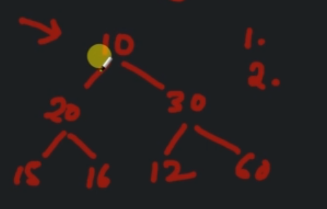
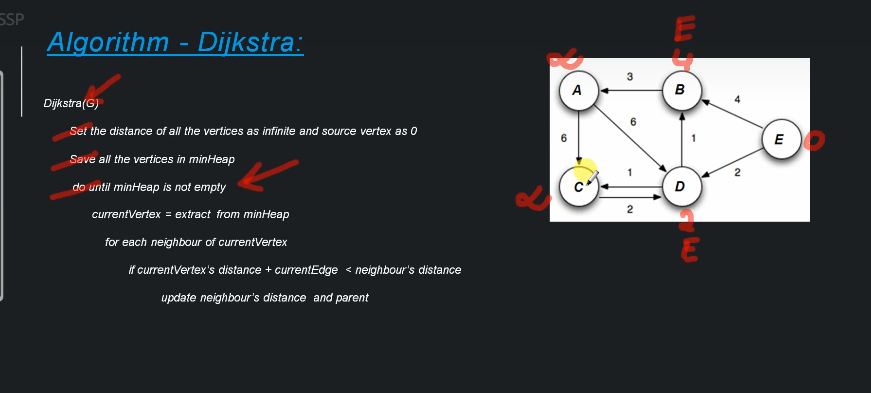


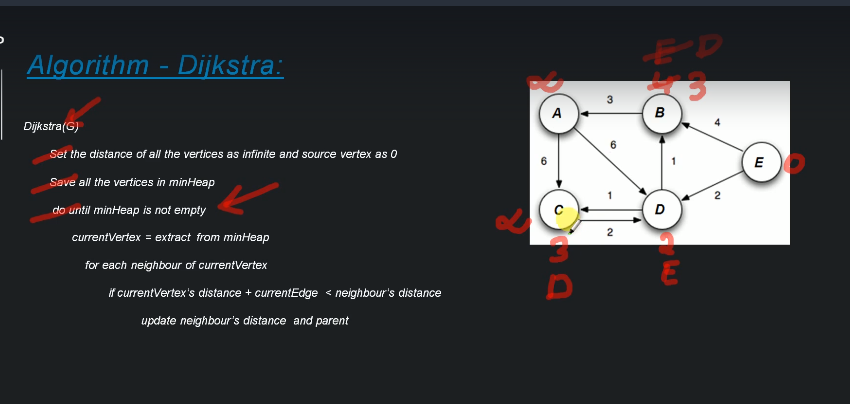
**Shortest path algorithm**

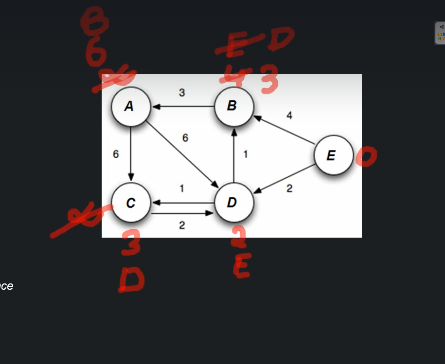
In min Heap the top most element is extracted first and the top most element is the smallest. But it is not applied recursively means the only the element 10 is the smallest, other elements such as 20 is also a root node but it is not the smallest among the child.

