In [1]:
```
##########################
# Project 5
# ESE 572
# Routing Greedy Algorithm
##########################

import numpy as np
import math
import networkx as nx
import matplotlib.pyplot as plt
from heapq import heappush, heappop

###### Necessary Installations for Visual ######
# pip install networkx
# pip install decorator==5.0.9
# pip install --upgrade networkx
# pip install --upgrade matplotlib
###############################################

###### Basic Graph Visual Example ######
# G = nx.Graph()
# G.add_node(1)
# G.add_node(2)
# G.add_edge(1, 2)
# G.number_of_nodes()
# G.number_of_edges()
# nx.draw_networkx(G)
########################################
```

In [2]:
```
########################################
# Step 1: Minimum Spanning Tree
# Kruskal's Algorithm
# Undirected and weighted graph
########################################

class Graph:
    def __init__(self,nodes):
        self.nodes = nodes
        self.graph = []

    def add_e(self,source,dest,weight):
        self.graph.append([source,dest,weight])

    def search(self,parent,i):
        if parent[i] != i:
            # reassign node's parent to root node
            parent[i] = self.search(parent,parent[i])
        return parent[i]

    def union(self,parent,rank,x,y):
        # attach smaller tree to higher rank tree
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x
        else:
            parent[y] = x
            rank[x] += 1

    def kruskalMST(self):
        result = []
        pos = {0: (10, 20), 1: (15, 27.5), 2: (15, 22.5), 3: (15, 17.5), 4: (15, 12.5),
               5: (25, 27.5), 6: (25, 22.5), 7: (25, 17.5), 8: (25, 12.5), 9: (30, 20)}
        i = 0
        e = 0
        self.graph = sorted(self.graph, key=lambda x: x[2])
        parent = []
        rank = []

        for node in range(self.nodes):
            parent.append(node)
            rank.append(0)

        while e < self.nodes - 1:
            # pick smallest edge
            source,dest,weight = self.graph[i]
            i = i + 1
            x = self.search(parent,source)
            y = self.search(parent,dest)

            ns = [] # list of nodes to plot
            G_visual = nx.Graph()
            if x != y:
```

```
                e = e + 1
                result.append([source,dest,weight])
                self.union(parent,rank,x,y)

                for lst in result:
                    if lst[0] not in ns:
                        ns.append(lst[0])
                        G_visual.add_node(lst[0])
                    if lst[1] not in ns:
                        ns.append(lst[1])
                        G_visual.add_node(lst[1])

                iteration_cost = 0
                for lst in result:
                    G_visual.add_edge(lst[0], lst[1], weight=lst[2])
                    iteration_cost += lst[2]

                labels = nx.get_edge_attributes(G_visual,'weight')
                plt.figure()
                nx.draw_networkx(G_visual, pos)
                nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)
                plt.title('Iteration %d, %d Nodes, %d Edges, Iteration Cost: %d' %
                          (e, G_visual.number_of_nodes(), G_visual.number_of_edges(), iteration_cost),
                          fontsize=12)

        min_cost = 0
        print("Kruskal's Algorithm for Minimum Spanning Tree: \nMST Edges")
        print("source -- dest -- weight")
        for source,dest,weight in result:
            min_cost += weight
            print("  %d    --  %d   --   %d" % (source,dest,weight))
        print("Minimum Spanning Tree Total Weight:",min_cost)

    def dijkstra(self, source, end):
        pos = {0: (10, 20), 1: (15, 27.5), 2: (15, 22.5), 3: (15, 17.5), 4: (15, 12.5),
               5: (25, 27.5), 6: (25, 22.5), 7: (25, 17.5), 8: (25, 12.5), 9: (30, 20)}
        dist = [float('inf')] * self.nodes # Initialize distances to all nodes to infinity
        dist[source] = 0  # Set distance to source node to 0
        visited = [False] * self.nodes # Create an array to keep track of visited nodes
        path = [[] for _ in range(self.nodes)] # Create an array to keep track of the shortest path to each node
        pq = [(0, source)] # Create a priority queue to store nodes to visit
        it_num = 1
        while pq: # Loop through the priority queue until it's empty
            d, u = heappop(pq) # Pop the node with the smallest distance from the priority queue
            if visited[u]: # If the node has already been visited, skip it
                continue
            visited[u] = True # Mark the node as visited

            for edge in self.graph: # Update the distances to all neighboring nodes
                if edge[0] == u or edge[1] == u: # Check if the edge starts or ends at the current node
                    v = edge[1] if edge[0] == u else edge[0] # Get the other node in the edge
                    w = edge[2] # Get the weight of the edge
                    if dist[u] + w < dist[v]:
                        dist[v] = dist[u] + w
                        path[v] = path[u] + [u] # Add the neighboring node to the priority queue
                        heappush(pq, (dist[v], v)) # If we've reached the end node, we can stop searching
            if visited[end]:
                break

        full_path = path[end] + [end]
        drawn_edges = []
        for fp in range(len(full_path)-1):
            s_node = full_path[fp]
            d_node = full_path[fp+1]
            for lst in self.graph:
                if (lst[0]==s_node and lst[1]==d_node) or (lst[1]==s_node and lst[0]==d_node):
                    path_w = lst[2]
                    break
            drawn_edges.append([s_node,d_node,path_w])

            # Plotting iterations
            G_visual = nx.Graph()
            iteration_cost = 0
            for vn in range(len(visited)):
                G_visual.add_node(vn) # draw all nodes
            for each in drawn_edges:
                G_visual.add_edge(each[0], each[1], weight=each[2])
                iteration_cost += each[2]

            labels = nx.get_edge_attributes(G_visual,'weight')
            plt.figure()
            nx.draw_networkx(G_visual, pos)
            nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)
            plt.title('Iteration %d, Source: %d, Dest: %d, Iteration Cost: %d' %
                      (it_num, source, end, iteration_cost), fontsize=12)
            it_num += 1
```

