

**Programming, Data Structures and Algorithms**  
**Prof. N. S. Narayanaswamy**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Module – 11**

**Lecture – 58**

**Problem: single source shortest path**

**Assumptions on graph, weights**

**Data structure: Priority queue and operations**

**Distance array**

**Algorithm and example**

**Correctness argument**

In today's algorithms lecture, we are going to study the Dykestra's algorithm. Let us start off with a pronunciation of the inventor of this algorithm, his name is Dyke Stra and you can, his name is Dyke Stra, think of it as two parts Dyke and Stra that is how the name is pronounced.

(Refer Slide Time: 00:18)

### Dijkstra's (Dyke-Stra's) Algorithm

**Problem :** The single source shortest path problem in a graph  $G=(V, E)$

finding shortest paths from a single designated source vertex to every other vertex in  $G$ .

**Input:** Weighted graph  $G=(V, E)$  with real valued edge weights, a source vertex  $s$ .

**Output:** Shortest paths form source  $s \in V$  to every vertex  $v \in V$ .

**Algorithmic approach:** Greedy Algorithm

Dykestra's Algorithm is used to solve the single source shortest path problem in a given graph  $G$ , the vertices of  $G$  are  $V$  and the edges are  $E$ . The goal is to find the shortest path from a single designated source vertex to every other vertex in  $G$ . This is the definition of the problem. The input to this problem is a weighted graph  $G$  where every edge has a real valued weight and a special source vertex denoted by  $s$ . The output of the algorithm should consist of shortest path from the given source vertex  $s$  to every other vertex  $v$  in

the graph. Dykestra's algorithm is an algorithm that uses the greedy strategy and we will see how it uses the greedy strategy.

(Refer Slide Time: 01:39)

## Some Assumptions

- Graph may be directed or undirected.
- All edge weights must be **non-negative**.
- **Data structure used: Priority Queue Q**
  - Will return the least value element from the Queue and update Q in  $\log(n)$  time – **DeleteMin function**
  - Will update the priority of an element and maintain a priority queue in  $\log(n)$  time - **Decrease**
  - The initial priority queue of  $n$  elements can be constructed in time  $O(n)$  time - **Insert**

We make some assumptions about the graph and we also make some assumptions about the data structure which is used by the algorithm. First of all, the graph may be either a directed graph or an undirected graph. We also assume that all the edge weights are non-negative. We will see in the analysis of algorithm that this is very crucial to guarantee the correctness of the algorithm. We will also assume that the algorithm has access to a data structure which is a priority queue, we will use the letter Q to denote the priority queue.

A priority queue of  $n$  elements has the following features which we will assume exists, we will not worry about the implementation of the priority queue, when we design and study the Dykestra's algorithm. We will assume that the priority queue has a DeleteMin function which will return the least valued element from the queue in order of  $\log n$  time. Not only will it return, it will also remove that element from the queue. The second operation that we assume comes with a priority queue is the operation of decreasing the priority of a specific element in the queue.

We assume that this can also be done in  $\log n$  time. The initial priority queue of the  $n$  elements can be constructed using an insert function in order of  $n$  time, recall that  $n$  is the total number of elements in the queue. As an additional point, the interested student may refer to the data structure called heap or a min heap to understand the implementation of

a priority queue. However, that is not an important for us to analyse the Dykestra's algorithm.

(Refer Slide Time: 04:04)

## Preliminaries

- For each vertex  $v \in V$  maintain the  $\text{dist}[v]$ 
  - ▣ Always represents an upper bound on the weight of the shortest path from  $s$  to  $v$
  - ▣  $\text{dist}[s]=0$
  - ▣ For each  $v \in V$   $\text{dist}[v]=\infty$
- For each vertex  $v \in V$  maintain the  $\text{pred}[v]$ 
  - ▣ Denotes the predecessor of  $v$  in the path from  $s$ .
  - ▣  $\text{pred}[v]$  is either another vertex or NIL.

Here, at the data structures that are maintained by the algorithm, we assumed that the algorithm has access to an array called  $\text{dist}$ , short form for distance and for every vertex  $v$ , the element to the array  $\text{dist}$  index by  $v$  that is  $\text{dist}[v]$  contains an upper bound on the length of the shortest path from  $s$  to  $v$ . This is an invariant that the algorithm maintains repeatedly. The distance of  $s$  from itself is equal to zero and initially the distance of every vertex from the source vertex  $s$  is taken to be infinity in the description of the algorithm.

For each vertex  $v$ , we also maintain the predecessor on the shortest path from  $s$ . This is maintained in an array called  $\text{pred}$  which is a short form for predecessor and initially the predecessor over every vertex is chosen to be nil. Except for the vertex  $s$  which will be its own predecessor.