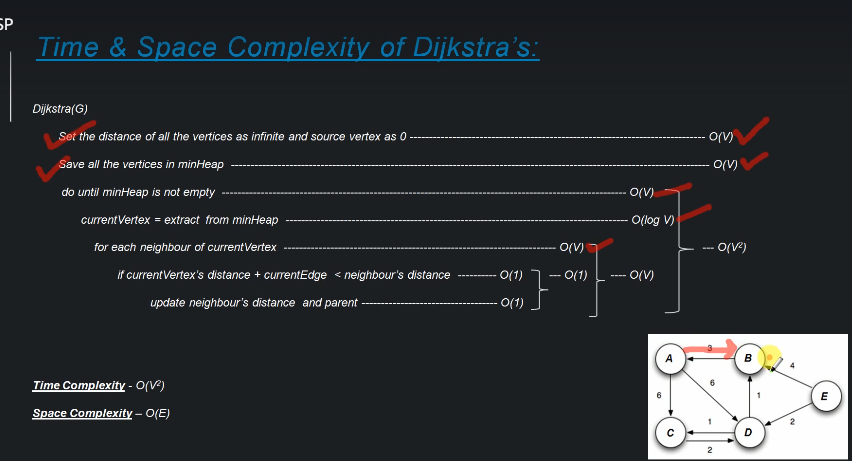


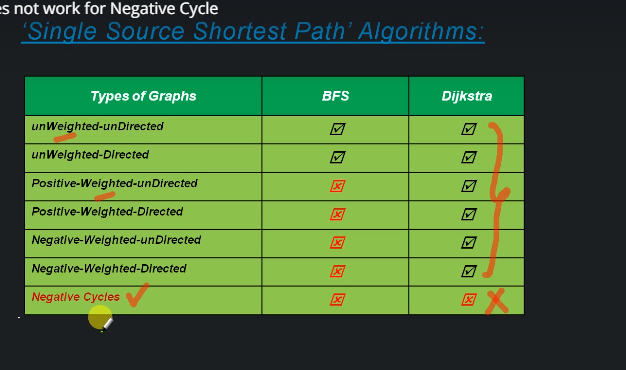


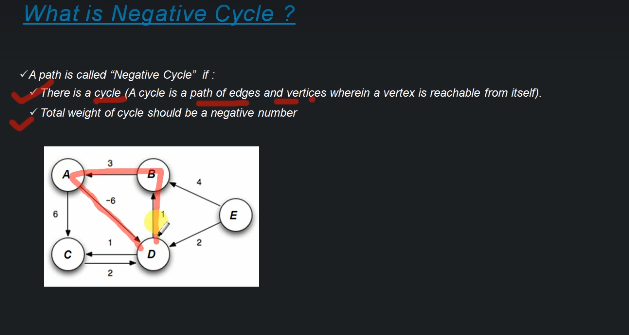
Now we will take D and find out the above formula

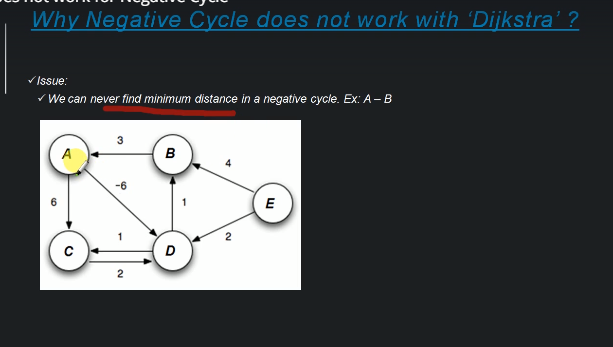


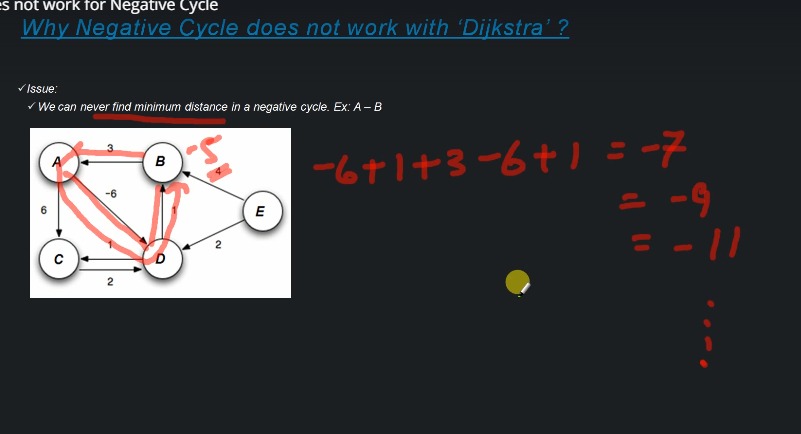












## **Dijkstra’s Algorithm**

The algorithm we are going to use to determine the shortest path is called “Dijkstra’s algorithm.” Dijkstra’s algorithm is an iterative algorithm that provides us with the shortest path from one particular starting node to all other nodes in the graph. Again this is similar to the results of a breadth first search.

To keep track of the total cost from the start node to each destination we will make use of a distances dictionary which we will initialize to 0 for the start vertex, and infinity for the other vertices. Our algorithm will update these values until they represent the smallest weight path from the start to the vertex in question, at which point we will return the distances dictionary.

