



**ALGORITHM AND DATA STRUCTURE PROJECT**  
**DOCUMENTATION REPORT**

PROJECT 2  
ITCS 6114

**DEPARTMENT OF COMPUTER SCIENCE SUBMITTED TO**  
**Dewan T. Ahmed, Ph.D.**

**SUBMITTED BY:**

**Harshitha Keshavaraju Vijayalakshmi**

**801084151**

**Urma Haldar**

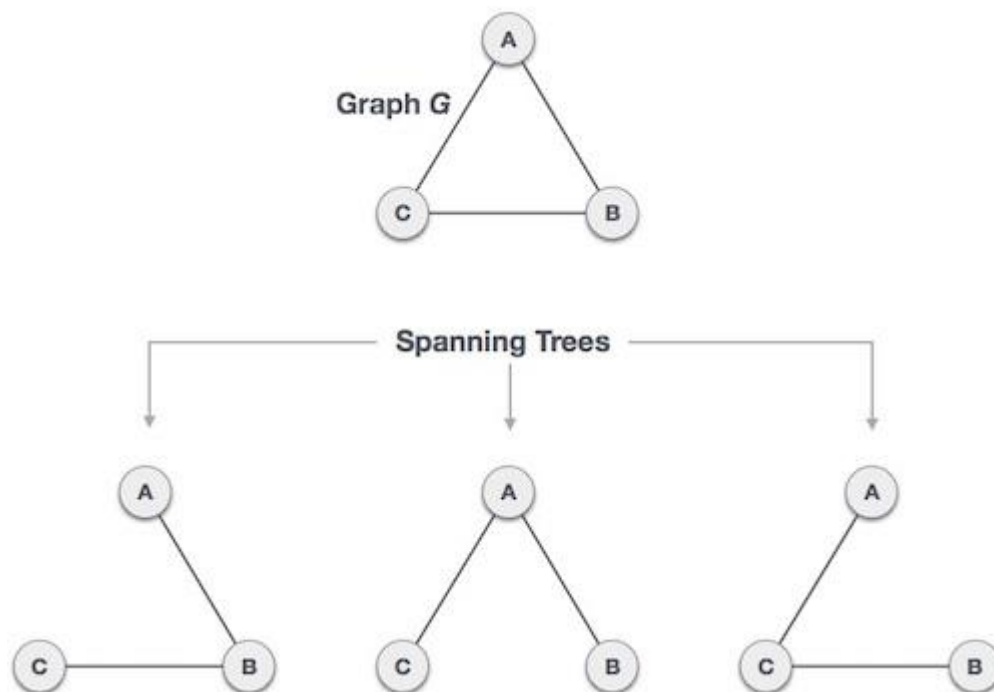
**801074045**

## Table of Contents:

<b>Data Structure &amp; Algorithms - Spanning Tree.....</b>	<b>2</b>
General Properties of Spanning Tree.....	2
Mathematical Properties of Spanning Tree.....	3
<b>Application of Spanning Tree.....</b>	<b>3</b>
Minimum Spanning Tree (MST).....	3
Minimum Spanning-Tree Algorithm.....	3
<b>Prim's Spanning Tree Algorithm.....</b>	<b>4</b>
Pseudo code .....	5
Source Code and output.....	9
<b>Dijkstra's Algorithm .....</b>	<b>12</b>
Relaxation.....	13
Pseudo code and implementation.....	14
Source code and output for directed and undirected. ....	18
Conclusion .....	28

# Data Structure & Algorithms - Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices secured with least conceivable number of edges. Thus, a traversing tree does not have cycles and it can't be disconnected. By this definition, we can reach a determination that each associated and undirected Graph G has something like one spanning tree. A disconnected diagram does not have any spanning tree, as it can't be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum  $n^{n-2}$  number of spanning trees, where  $n$  is the number of nodes. In the above addressed example,  $n$  is 3, hence  $3^{3-2} = 3$  spanning trees are possible.

## General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).

- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

## Mathematical Properties of Spanning Tree

- Spanning tree has  **$n-1$**  edges, where  **$n$**  is the number of nodes (vertices).
- From a complete graph, by removing maximum  **$e - n + 1$**  edges, we can construct a spanning tree.
- A complete graph can have maximum  **$n^{n-2}$**  number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph  $G$  and disconnected graphs do not have spanning tree.

## Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

## Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

## Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here –

- Kruskal's Algorithm
- Prim's Algorithm

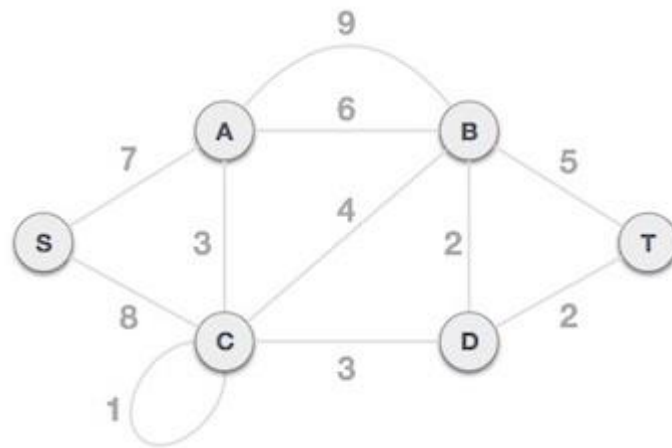
Both are greedy algorithms.

