave the queue only once.

So, then the question is, how much time does it take for the second step? See, putting it into the queue is a one, is a kind of single operation, I set visited to true and put it into the queue. When I remove it and explore it, I have to explore all its neighbours. So, how long does that take?

So, in this adjacency matrix representation, we have to scan all the neighbours? So, we have to look at all n entries in the row of i regardless of how many neighbours it actually has. So, I have no option but to spend time looking at all the neighbours. Whereas, if I had a cleverer version, which looked at the adjacency list, then I would only pick up the neighbours that are there I would not look at the neighbours, which are the non-existent neighbours, I would not have to look at what are the 0s and my adjacency matrix. So, it becomes faster.

But of course, now the question is, if I am doing it with an adjacency list, depending on the degree, it could take more or less time to process one vertex. So, if I am doing it with an adjacency matrix, it is easier to calculate what happens. I have put each vertex in once, and each vertex I spend n time processing its neighbours. So, I have n vertices I put in, and each one takes time order n to scan its neighbours. So, I have n times n is n squared.

Now, for adjacency list representation, I put each of these things into the queue once, but when I pull it out how much time I spend on it depends on how many vertices it is connected to. So, this is a little bit problematic. So, let us see how to analyse this. So, one thing we can observe is that even though the sum the degrees of individual vertices vary, I can tell you something about the total degrees added up across all the vertices. The claim is that if I add up the degrees of all the vertices in my graph, it must be two times the number of edges.

And why is this, that is because if I take a given edge, i comma j , then it contributes the degree of i and it contributes the degree of j . So, each edge gives me contribution to two vertices degrees. So, across all the m edges, I have 2 times m added to some degree or the other. So that means the sum of the degrees across all the vertices in a graph is always going to be 2 times m , regardless of how it is individually distributed among the given vertices.

(Refer Slide Time: 18:06)

Complexity of BFS

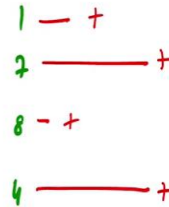


BFS with adjacency matrix

- n steps to initialize each vertex
- n steps to explore each vertex
- Overall time is $O(n^2)$

BFS with adjacency list

- n steps to initialize each vertex ✓
- $2m$ steps (sum of degrees) to explore all vertices
- An example of amortized analysis
- Overall time is $O(n + m)$



Navigation icons: back, forward, search, etc.

Madhavan Mukund

Breadth First Search

Complexity of BFS



BFS with adjacency matrix

- n steps to initialize each vertex
- n steps to explore each vertex
- Overall time is $O(n^2)$

BFS with adjacency list

- n steps to initialize each vertex
- $2m$ steps (sum of degrees) to explore all vertices
- An example of amortized analysis
- Overall time is $O(n + m)$

- If $m \ll n^2$, working with adjacency lists is much more efficient
 - This is why we treat m and n as separate parameters
- For graphs, $O(m + n)$ is typically the best possible complexity
 - Need to see each vertex and edge at least once
 - Linear time

Navigation icons: back, forward, search, etc.

Madhavan Mukund

Breadth First Search

So now, to come back to the complexity, if I use this adjacency matrix, as we said, in the worst case, we are going to visit everything. So, we are going to n times we are going to put them into the queue. But even before that, we have to first initialize this thing. So initially, we have to set visited i equal to false for everything. So, we have to look at all the vertices and initialize each vertex.

And then in the worst case, I am going to visit all of them and explore each of them. But it takes n steps to explore a vertex. So, I visit n vertices, and I take n time to explore each vertex. So, n times n is n squared. So, the overall time becomes n squared. Whereas, if I have an adjacency list, I still need to initialize the vertices. So, I will take n steps. But now, we can do something a little bit more sophisticated.

So, we say that, supposing I visited it as V_1 , V_7 , then V_8 , then V_4 , and so on. So, supposing this is my sequence, so let me just write it as 1, 7, 8, 4. So, I spend a certain amount of time doing this, I spend a certain amount of time doing that I spend less time doing this, and so on. So, this is how much time the inner part of that exploration takes place. It depends on the degree.

But if I add up all these red lines, what am I doing, I am adding up the degrees of all the vertices, I am adding up the degree of 1 plus a degree of 7. So, the total time I take is a sum of degree 1 plus degree 7 plus degree 8 plus degree 4. So, even though I cannot tell you which ones were fast, and which ones were slow, I can tell you that across these n vertices, the total time that I took was the sum of the degrees and that is $2m$.

