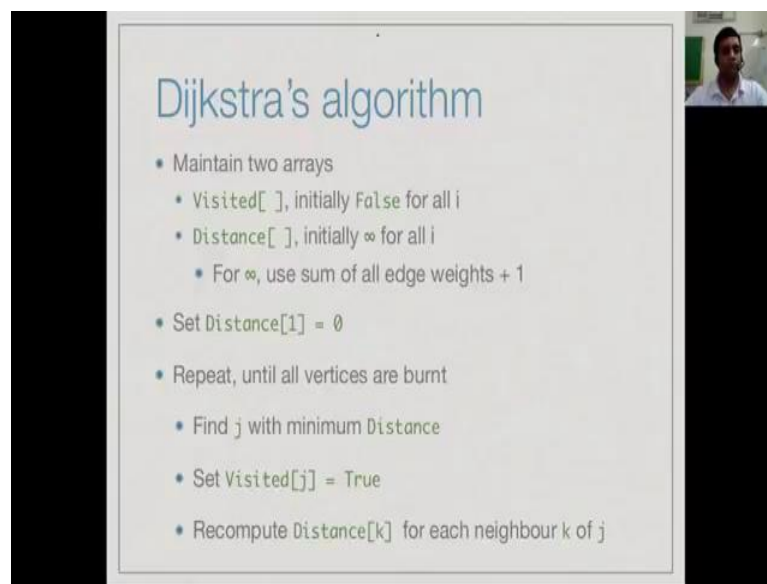


Week- 04
Module - 02
Lecture - 26

So, now let us analyze Dijkstra's algorithm for the single source shortest path problem.

(Refer Slide Time: 00:06)

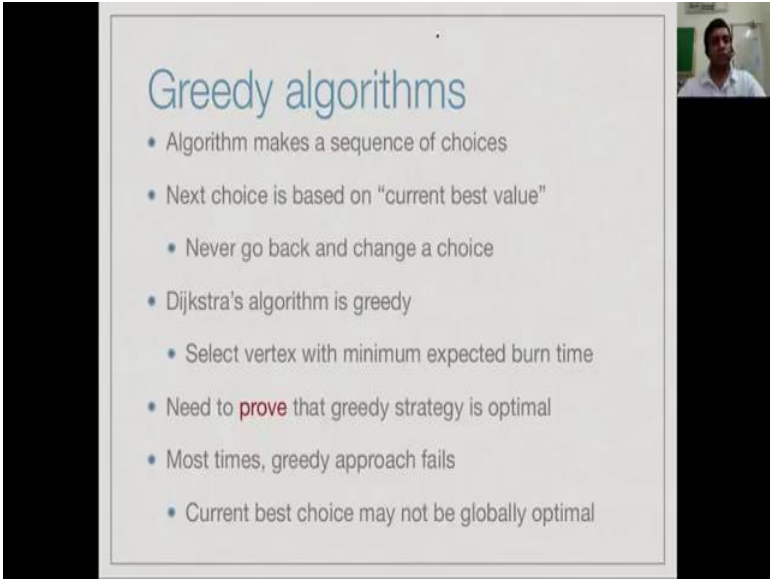


Dijkstra's algorithm

- Maintain two arrays
 - Visited[], initially False for all i
 - Distance[], initially ∞ for all i
 - For ∞ , use sum of all edge weights + 1
- Set Distance[1] = 0
- Repeat, until all vertices are burnt
 - Find j with minimum Distance
 - Set Visited[j] = True
 - Recompute Distance[k] for each neighbour k of j

So, recall that Dijkstra's algorithms operate by burning vertices in the analogy we used. So, we keep track of vertices which have burnt or ((Refer Time: 00:17)) initially nothing is visited and initially we do not know any distance to any vertex. So, we have assumed all distances that at infinity. When we start at the source vertex, let us call it 1 by assumption. So, we set its distance to 0. Then, we repeatedly pick up the first vertex which is not burnt and which has the minimum distance among those which are not burnt, visited and recomputed the distance to all its neighbors.