# Prim's Spanning Tree Algorithm

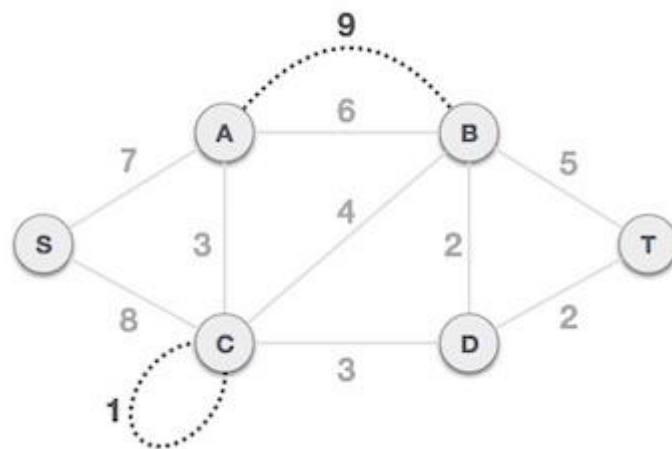
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

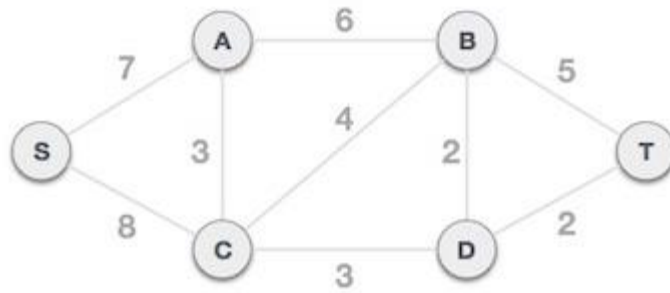
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

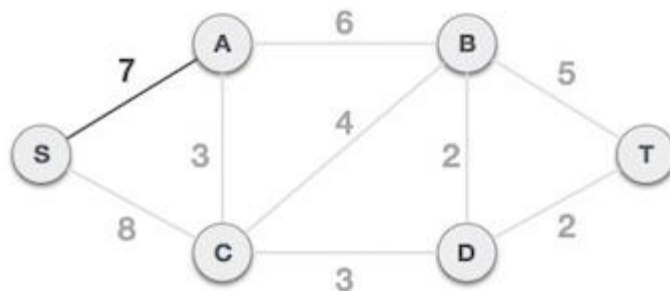


Step 2 - Choose any arbitrary node as root node

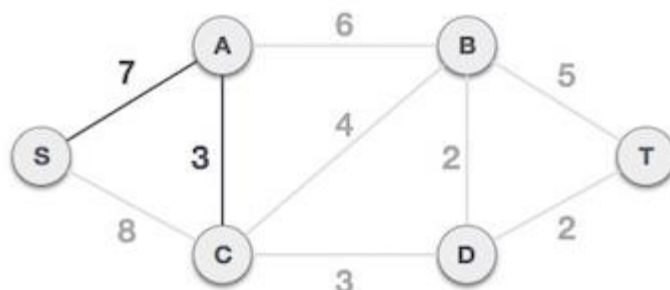
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

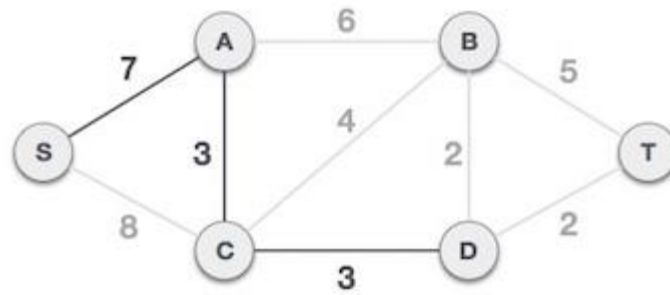
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



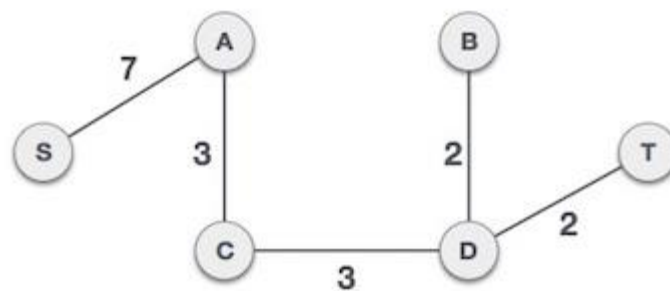
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

