```
import sys, copy
INFINITY = sys.maxint
#Set number of elements that affect the link cost between nodes
element_num = 1

class Vertex:
    #Define a class vertex to hold a node
    def __init__ (self, id):
        self.id = id
        #Set the length value to INFINITY
        self.length_value = INFINITY
        self.visited = False
        #Use a dictionary to hold neighbors nodes
        self.neighbors  = {}
        #Initialize the predecessor of the node
        self.predecessor = None

    def add_neighbor(self, neighbor, weight):
        #Add a neighbor with given weight
        self.neighbors[neighbor] = float(weight)

    def set_length_value(self, length):
        #Set the length value of the node
        self.length_value = length

    def set_visited(self):
        self.visited = True

    def set_predecessor(self,prev):
        self.predecessor = prev
```

```
    def get_id(self):
        return self.id

    def get_neighbors(self):
        return self.neighbors.keys()

    def get_weight(self, neighbor):
        return self.neighbors[neighbor]

    def get_length_value(self):
        return self.length_value

    def get_predecessor(self):
        return self.predecessor

class Graph:
    #Define a Graph class that holds vertices
    def __init__(self):
        self.vertices = {}
        self.vert_count = 0

    def add_vertex(self, vertex_id):
        self.vert_count += 1
        node = Vertex(vertex_id)
        self.vertices[vertex_id] = node

    def add_edge(self, node1, node2, link_cost):
        if node1 not in self.vertices:
            self.add_vertex(node1)
```

```
        if node2 not in self.vertices:
            self.add_vertex(node2)

        self.vertices[node1].add_neighbor(self.vertices[node2], link_cost)
        self.vertices[node2].add_neighbor(self.vertices[node1], link_cost)

    def get_vertex(self, node):
        if node not in self.vertices:
            return None
        else:
            return self.vertices[node]

    def get_vertices(self):
        return self.vertices.keys()

def dijkstra(Network, start):
    current = Network.get_vertex(start)
    visited = 1
    current.set_length_value(0)
    current.set_visited()
    while (visited < len(Network.get_vertices())):
        min_dist = INFINITY
        for neighbor in current.get_neighbors():
            if neighbor.visited:
                continue
            else:
                current.set_visited()
                dist = current.get_length_value() +
neighbor.get_weight(current)
                if dist < neighbor.get_length_value():
```

```
                    neighbor.set_length_value(dist)
                    neighbor.set_predecessor(current)
                if min_dist > dist:
                    min_dist = dist
                    next_vert = neighbor
        current = next_vert
        visited += 1
    #Using Dijstra algorithm to create routing tables for entries
    return 0

def shortest_path(Network, dest, path):
    cost = dest.get_length_value()
    while dest.get_predecessor():
        path.append(dest.get_predecessor().get_id())
        dest = dest.get_predecessor()
    return cost

def link_cost_calc(weight):
    #Calculate the total weight of a link

    return weight

def input_process(filename):
    #Process the input and parse it into a data structure
    Network = Graph()
    fopen = open(filename,'r')
    for line in fopen:
        elem = line.strip("\n").split(" ")
        Network.add_edge(elem[0],elem[1],link_cost_calc(elem[2]))
```

```
    return Network


def output(g_ini, fout):
    for destination in g_ini.get_vertices():
        #Deep copy the graph to find routing table for each destination
        g = copy.deepcopy(g_ini)
        dijkstra(g, destination)
        fout.write("Destination: " + str(destination) + "\n")
        for node in g.get_vertices():
            path = []
            target = g.get_vertex(node)
            path.append(node)
            cost = shortest_path(g, target, path)
            if len(path) > 1:
                fout.write("Source: " + node + "\t\t Destination: " +
destination + "\t \t Route: " + str(path[1]) + "\t\t Cost: " + str(cost) +
"\n")
            else:
                fout.write("Source: " + node + "\t\t Destination: " +
destination + "\t \t Route: -1" + "\t\t Cost: " + str(cost) + "\n")
        fout.write("\n")
    #Output the results into a file
    return 0


def main():
    #Initialize variables
    input_file = "zero.net"
    output_file = "results.net"
    fout = open (output_file,"w")
    #Initial graph
    g_ini = Graph()
```

```
    #Process the input to add the edges of the graph
    g_ini = input_process(input_file)
    output(g_ini, fout)
    #Call main functions

    return 0
main()
```