The algorithm iterates once for every vertex in the graph; however, the order that we iterate over the vertices is controlled by a priority queue. The value that is used to determine the order of the objects in the priority queue is the distance from our starting vertex. By using a priority queue, we ensure that as we explore one vertex after another, we are always exploring the one with the smallest distance.

The code for Dijkstra’s algorithm is shown below.

import heapq

def calculate\_distances(graph, starting\_vertex):

distances = {vertex: float('infinity') for vertex in graph}

distances[starting\_vertex] = 0

pq = [(0, starting\_vertex)]

while len(pq) > 0:

current\_distance, current\_vertex = heapq.heappop(pq)

# Nodes can get added to the priority queue multiple times. We only

# process a vertex the first time we remove it from the priority queue.

if current\_distance > distances[current\_vertex]:

continue

for neighbor, weight in graph[current\_vertex].items():

distance = current\_distance + weight

# Only consider this new path if it's better than any path we've

# already found.

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(pq, (distance, neighbor))

return distances

example\_graph = {

'U': {'V': 2, 'W': 5, 'X': 1},

'V': {'U': 2, 'X': 2, 'W': 3},

'W': {'V': 3, 'U': 5, 'X': 3, 'Y': 1, 'Z': 5},

'X': {'U': 1, 'V': 2, 'W': 3, 'Y': 1},

'Y': {'X': 1, 'W': 1, 'Z': 1},

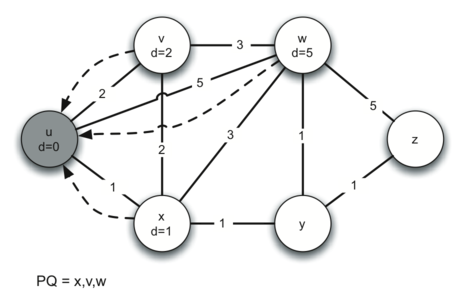
'Z': {'W': 5, 'Y': 1},

}

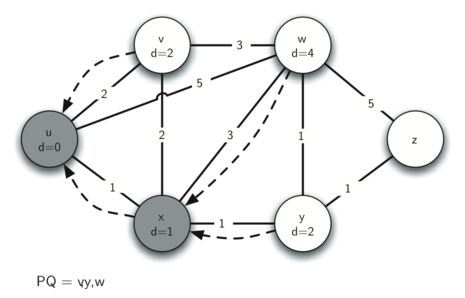
print(calculate\_distances(example\_graph, 'X'))

# => {'U': 1, 'W': 2, 'V': 2, 'Y': 1, 'X': 0, 'Z': 2}

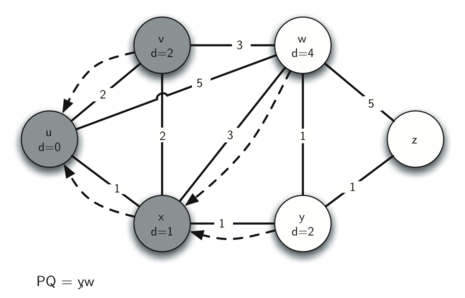
Let’s walk through an application of Dijkstra’s algorithm one vertex at a time using the following sequence of diagrams as our guide. We begin with the vertex u*u*. The three vertices adjacent to u*u* are v,w,*v*,*w*, and x*x*. Since the initial distances to v,w,*v*,*w*, and x*x* are all initialized to infinity, the new costs to get to them through the start node are all their direct costs. So we update the costs to each of these three nodes. The state of the algorithm is:



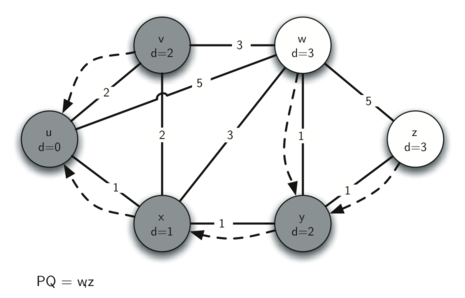
In the next iteration of the while loop we examine the vertices that are adjacent to u*u*. The vertex x*x* is next because it has the lowest overall cost and therefore will be the first entry removed from the priority queue. At x*x* we look at its neighbors u,v,w*u*,*v*,*w* and y*y*. For each neighboring vertex we check to see if the distance to that vertex through x*x* is smaller than the previously known distance. Obviously this is the case for y*y* since its distance was infinity. It is not the case for u*u* or v*v* since their distances are 0 and 2 respectively. However, we now learn that the distance to w*w* is smaller if we go through x*x* than from u*u* directly to w*w*. Since that is the case we update w*w* with a new distance and add another entry to the priority queue. The state of the algorithm is now:



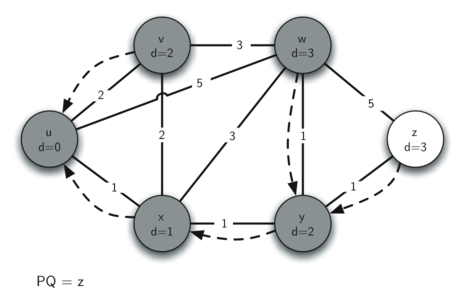
The next step is to look at the vertices neighboring v*v* (below). This step results in no changes to the graph, so we move on to node y*y*.

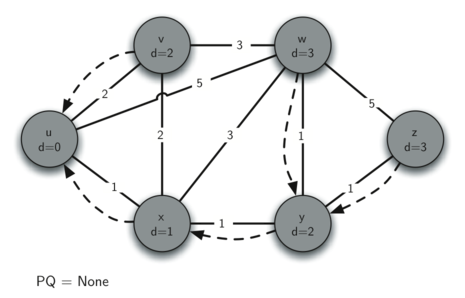


At node y*y* (below) we discover that it is cheaper to get to both w*w* and z*z*, so we adjust the distances accordingly.



Finally we check nodes w*w* and z*z*. However, no additional changes are found and so the priority queue is empty and Dijkstra’s algorithm exits.





It is important to note that Dijkstra’s algorithm works only when the weights are all positive. You should convince yourself that if you introduced a negative weight on one of the edges to the graph that the algorithm would never exit.

If we take more rounds then the distance will keep on decreasing, so we cannot find a minimum path.

Algorithm  
**1)** Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.  
**2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.  
**3)** While sptSet doesn’t include all vertices  
….**a)** Pick a vertex u which is not there in sptSet and has minimum distance value.  
….**b)** Include u to sptSet.  
….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

# Python program for Dijkstra's single

# source shortest path algorithm. The program is

# for adjacency matrix representation of the graph

# Library for INT\_MAX

import sys

class Graph():

    def \_\_init\_\_(self, vertices):

        self.V = vertices

        self.graph = [[0 for column in range(vertices)]

                    for row in range(vertices)]

    def printSolution(self, dist):

        print "Vertex tDistance from Source"

        for node in range(self.V):

            print node, "t", dist[node]

    # A utility function to find the vertex with

    # minimum distance value, from the set of vertices

    # not yet included in shortest path tree

    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next node

        min = sys.maxint

        # Search not nearest vertex not in the

        # shortest path tree

        for v in range(self.V):

            if dist[v] < min and sptSet[v] == False:

                min = dist[v]

                min\_index = v

        return min\_index

    # Funtion that implements Dijkstra's single source

    # shortest path algorithm for a graph represented

    # using adjacency matrix representation

    def dijkstra(self, src):

        dist = [sys.maxint] \* self.V

        dist[src] = 0

        sptSet = [False] \* self.V

        for cout in range(self.V):

            # Pick the minimum distance vertex from

            # the set of vertices not yet processed.

            # u is always equal to src in first iteration

            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the

            # shotest path tree

            sptSet[u] = True

            # Update dist value of the adjacent vertices

            # of the picked vertex only if the current

            # distance is greater than new distance and

            # the vertex in not in the shotest path tree

            for v in range(self.V):

                if self.graph[u][v] > 0 and \

                     sptSet[v] == False and \

                     dist[v] > dist[u] + self.graph[u][v]:

                    dist[v] = dist[u] + self.graph[u][v]

        self.printSolution(dist)

# Driver program

g = Graph(9)

g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],

        [4, 0, 8, 0, 0, 0, 0, 11, 0],

        [0, 8, 0, 7, 0, 4, 0, 0, 2],

        [0, 0, 7, 0, 9, 14, 0, 0, 0],

        [0, 0, 0, 9, 0, 10, 0, 0, 0],

        [0, 0, 4, 14, 10, 0, 2, 0, 0],

        [0, 0, 0, 0, 0, 2, 0, 1, 6],

        [8, 11, 0, 0, 0, 0, 1, 0, 7],

        [0, 0, 2, 0, 0, 0, 6, 7, 0]

        ];

g.dijkstra(0);