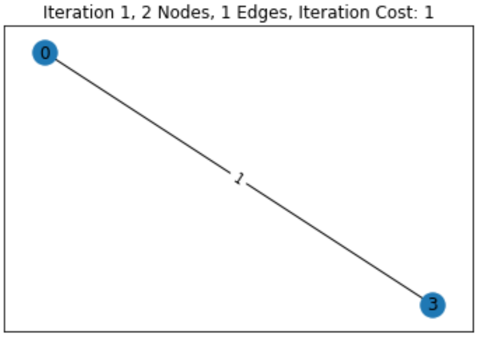
```
        return dist[end], path[end] + [end] # Return the shortest distance and the shortest path to the end node

if __name__ == '__main__':
    G = Graph(10)
    G.add_e(0,1,2)
    G.add_e(0,3,1)
    G.add_e(0,4,2)
    G.add_e(1,2,1)
    G.add_e(2,3,1)
    G.add_e(3,4,4)
    G.add_e(1,5,6)
    G.add_e(2,6,3)
    G.add_e(3,7,2)
    G.add_e(4,8,1)
    G.add_e(5,6,1)
    G.add_e(6,7,3)
    G.add_e(7,8,4)
    G.add_e(5,9,1)
    G.add_e(6,9,3)
    G.add_e(8,9,2)
```

In [3]:

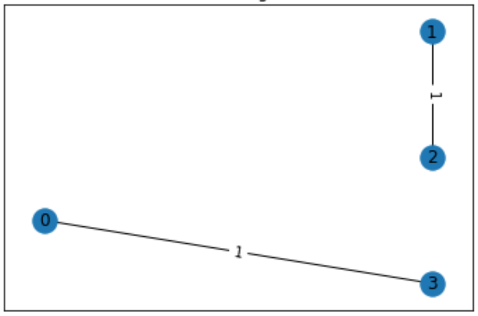
```
################################
# Step 1: Minimum Spanning Tree
# RESULTS
################################

G.kruskalMST()

# Kruskal's Algorithm for Minimum Spanning Tree:
# MST Edges
# source -- dest -- weight
#   0    --  3   --   1
#   1    --  2   --   1
#   2    --  3   --   1
#   4    --  8   --   1
#   5    --  6   --   1
#   5    --  9   --   1
#   0    --  4   --   2
#   3    --  7   --   2
#   8    --  9   --   2
# Minimum Spanning Tree Total Weight: 12
```

```
Kruskal's Algorithm for Minimum Spanning Tree:
MST Edges
source -- dest -- weight
  0    --  3   --   1
  1    --  2   --   1
  2    --  3   --   1
  4    --  8   --   1
  5    --  6   --   1
  5    --  9   --   1
  0    --  4   --   2
  3    --  7   --   2
  8    --  9   --   2
Minimum Spanning Tree Total Weight: 12
```

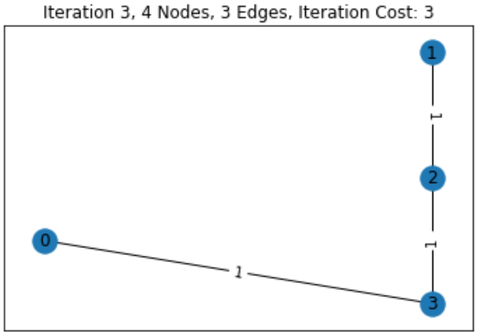
Iteration 1, 2 Nodes, 1 Edges, Iteration Cost: 1



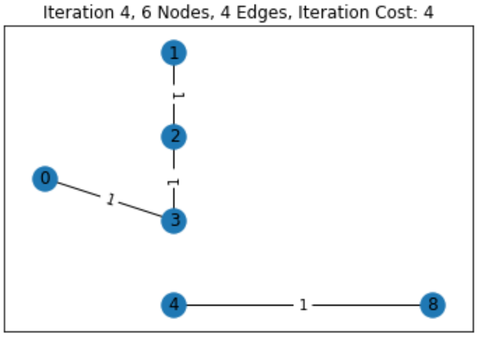
Iteration 2, 4 Nodes, 2 Edges, Iteration Cost: 2



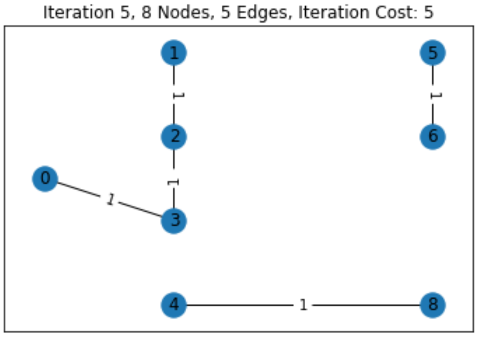
Iteration 3, 4 Nodes, 3 Edges, Iteration Cost: 3



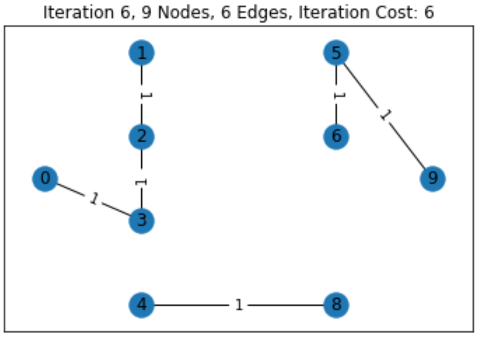
Iteration 4, 6 Nodes, 4 Edges, Iteration Cost: 4



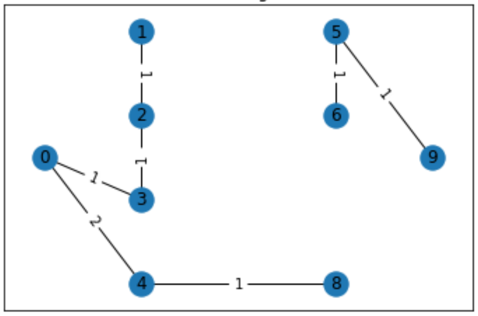
Iteration 5, 8 Nodes, 5 Edges, Iteration Cost: 5