(Refer Slide Time: 00:45)



Greedy algorithms

- Algorithm makes a sequence of choices
- Next choice is based on "current best value"
 - Never go back and change a choice
- Dijkstra's algorithm is greedy
 - Select vertex with minimum expected burn time
- Need to **prove** that greedy strategy is optimal
- Most times, greedy approach fails
 - Current best choice may not be globally optimal

So, before we look at the complexity of these algorithms, we actually first have to verify that it is correct. So, Dijkstra's algorithms now make these sequences of updates. It is a bit analysis to bfs and dfs, because it keep visit in vertices and it visits every vertex only once. Now, its visits vertices in a particular order which is different from breadth first or the depth first, and we need to justify that this order is actually correct. So, in some sense at every point, breadth first search looks all its neighbors and visits them say in the order of their vertex number. Depth first search will pick the first neighbor, and then explore it further.

Now, Dijkstra algorithm uses the different strategy which is to pick the first, the smallest distance and visited neighbor. So, we have to justify that this choice which is never under, that is which keep going forward, and you never go back and say hope maybe I should have chose a different one that this kind of strategy actually is correct. So, this is a general class of algorithms called greedy algorithms where you have a number of possible trajectories of parts that you can choose to solve the problem. At each stage you make the next choice based on some local information. At this point, this looks like the best choice to make, and then somehow magically this best choice that you make on local information turns out to be globally the best trajectory to take. So, for such greedy algorithm, it is important to establish that this local choice of the next step actually gives

us a global optimum, because very often this kind of a local juristic does not give us a correct value.

(Refer Slide Time: 02:20)

Correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have identified shortest paths to all vertices already burnt

Diagram illustrating the correctness of Dijkstra's algorithm. A set of vertices is shown, with some labeled 'Burnt vertices' (s, x, y) and others (v, w) not yet burnt. A path is shown from s to x to v. Handwritten notes include 'Invariant' and 'Correctly solved' in red, and the equations $d(x) + w(x, v)$ and $d(y) + w(y, w)$ in green and purple respectively. A comparison shows $d(x) + w(x, v) < d(y) + w(y, w)$.

- Next vertex to burn is v, via x
- Cannot later find a shorter path from y to w to v

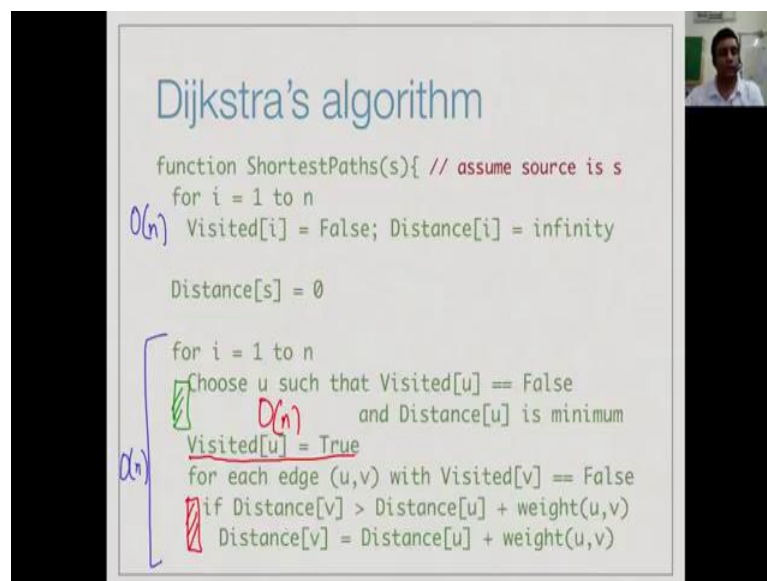
So, let us first look at the correctness of Dijkstra algorithm. So, the key to establishing correctness in this particular case is which establishes what is called an invariant, right. So, now, the Dijkstra algorithm proceeds as you saw in a sequence of iteration. They are n iteration. The invariant is that at each iteration we have this correctness of vertices as burnt and unburnt. What you want to claim is that the burnt vertices are correctly solved, that is at any point if we look at the distances assigned to the vertices that the burnt set, then those distances are actually the shortest distance in the source vertices to that burnt vertex. So, if we assume that this inductive invariant is true, now it is certainly true at the beginning because at the beginning, the only vertex that we have burnt is the start vertex s, right. So, the first vertex we burnt is s and we said it is distance to 0. So, certainly at that point this property is true that among the burnt vertices, the distances or in fact the shortest distances.