Vertex tDistance from Source

0 t 0

1 t 4

2 t 12

3 t 19

4 t 21

5 t 11

6 t 9

7 t 8

8 t 14

Namedtuple in Python

Python supports a type of container like [dictionaries](http://quiz.geeksforgeeks.org/python-set-4-dictionary-keywords-python/) called “**namedtuples()**” present in module, “**collection**“. Like dictionaries they contain keys that are hashed to a particular value. But on contrary, it supports both access from key value and iteration, the functionality that dictionaries lac

|  |
| --- |
| # Python code to demonstrate namedtuple() and  # Access by name, index and getattr()    # importing "collections" for namedtuple()  import collections    # Declaring namedtuple()  Student = collections.namedtuple('Student',['name','age','DOB'])    # Adding values  S = Student('Nandini','19','2541997')    # Access using index  print ("The Student age using index is : ",end ="")  print (S[1])    # Access using name  print ("The Student name using keyname is : ",end ="")  print (S.name)    # Access using getattr()  print ("The Student DOB using getattr() is : ",end ="")  print (getattr(S,'DOB')) |

Output :

The Student age using index is : 19

The Student name using keyname is : Nandini

The Student DOB using getattr() is : 2541997

# importing "collections" for namedtuple()

import collections

# Declaring namedtuple()

Student = collections.namedtuple('Student',['name','age','DOB'])

# Adding values

S = Student('Nandini','19','2541997')

# initializing iterable

li = ['Manjeet', '19', '411997' ]

# initializing dict

di = { 'name' : "Nikhil", 'age' : 19 , 'DOB' : '1391997' }

# using \_make() to return namedtuple()

print ("The namedtuple instance using iterable is  : ")

print (Student.\_make(li))

# using \_asdict() to return an OrderedDict()

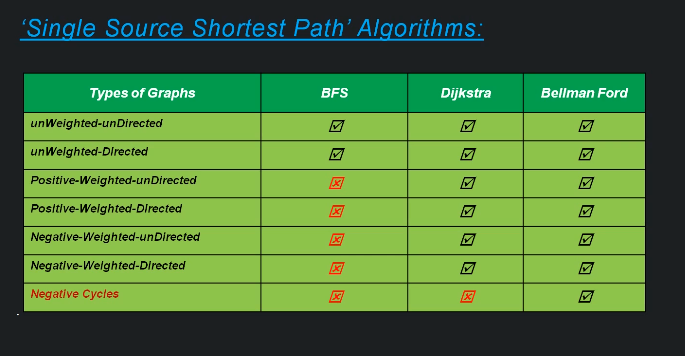
print ("The OrderedDict instance using namedtuple is  : ")

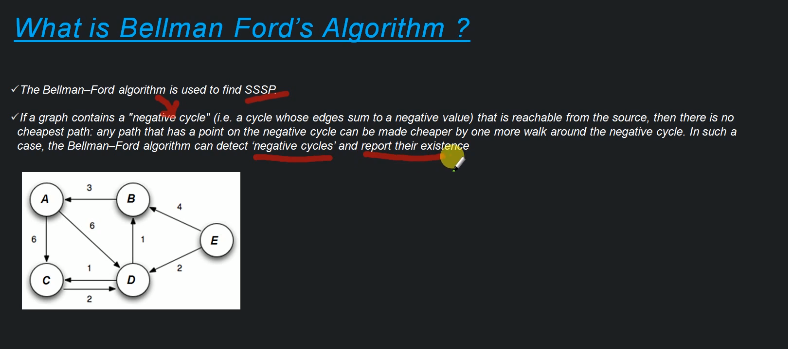
print (S.\_asdict())

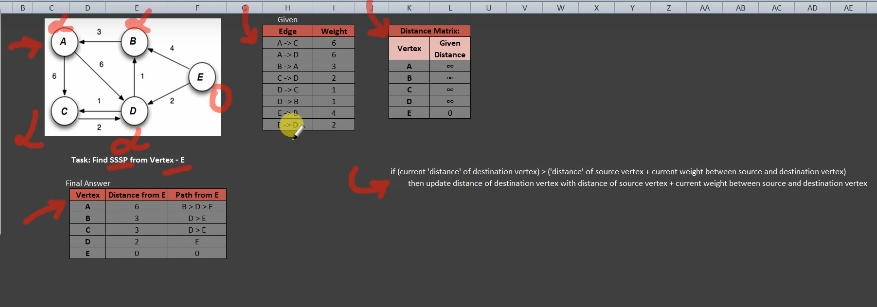
# using \*\* operator to return namedtuple from dictionary