(Refer Slide Time: 05:26)

```
Algorithm
Algorithm Dijkstra(G,s)
{
//Initialize vertex priority queue to empty
Initialize(Q)
for every v ∈ V do
    dist[v] ← ∞
    pred[v] ← NIL
//initialize vertex priority in Q
    Insert(Q, v, dist[v])
dist[s] ← 0
pred[s] ← s
//update priority of s with dist[s]=0
Decrease(Q, s, dist[s])
//vertices for which shortest path from s
//has been computed
Vt ← ∅
while Q ≠ ∅
{
//Delete from Q element with minimum value
u ← DeleteMin(Q)
Vt ← Vt ∪ {u}
//update distance upper bounds from s
for every v ∈ V - Vt such that (u,v) ∈ E(G)
    if dist[u]+w(u,v) < dist[v]
        dist[v] ← dist[u]+w(u,v)
        pred[v] ← u
//update priorities in Q
        Decrease(Q, v, dist[v])
}
```

Here, is the description of the algorithm and all the statements which you see in red color are commands and these are not executed. All the statements that you see in red are commands and these are not executed. These are to help the reader, understand the operations performed in the algorithm. On this slide, the algorithm is described in two parts, the left part of the side is a starting point of the algorithm and what is on the right part of the side is a continuation of the left hand side part of the code.

In other words, this slide is essentially the whole description of the algorithm. In other words, this slide contains the description of the whole algorithm. It is logically split in to the two parts, the first part can be imagine that is the initialization phase for the algorithm to run and the second part of the slide, that is the right hand side part of the slide consists of a computation of the shortest path from the source vertex  $s$ .

So, let us focus on the left hand side part. The first step is to initialize the priority queue. It is initialized to empty and then the elements and then for every vertex  $v$  in the graph, the array distance is organized as a priority queue. For every vertex  $v$ , the distance is set to be infinity from  $s$  and the predecessor of  $v$  is initialized to the value NIL and now we initialize the priority queue. And to do this, we use the distance value as the value that is present in the queue.

The distance of  $s$  to itself is then set to 0 and the predecessor of  $s$  is taken to be the vertex  $S$  itself. After doing this, we decrease the priority of the vertex  $s$  to 0 using the decrease

function. Let us just look at this function call. It is to decrease the priority of the vertex  $s$  to be the value, distance of  $s$  in the queue  $Q$ . After the initialization phase, we move to the right hand side part of this slide and now we compute iteratively the distance of the each vertex from  $s$ .

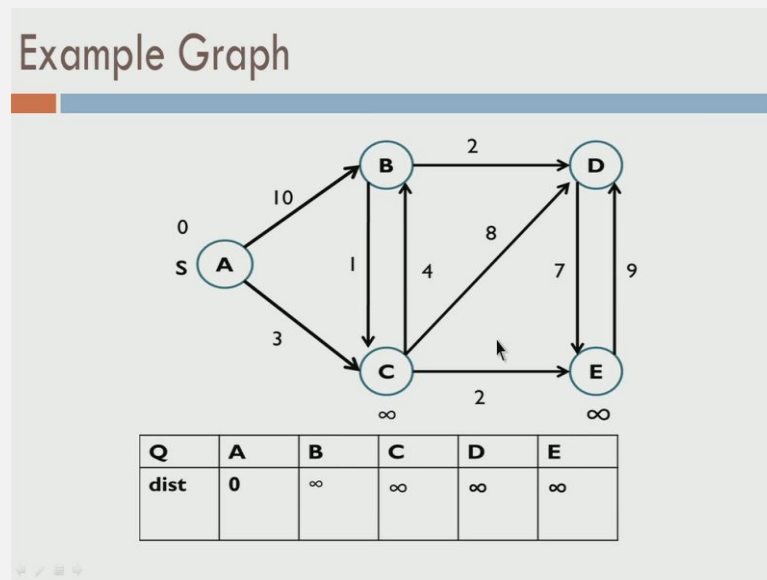
Initially,  $V_t$  is chosen to be empty that is we assert that for no vertex the distance for  $s$  has been computed and here is a while loop, till the queue becomes empty. As long as the queue is non empty, the while loop will be executed. What are the steps that are performed inside this loop? The first step is to pick a vertex which has the least value in the queue. This is achieved using the DeleteMin operation from the queue and let us say  $u$  is the vertex which is returned.

In other words,  $u$  is the vertex for which the distance value is the least among all the elements in the queue.  $V_t$  is now augmented to contain  $u$ , the invariant is that  $V_t$  always contains all the vertices for which the shortest path from  $s$  has been computed. After having added  $u$  to the set  $V_t$ , we update the distance bounds from the vertex  $s$  in the fourth coming loop. The way this is done is to consider the every vertex  $v$  which is outside the set  $V_t$  that is in  $V$  minus  $V_t$ .

With the property that  $v$  has an edge to  $u$ . Recall that  $u$  is the vertex which I just been added into the set  $V_t$ , that is  $u, v$  must be an edge and  $v$  must be outside the set  $V_t$ . Now, we update the distances. If we observe that the distance from  $s$  to  $u$  plus the weight of the edge  $u, v$  is smaller than the current estimate of distance of  $v$ , then distance of  $v$  is updated. The predecessor of  $v$  is also updated to be  $u$ . If this condition fails, no action is taken and nothing is changed.

Indeed, if this condition succeeds, the priority of  $v$  is updated in the queue using the decrease function. This completes the description of the algorithm. Observe that the algorithm will definitely terminate, this is very important because, at every step one element is removed from the  $Q$ , therefore eventually the  $Q$  will become empty.