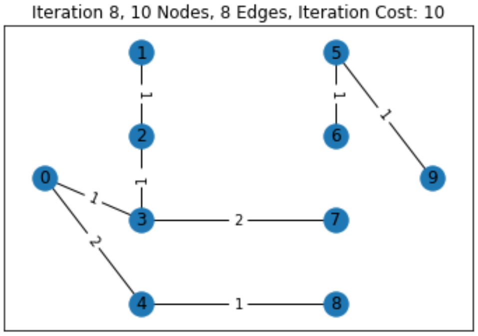
Iteration 6, 9 Nodes, 6 Edges, Iteration Cost: 6



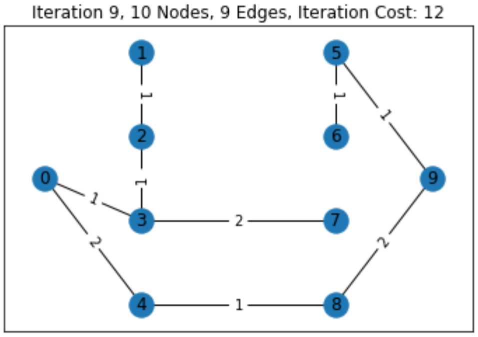
Iteration 7, 9 Nodes, 7 Edges, Iteration Cost: 8



Iteration 8, 10 Nodes, 8 Edges, Iteration Cost: 10



Iteration 9, 10 Nodes, 9 Edges, Iteration Cost: 12

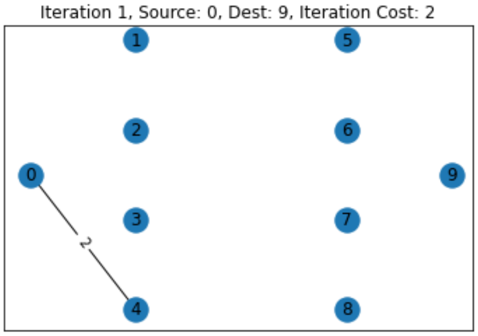


In [4]:

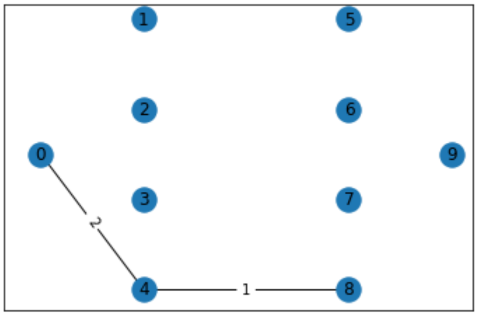
```
##########################################
# Step 2: Shortest Path Between Nodes
# Dijkstra's Algorithm
##########################################

# Nodes: (A,J), (B,I), (A,G), (E,F), (B,J), (D,G)
#        (0,9), (1,8), (0,6), (4,5), (1,9), (3,6)

# Show each iteration as part of determining the final routing patterns

print('A to J')
dist, path = G.dijkstra(0, 9)
print(f'Shortest path distance: {dist}')
print('Shortest path:', path)
print()
```

```
A to J
Shortest path distance: 5
Shortest path: [0, 4, 8, 9]
```

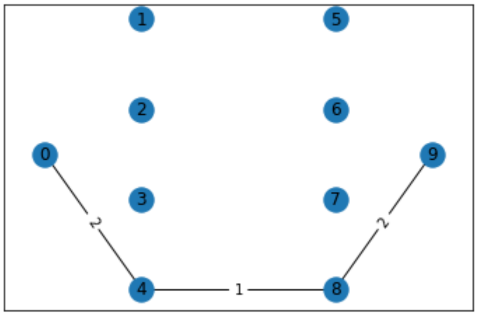
Iteration 1, Source: 0, Dest: 9, Iteration Cost: 2