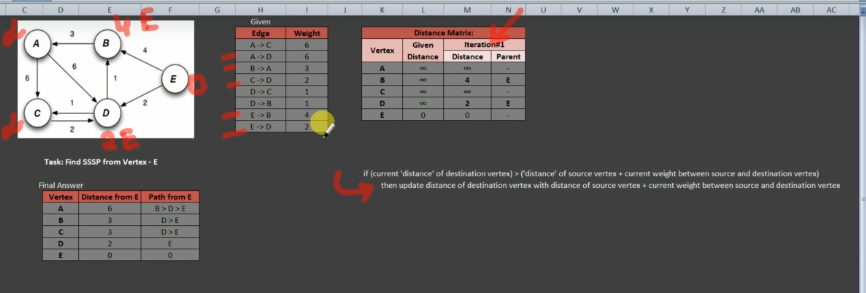
print ("The namedtuple instance from dict is  : ")

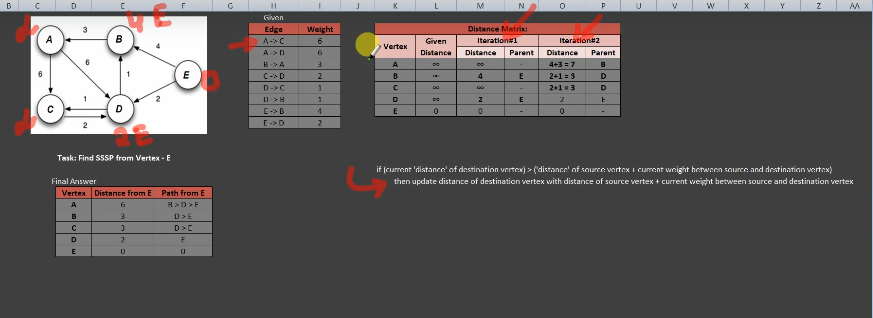
print (Student(\*\*di))