## Pseudocode for Prim's Algorithm:

```
int *MinimumSpanningTree( Graph g, int n, double **costs ) {
```

```
    Queue q;
```

```
    int u, v;
```

```
    int d[n], *pi;
```

```
    q = ConsEdgeQueue( g, costs );
```

```
    pi = ConsPredList( n );
```

```
    for(i=0;i<n;i++) {
```

```
        d[i] = INFINITY;
```

```
    }
```

```
    /* Choose 0 as the "root" of the MST */
```

```
    d[0] = 0;
```

```
    pi[0] = 0;
```

```
    while ( !Empty( q ) ) {
```

```

u = Smallest( g );
for each v in g->adj[u] {
    if ( (v in q) && costs[u][v] < d[v] ) {
        pi[v] = u;
        d[v] = costs[u][v];
    }
}
}
return pi;
}

```

The time complexity is  $O(V \log V + E \log V) = O(E \log V)$

Prim's algorithm grows a single tree until it becomes the minimum spanning tree and uses greedy approach.

## Source Code

```

from collections import defaultdict

from heapq import *

def prims( nodes, edges ):

    curr = defaultdict( list )

    for v1,v2,c in edges:

        curr[ v1 ].append( (c, v1, v2) )

        curr[ v2 ].append( (c, v2, v1) )

    mst = []

    used = set( nodes[ 0 ] )

    available_edges = curr[ nodes[0] ][:]

    heapify( available_edges )

```



```

while available_edges:

    cost, v1, v2 = heappop( available_edges )

    if v2 not in used:

        used.add( v2 )

        mst.append( ( v1, v2, cost ) )

        for q in curr[ v2 ]:

            if q[ 2 ] not in used:

                heappush( available_edges, q )

return mst

nodes =[]

varlist = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

varlist =list(varlist)

f1 = open(r'inp2_undirected.txt','r')

a = []

for i in f1.readlines():

    p=i.split()

    a.append(p)

cost = 0

length = len(a)

no_of_vertices =int(a[0][0])

no_of_edges = int(a[0][1])

undirected = str(a[0][2])

edges = a[1:length-1]

if(undirected == 'U'):

    print("The minimum spanning tree using Prim's Algorithm")

    print("The number of Vertices is :", no_of_vertices)

```

```
print("The number of Edges is :", no_of_edges)
```

```
for i in range(0,no_of_vertices):
```

```
    nodes.append(varlist[i])
```

```
out = prims( nodes, edges )
```

```
j=0
```

```
for i in out:
```

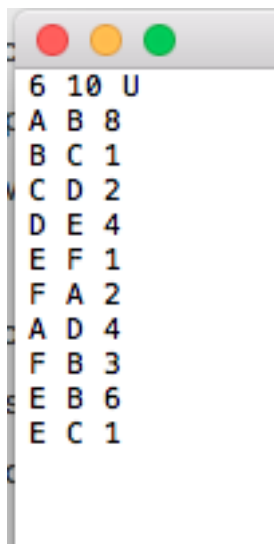
```
    print (i)
```

```
    cost += int(out[j][2])
```

```
    j +=1
```

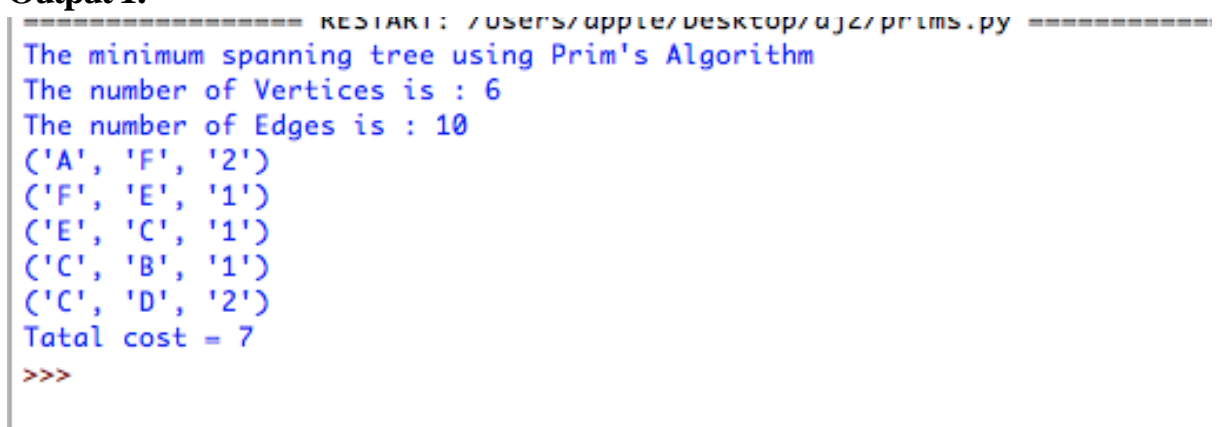
```
print("Tatal cost =" ,cost)
```

Input file 1:



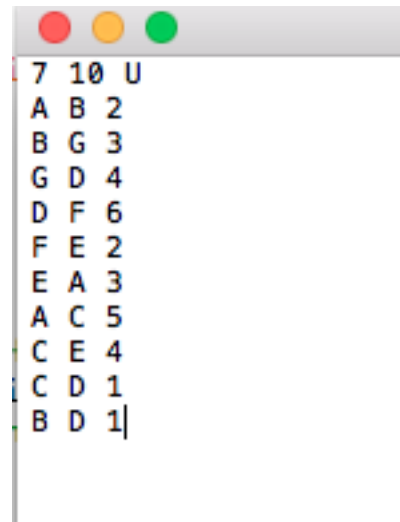
```
6 10 U
A B 8
B C 1
C D 2
D E 4
E F 1
F A 2
A D 4
F B 3
E B 6
E C 1
```

**Output 1:**



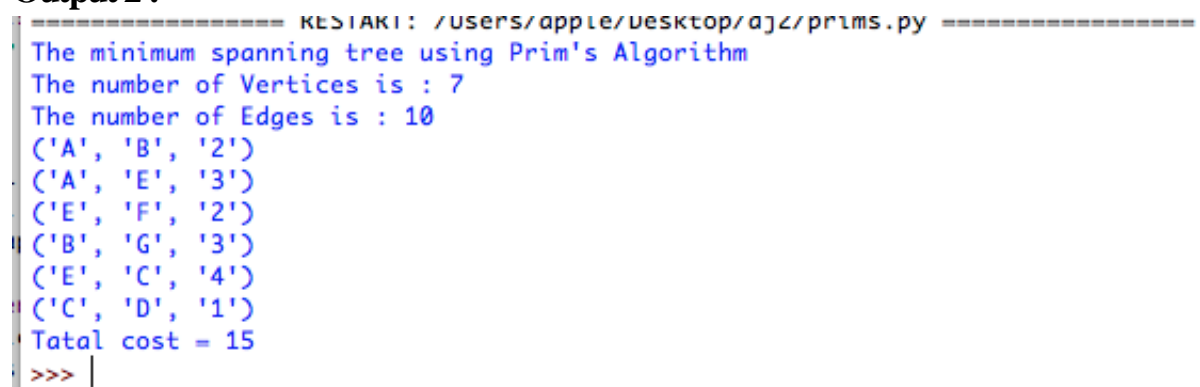
```
===== RESTART: /Users/apple/Desktop/aj2/prims.py =====
The minimum spanning tree using Prim's Algorithm
The number of Vertices is : 6
The number of Edges is : 10
('A', 'F', '2')
('F', 'E', '1')
('E', 'C', '1')
('C', 'B', '1')
('C', 'D', '2')
Total cost = 7
>>>
```

Input file 2:



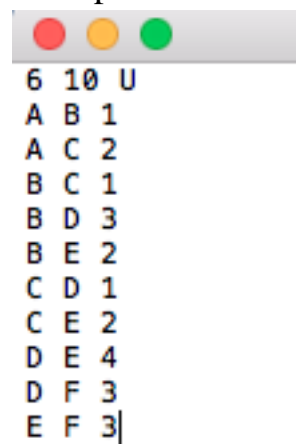
```
7 10 U
A B 2
B G 3
G D 4
D F 6
F E 2
E A 3
A C 5
C E 4
C D 1
B D 1|
```

Output 2:



```
===== RESTART: /users/apple/desktop/ajz/prims.py =====
The minimum spanning tree using Prim's Algorithm
The number of Vertices is : 7
The number of Edges is : 10
('A', 'B', '2')
('A', 'E', '3')
('E', 'F', '2')
('B', 'G', '3')
('E', 'C', '4')
('C', 'D', '1')
Total cost = 15
>>> |
```

File input 3:



```
6 10 U
A B 1
A C 2
B C 1
B D 3
B E 2
C D 1
C E 2
D E 4
D F 3
E F 3|
```

### Output 3:

```
The minimum spanning tree using Prim's Algorithm
```

```
The number of Vertices is : 6
```

```
The number of Edges is : 10
```

```
('A', 'B', '1')
```

```
('B', 'C', '1')
```

```
('C', 'D', '1')
```

```
('B', 'E', '2')
```

```
('D', 'F', '3')
```

```
Total cost = 8
```

```
>>> |
```

# Dijkstra's Algorithm

Dijkstra's algorithm (named after its discover, E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination. It turns out that one can find the shortest paths from a given source to *all* points in a graph in the same time, hence this problem is sometimes called the **single-source shortest paths** problem.

The somewhat unexpected result that *all* the paths can be found as easily as one further demonstrates the value of reading the literature on algorithms.