So, this is a kind of interesting analysis. It is not saying anything about each individual thing is rather saying something about the thing as a whole and saying that if I do more work on one vertex, I must corresponding be doing less work on some other vertex. So, this is the form of accounting which so we are really doing some accounting, we are counting how many times these things happen.

So, the simpler form of accounting is I do this n times, and each time I spend so much time, so n times something. Whereas here, I am not saying that. I am not saying I do, I am not telling you precisely how much I spend each time, but I am saying that across these n vertices are explored, I am going to add up totally to only $2m$ steps, it cannot be more than $2m$, even if some of them are very large, others will have to be correspondingly small, because there is an upper bound on the sum of the degrees.

So, this is what is sometimes called amortized analysis. So, rather than looking at individual operations and adding them up, I look at a cumulative set of operations. And I say across the whole thing, I do not know that individual distribution very clearly, but I know that overall, it cannot take more. So, if I lose more, if I gain something there, I will lose it here. If I lose something here, I will gain it there. So, I must take $2m$ steps.

But still, this is not bad, because what it is saying is that I spend n steps visiting the vertices, initializing and visiting them. And the exploration step across all the n explorations takes $2m$ steps. So, the total time complexity now is not n squared, but n plus $2m$. And of course, I can drop the 2. And I can say it is order of n plus m . So, if I use an adjacency list, my amortized complexity is n plus m rather than n squared.

So, this is important for us, because in many situations, m is actually quite small compared to n squared. So, most of the graphs that we actually draw and many of the graphs we encounter in practice are what are called sparse graphs. In a sparse graph, I have edges which are proportional to n usually, rather than n squared. So, if I am doing m plus n time, I am doing much better than if I am doing an n squared time.

So, for graphs, typically, this is the best possible complexity order m plus n . So, this is considered to be linear time for a graph. Now, why is it the best possible time? Well, it is difficult to imagine any serious graph problem in which you do not examine the whole graph. So, if you examine all the graphs, and all the edges, all the vertices and all the edges, then you need to spend time order m plus n , you have to look at the whole graph. So, you cannot realistically do much without looking at the whole graph.

But on the other hand, if we separate out m from n , and we do not just take m to be its worst case, which is n squared, then I can show that certain implementations like this adjacency list actually worked much better.

(Refer Slide Time: 22:30)

Enhancing BFS to record paths

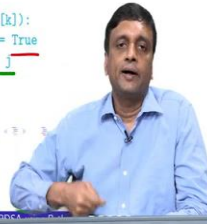
- If BFS from i sets $visited(k) = \text{True}$, we know that k is reachable from i
- How do we recover a path from i to k ?
- $visited(k)$ was set to True when exploring some vertex j

Madhavan Mukund Breadth First Search IIT Madras BSc Degree

- If BFS from i sets $\text{visited}(k) = \text{True}$, we know that k is reachable from i
- How do we recover a path from i to k ?
- $\text{visited}(k)$ was set to True when exploring some vertex j
- Record $\text{parent}(k) = j$
- From k , follow parent links to trace back a path to i

```
def BFSListPath(AList, v):
    (visited, parent) = ({}, {})
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    q = Queue()
    visited[v] = True
    q.addq(v)
    while(not q.isEmpty()):
        j = q.deq()
        for k in AList[j]:
            if (not visited[k]):
                visited[k] = True
                parent[k] = j
                q.addq(k)
    return visited, parent
```

0 - n-1



So now, let us see what more we can do with BFS. So, with BFS so far, we have just said that we can do reachability. But one of the things that we were interested in terms of reachability was how to get from one place to another. So, it would be nice if we could also record the path. So, how do we recover a path from i to k ?

So, in order to recover a path, we need to, we know that there is a path because the visited value of k is true. But when visited value of k was set to true, it was set to true from some previous thing. It was set to true because there was an edge j comma k . So therefore, the step to reach k , last pass through j .

So, I can now record that k is the parent of j . Now from k , I can follow parent links back to trace back to path to i . So, if I, every time I add an edge to the visited thing, if I record from where I added that edge, then I can follow these links back. So, I keep track of one extra dictionary called parent. So, here is a version of breadth first search which uses adjacency lists and maintains this parent information.

So as before, we keep this visited dictionary and now we have a new dictionary called parent which are both initialized to empty. Now, when we did an adjacency matrix, we had to basically initialize the visited thing from based on the size of the matrix. So, we looked at the shape that is here, I can just take the keys of this. So basically, it is the same thing, but I ran through a loop from 0 to n minus 1 and I say that all the vertices are initially not visited and all of them have no calculated parents.

