IIT Madras
BSc Degree

[Home](#) [Week-4](#) [Week-6](#)

PDSA - Week 5

Weighted Graph

Weighted directed graph

Adjacency matrix representation in Python

```
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in dedges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(W)
```

Adjacency list representation in Python

```python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),
(5,6,10)]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in dedges:
    WL[i].append((j,d))
print(WL)
```

Weighted undirected graph

Adjacency matrix representation in Python

```python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),
(5,6,10)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in edges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(W)
```

Adjacency list representation in Python

```python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),
(5,6,10)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in edges:
    WL[i].append((j,d))
print(WL)
```

Shortest Path

Single source shortest path algorithm

Find shortest paths from a fixed vertex to every other vertex.

- Dijkstra's Algorithm
- Bellman Ford algorithm

Working visualization of both algorithm

7 VISUALGO.NET / en / sssp

SINGLE-SOURCE SHORTEST PATHS LOGIN

We use cookies to improve our website.

By clicking ACCEPT, you agree to our use

of Google Analytics for analysing user

behaviour and improving user experience

as described in our Privacy Policy.

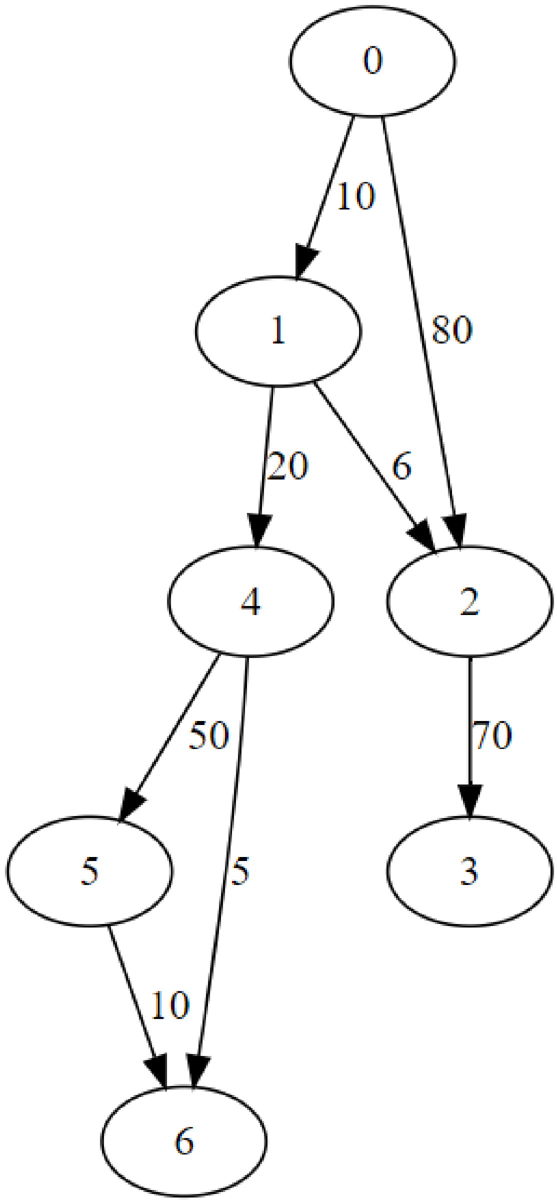
By clicking reject, only cookies necessary

Source:- https://visualgo.net/en/sssp

Dijkstra's Algorithm

- Dijkstra's algorithm works for both directed and undirected graphs.
- Dijkstra's algorithm doesn't work for graphs with negative weights or negative weight cycles.
- This algorithm returns the shortest distance from the source to all other nodes, but after some modification like maintaining parent information of each node we can find out the shortest path.

For given graph



For Adjacency matrix

```python
def dijkstra(WMat,s):
    (rows,cols,x) = WMat.shape
    infinity = np.max(WMat)*rows+1
    (visited,distance) = ({},{})
    for v in range(rows):
        (visited[v],distance[v]) = (False,infinity)

    distance[s] = 0

    for u in range(rows):
        nextd = min([distance[v] for v in range(rows) if not visited[v]])
        nextvlist = [v for v in range(rows)if (not visited[v]) and distance[v] == nextd]
        if nextvlist == []:
            break
        nextv = min(nextvlist)
        visited[nextv] = True
```

```
        for v in range(cols):
            if WMat[nextv,v,0] == 1 and (not visited[v]):
                distance[v] = min(distance[v],distance[nextv] + WMat[nextv,v,1])
    return(distance)


dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in dedges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(dijkstra(W,0))
```