This problem is related to the spanning tree one. The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex. For a graph,

$$G = (V, E) \quad \text{where} \quad \begin{array}{l} \bullet \quad V \text{ is a set of vertices and} \\ \bullet \quad E \text{ is a set of edges.} \end{array}$$

Dijkstra's algorithm keeps two sets of vertices:

**S** the set of vertices whose shortest paths from the source have already been determined *and*

**V-** The remaining vertices

The other data structures needed are:

**d** array of best estimates of shortest path to each vertex

**pi** an array of predecessors for each vertex

The basic mode of operation is:

1. Initialise **d** and **pi**,
2. Set **S** to empty,
3. While there are still vertices in **V-S**,
  - i. Sort the vertices in **V-S** according to the current best estimate of their distance from the source,
  - ii. Add **u**, the closest vertex in **V-S**, to **S**,
  - iii. **Relax** all the vertices still in **V-S** connected to **u**

Relaxation :

The **relaxation** process updates the costs of all the vertices, **v**, connected to a vertex, **u**, if we could improve the best estimate of the shortest path to **v** by including (**u,v**) in the path to **v**.

The relaxation procedure proceeds as follows:

```
initialise_single_source( Graph g, Node s )
  for each vertex v in Vertices( g )
    g.d[v] := infinity
    g.pi[v] := nil
  g.d[s] := 0;
```

This sets up the graph so that each node has no predecessor (**pi[v] = nil**) and the estimates of the cost (distance) of each node from the source (**d[v]**) are infinite, except for the source node itself (**d[s] = 0**).

## Logic for the code and Implementation:

For all the programs we are taking graph inputs from a text file, as vertex, edges and weights. The data structures used are arrays, lists and heaps.

Runtime for djikstras is  $O(E \lg V)$  . Every time the main loop executes, one vertex is extracted from the queue. Assuming that there are  $V$  vertices in the graph, the queue may contain  $O(V)$  vertices. Each pop operation takes  $O(\lg V)$  time assuming the heap implementation of priority queues. So the total time required to execute the main loop itself is  $O(V \lg V)$ . In addition, we must consider the time spent in the function expand, which applies the function handle\_edge to each outgoing edge. Because expand is only called once per vertex, handle\_edge is only called once per edge. It might call push(v'), but there can be at most  $V$  such calls during the entire execution, so the total cost of that case arm is at most  $O(V \lg V)$ . The other case arm may be called  $O(E)$  times, however, and each call to increase\_priority takes  $O(\lg V)$  time with the heap implementation. Therefore the total run time is  $O(V \lg V + E \lg V)$ , which is  $O(E \lg V)$  because  $V$  is  $O(E)$  assuming a connected graph.

## Pseudocode for Djikstra's Algorithm:

(\* Djikstra's Algorithm \*)

```
let val q: queue = new_queue()
```

```
val visited: vertexMap = create_vertexMap()
```

```
fun expand(v: vertex) =
```

```
  let val neighbors: vertex list = Graph.outgoing(v)
```

```
    val dist: int = valOf(get(visited, v))
```

```
    fun handle_edge(v': vertex, weight: int) =
```

```
      case get(visited, v') of
```

```

SOME(d') =>
    if dist+weight < d'
    then ( add(visited, v', dist+weight);
           incr_priority(q, v', dist+weight) )
    else ()
| NONE => ( add(visited, v', dist+weight);
           push(q, v', dist+weight) )

in

app handle_edge neighbors

end

in

add(visited, v0, 0);

expand(v0);

while (not (empty_queue(q))) do expand(pop(q))

end

```

## Source code :

### For directed Dijkstra :

```

from collections import deque, namedtuple

```

```

# initialize all nodes to infinity

```

```

inf = float('inf')

```

```

Edge = namedtuple('Edge', 'start, end, cost')

```

```

def make_edge(start, end, cost=1):

```

```

    return Edge(start, end, cost)

```

```

class Graph:

    def __init__(self, edges):

        incorrect_edges = [i for i in edges if len(i) not in [2, 3]]

        if incorrect_edges:

            raise ValueError('Wrong edges data: {}'.format(incorrect_edges))

        self.edges = [make_edge(*edge) for edge in edges]

    @property
    def vertices(self):

        return set(

            sum(

                ([edge.start, edge.end] for edge in self.edges), []) )

    def get_node_pairs(self, n1, n2, both_ends=True):

        if both_ends:

            node_pairs = [[n1, n2], [n2, n1]]

        else:

            node_pairs = [[n1, n2]]

        return node_pairs

    def remove_edge(self, n1, n2, both_ends=True):

        node_pairs = self.get_node_pairs(n1, n2, both_ends)

        edges = self.edges[:]

        for edge in edges:

            if [edge.start, edge.end] in node_pairs:

                self.edges.remove(edge)

    def add_edge(self, n1, n2, cost=1, both_ends=True):

        node_pairs = self.get_node_pairs(n1, n2, both_ends)