So, remember that the vertices run from 0 to n minus 1. So, minus 1 cannot be the parent because there is no vertex called minus 1 in my thing. So, it is safe to use minus 1 as a invalid

value to say that parent for this vertex has not been calculated yet. Now, this is BFS so I set up an empty queue, I mark the initial vertex to be visited. I add it to the queue. Now the initial vertex is where I start from.

So, it was not visited from anywhere it was given to me to start from there. So, I will not mark its parent to be anything. So, I will leave its parent as minus 1 for now. We know That is if something is so it will be the only vertex that will be visited, which was not visited from somewhere. So, if something is visited, but its parent is minus 1, I know that is where I started. For every other vertex that I visit, I will set its parent as we will see.

So, the rest of the thing is the same as the earlier BFS. So, while the q is not empty, I take out the head of the q. And now I am using an adjacency list. So, I do not have to go through all the rows in the matrix, I can just pick up this list of neighbours of j. And for every such k, which is a neighbour of j, if it has not been visited, I set its visited status to true, I add it to the q. And now here is the new thing that I do. I set its parent to be j because j is the reason why k got added to this list.

So, if I visited k, by following an edge from j to k, then I set the parent of k to be j. And now I want to keep track of both of these quantities. So, I return not just which ones are visited, but I also return this parent dictionary, which tells me if I say, you claim that vertex 7 was visited, or vertex 9 was visited, how did you get there? So, I will say parent, 9 was something the parent of that do something, and so on. So, I will kind of trace back apart from the target back to the source.

(Refer Slide Time: 26:21)

BFS from vertex 7 with parent information

| | Visited | Parent |
|---|---------|--------|
| 0 | True | 4 |
| 1 | False | -1 |
| 2 | False | -1 |
| 3 | True | 4 |
| 4 | True | 7 |
| 5 | True | 7 |
| 6 | False | -1 |
| 7 | True | -1 |
| 8 | True | 7 |
| 9 | False | -1 |

| To explore queue | | | | | |
|------------------|---|---|---|--|--|
| 5 | 8 | 0 | 3 | | |

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}

Mathavan Mukund

Breadth First Search

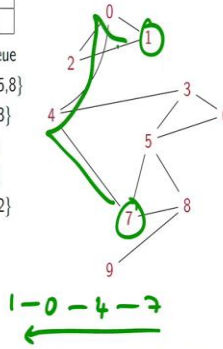
BFS from vertex 7 with parent information



| | Visited | Parent |
|---|---------|--------|
| 0 | True | 4 |
| 1 | True | 0 |
| 2 | True | 0 |
| 3 | True | 4 |
| 4 | True | 7 |
| 5 | True | 7 |
| 6 | True | 5 |
| 7 | True | -1 |
| 8 | True | 7 |
| 9 | True | 8 |

| To explore queue |
|------------------|
| |

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



Madhavan Mukund

Breadth First Search

DDCA

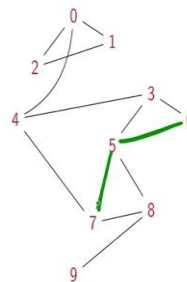
BFS from vertex 7 with parent information



| | Visited | Parent |
|---|---------|--------|
| 0 | True | 4 |
| 1 | True | 0 |
| 2 | True | 0 |
| 3 | True | 4 |
| 4 | True | 7 |
| 5 | True | 7 |
| 6 | True | 5 |
| 7 | True | -1 |
| 8 | True | 7 |
| 9 | True | 8 |

| To explore queue |
|------------------|
| |

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



Path from 7 to 6 is
7-5-6



Madhavan Mukund

Breadth First Search

DDCA

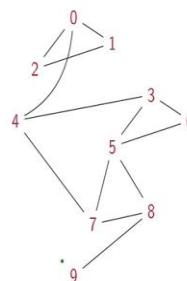
BFS from vertex 7 with parent information



| | Visited | Parent |
|---|---------|--------|
| 0 | True | 4 |
| 1 | True | 0 |
| 2 | True | 0 |
| 3 | True | 4 |
| 4 | True | 7 |
| 5 | True | 7 |
| 6 | True | 5 |
| 7 | True | -1 |
| 8 | True | 7 |
| 9 | True | 8 |

| To explore queue |
|------------------|
| |

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



Path from 7 to 2 is
7-4-0-2



Madhavan Mukund

Breadth First Search

DDCA

So, here is the same BFS we did earlier, but keeping track of the parent information, so we want to start from 7. So, this is where we are starting our BFS. So initially, all parents are minus 1, all vertices are false. So, now when we initialize, we mark 7 to be true, we do not change its parent, but we put it in the queue.