Output

```
{0: 0, 1: 10.0, 2: 16.0, 3: 86.0, 4: 30.0, 5: 80.0, 6: 35.0}
```

Complexity

$O(n^2)$

For Adjacency list

```
def dijkstralist(WList,s):
    infinity = 1 + len(WList.keys())*max([d for u in WList.keys() for (v,d) in WList[u]])
    (visited,distance) = ({},{})
    for v in WList.keys():
        (visited[v],distance[v]) = (False,infinity)

    distance[s] = 0

    for u in WList.keys():
        nextd = min([distance[v] for v in WList.keys() if not visited[v]])
        nextvlist = [v for v in WList.keys() if (not visited[v]) and distance[v] == nextd]
        if nextvlist == []:
            break
        nextv = min(nextvlist)
        visited[nextv] = True
        for (v,d) in WList[nextv]:
            if not visited[v]:
                distance[v] = min(distance[v],distance[nextv]+d)
    return(distance)
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
WL = {}
for i in range(size):
```

```
    WL[i] = []
for (i,j,d) in dedges:
    WL[i].append((j,d))
print(dijkstralist(WL,0))
```

Output

```
{0: 0, 1: 10, 2: 16, 3: 86, 4: 30, 5: 80, 6: 35}
```

Complexity

$O(n^2)$

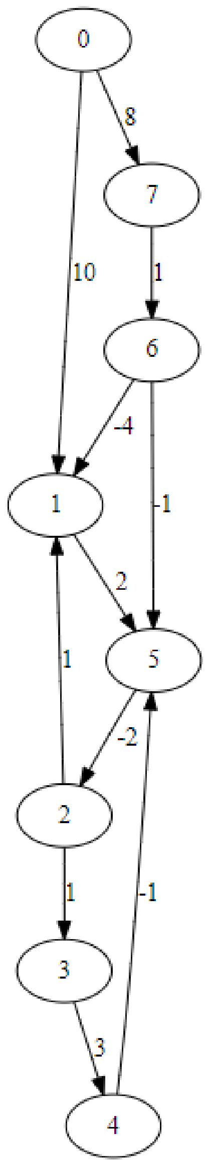
Bellman Ford algorithm

- Bellman-Ford works for both directed and undirected graphs with non-negative edges weights.
- Bellman-Ford does not work with an undirected graph with negative edges weight, as it will be declared as a negative weight cycle.
- Bellman-Ford works for a directed graph with negative edge weight, but not with negative weight cycle.

Working visualization

https://visualgo.net/en/sssp

For given graph



For adjacency matrix

```
def bellmanford(WMat,s):
    (rows,cols,x) = WMat.shape
    infinity = np.max(WMat)*rows+1
    distance = {}
    for v in range(rows):
        distance[v] = infinity

    distance[s] = 0

    for i in range(rows):
        for u in range(rows):
            for v in range(cols):
                if WMat[u,v,0] == 1:
                    distance[v] = min(distance[v], distance[u] + WMat[u,v,1])
    return(distance)
edges = [(0,1,10),(0,7,8),(1,5,2),(2,1,1),(2,3,1),(3,4,3),(4,5,-1),(5,2,-2),(6,1,-4),(6,5,-1),(7,6,1)]
size = 8
```

```
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in edges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(bellmanford(W,0))
```

Output

```
{0: 0, 1: 5.0, 2: 5.0, 3: 6.0, 4: 9.0, 5: 7.0, 6: 9.0, 7: 8.0}
```

Complexity

$O(n^3)$

For adjacency list

```
def bellmanfordlist(WList,s):
    infinity = 1 + len(WList.keys())*max([d for u in WList.keys() for (v,d) in WList[u]])
    distance = {}
    for v in WList.keys():
        distance[v] = infinity

    distance[s] = 0

    for i in WList.keys():
        for u in WList.keys():
            for (v,d) in WList[u]:
                distance[v] = min(distance[v], distance[u] + d)
    return(distance)
edges = [(0,1,10),(0,7,8),(1,5,2),(2,1,1),(2,3,1),(3,4,3),(4,5,-1),(5,2,-2),(6,1,-4),(6,5,-1),(7,6,1)]
size = 8
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in edges:
    WL[i].append((j,d))
print(bellmanfordlist(WL,0))
```

Output

```
{0: 0, 1: 5, 2: 5, 3: 6, 4: 9, 5: 7, 6: 9, 7: 8}
```

Complexity

$O(mn)$- where m is number of edges and n is number of vertices.