```



```

for edge in self.edges:
    if [edge.start, edge.end] in node_pairs:
        return ValueError('Edge { } { } already exists'.format(n1, n2))
self.edges.append(Edge(start=n1, end=n2, cost=cost))
if both_ends:
    self.edges.append(Edge(start=n2, end=n1, cost=cost))

@property
def neighbours(self):
    neighbours = {vertex: set() for vertex in self.vertices}
    for edge in self.edges:
        neighbours[edge.start].add((edge.end, edge.cost))
    return neighbours

def dijkstra(self, source, dest):
    global distances
    assert source in self.vertices, 'Such source node doesnt exist'
    distances = {vertex: inf for vertex in self.vertices}
    pre_node = {
        vertex: None for vertex in self.vertices
    }
    distances[source] = 0
    vertices = self.vertices.copy()

    while vertices:
        present_node = min(
            vertices, key=lambda vertex: distances[vertex])
        vertices.remove(present_node)
        if distances[present_node] == inf:
            break

```

```

    for neighbour, cost in self.neighbours[present_node]:

        other_path = distances[present_node] + int(cost)

        if other_path < distances[neighbour]:

            distances[neighbour] = other_path

            pre_node[neighbour] = present_node

    path, present_node = deque(), dest

    while pre_node[present_node] is not None:

        path.appendleft(present_node)

        present_node = pre_node[present_node]

    if path:

        path.appendleft(present_node)

    return path,distances

```

```

f1 = open(r'inp2.txt','r')

distances = {}

a = []

var =[]

for i in f1.readlines():

    p=i.split()

    a.append(p)

```

```

inp =[]

length = len(a)

no_of_vertices =int( a[0][0])

no_of_edges = int(a[0][1])

directed = str(a[0][2])

graph = a[1:length-1]

```

```

for i in range(1,no_of_edges+1):

    if(a[i][0] not in inp):

        inp.append(a[i][0])

if(directed == 'D'):

    print("The number of Vertices is :",no_of_vertices)

    print("The number of Edges is :",no_of_edges)

    graph = Graph(graph)

    print("The path from the source to every other node in the directed \n")

    for q in range(1,len(inp)):

        out =graph.dijkstra("A",inp[q])

        print(out[0])

    print ("The shortest path code from source A to every other node is:\n ")

    print(distances)

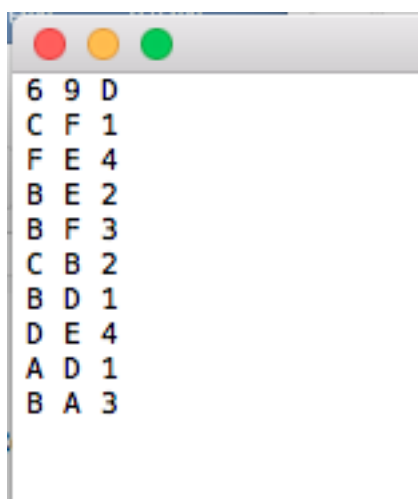
else:

    print(" This program is for directed graphs")

    exit()

```

Input file 1:



```

6 9 D
C F 1
F E 4
B E 2
B F 3
C B 2
B D 1
D E 4
A D 1
B A 3

```

**Output 1:**

```

===== RESIAKI: /Users/apple/desktop/ajz/dijkstra_directed.py =====
The number of Vertices is : 6
The number of Edges is : 9

NOTE :
The source node is A, we traverse to the destination node (could be any node),and then we also get the shortest path to all other nodes in the graph from A. If there is no path, it would be infinity, as it is a directed graph.
Directed Graph - Dijkstra's - Output:
The path from the source to every other node in the directed


deque([])
deque([])
deque(['A', 'D'])
deque([])
The shortest path cost from source A to every other node is:

{'C': inf, 'F': inf, 'A': 0, 'E': 5, 'D': 1, 'B': inf}
>>>

```

Here, the deque [] means there is no path from source to that particular node.

Input file 2:



```

5 8 D
A C 2
C B 6
D B 4
A B 3
A D 1
E C 2
E A 1
D E 3

```

Output 2:

```

1 The number of Vertices is : 5
2 The number of Edges is : 8
3
4 NOTE :
5 The source node is A, we traverse to the destination node (could be any node),and then we also get the shortest path to all other nodes in the graph from A. If there is no path, it would be infinity, as it is a directed graph.
6 Directed Graph - Dijkstra's - Output:
7 The path from the source to every other node in the directed
8
9 deque(['A', 'C'])
10 deque(['A', 'C'])
11 deque(['A', 'B'])
12 deque([])
13 deque(['A', 'D'])
14 deque(['A', 'D'])
15 deque(['A', 'D', 'E'])
16 The shortest path cost from source A to every other node is:
17 {'B': 3, 'C': 2, 'E': 4, 'A': 0, 'D': 1}
18
19