Iteration 2, Source: 0, Dest: 9, Iteration Cost: 3



Iteration 3, Source: 0, Dest: 9, Iteration Cost: 5

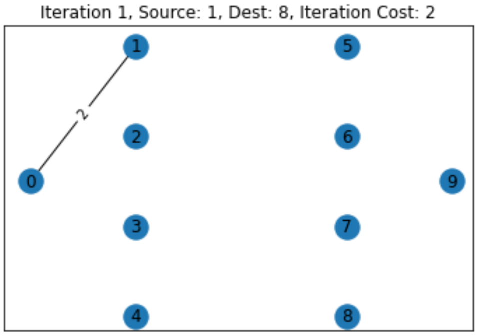


In [5]:

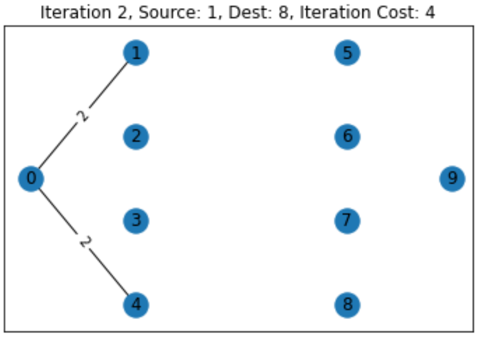
```
print('B to I')
dist, path = G.dijkstra(1, 8)
print(f'Shortest path distance: {dist}')
print('Shortest path:', path)
print()
```

```
B to I
Shortest path distance: 5
Shortest path: [1, 0, 4, 8]
```

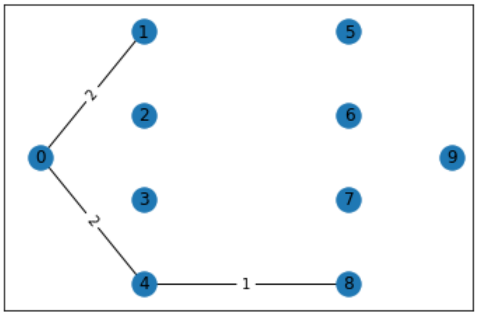
Iteration 1, Source: 1, Dest: 8, Iteration Cost: 2



Iteration 2, Source: 1, Dest: 8, Iteration Cost: 4



Iteration 3, Source: 1, Dest: 8, Iteration Cost: 5

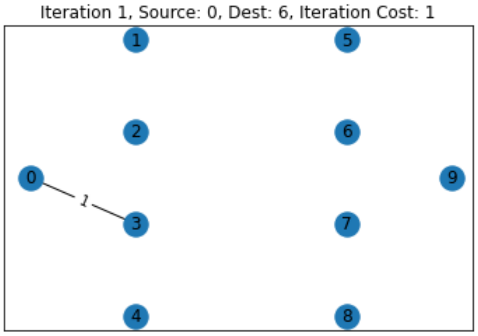


In [6]:

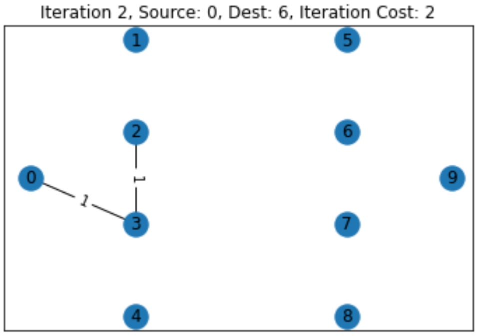
```
print('A to G')
dist, path = G.dijkstra(0, 6)
print(f'Shortest path distance: {dist}')
print('Shortest path:', path)
print()
```

```
A to G
Shortest path distance: 5
Shortest path: [0, 3, 2, 6]
```