Now, assuming that we have extended it say for a few vertices, now we want to add a new one. So, what we said is that we will pick a vertex v, such that if we look at the distance to x plus the weight of x v, if you look at this total sum which will be updated

distance to v , this is smallest among all the vertices which are not burnt. So, the claim is that if we now add this to our burnt set, right so we extended our burnt set like this by include v , then, the distance of v cannot be actually smaller than what we have computed now and we could not change it at later update. So, how did you change it because at the later update, it could be that some later point we actually extend a burnt set and included a new one. Now, can it be that there was a path from y to w to v which is shorter than a path we claimed just now via x to v . So, it is easy to see that the distance to y plus the cost of wxy . Now, this must be bigger than or equal to the earlier thing because we choose v and not w because at this point w was not a smallest one, v was. Perhaps they were equal, but certainly w was not less than v .

So, at a later stage if you go to w , we know that we will take at least that much time to reach w . Then, we have to additionally incur a cost of going from w to v . So, there is no way that a later path if we can discover that y to w to v can be better than the correct path next, right. So, this reestablishing the invariant that the vertex we burnt next is correctly solved. So, this is a way to show that Dijkstra greedy strategy actually solves this problem correctly.

(Refer Slide Time: 05:01)



Dijkstra's algorithm

```

function ShortestPaths(s){ // assume source is s
  for i = 1 to n
    Visited[i] = False; Distance[i] = infinity
    Distance[s] = 0

  for i = 1 to n
    [Choose u such that Visited[u] == False
     and Distance[u] is minimum]
    Visited[u] = True
    for each edge (u,v) with Visited[v] == False
      if Distance[v] > Distance[u] + weight(u,v)
        Distance[v] = Distance[u] + weight(u,v)
  
```

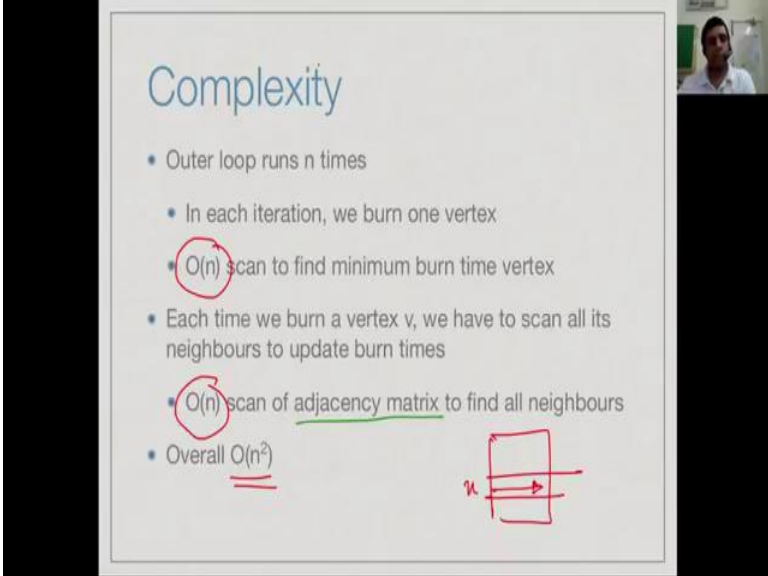
Handwritten annotations:

- $O(n)$ next to the first loop.
- $O(n^2)$ next to the selection of u .
- A bracketed $O(n^2)$ covering the entire algorithm.

So, now having established that it is correct, let us look at the complexity. So, there are

some obvious loops in this. So, there is a first, then there is an order n loop of initializing the visited values to false and instances to infinity. Now, there is another order n loop here and inside this order n loop, there are two loops. One is more obvious than the other. So, here once we have visited about x and new vertex, then we have to go through all its outgoing neighbors, right. There is a scan of all edges which are going out of q . So, that is one loop, but also there is a loop here which is kind of implicit which is before we visit a vertex, we have to choose it, right. So, we have to go through all the unvisited vertices and pick the one whose distance is minimum. So, this in general we have seen that if we have a list or an array which is not in any particular order, we have to find the minimum and scan the entire array. So, this is implicitly an order n square. So, this is an order n step and that time it takes here, it depends on how we represent the edge.