Now, each time we now pull out something from the queue and process it, we not only mark visited thing, we also mark parent. So, we pull out 7, we mark 4, 5 and 8 as before, but the important thing is that we have now put the parent of 4, 5 and 8 to be 7 saying that I reached these vertices from 7, the last thing I did to reach them with 7. Similarly, I pick out 4, and I put 3 back and 0 back.

And for each of them, I set the parent to be 4 so I not just in addition to making them true and putting them in the queue, I also set the parent vertex. So, now from 5, I get to 6. So, I say the parent of 6 is 5. From 8, I get to 9. So, I say the parent of 9 is 8. So, that is here. Then from 0, I get to 1 and 2. So, I say the parents of both 1 and 2 are both 0. So, that is here. And then after this, nothing new happens. And so I come out with this.

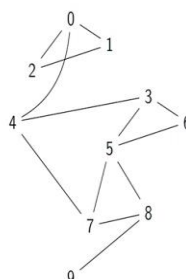
So, now I have the fact that everything is visited. But if I asked, for instance, how do I get from 7 to say, how did I get from 7 to say 1? So, then it says I got from 1, I came back from 0, and 0, I came back from 4, and 4, I came back from 7. So, this is my path. 7, 4, 0, 1. So 7, 4, 0, 1 is my path. So, this is how I do it. So, set the path from 7 to 6, for instance, 7, 5, 6. Before I start from 6, it says 6 came from 5, and 5 says 5 came from 7. Similarly, if I say, 2, it says 7, 4, 0, 2. So, this is how I use this.

(Refer Slide Time: 28:18)

Enhancing BFS to record distance



- BFS explores neighbours level by level
- By recording the level at which a vertex is visited, we get its distance from the source vertex



Navigation icons: back, forward, search, etc.



So, this is one thing I can do, which is recover the parent information, I can also recover how far it is? So, of course, if I trace out the path, then I get automatically how far it is, because that is telling me how many steps it took to reach there. But I can also keep this information directly. So, since I am exploring level by level, I know that my first neighbours were one step away, the neighbours of those neighbours are two steps away, and so on.

So, each time I explore a vertex, I can store how many steps away it is. And with this, I can keep track of the shortest number of steps, I need to get to every vertex. So, I can keep the level and if I keep the level and the level is not 0, then it is visited. If it is not, if it is set, then it is not visited. So, I do not need to keep visited and level I can keep just the level.

(Refer Slide Time: 29:04)

Enhancing BFS to record distance

- BFS explores neighbours level by level
- By recording the level at which a vertex is visited, we get its distance from the source vertex
- Instead of `visited(j)`, maintain `level(j)`

```
def BFSListPathLevel(AList,v):
    (level,parent) = ({},{})
    for i in AList.keys():
        level[i] = -1
        parent[i] = -1
    q = Queue()

    level[v] = 0
    q.addq(v)

    while(not q.isempty()):
        j = q.delq()
        for k in AList[j]:
            if (level[k] == -1):
                level[k] = level[j]+1
                parent[k] = j
                q.addq(k)

    return (level,parent)
```

Madhavan Mukund | Breadth First Search

Enhancing BFS to record distance

- BFS explores neighbours level by level
- By recording the level at which a vertex is visited, we get its distance from the source vertex
- Instead of `visited(j)`, maintain `level(j)`
- Initialize `level(j) = -1` for all `j`
- Set `level(i) = 0` for source vertex
- If we visit `k` from `j`, set `level(k)` to `level(j) + 1`
- `level(j)` is the length of the shortest path from the source vertex, in number of edges

```
def BFSListPathLevel(AList,v):
    (level,parent) = ({},{})
    for i in AList.keys():
        level[i] = -1
        parent[i] = -1
    q = Queue()

    level[v] = 0
    q.addq(v)

    while(not q.isempty()):
        j = q.delq()
        for k in AList[j]:
            if (level[k] == -1):
                level[k] = level[j]+1
                parent[k] = j
                q.addq(k)

    return (level,parent)
```

Madhavan Mukund | Breadth First Search

So, instead of visited j , I maintain level of j . So, here I change the terminology. So, it is the same BFS, but I am using level and in i . In earlier, we initialize the visited to be false. But now level is going to be a number. It is how many steps I took. So, I am going to say that level is now a number. And if it is minus 1, it means that this vertex has not been visited. So, initially, no vertex is visited, same as visited false level is minus 1. And all vertices have no parents, or parents are minus 1.