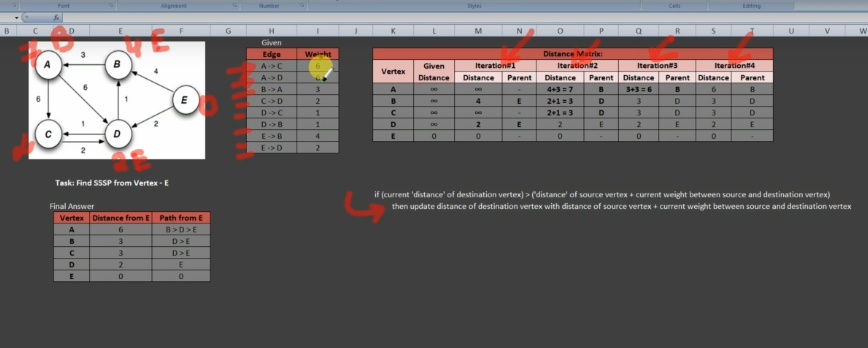




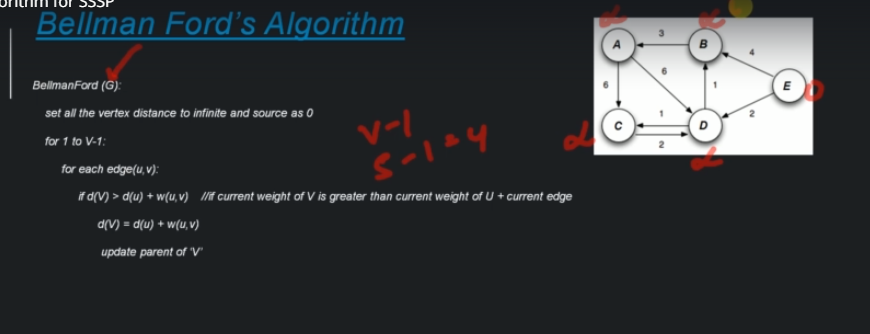


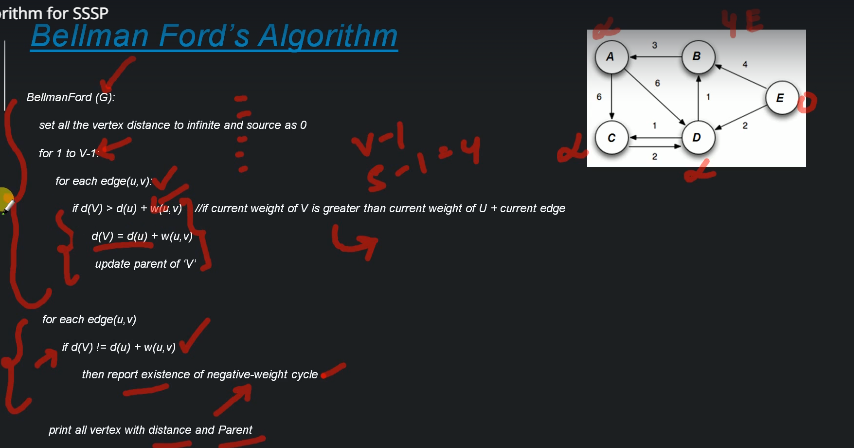


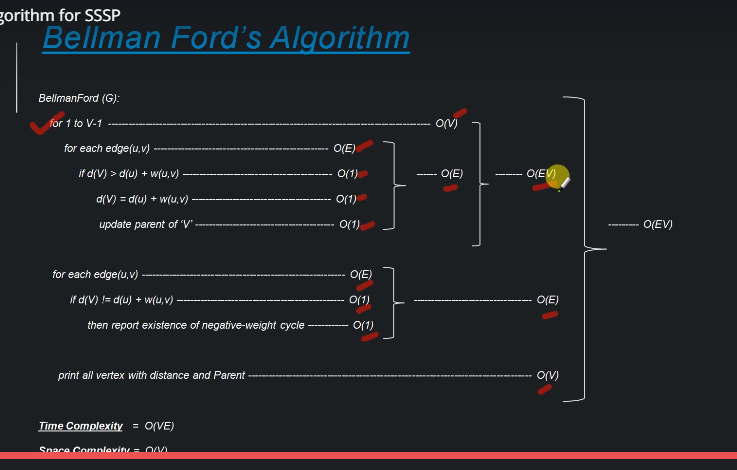




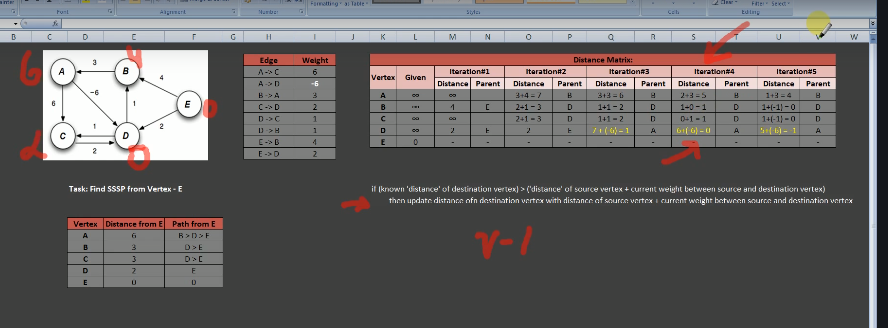
We have to run this algorithm with V-1 times where V is the number of vertices.

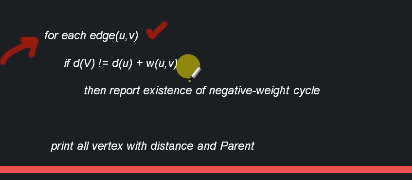




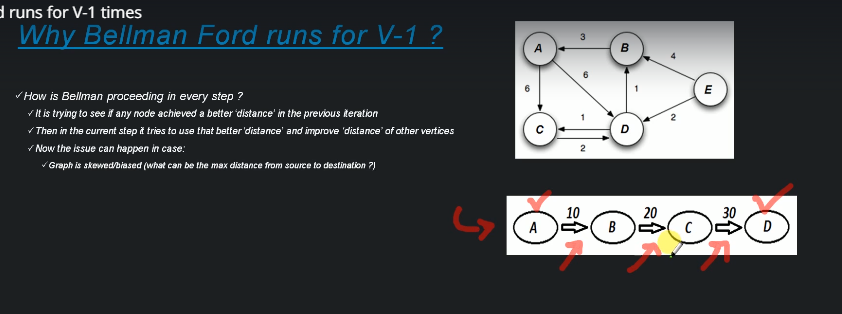


When there is a negative cycle, we are getting a negative value.