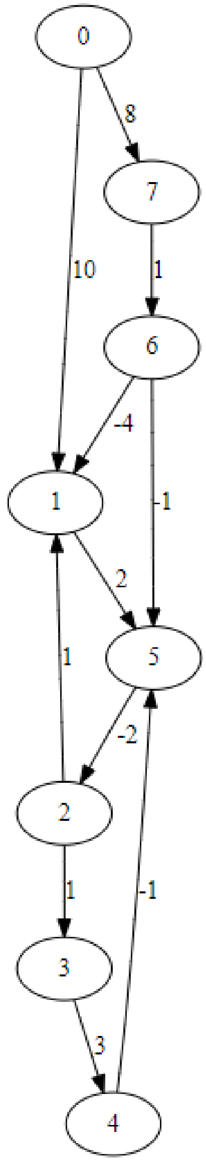
All pair of shortest path

- Find the shortest paths between every pair of vertices `i` and `j`.
- It is equivalent to if run Dijkstra or Bellman-Ford from each vertex.

Floyd-Warshall algorithm

- Floyd-Warshall's works for both directed and undirected graphs with non-negative edges weights.
- Floyd-Warshall's does not work with an undirected graph with negative edges weight, as it will be declared as a negative weight cycle.
- Floyd-Warshall’s algorithm is an alternative way to compute transitive closure `B k [i, j] = 1` if we can reach `j` from `i` using vertices in {0, 1, . . . , k−1}
- Floyd-Warshall works for a directed graph with negative edge weight, but not with a negative weight cycle.
- Formula for Floyd-Warshall algorithm is given below:-
- $$SP^k[i,j] = min[SP^{k-1}[i,j],\ SP^{k-1}[i,k] + SP^{k-1}[k,j]]$$

For given input graph



For adjacency matrix

```
def floydwarshall(WMat):
    (rows,cols,x) = WMat.shape
    infinity = np.max(WMat)*rows*rows+1

    SP = np.zeros(shape=(rows,cols,cols+1))
    for i in range(rows):
        for j in range(cols):
            SP[i,j,0] = infinity

    for i in range(rows):
        for j in range(cols):
            if WMat[i,j,0] == 1:
                SP[i,j,0] = WMat[i,j,1]

    for k in range(1,cols+1):
        for i in range(rows):
            for j in range(cols):
                SP[i,j,k] = min(SP[i,j,k-1],SP[i,k-1,k-1]+SP[k-1,j,k-1])

```

```
    return(SP[:,:,cols])
edges = [(0,1,10),(0,7,8),(1,5,2),(2,1,1),(2,3,1),(3,4,3),(4,5,-1),(5,2,-2),(6,1,-4),(6,5,-1),(7,6,1)]
size = 8
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in edges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(floydwarshall(W))
```

Output

```
[[641.   5.   5.   6.   9.   7.   9.   8.]
 [641.   1.   0.   1.   4.   2. 641. 641.]
 [641.   1.   1.   1.   4.   3. 641. 641.]
 [641.   1.   0.   1.   3.   2. 641. 641.]
 [638.  -2.  -3.  -2.   1.  -1. 638. 638.]
 [639.  -1.  -2.  -1.   2.   1. 639. 639.]
 [637.  -4.  -4.  -3.   0.  -2. 637. 637.]
 [638.  -3.  -3.  -2.   1.  -1.   1. 638.]]
```

Here all large value(>=637) representing no reachability from row index node to column index node.

Complexity

$O(n^3)$

Spanning Tree(ST)

- Retain a minimal set of edges so that graph remains connected
- Recall that a minimally connected graph is a tree
- Adding an edge to a tree creates a loop
- Removing an edge disconnects the graph
- Want a tree that connects all the vertices — **spanning tree**
- More than one spanning tree, in general

Minimum Cost Spanning Tree(MCST)

- Add the cost of all the edges in the tree
- Among the different spanning trees, choose one with minimum cost
- Some facts about trees
  - A tree on n vertices has exactly n − 1 edges
  - Adding an edge to a tree must create a cycle.
  - In a tree, every pair of vertices is connected by a unique path
- Algorithms:-
  - Prim's Algorithm
  - Kruskal's Algorithm

Working visualization of both algorithm

VISUALGO.NET / en /mst

MIN SPANNING TREE

LOGIN

We use cookies to improve our website.