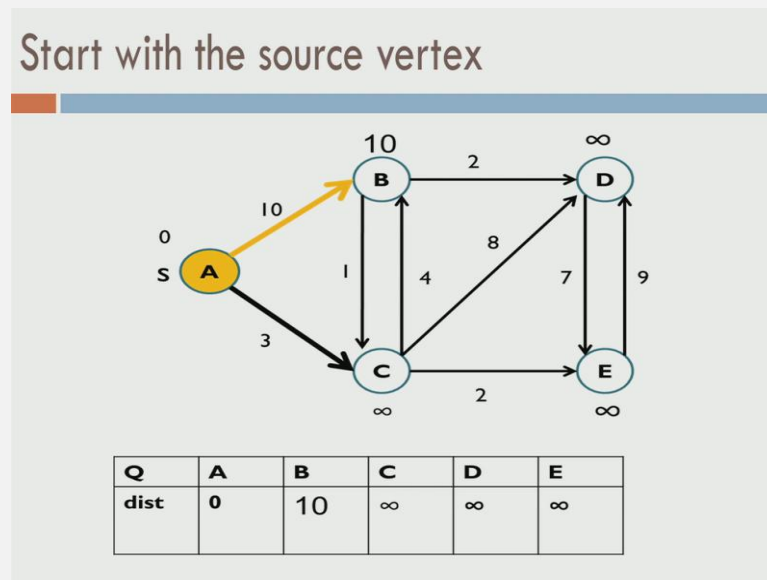
(Refer Slide Time: 11:18)



We need to understand, what the algorithm does, we do this initially by an example before going to a formal argument. So, this is an example graph that you can see on the slide. The edge weights are written near every edge. The vertices are A, B, C, D and E and the queue which is based on the distances are essentially the elements of this particular array, S is the source vertex in this example.

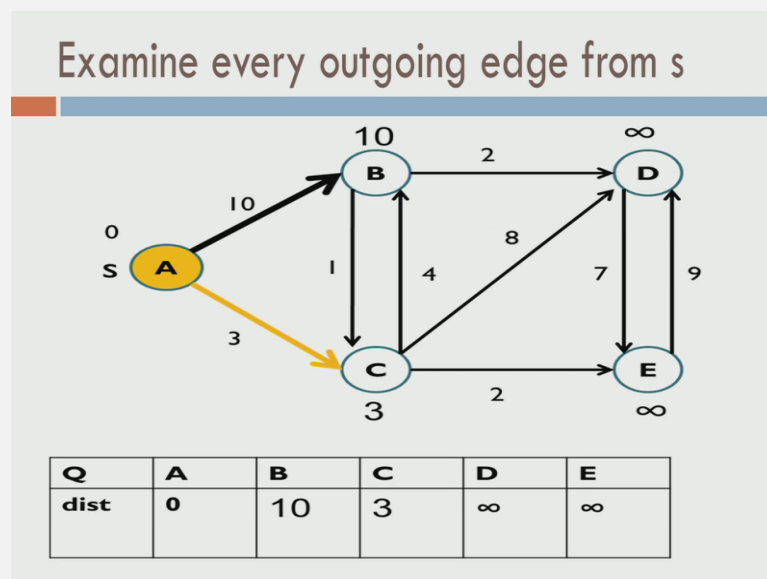
We will see that the Dykestra's algorithm that we have just see in the previous slide, eventually computes the distance of every vertex from the source vertex A, when the algorithm terminates. Initially, the distances of A to itself is set to 0 which is indeed correct, which is the distance of A to itself.

(Refer Slide Time: 12:20)



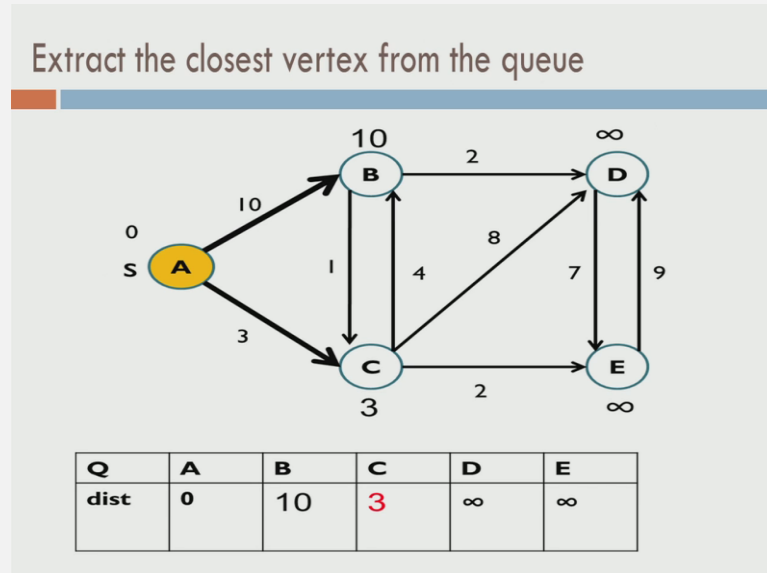
In the next step, upper bounds of the distances 2, vertices which are adjacent to S are obtained. In particular, the vertex B is adjacent to the vertex A, where an edge whose weight is 10. Therefore, the distance value of B is updated to 10 which is a current upper bound on the length of the shortest path from A to B. Similarly, further the yellow colour on the arrow indicates that the current predecessor of B is A. The orange coloured vertexes are also those vertexes for which the distance from S has already been calculated.