```

Input file 3:

```

5 10 D
A E 1
A B 2
B C 3
C D 4
D E 2
F A 4
F E 2
F D 1
F C 1
F B 3

```

Output 3:

```

===== RESTART: /Users/apple/Desktop/dj2/dijkstra_directed.py =====
The number of Vertices is : 5
The number of Edges is : 10

NOTE :
The source node is A, we traverse to the destination node (could be any node), and then we also get the shortest path to all other nodes in the graph from A. If there is no path, it would be infinity, as it is a directed graph.
Directed Graph - Dijkstra's - Output:
The path from the source to every other node in the directed

deque(['A', 'E'])
deque([])    Since there is no edge between A -> F hence its infinite
deque(['A', 'B'])
deque(['A', 'B'])
deque(['A', 'B', 'C'])
deque(['A', 'B', 'C'])
deque(['A', 'B', 'C', 'D'])
The shortest path cost from source A to every other node is:

{'C': 5, 'E': 1, 'F': inf, 'B': 2, 'A': 0, 'D': 9}
>>>

```

## For undirected graph :

```

from collections import defaultdict
from heapq import *

```

```

def dijkstra(edges, f, t):
    g = defaultdict(list)
    for l,r,c in edges:
        g[l].append((c,r))

```

```

q, seen, mins = [(0,f,())], set(), {f: 0}
while q:
    (cost,v1,path) = heappop(q)
    if v1 not in seen:
        seen.add(v1)
        path = (v1, path)
        print(path,cost)
        if v1 == t: return (cost, path)

    for c, v2 in g.get(v1, ()):
        if v2 in seen: continue
        prev = mins.get(v2, None)
        next = cost + int(c)
        if prev is None or next < prev:
            mins[v2] = next
            heappush(q, (next, v2, path))

return (dijkstra(edges,t,f))

f1 = open(r'inp2_undirected.txt','r')
a = []

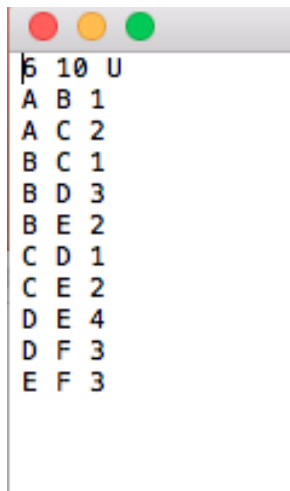
for i in f1.readlines():
    p=i.split()
    a.append(p)

inp = []
length = len(a)
no_of_vertices =int( a[0][0])
no_of_edges = int(a[0][1])
undirected = str(a[0][2])
edges = a[1:length-1]
if(undirected == 'U'):
    print("The number of Vertices is :",no_of_vertices)
    print("The number of Edges is :",no_of_edges)

    print ("=== Dijkstra ===")
    print("The path traversal is:")
    out = (dijkstra(edges, "E", "A"))
    #out2 = (dijkstra(edges, "A", "D"))
    print(out)

```


Input file 1:



## Output 1:

```
The source node is A, we traverse to the destination node (could be any node). Since, this is an undirected graph, the source to destination as well as reverse would have the same shortest path and we could traverse bothways. We print them according to all the subpaths and costs.
The number of Vertices is : 6
The number of Edges is : 10
('A', ()) 0
('B', ('A', ())) 1
The path traversal from source A to B
1
('A', ()) 0
The path traversal from source A to A
0
('A', ()) 0
('B', ('A', ())) 1
('C', ('A', ())) 2
The path traversal from source A to C
2
('A', ()) 0
('B', ('A', ())) 1
The path traversal from source A to B
1
('A', ()) 0
('B', ('A', ())) 1
('C', ('A', ())) 2
('D', ('C', ('A', ()))) 3
The path traversal from source A to D
3
('A', ()) 0
('B', ('A', ())) 1
The path traversal from source A to B
1
('A', ()) 0
('B', ('A', ())) 1
('C', ('A', ())) 2
('D', ('C', ('A', ()))) 3
('E', ('B', ('A', ()))) 3
The path traversal from source A to E
3
('A', ()) 0
('B', ('A', ())) 1
('C', ('A', ())) 2
('D', ('C', ('A', ()))) 3
The path traversal from source A to D
3
('A', ()) 0
('B', ('A', ())) 1
('C', ('A', ())) 2
('D', ('C', ('A', ()))) 3
('E', ('B', ('A', ()))) 3
('F', ('D', ('C', ('A', ()))) 6
The path traversal from source A to F
6
(6, ('F', ('D', ('C', ('A', ())))))
```

## Input file 2:



```
6 10 U
A B 8
B C 1
C D 2
D E 4
E F 1
F A 2
A D 4
F B 3
E B 6
E C 1
```

Output2 :