(Refer Time Slide: 06:10)



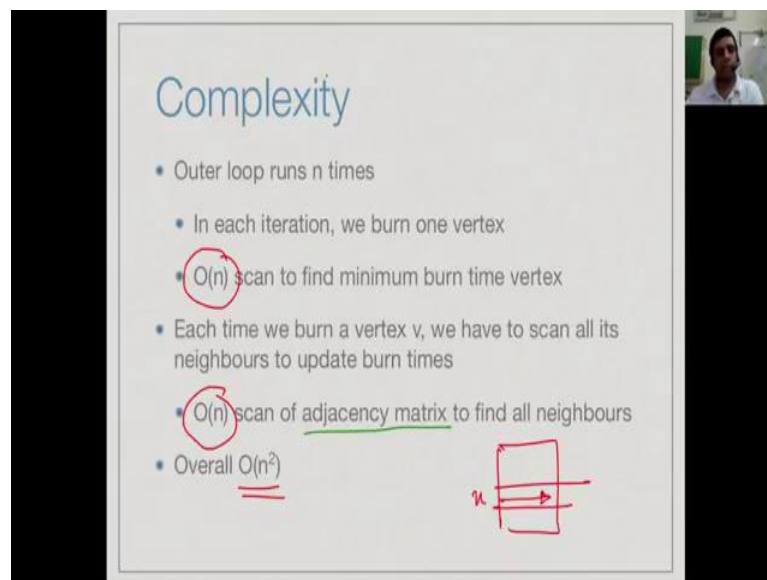
Complexity

- Outer loop runs n times
- In each iteration, we burn one vertex
- $O(n)$ scan to find minimum burn time vertex
- Each time we burn a vertex v , we have to scan all its neighbours to update burn times
- $O(n)$ scan of adjacency matrix to find all neighbours
- Overall $O(n^2)$

So, the outer loop run time we saw. In each iteration, we burnt a vertex. This requires an order n scan to find the minimum vertex to burn. After we burnt the vertex, we have to scan its neighbors to update the burnt times of those vertices. Now, if we have an adjacency matrix as we have seen, then you are going to burn u now, then we have find its new neighbor now. We have to scan the row for u in the adjacency matrix. So, this will take order n time, right. So, we are in outer order n and inside we have two order n things independent of each other. One is to find the minimum burnt vertex, unburnt

vertex to burn next, and the second is to update all its neighbors, right. So, because we have this order n thing inside a order n loop, overall its order n square. So, now one of the bottom x certainly is this adjacency matrix, right.

(Refer Slide Time: 07:00)



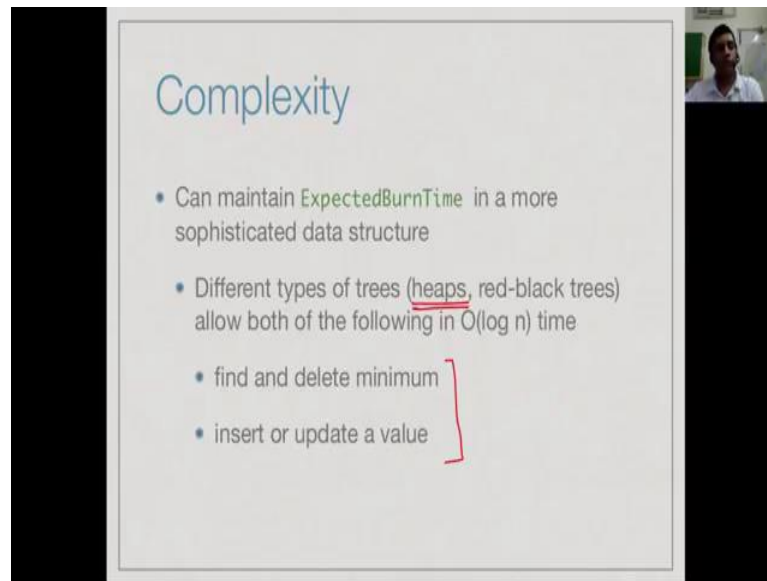
Complexity

- Outer loop runs n times
 - In each iteration, we burn one vertex
 - $O(n)$ scan to find minimum burn time vertex
 - Each time we burn a vertex v , we have to scan all its neighbours to update burn times
 - $O(n)$ scan of adjacency matrix to find all neighbours
- Overall $O(n^2)$

n

So, we have seen in bfs and dfs that if we move from an adjacency matrix, when adjacency list scanning out the vertices of the particular all the neighbors of the particular vertex becomes more efficient. So, we can do the same thing here, right. So, if we move from adjacency matrix, from adjacency list representation, certainly the second order n loop within each iterations can be now counted as an overall order n cost because across all the iterations, we would explore every edge only once because we explore when we burn its source vertex, right. So, therefore, the second contribution of order n gets solved by using an adjacency list, but unfortunately we still have the first order n step which is to find the minimum. So, we have a list of burnt time associated with the unburn unvisited vertices, and we have to find the minimum among them and this remains the order n step. So, overall though we have moved to an adjacency list, this alone does not help us because we still have an order n step inside the big loop. So, we are still at order n square.