(Refer Slide Time: 13:25)



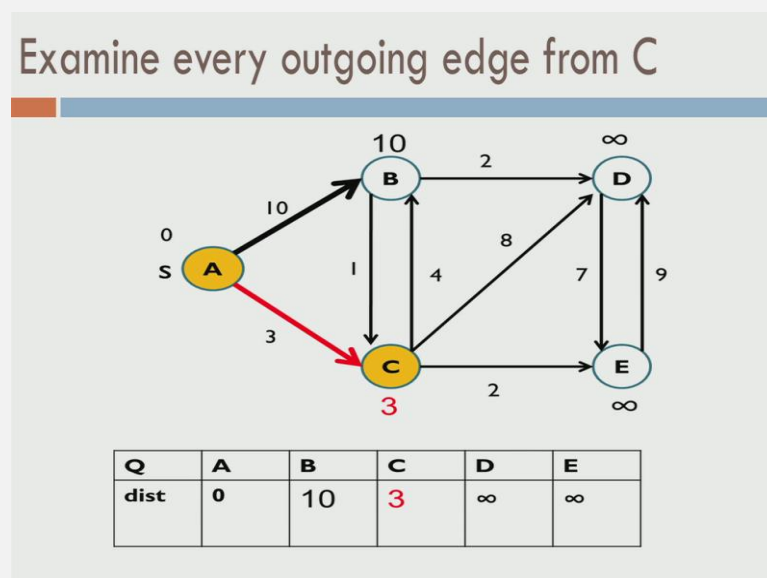
Similarly, the upper bound in the distance from S to C in this iteration is the distance 3 and the predecessor of C is selected to be A.

(Refer Slide Time: 13:49)



In the next iteration, we will now select the closest vertex from the Q, in other words the vertex which has a least value. This is coloured red and observe that in this particular Q, C is the vertex with the least value which is selected. As you can see, C is the vertex which is closest to S, based on the current distant estimates.

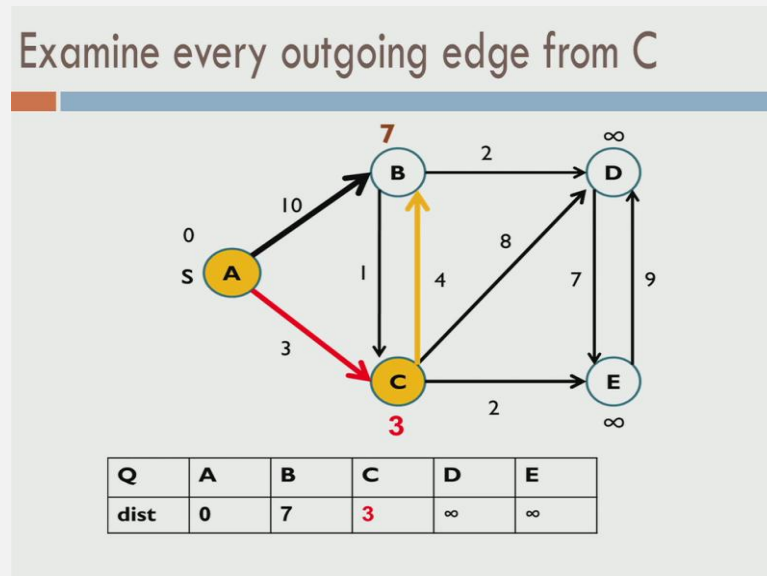
(Refer Slide Time: 14:15)





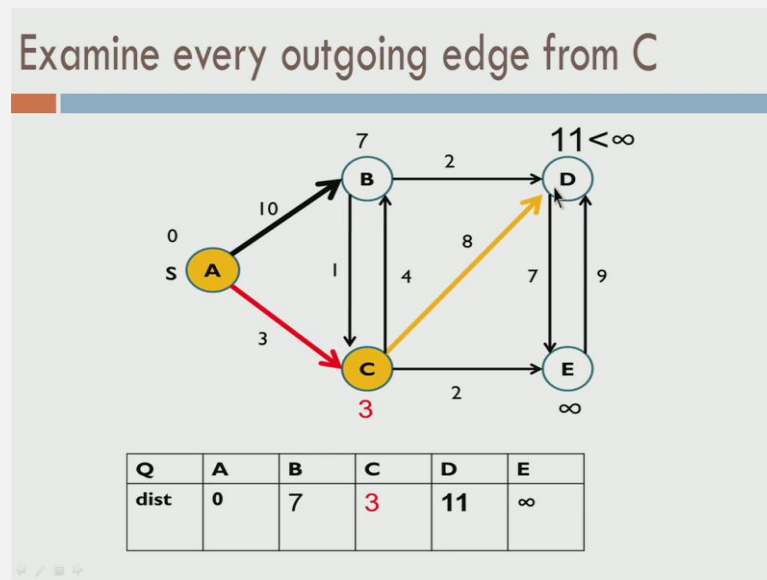
Now, C enters the set of vertexes for which the distance from S has already been calculated. In other words, the length of the shortest path from S has already been calculated. Now, C is a current vertex, we go into the iteration where we update the distances of all the other vertices which are outside the set V t which is, we will now update the distances for the vertices B, D and E.