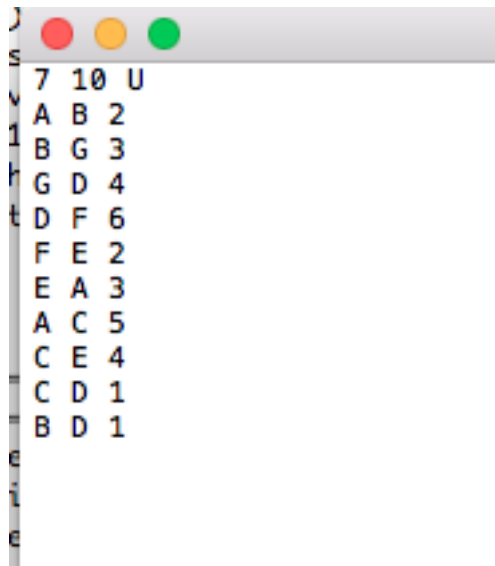


```

('A', ()) 0
('D', ('A', ())) 4
('B', ('A', ())) 8
The path traversal from source A to B
8
('A', ()) 0
('D', ('A', ())) 4
('B', ('A', ())) 8
('E', ('D', ('A', ()))) 8
('C', ('B', ('A', ()))) 9
The path traversal from source A to C
9
('A', ()) 0
('D', ('A', ())) 4
('B', ('A', ())) 8
('E', ('D', ('A', ()))) 8
('C', ('B', ('A', ()))) 9
The path traversal from source A to C
9
('A', ()) 0
('D', ('A', ())) 4
The path traversal from source A to D
4
('A', ()) 0
('D', ('A', ())) 4
The path traversal from source A to D
4
('A', ()) 0
('D', ('A', ())) 4
('B', ('A', ())) 8
('E', ('D', ('A', ()))) 8
The path traversal from source A to E
8
('A', ()) 0
('D', ('A', ())) 4
('B', ('A', ())) 8
('E', ('D', ('A', ()))) 8
The path traversal from source A to E
8
('A', ()) 0
('D', ('A', ())) 4
('B', ('A', ())) 8
('E', ('D', ('A', ()))) 8
('C', ('B', ('A', ()))) 9
('F', ('E', ('D', ('A', ()))) 9
The path traversal from source A to F
9
(9, ('F', ('E', ('D', ('A', ())))))

```

Input file 3:



OUTPUT 3:

```
===== RESIAKI: /users/apple/desktop/ajc/dijkstra_undirected.py =====
=== Dijkstra for undirected ===
NOTE :
The source node is A, we traverse to the destination node (could be any node). Since, this is an undirected graph, the source to destination as well as reverse would have the same shortest path and we could traverse bothways. We print them according to all the subpaths and costs.
The number of Vertices is : 7
The number of Edges is : 10
('A', ()) 0
('B', ('A', ())) 2
The path traversal from source A to B
2
('A', ()) 0
('B', ('A', ())) 2
_ _ _ _ _
```

The path traversal from source A to G

5

('A', ()) 0

('B', ('A', ())) 2

('C', ('A', ())) 5

('G', ('B', ('A', ()))) 5

('D', ('C', ('A', ()))) 6

The path traversal from source A to D

6

('A', ()) 0

('B', ('A', ())) 2

('C', ('A', ())) 5

('G', ('B', ('A', ()))) 5

('D', ('C', ('A', ()))) 6

The path traversal from source A to D

6

('A', ()) 0

('B', ('A', ())) 2

('C', ('A', ())) 5

('G', ('B', ('A', ()))) 5

('D', ('C', ('A', ()))) 6

('E', ('C', ('A', ()))) 9

('F', ('D', ('C', ('A', ()))) 12

The path traversal from source A to F

12

('A', ()) 0

('B', ('A', ())) 2

('C', ('A', ())) 5

('G', ('B', ('A', ()))) 5

('D', ('C', ('A', ()))) 6

('E', ('C', ('A', ()))) 9

('F', ('D', ('C', ('A', ()))) 12

The path traversal from source A to F

12

('A', ()) 0

('B', ('A', ())) 2

('C', ('A', ())) 5

('G', ('B', ('A', ()))) 5

('D', ('C', ('A', ()))) 6

('E', ('C', ('A', ()))) 9

The path traversal from source A to E

9

('A', ()) 0

The path traversal from source A to A

0

('A', ()) 0

('B', ('A', ())) 2

('C', ('A', ())) 5

The path traversal from source A to C

5

(5, ('C', ('A', ())))

## CONCLUSION

The Dijkstra's shortest path algorithm is the most commonly used to solve the single source shortest path problem today. For a graph  $G(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, the running time for finding a path between two vertices varies when different data structure are used. This project uses binary heap to implement Dijkstra's algorithm although there are some data structures that may slightly improve the time complexity.

Prim's (also known as Jarník's) algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

## REFERENCES

1. [Geeksforgeeks.com](https://www.geeksforgeeks.com)
2. [stackoverflow.com](https://stackoverflow.com)
3. [Wikipedia.org](https://en.wikipedia.org)