By clicking ACCEPT, you agree to our use of Google Analytics for analysing user behaviour and improving user experience as described in our Privacy Policy.

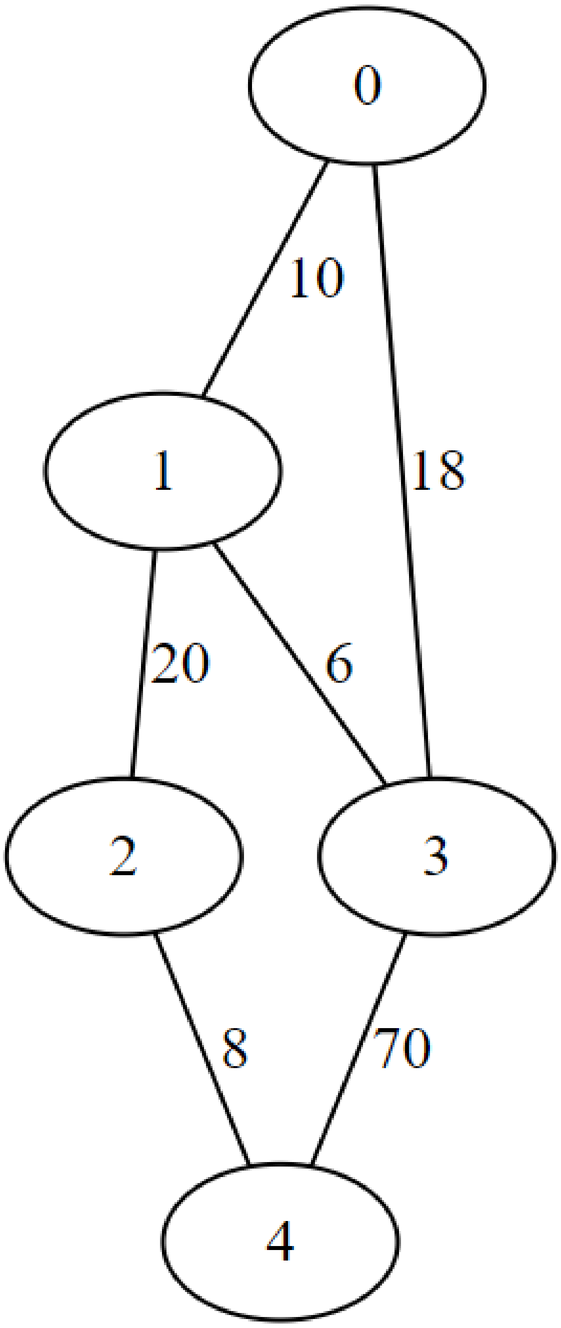
By clicking reject, only cookies necessary

Source:- https://visualgo.net/en/mst

Prim's Algorithm

- An implementation is similar to Dijkstra's algorithms, only update rule for distance is different.