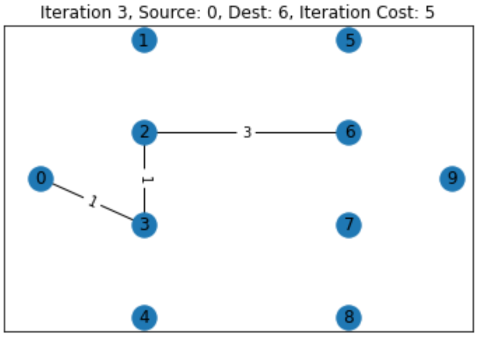
Iteration 1, Source: 0, Dest: 6, Iteration Cost: 1



Iteration 2, Source: 0, Dest: 6, Iteration Cost: 2



Iteration 3, Source: 0, Dest: 6, Iteration Cost: 5

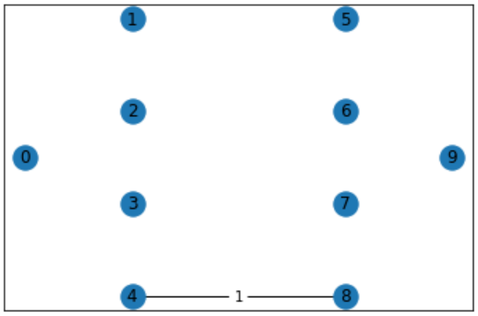


In [7]:

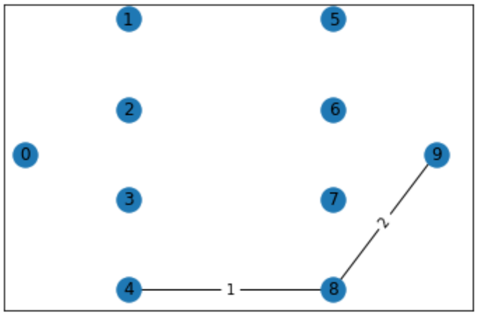
```
print('E to F')
dist, path = G.dijkstra(4, 5)
print(f'Shortest path distance: {dist}')
print('Shortest path:', path)
print()
```

```
E to F
Shortest path distance: 4
Shortest path: [4, 8, 9, 5]
```

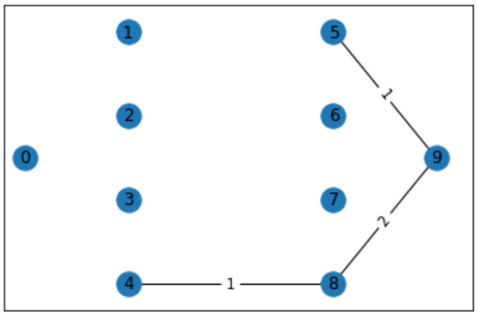
Iteration 1, Source: 4, Dest: 5, Iteration Cost: 1



Iteration 2, Source: 4, Dest: 5, Iteration Cost: 3



Iteration 3, Source: 4, Dest: 5, Iteration Cost: 4

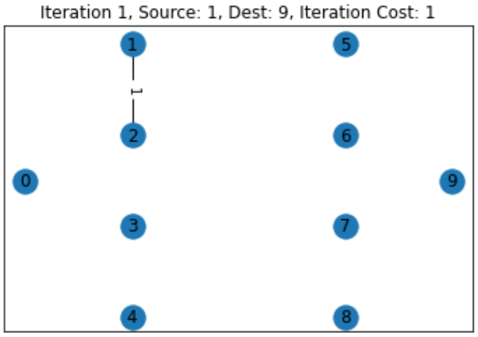


In [8]:

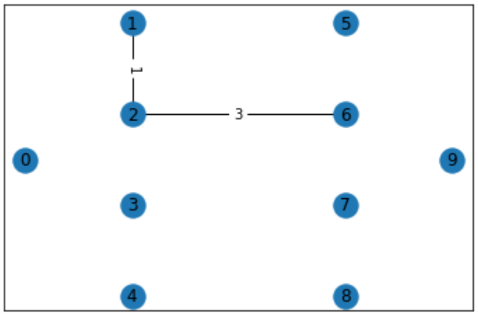
```
print('B to J')
dist, path = G.dijkstra(1, 9)
print(f'Shortest path distance: {dist}')
print('Shortest path:', path)
print()
```

```
B to J
Shortest path distance: 6
Shortest path: [1, 2, 6, 5, 9]
```