(Refer Time Slide: 08:05)



The slide is titled "Complexity" in a blue font. It contains a list of bullet points. The first bullet point is "Can maintain ExpectedBurnTime in a more sophisticated data structure". The second bullet point is "Different types of trees (heaps, red-black trees) allow both of the following in $O(\log n)$ time". This second bullet point is followed by two sub-bullets: "find and delete minimum" and "insert or update a value". A red bracket is drawn to the right of these two sub-bullets, grouping them together. In the top right corner of the slide, there is a small video inset showing a person's face.

- Can maintain ExpectedBurnTime in a more sophisticated data structure
- Different types of trees (heaps, red-black trees) allow both of the following in $O(\log n)$ time
 - find and delete minimum
 - insert or update a value

So, activate to get around this bottle neck, we need to maintain the burnt times in a most sophisticated data structure. So, it turns out as we need data structure in which we can of course find and remove the minimum element quickly, but once we have that, we also need to be able to update the values quickly, so that overall both updating and extracting are roughly equally the same. So, this can be done using tree like structure in particular we will see in a later lecture that there is a nice data structure called heap which precisely allows us to do these two operations in $\log n$ time. So, if you have n values, then $\log n$ time you can find and read the minimum value and we can insert a new value or update a value which is already in a heap. All these operations take only $\log n$ time.

(Refer Slide Time: 08:58)

Complexity

- With such a tree
 - Finding minimum burn time vertex takes $O(\log n)$
- With adjacency list, updating burn times take $O(\log n)$ each, total $O(m)$ edges
- Overall $O(n \log n + m \log n) = O((n+m) \log n)$

Handwritten annotations on the slide:

- A blue arrow points from the first bullet point to the text "choosing min vertex".
- A blue arrow points from the second bullet point to the text "Updating distance".
- A red arrow points from the overall complexity formula to the text $O(n^2)$.

So, if this which we will see later and if this can be done, then what it says is that finding the minimum burn time takes $\log n$ time. Now, when we update the vertices which are adjacent to the current burn vertex, overall we do this $O m$ times, right, because of $O m$ edges, but each updates again takes $O \log n$. So, we have now two contributions to our complexity. So, this comes from choosing the minimum vertex, right. In order to extract the minimum vertex, we do this n times and each time it process $\log n$ time, and then in order to update the distances once we burnt a vertex, each time it takes the $\log n$ time, but we do this for each edge. So, it is $n \log n$. So, we have $n \log n$ for choosing a vertex and this is for updating a distance, right. So, overall are the algorithm becomes n plus $m \log n$. So, remember that in a graph n plus m is really a fact is the size of the graph. So, this is really a $n \log n$ algorithm as a post to the $(()) O n$ square, and we are seeing sorting and other such problems. That is a huge practical jump going from $O n$ square to $n \log n$. It is a huge jump in term of the size of the problem we can hope to solve.

(Refer Slide Time: 10:13)

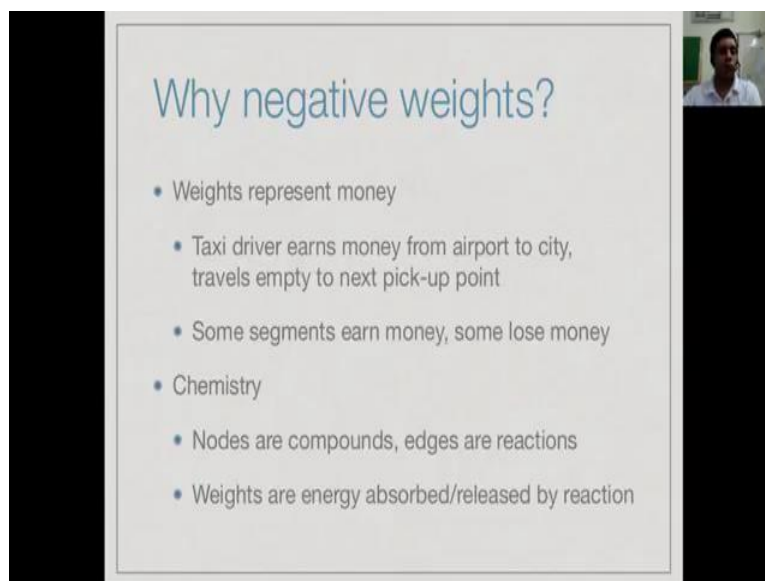
Limitations

- What if edge weights can be negative?
- Our correctness argument is no longer valid