(Refer Slide Time: 14:48)



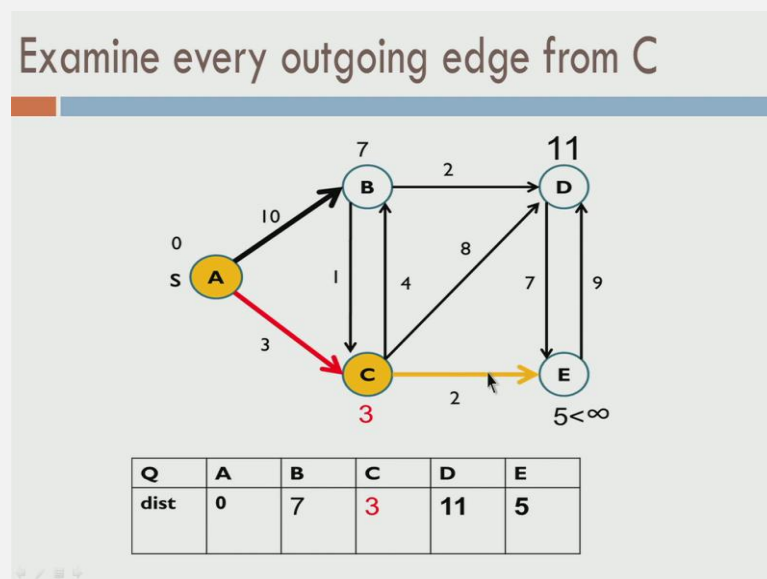
Let us start this with the vertex B. Observe that the distance estimates from S to B is now changed to the value 7. It was recall, earlier the value 10. Now, the distance from S to B is found to be shorter via C and this is updated to the value 7 from the previous estimate which was 10. The predecessor of B is also now updated to be the vertex C.

(Refer Slide Time: 15:29)



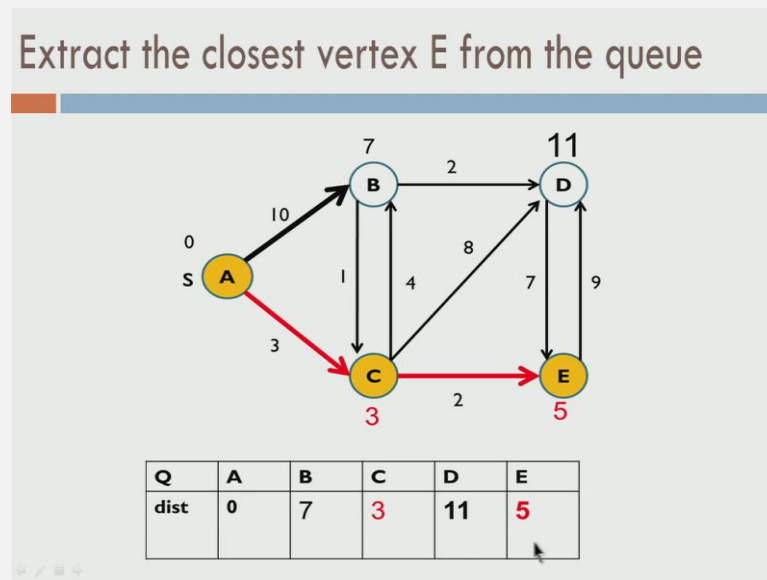
This is also done next for the vertex D, the distance estimate from S to D was infinity earlier. Now, the distances is found to be not larger than 11, because the distance from A to D via C is found to be 11, this is updated.

(Refer Slide Time: 15:54)



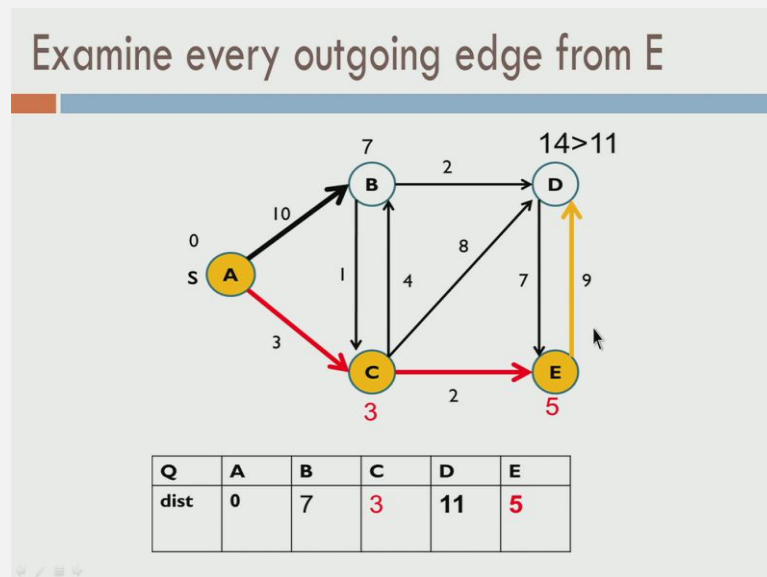
The same is done for the vertex E. Observe that the predecessor of D and E ((Refer Time: 15:58)) are both the vertex C. The next step was the algorithm we will now choose the vertex which is the closest to S. In this case, it is going to be E.