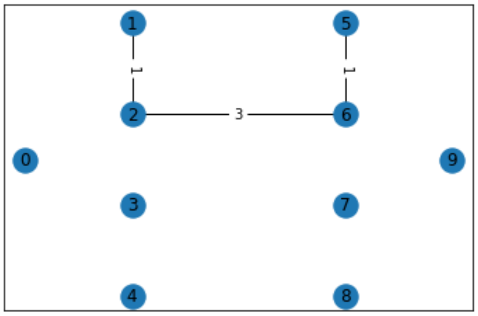
Iteration 1, Source: 1, Dest: 9, Iteration Cost: 1



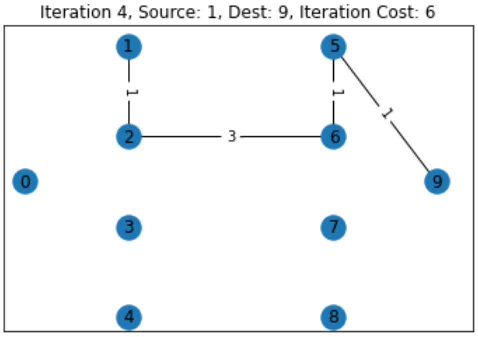
Iteration 2, Source: 1, Dest: 9, Iteration Cost: 4



Iteration 3, Source: 1, Dest: 9, Iteration Cost: 5



Iteration 4, Source: 1, Dest: 9, Iteration Cost: 6

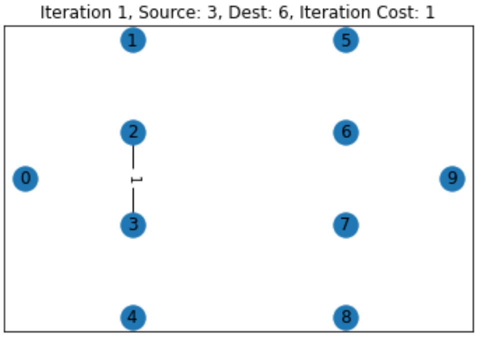


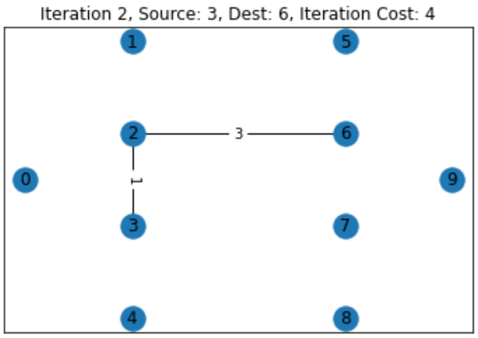
In [9]:

```
print('D to G')
dist, path = G.dijkstra(3, 6)
print(f'Shortest path distance: {dist}')
print('Shortest path:', path)
print()
```

```
D to G
Shortest path distance: 4
Shortest path: [3, 2, 6]
```

Iteration 1, Source: 3, Dest: 6, Iteration Cost: 1





In [10]:

```
################################
# Step 2: Shortest Path
# RESULTS
################################

# A to J
# Shortest path distance: 5
# Shortest path: [0, 4, 8, 9]

# B to I
# Shortest path distance: 5
# Shortest path: [1, 0, 4, 8]

# A to G
# Shortest path distance: 5
# Shortest path: [0, 3, 2, 6]

# E to F
# Shortest path distance: 4
# Shortest path: [4, 8, 9, 5]

# B to J
# Shortest path distance: 6
# Shortest path: [1, 2, 6, 5, 9]

# D to G
# Shortest path distance: 4
# Shortest path: [3, 2, 6]
```