So, when I now start with V , the first thing I do instead of setting visited v to true is I set the level of V to 0. So, I reached V , I started at V . So, in 0 steps, I reached v . And then I do the usual thing, I put it into the queue and I process the queue. So, while processing the queue, as before I pick out the first element in the queue, I look at all its neighbours. And now instead of checking whether the vertex has been visited, which is what I was doing before, I will check whether the level has been assigned.

So, if the level has not been assigned, then it means it has not been visited. So, if the level has not been assigned the level still minus 1. So, if the level is minus 1, then I have to set the level. So, what do I set the level to? Well, I have to set the level to be 1 more than the level from where I am looking at it. So, if I am coming from j to k , and if j was at level 7, for example, then the new vertex k must be at level 8. So, I set parent of k to j and I set level of k to be level of j plus 1. So, it is 1 more than where I came from.

And at the end, now, I will return level and parent. In level if I find entries minus 1 it is false. If it is not minus 1, it is true. So, I implicitly have the visited information given back to me. And in addition, of course, I have the actual path which is given by parent. So, notice that from the parent thing, I can actually go backwards and calculate the level. But this has now given me the level in one step. So, this is what we said we initialize the level set it to 0 for the source vertex. And every time we visit a new vertex k , we increment the level compared to where we came from.

(Refer Slide Time: 31:14)

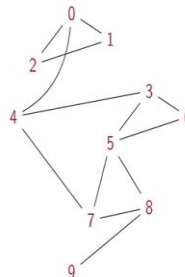
BFS from vertex 7 with parent and distance information



| | Level | Parent |
|---|-------|--------|
| 0 | 2 | 4 |
| 1 | 3 | 0 |
| 2 | 3 | 0 |
| 3 | 2 | 4 |
| 4 | 1 | 7 |
| 5 | 1 | 7 |
| 6 | 2 | 5 |
| 7 | 0 | -1 |
| 8 | 1 | 7 |
| 9 | 2 | 8 |

| To explore queue |
|------------------|
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



Navigation icons: back, forward, search, etc.



Madhavan Mukund

Breadth First Search

So, just our same example as before, so we start at 7, initially, all levels and all parents are minus 1, when I start with 7, now, I set this to be 0, so this is my initial thing. And now as I go ahead, I mark these to be 1, so this is now 1, because these were marked from 7, then if I go from 4, then I mark the level of 3 and 0 to be 2, because they were reachable from 4, which was level 1. And then if I go from 5 to 6, again, 5 was already at level 1, so 6 is at level 2.

And now if I go to 8, for instance, 8 was at level 1, so 9 is at level 2. And now I get to level 3, because 0 is at level 2. And if I explore the neighbours of 0, then its neighbours will be at level 3. So, 1 and 2, which are going to be explored next will now be at level 3. So, it means it takes me 3 steps to get to 1 and 2. And now you can see on the left hand side that all the entries have become different from minus 1. So, I have actually reached everything. So, we are as usual just going to process the queue and find no changes.

सिद्धिर्भवति कर्मजा

(Refer Slide Time: 32:26)

Summary



- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m+n)$ using adjacency list
- Add parent information to recover the path to each reachable vertex
- Maintain level information to record length of the shortest path, in terms of number of edges
 - In general, edges are labelled with a **cost** (distance, time, ticket price, ...)
 - Will look at **weighted graphs**, where shortest paths are in terms of cost, not number of edges

Navigation icons: back, forward, search, etc.



Madhavan Mukund

Breadth First Search

So, to summarize, breadth first search is a systematic way to explore a graph. And essentially, we record which vertices have been visited and we use a queue to keep track of which vertices are yet to be explored. If we use an adjacency matrix, then we end up using n^2 time regardless of how many edges that are in the graph. So, this is not optimal, because very often, you will have a smaller number of edges.

So, we use adjacency lists to get m plus n because the work across all the vertices turns out to be proportional to the sum of the degrees which is bounded by $2m$. We saw that we can add parent information to recover the path and we can maintain the level information to record the length of the shortest path in terms of number of edges. Now, the shortest path in terms of number of edges may not be the actual shortest path we are interested in as we will see later on.

S

o

,

l

a

t

e

r

o

n

,