(Refer Slide Time: 16:16)



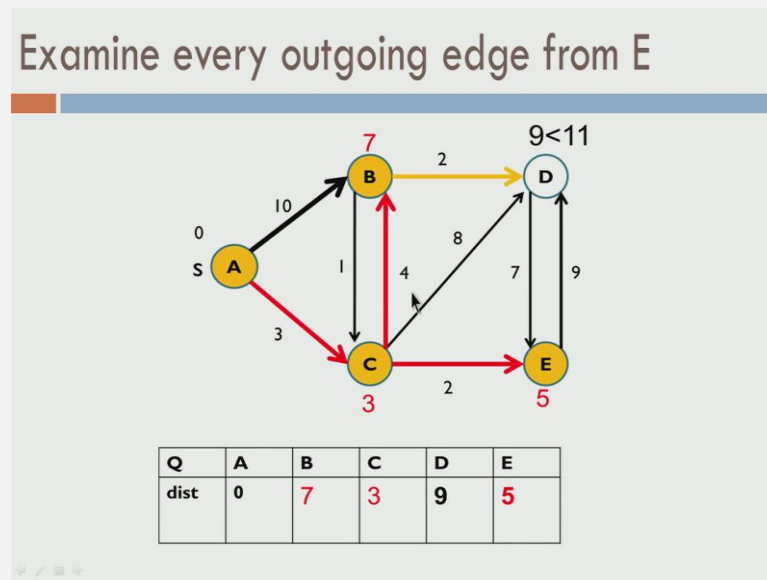
Therefore, we extract the closest vertex E from the priority queue and now E is added into the set of vertices for which distance from S has been calculated, E is now coloured orange. And now, we go ahead and estimate the distances of, now we go ahead and update the distances of the vertices B and D.

(Refer Slide Time: 16:41)



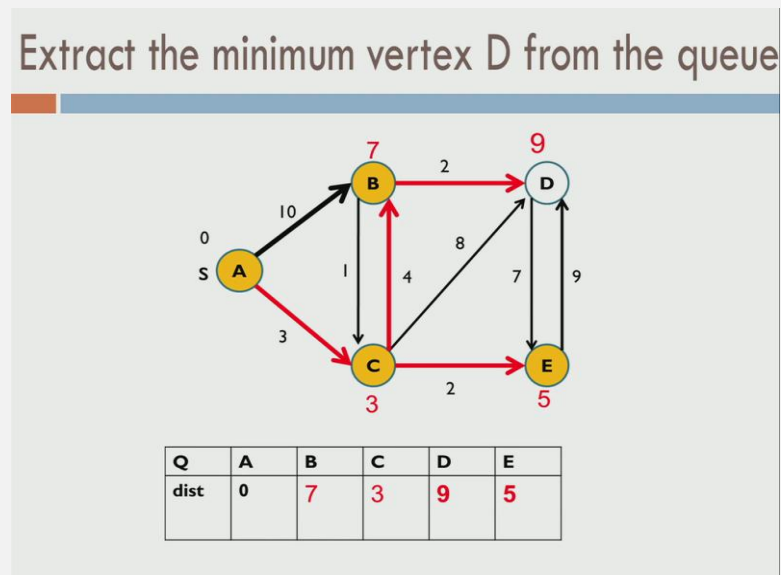
Now, observe that the distance from S to D going via E is 14 which is more than 11, therefore the distance value of D does not change.

(Refer Slide time: 17:14)



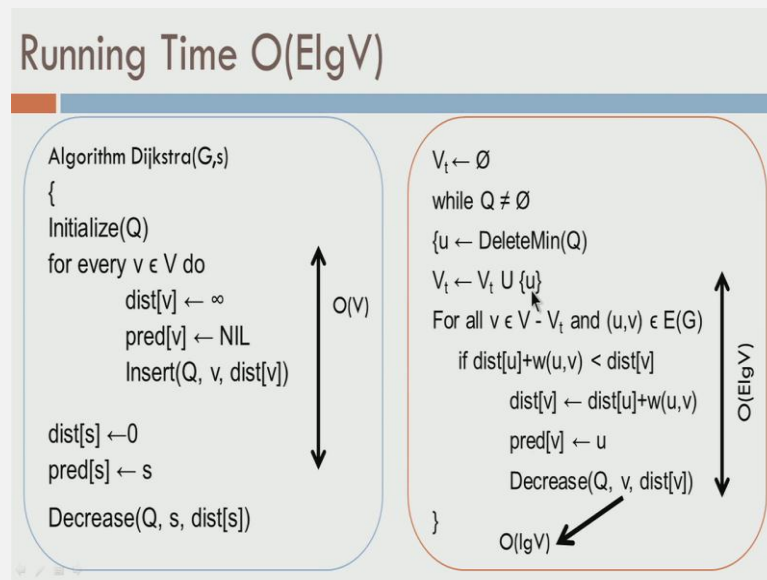
Similarly, the distance value of B also does not change. The next step, we extract the closest vertex which is B from the queue which is now coloured orange and the predecessor of B is taken to be C. And now we update the distance of D via B which is only 9 which is smaller than 11 and now the whole procedure has come to an end.

(Refer Slide Time: 17:32)



Because, D is now removed from the priority queue, because that was the only element in the queue, the queue has become empty the algorithm comes to an end.