For given input graph



For adjacency list

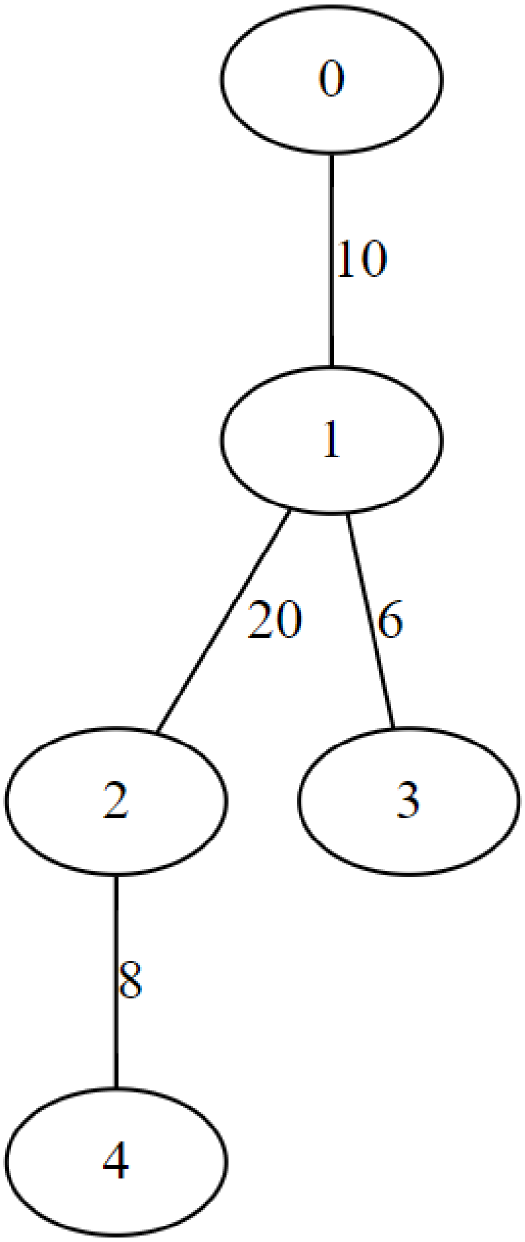
```
def primlist(WList):
    infinity = 1 + max([d for u in WList.keys()
                          for (v,d) in WList[u]])
    (visited,distance) = ({},{})
    for v in WList.keys():
        (visited[v],distance[v]) = (False,infinity)

    TreeEdges = []
    visited[0] = True
    for (v,d) in WList[0]:
        distance[v] = d

    for i in WList.keys():
        mindist = infinity
```

```python
        nextv = None
        for u in WList.keys():
            for (v,d) in WList[u]:
                if visited[u] and (not visited[v]) and d < mindist:
                    mindist = d
                    nextv = v
                    nexte = (u,v)

        if nextv is None:
            break

        visited[nextv] = True
        TreeEdges.append(nexte)
        for (v,d) in WList[nextv]:
            if not visited[v]:
                distance[v] = min(distance[v],d)
    return(TreeEdges)
dedges = [(0,1,10),(0,3,18),(1,2,20),(1,3,6),(2,4,8),(3,4,70)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 5
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in edges:
    WL[i].append((j,d))
print(primlist(WL))
```

Output

```
[(0, 1), (1, 3), (1, 2), (2, 4)]
```

Output minimum spanning tree with cost 44



or

```
def primlist2(WList):
    infinity = 1 + max([d for u in WList.keys()
                              for (v,d) in WList[u]])
    (visited,distance,nbr) = ({},{},{})
    for v in WList.keys():
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)

    visited[0] = True
    for (v,d) in WList[0]:
        distance[v] = d
        nbr[v] = 0

    for i in range(1,len(WList.keys())):
        nextd = min([distance[v] for v in WList.keys() if not visited[v]])
        nextvlist = [v for v in WList.keys() if (not visited[v]) and
distance[v] == nextd]
```

```python
        if nextvlist == []:
            break
        nextv = min(nextvlist)

        visited[nextv] = True
        for (v,d) in WList[nextv]:
            if not visited[v]:
                if d < distance[v]:
                    nbr[v] = nextv
                    distance[v] = d
    return(nbr)
dedges = [(0,1,10),(0,3,18),(1,2,20),(1,3,6),(2,4,8),(3,4,70)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 5
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in edges:
    WL[i].append((j,d))
print(primlist2(WL))
```