- Next vertex to burn is v, via x
- Might find a shorter path later with negative weights from y to w to v

So, Dijkstra algorithm makes a very crucial assumption which underlies the correctness of its greedy step, and that is that there is no negative cost associated to the nature, right. Our argument was that if we choose v as the next neighbor to add to our burn set, then we can never find the shorter path coming this way, but obviously we have negative edges, then we could have a situation where had say a path of length to here, right and then I chose not to follow another path because maybe it is have of length 4, but if I have come back from there, right maybe this has length minus 3. So, if I come around this way, then I incur net cost of plus 1 whereas if I go this way, I get plus 2, right. So, therefore, by locally looking for a nearest neighbor, I do not necessarily find the shortest distance.

(Refer Slide Time: 11:15)



Why negative weights?

- Weights represent money
 - Taxi driver earns money from airport to city, travels empty to next pick-up point
 - Some segments earn money, some lose money
- Chemistry
 - Nodes are compounds, edges are reactions
 - Weights are energy absorbed/released by reaction

So, now of course you might want to ask why you want to put a negative ways in a graph at all. Well, remember that the graphs are very general model. There are many situations where you can actually interpret negative ways in a sensible way. So, supposing we are looking at say at a taxi driver and we are looking at the graph as a list of places, where we picks up and drop off people. Now, obviously there are segments where that the passenger is in the car and there are segments where we may not have to be may not have a passenger, he may have been returning to a place to pick up. For instance, a taxi driver operation there from the airport might typically go back to the airport after a free trip because he expects to find longer trips from there can within the city, right. So, there are segments where he travels empty, there are segments where he has passengers. So, some segments earn money, some segments use money. So, there are some positive edges and there are some negative edges, and the other situation is completely unrelated to anything we have seen so far.

You can think of say chemicals, chemical compounds and we can represent the graph saying how we transform one compound to other compound, and here for instance age weight can represent the energy which is either released or absorbed in this process. So, again it could be positive or negative, right. So, there are many situations in which edge, negative edge which actually makes sense. So, what do we do here negative edge

weights?

(Refer Slide Time: 12:32)

The slide is titled "Handling negative edges" in blue text. To the right of the title is a diagram of a cycle with three nodes. The edges are labeled with weights: -3, -2, and +1. A purple arrow indicates a clockwise path around the cycle, and a bracket next to it is labeled -4, representing the total weight of the cycle. Below the title is a list of bullet points:

- **Negative cycle:** loop with a negative total weight
- Problem is not well defined with negative cycles
- Repeatedly traversing cycle pushes down cost without a bound
- With negative edges, but no negative cycles, other algorithms exist (will see later)
- Bellman-Ford
- Floyd-Warshall all pairs shortest path

So, first thing to notice that if you have a negative cycle, so if I have a graph which I have say a loop like this and all of the things add minus 3, minus 2 and plus 1, then if I go around the cycle once, right and come back, then I will incur a total cost of minus 4. So, this means that if I had a path which started somewhere else, came here and left the cycle somewhere else. Suppose we had a source and a target, then I can make the distance in a source to the target arbitrarily small by going round and round this loop many number of times, right. At each time I go round the loop, the cost reduces by minus 4. So, I keep adding minus 4 to my cost. So, the cost will be as low as I want just depending on how many times I go around you.

So, if I have negative cycles in a graph, the question of the shortest path does not even make a sense. There is no notion of a shortest path, because the quantity is not well defined, but it turns out that if I rule out this cycle, it could have negative edges, but not negative cycle. So, for instance here if instead of plus 1, I said that this was plus 7, right. If I have made this way plus 7 and going around the cycle will cause me plus 2. So, therefore, it does not help me to go around the cycle because it only adds to my cost. Therefore, in a situation where I have no negative cycle, but I do have negative edges, it

still makes sense to talk of shortest path, and we will see later that there are other algorithms, other than Dijkstra's algorithm which can handle these. In particular we will look at Bellman-Ford algorithm, and we will also see that all pair shortest path problems which generalize the single source short path problem called the Floyd Marshall algorithm. We will also solve these things regardless of whether the paths are negative or the weights are negative or not, provided there are no negative cycles.