1. Function `WeightedPath(strArr)` takes `strArr`, an array of strings, of the following structure as its input. The first element in the array is the number of nodes N in the graph. The next N elements are node names (e.g. A, B, C .. BS, MS .. etc.). The rest of the elements in the array are the connections between all of the nodes along with their weights separated by a pipe symbol (|): A|B|3, B|C|12 .. BS|MS|14 .. etc. The graph is undirected and node weights are always positive. Note, there may exist no connections at all.

Your program should implement Dijkstra’s algorithm and return the shortest path (on the graph encoded by `strArr`) from the first node in the list to the last node in the list. There will only ever be one shortest path for any given graph. If no path between the first and the last node exists, your code should return -1. The array will have at least two nodes.

Note. File `./q1/utilities.py` contains useful implementations of: (i) a priority queue based on Heap data structure, (ii) a `Node` class storing properties such as `parent`, `neighbours`, `visited` flag and shortest `distance` to this node as well as other useful properties, and (iii) a simple function `adj_mat_and_s_e_from_str` which convert string array (`strArr`) into a tuple containing the graph adjacency list `G` and start node `s` and end node `e` (see code below).

**Examples**

Input:
```
```
Desired output: A-B-D

Input:
```
[“4”, “x”, “y”, “z”, “w”, “x|y|2”, “y|z|14”, “z|y|31”]
```
Desired output: -1

(a) Extend function `Dijkstra` in order to solve the aforementioned task. [40%]
In [2]:

from utilities import Heap, Node, adj_mat_and_s_e_from_str

def Dijkstra(G, s, e):
    #G - adjacency list, s - start node, e - end node.
    # Returns a tuple (distance, path), where distance is the length of the
    # shortest path and **path** is a string containing node names in the
    # shortest path from s to e, separated by "-". If no path exists, it
    #returns None.

    nodes = {name: Node(name, neighbors for name, neighbors in G.items())
             for name, neighbors in G.items()}
    nodes[s].distance = 0
    pq = Heap()
    for node in nodes.values():
        pq.push(node, node.distance)

    while len(pq):
        node = pq.pop()[0]
        node.visited = True
        if node.name == e:
            break
        for name in node.neighbors:
            neighbor = nodes[name]
            if (not neighbor.visited and
                (node.distance + node.neighbors[name] < neighbor.distance)
                neighbor.distance = node.distance + node.neighbors[name]
                neighbor.parent = node
                pq.push(neighbor, neighbor.distance)

    path = []
    node = nodes[e]
    if not node.parent:
        return None
    while node:
        path.append(node)
        node = node.parent
    return (nodes[e].distance, '-'.join([node.name for node in path[::-1]]))

In [3]:

Example Test Case - 1

In [4]:

Automatic Evaluation

Function evaluate_solution evaluates your function ShortestPath on 10 pre-set test cases. Note, when
debugging you may want to run your code on a subset of tests.

Do not forget to run on all test cases when your final implementation is finished!

Example Test Case - 2

In [3]:

Example Test Case - 2

In [4]:

input_value = ["4","x","y","z","w","x|y|2","y|z|14","z|y|31"]
2 print (ShortestPath(input_value))

-1
(b) Answer questions about the graph illustrated below:

(i) List all strongly connected components of this graph.
   
   Obtained output:
   c-a-b-e
   <<CORRECT>>
   
   Running question 1, part 1, test id: 10
   Input:
   Obtained output:
   C-E-B-H
   <<CORRECT>>
   
   Total correct outputs:
   10 out of 10

(ii) Calculate the maximum flow from node A to node F.

   Augmented paths:
   A-C-E-F -> 4
   A-C-E-D-F -> 7
   A-B-D-F -> 12

   Total flow: 23

(iii) Assume this graph is transformed into an undirected graph by replacing all directed edges with equivalent undirected edges of same corresponding weights. Write out the edges of the Minimum Spanning Tree discovered by Kruskal’s algorithm in the same order as they would be found.