(Refer Slide Time: 17:42)



Now, let us analyse the running time of this algorithm. Let us look at the initialization phase of the algorithm and let us look at the work that is done. For every vertex  $v$ , the distance value is initially set to infinity. There are  $n$  vertexes or if we take  $v$  to be the cardinality of the set  $v$  itself, then there are  $V$  vertexes, capital  $V$  number of vertices and they are all initialized to once. The predecessor of each vertexes also set once in an initialization phase, the insertion into the  $Q$  for every vertexes also done once.

Therefore, the initialization phase takes order of  $V$  time, where  $V$  is a total number of vertexes in the graph  $G$ . In this iteration, observe that this iteration, the outer iteration runs  $v$  number of times, in every iteration 1 vertex is removed, however the dominating factor in the running time is in this while loop.

However, the dominating factor of the running time is in this for loop, where every edges inspected once and the primary operation which is done here is, further the most expensive operation, among the three operations here is to decrease the priority in the queue which takes  $\log v$  time and there are  $E$  edges,  $E$  number of edges therefore, the total running time is  $E$  times  $\log v$ . Therefore, the running time of the algorithm is order of  $v$  plus order of  $\log v$ . Recall that the  $\log v$  guarantee comes from the property of the priority queue data structure.

(Refer Slide Time: 20:07)

## Correctness Arguments

- For every vertex  $v$ , when it is removed from the priority queue  $Q$ ,  $\text{dist}[v]$  is the distance from  $s$  to  $v$ 
  - The claim is true for  $s$ , the first vertex to be removed from the priority queue!!
  - To prove our claim by induction, let it be true for all vertices removed from the queue  $Q$  before the  $i$ -th step. Here  $i \geq 1$ .
  - Let this set be called  $F$  (for distance FOUND)
  - Let  $v$  be the  $i$ -th vertex removed from  $Q$ .
  - Let  $u$  be the predecessor of  $v$ , when  $v$  is removed,  $\text{pred}[v] = u$
  - We know that  $\text{dist}[v] = \text{dist}[u] + w(u, v)$

We now have to conclude that the algorithm is indeed correct and we have a sequence of correctness arguments. The proof of correctness of the algorithm is guaranteed by the claim which is, for every vertex  $v$  when it is removed from the priority queue, the value  $\text{dist}[v]$  is the distance from  $s$  to  $v$  in the graph. This is the claim that we need to prove. Let us start off with the base case, the claim indeed is true for  $s$  which is the first vertex to be removed from the priority queue, because the distance from  $s$  to itself is 0.

We now prove our claim by induction on the index of the vertex to be removed from queue. Let us assume that the claim is true for all vertices which are removed from the queue before the  $i$ th step, where  $i$  is at least 1. Let us call the set of vertices that have been removed, the set  $F$  and  $F$  stands for the word FOUND for which the shortest distances have been found already. Let  $v$  be the  $i$ th vertex to be removed from the  $Q$ .

Now, let  $u$  be the predecessor of the vertex  $v$ , when  $v$  is removed, in other words, let  $\text{pred}[v]$  be equal to the vertex  $u$ . We know that from the description of the algorithm, we know that the distance of  $v$ ,  $\text{dist}[v]$  is equal to  $\text{dist}[u]$  plus the weight of the edge  $u, v$ , this is from the description of the algorithm.

(Refer Slide Time: 22:22)

### Correctness continued

- Let  $P$  be any other path from  $s$  to  $v$  in  $G$
- Let  $(x,y)$  be the first edge such that  $x$  is in  $F$  and  $y$  is not
  - Important: Note  $F$  is the set before  $v$  is added to it.
- Let us observe that  $w(P) \geq \text{dist}[v]$ 
  - $w(P) = \text{dist}[x] + w(x,y) + w(P_{xv}) \geq \text{dist}[y] + w(P_{xv}) \geq \text{dist}[v]$
  - First equality follows from definition of  $w(P)$ .
  - The second inequality follows because of definition of  $\text{dist}[y]$
  - The third inequality follows because
    - $\text{dist}[v] \leq \text{dist}[y]$ , as the algorithm chose  $v$  as the one with minimum  $d$  value
    - And  $w(P_{xv})$  is not negative!!!

We now show that any other path from  $s$  to  $v$  has the property that the length of that path given by  $w$  of  $P$  is at least as large as the value  $\text{dist}$  of  $v$ . To understand this analysis, let  $x$  comma  $y$  be the first edge on the path  $P$ , where with the property that  $x$  is in  $F$  and  $y$  is not in  $F$ . It is important to note that  $F$  is the current set in the analysis, in this iteration in the algorithm that is  $F$  is the set before  $v$  is added to  $F$ . We now argue that the length of  $P$  is at least as large as distance of  $v$ .

We now argue that the length of  $P$  denoted by  $w$  of  $P$  is at least as large as  $\text{dist}$  of  $v$  which is our estimate of the distance of  $v$  from the vertex  $s$ . To make this observation, let us write down what  $w$  of  $P$  is. Clearly,  $w$  of  $P$  is the distance from  $s$  to  $x$  plus the weight of the edges  $x$  comma  $y$  and the weight of the path  $P_{xv}$  [FL].