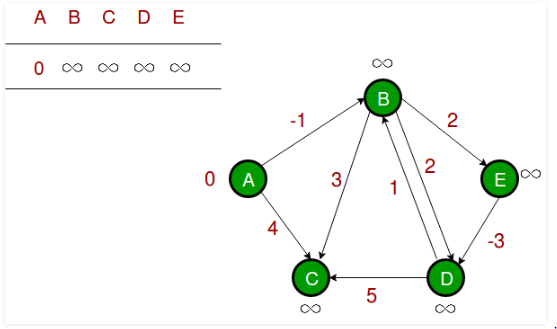


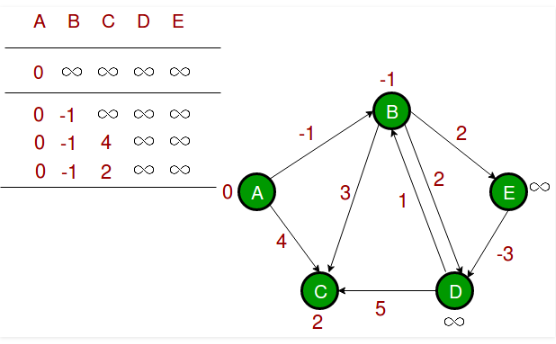
Here we are checking the condition that if any of the weight is decreased with iteration, then we will tell that there is a negative cycle that is present only in the vth iteration.

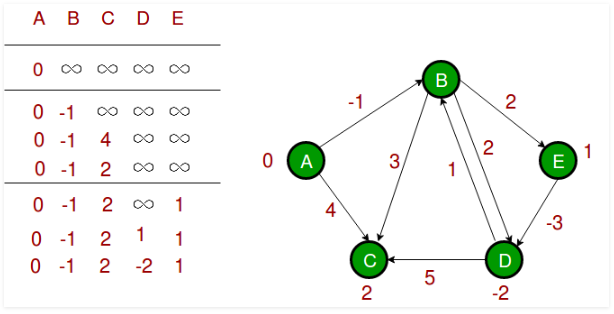
Why BellmanFord runs for V-1 times

We can have maximum 3 edges in the worse case when there is 4 nodes.

In v-1 iteration we can see that we can find the optimum distances. If in the vth iteration, there is any change in any of the nodes then we can say that there is a negative value in the edge else in the vth iteration also the distance will remain same.







# Python program for Bellman-Ford's single source

# shortest path algorithm.

I can keep reducing the shortest path by going through the loop multiple times.

**It doesn’t use priority queue**

from collections import defaultdict

# Class to represent a graph

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices # No. of vertices

        self.graph = [] # default dictionary to store graph

    # function to add an edge to graph

    def addEdge(self, u, v, w):

        self.graph.append([u, v, w])

    # utility function used to print the solution

    def printArr(self, dist):

        print("Vertex   Distance from Source")

        for i in range(self.V):

            print("% d \t\t % d" % (i, dist[i]))

    # The main function that finds shortest distances from src to

    # all other vertices using Bellman-Ford algorithm.  The function

    # also detects negative weight cycle

    def BellmanFord(self, src):

        # Step 1: Initialize distances from src to all other vertices

        # as INFINITE

        dist = [float("Inf")] \* self.V

        dist[src] = 0

        # Step 2: Relax all edges |V| - 1 times. A simple shortest

        # path from src to any other vertex can have at-most |V| - 1

        # edges

        for i in range(self.V - 1):

            # Update dist value and parent index of the adjacent vertices of

            # the picked vertex. Consider only those vertices which are still in

            # queue

            for u, v, w in self.graph:

                if dist[u] != float("Inf") and dist[u] + w < dist[v]:

                        dist[v] = dist[u] + w

        # Step 3: check for negative-weight cycles.  The above step

        # guarantees shortest distances if graph doesn't contain

        # negative weight cycle.  If we get a shorter path, then there

        # is a cycle.

        for u, v, w in self.graph:

                if dist[u] != float("Inf") and dist[u] + w < dist[v]:

                        print "Graph contains negative weight cycle"

                        return

        # print all distance

        self.printArr(dist)

g = Graph(5)

g.addEdge(0, 1, -1)

g.addEdge(0, 2, 4)

g.addEdge(1, 2, 3)

g.addEdge(1, 3, 2)

g.addEdge(1, 4, 2)

g.addEdge(3, 2, 5)

g.addEdge(3, 1, 1)

g.addEdge(4, 3, -3)

# Print the solution

g.BellmanFord(0)

**Output:**

Vertex Distance from Source

0 0

1 -1

2 2

3 -2

4 1

