The time complexity for the matrix representation is O(V^2). In this post, O(ELogV) algorithm for adjacency list representation is discussed.

In Dijkstra's algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in O(V+E) time using BFS. The idea is to traverse all vertices of graph using BFS and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet). Min Heap is used as a priority queue to get the minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is O(LogV) for Min Heap.

Following are the detailed steps.

1. Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.
2. Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).
3. While Min Heap is not empty, do following
    1. Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.
    2. For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of u-v plus distance value of u, then update the distance value of v.

```
# A Python program for Dijkstra's shortest
# path algorithm for adjacency
# list representation of graph

from collections import defaultdict
import sys

class Heap():

    def __init__(self):
        self.array = []
        self.size = 0
        self.pos = []

    def newMinHeapNode(self, v, dist):
        minHeapNode = [v, dist]
        return minHeapNode
```

```
    # A utility function to swap two nodes
    # of min heap. Needed for min heapify
    def swapMinHeapNode(self,a, b):
        t = self.array[a]
        self.array[a] = self.array[b]
        self.array[b] = t

    # A standard function to heapify at given idx
    # This function also updates position of nodes
    # when they are swapped.Position is needed
    # for decreaseKey()
    def minHeapify(self, idx):
        smallest = idx
        left = 2*idx + 1
        right = 2*idx + 2

        if left < self.size and self.array[left][1] \
                                < self.array[smallest]
[1]:
            smallest = left

        if right < self.size and self.array[right][1]\
                                < self.array[smallest]
[1]:
            smallest = right

        # The nodes to be swapped in min
        # heap if idx is not smallest
        if smallest != idx:

            # Swap positions
            self.pos[ self.array[smallest][0] ] = idx
            self.pos[ self.array[idx][0] ] = smallest

            # Swap nodes
            self.swapMinHeapNode(smallest, idx)

            self.minHeapify(smallest)

    # Standard function to extract minimum
    # node from heap
    def extractMin(self):

        # Return NULL wif heap is empty
        if self.isEmpty() == True:
            return

        # Store the root node
```

```
        root = self.array[0]

        # Replace root node with last node
        lastNode = self.array[self.size - 1]
        self.array[0] = lastNode

        # Update position of last node
        self.pos[lastNode[0]] = 0
        self.pos[root[0]] = self.size - 1

        # Reduce heap size and heapify root
        self.size -= 1
        self.minHeapify(0)

        return root

    def isEmpty(self):
        return True if self.size == 0 else False

    def decreaseKey(self, v, dist):

        # Get the index of v in  heap array

        i = self.pos[v]

        # Get the node and update its dist value
        self.array[i][1] = dist

        # Travel up while the complete tree is
        # not hepified. This is a O(Logn) loop
        while i > 0 and self.array[i][1] < self.array[(i
- 1) / 2][1]:

            # Swap this node with its parent
            self.pos[ self.array[i][0] ] = (i-1)/2
            self.pos[ self.array[(i-1)/2][0] ] = i
            self.swapMinHeapNode(i, (i - 1)/2 )

            # move to parent index
            i = (i - 1) / 2;

    # A utility function to check if a given
    # vertex 'v' is in min heap or not
    def isInMinHeap(self, v):

        if self.pos[v] < self.size:
            return True
        return False
```

```
def printArr(dist, n):
    print "Vertex\tDistance from source"
    for i in range(n):
        print "%d\t\t%d" % (i,dist[i])


class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    # Adds an edge to an undirected graph
    def addEdge(self, src, dest, weight):

        # Add an edge from src to dest.  A new node
        # is added to the adjacency list of src. The
        # node is added at the begining. The first
        # element of the node has the destination
        # and the second elements has the weight
        newNode = [dest, weight]
        self.graph[src].insert(0, newNode)

        # Since graph is undirected, add an edge
        # from dest to src also
        newNode = [src, weight]
        self.graph[dest].insert(0, newNode)

    # The main function that calulates distances
    # of shortest paths from src to all vertices.
    # It is a O(ELogV) function
    def dijkstra(self, src):

        V = self.V  # Get the number of vertices in
graph
        dist = []   # dist values used to pick minimum
                    # weight edge in cut

        # minHeap represents set E
        minHeap = Heap()

        #  Initialize min heap with all vertices.
        # dist value of all vertices
        for v in range(V):
            dist.append(sys.maxint)
```

```
            minHeap.array.append(
minHeap.newMinHeapNode(v, dist[v]) )
            minHeap.pos.append(v)

        # Make dist value of src vertex as 0 so
        # that it is extracted first
        minHeap.pos[src] = src
        dist[src] = 0
        minHeap.decreaseKey(src, dist[src])

        # Initially size of min heap is equal to V
        minHeap.size = V;

        # In the following loop, min heap contains all
nodes
        # whose shortest distance is not yet finalized.
        while minHeap.isEmpty() == False:

            # Extract the vertex with minimum distance
value
            newHeapNode = minHeap.extractMin()
            u = newHeapNode[0]

            # Traverse through all adjacent vertices of

            # u (the extracted vertex) and update their

            # distance values
            for pCrawl in self.graph[u]:

                v = pCrawl[0]

                # If shortest distance to v is not
finalized
                # yet, and distance to v through u is
less
                # than its previously calculated
distance
                if minHeap.isInMinHeap(v) and dist[u] !=
sys.maxint and \
                   pCrawl[1] + dist[u] < dist[v]:
                        dist[v] = pCrawl[1] + dist[u]

                        # update distance value
                        # in min heap also
                        minHeap.decreaseKey(v, dist[v])

        printArr(dist,V)
```

Time Complexity:

The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V + E)$ times (similar to BFS). The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is $O(E + V) * O(LogV)$ which is $O((E + V) * LogV) = O(ELogV)$. Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to $O(E + VLogV)$ using Fibonacci Heap. The reason is, Fibonacci Heap takes $O(1)$ time for decrease-key operation while Binary Heap takes $O(Logn)$ time.

Notes: 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated and use it show the shortest path from source to different vertices.

2) The code is for undirected graph, same dijekstra function can be used for directed graphs also.

3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.1 of algorithm).

4) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, Bellman-Ford can be used.