We observe that the weight of  $P$  is the distance from  $s$  to  $x$ , by induction we know that because  $x$  is already in  $F$ , the distance from  $s$  to  $x$  is indeed the length of the shortest path from  $s$  to  $x$  plus the weight of the edge  $w$  of  $x$  comma  $y$  plus the weight of the path  $P$  from  $x$  to  $v$ . Now, we will also show that this is at least as large as distance of  $y$  plus weight of  $P_{xv}$ .

(Refer Slide Time: 25:40)

### Correctness continued

- Let  $P$  be any other path from  $s$  to  $v$  in  $G$
- Let  $(x,y)$  be the first edge such that  $x$  is in  $F$  and  $y$  is not
  - Important: Note  $F$  is the set before  $v$  is added to it.
- Let us observe that  $w(P) \geq \text{dist}[v]$ 
  - $w(P) = \text{dist}[x] + w(x,y) + w(P_{yv}) \geq \text{dist}[y] + w(P_{yv}) \geq \text{dist}[v]$
  - First equality follows from definition of  $w(P)$ .
  - The second inequality follows because of definition of  $\text{dist}[y]$
  - The third inequality follows because
    - $\text{dist}[v] \leq \text{dist}[y]$ , as the algorithm chose  $v$  as the one with minimum  $d$  value
    - And  $w(P_{yv})$  is not negative!!!

We now to show that the weight of the path is at least  $\text{dist}$  of  $v$ , we now show you that the weight of the path  $P$  is at least  $\text{dist}$  of  $v$ . To do this, we proof the following sequence of inequalities. We first show that weight of  $P$  is equal to distance of  $x$  plus  $w$  of  $x$  comma  $y$  plus weight of the path from  $y$  to  $v$ , we then show that this is at least as large as  $\text{dist}$  of  $y$  plus weight of the path from  $y$  to  $v$  and then with this is at least distance of  $v$ .

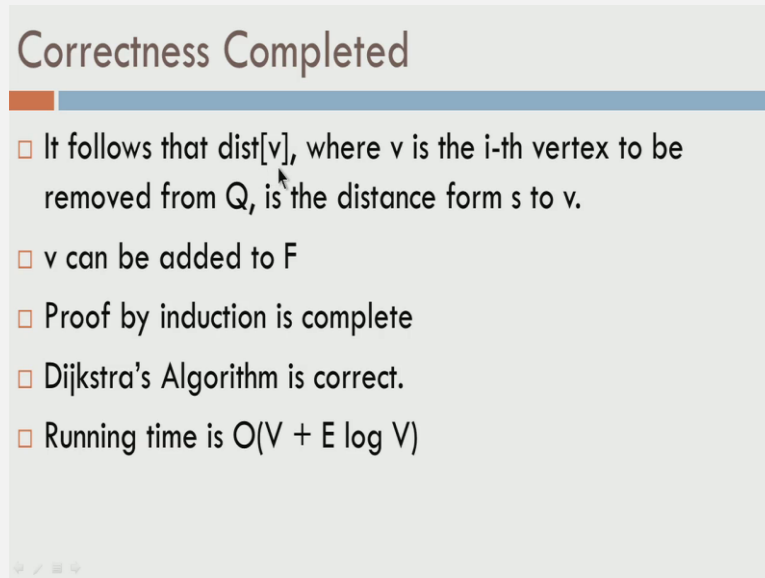
Now, let us focus on the first equality, this follows from the definition of  $w$  of  $p$  and the application of the induction hypothesis. How do we apply the induction hypothesis? We observe that  $x$  is already in  $F$  therefore, by induction  $\text{dist}$  of  $x$  contains the distance from by induction  $\text{dist}$  of  $x$  contains the value of the length of shortest path from  $s$  to  $x$ . Therefore, the length of  $P$  is distance from  $s$  to  $x$  plus the weight of the edge  $x$   $y$  plus the length of the path  $y$   $v$ , this is the length of the path  $P$ .

Now, the second inequality follows, because by definition distance of  $y$  is at most the distance of  $x$  plus the weight of  $x$  comma  $y$ , this is the definition of the distance in the algorithm. The third inequality follows because of two reasons. One is that we have chosen distance of  $v$  to be the one with the least value from the queue. Therefore, distance of  $v$  is smaller than or equal to distance of  $y$ . Observe that  $v$  and  $y$  are not in  $F$  and the algorithm choose distance of  $v$  to be the one with a smallest value of distance. Therefore, distance of  $v$  is smaller than or equal to distance of  $y$ .



It is very important now to observe that because, weight of  $P \rightarrow v$  is non negative, this inequality follows. Putting all the three together, it follows that the length of the path is at least distance of  $v$ .

(Refer Slide Time: 28:12)



### Correctness Completed

- It follows that  $\text{dist}[v]$ , where  $v$  is the  $i$ -th vertex to be removed from  $Q$ , is the distance from  $s$  to  $v$ .
- $v$  can be added to  $F$
- Proof by induction is complete
- Dijkstra's Algorithm is correct.
- Running time is  $O(V + E \log V)$

Consequently, it follows that distance of  $v$ , where  $v$  is  $i$ th vertex to be added is indeed the shortest length or the shortest path, in other words the distance from  $s$  to  $v$ . Now,  $v$  is to be added to  $F$ , the proof by induction is complete. In other words, for all the vertices in  $F$ , the distance from  $s$  is already calculated. This shows that Dijkstra's algorithm is correct and the running time of the algorithm is also something that we have analysed, the running time...