   Obtained output:
   C-B, E-F, E-D, D-C, A-C
(c) Provide brief answers to questions below: [20%]

(i) Assume that you are provided with an undirected graph which contains both positive and negative edges as well as an implementation of Prim’s Minimum Spanning Tree (MST) algorithm, which only accepts graphs with non-negative edge weights. Can you still obtain the MST for this graph without modifying the Prim’s algorithm implementation? Explain how.

Modify the graph by simply adding a positive constant C to all edges where C is the weight of the negative weight edge of largest magnitude in the original graph G. Then the total edge weight of the MST of the new graph will be equal to M+C(V-1). Here M is the total edge weight of the original graph and V is number of vertices (assuming all vertices are connected).

(ii) What are the advantages and disadvantages of the Bellman-Ford algorithm in comparison to Dijkstra’s algorithm?

Advantages: (1) It can be applied to graphs with non-negative edges. (2) It can also be used to detect negative-weight cycles.

Disadvantages: Bellman-Ford algorithm has a higher runtime - O(VE). Runtime of Dijkstra’s algorithm is O(V log V + E).

(iii) What is the order of growth of the memory used to represent a graph with V vertices and E edges using the adjacency-lists representation and the adjacency-matrix representation?

Adjacency list - O(V+E). Adjacency matrix - O(V^2).

(iv) Given a digraph where each edge is colored black or orange and two vertices s and t. Is it possible to find a path from s to t that uses the fewest number of black edges in time proportional to E + V in the worst case? Justify your answer.

This can be solved by a modified version of breadth-first search, where the vertices discovered by following black edges are put at the back of the queue (as usual) and the vertices discovered by following orange edges are put at the front of the queue.

(d) Extend your algorithm described in part (a) to find the second shortest path in the graph. [25%]

Inputs and expected outputs have the same format as in part (a).

```
Examples
Output: A-B-C-D

Input: ["4","x","y","z","w","A|y|2","y|z|14","z|y|81"]
Output: -1
```

```
In [5]:
def SecondShortestPath(strArr):
    G, s, e = adj_mat_and_s_e_from_strArr
    LARGE=100000
    shortest_path = Dijkstra(G, s, e)
    if not (shortest_path):
        return "-1"
    p = shortest_path[1].split('-')
    shortest_2nd=None
    cost=0
    for i in range(0,len(p)-1):
        tmp_edge=G[p[i]][p[i+1]]
        G[p[i]][p[i+1]]=LARGE
        new_p = Dijkstra(G,p[i],e)
        G[p[i]][p[i+1]]=tmp_edge
        if new_p and (shortest_2nd is None or shortest_2nd[0]>cost+new_p[0]):
            shortest_2nd = (cost+new_p[0], '-'.join(p[i]+new_p[1].split('-')))  
            cost=cost+tmp_edge
            if(cost>shortest_2nd[0]>LARGE):
                return "-1"
        else:
            return shortest_2nd[1]
```

Example test case - 1

```
In [6]:
1 input_value = ["4","A","B","C","D", "A|B|2", "B|C|11", "C|D|3", "B|D|2"]
2 print (SecondShortestPath(input_value))
```

A-B-C-D

Example test case - 2
Automatic evaluation

Function `evaluate_solution` evaluates your function `SecondShortestPath` on 10 pre-set test cases. Note, when debugging you may want to run your code on a subset of tests.

Do not forget to run on all test cases when your final implementation is finished!
Obtained output:
c-a-d-e
<<CORRECT>>
Running question 1, part 2, test id: 10
Input:
['8', 'C', 'B', 'A', 'D', 'E', 'F', 'G', 'H', 'C\|D\|1', 'D\|F\|2', 'G\|F\|2', 'G\|E\|1', 'E\|B\|1', 'H\|B\|1', 'C\|A\|13', 'B\|A\|2', 'C\|E\|1']
Obtained output:
C-D-F-G-E-B-H
<<CORRECT>>

----------------------------------
Total correct outputs:
10 out of 10