Output

```
{0: -1, 1: 0, 2: 1, 3: 1, 4: 2}
```

Complexity

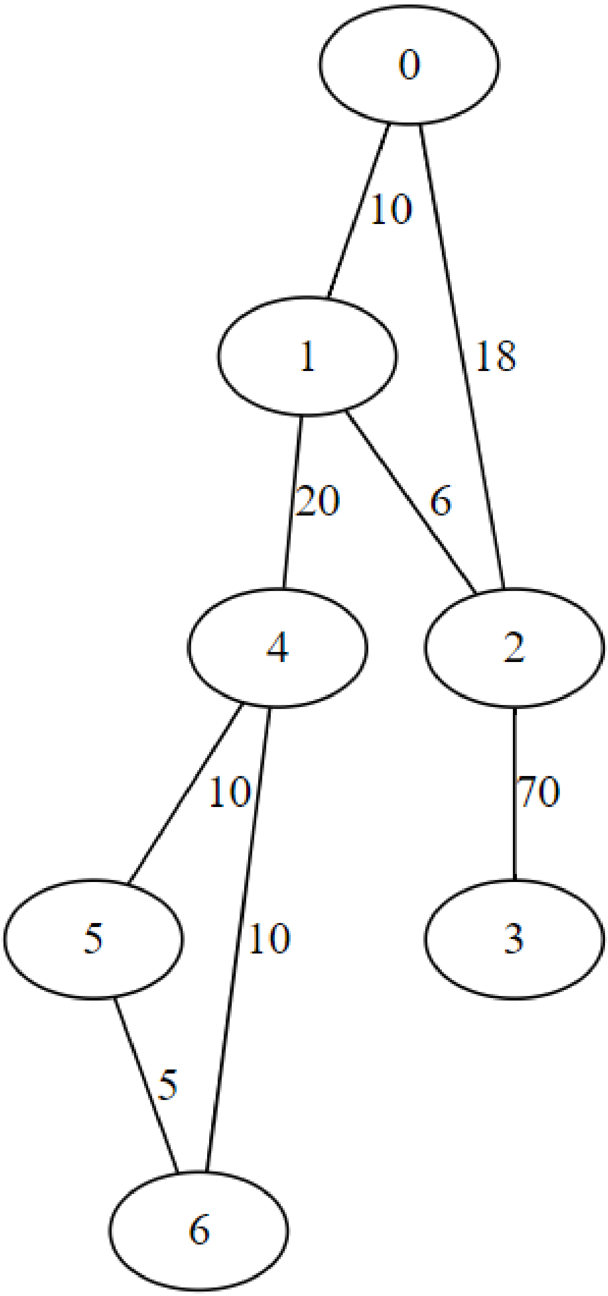
$O(n^2)$

Kruskal's Algorithm

Working visualization

https://visualgo.net/en/mst

For given input graph



For adjacency list

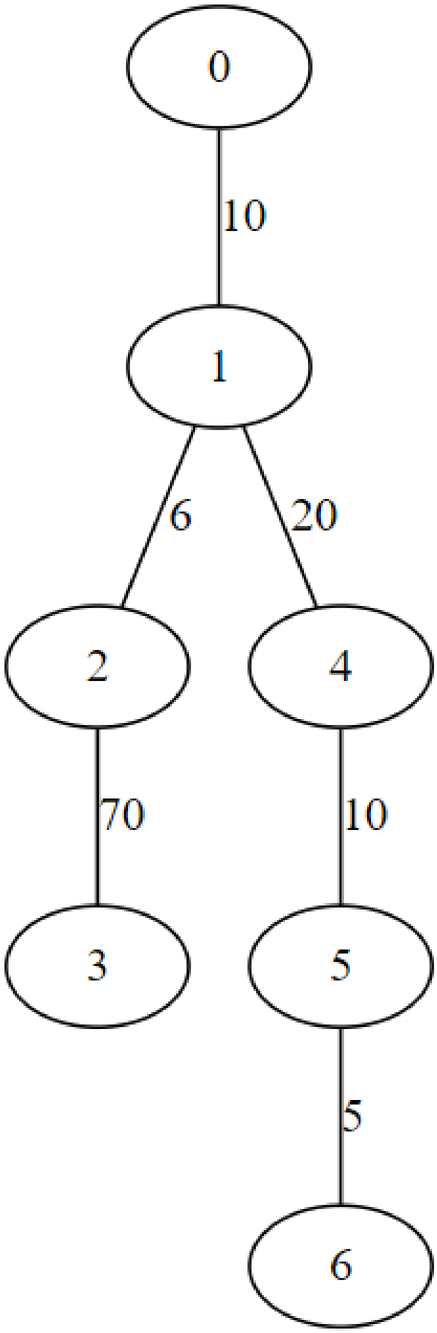
```
def kruskal(WList):
    (edges,component,TE) = ([],{},[])
    for u in WList.keys():
        # Weight as first component to sort easily
        edges.extend([(d,u,v) for (v,d) in WList[u]])
        component[u] = u
    edges.sort()
    #print(edges)

    for (d,u,v) in edges:
        if component[u] != component[v]:
            TE.append((u,v))
            c = component[u]
            for w in WList.keys():
                if component[w] == c:
```

```
                    component[w] = component[v]
    return(TE)
# Kruskak example
dedges = [(0,1,10),(0,2,18),(1,2,6),(1,4,20),(2,3,70),(4,5,10),(4,6,10),
(5,6,5)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in edges:
    WL[i].append((j,d))
print(kruskal(WL))
```

Output

```
[(5, 6), (1, 2), (0, 1), (4, 5), (1, 4), (2, 3)]
```

Output minimum spanning tree with cost 121



Complexity

$O(n^2)$