CHARLOTTE

ITCS 6114-Algorithms and Data Structures

Project 2

Graph Algorithms and Related Data Structures

Submitted by

Srushti Khot (801203532)

Sanket Revadigar (801203510)

Instructor

Prof. Dewan T. Ahmed, Ph.D.

PROBLEM 1: SINGLE-SOURCE SHORTEST PATH ALGORITHM

1 .1 INTRODUCTION:

Find shortest path tree in both directed and undirected weighted graphs for a given source vertex. Assume there is no negative edge in your graph. You will print each path and path cost for a given source.

Single-source Shortest Path

A connected weighted directed graph G (V, E), with each edge ⟨u, v⟩∈E and weight w (u, v). The *single source shortest paths* problem is to find a shortest path from a given source r to every other vertex v∈V-{r}. The weight of a path p=⟨ v0, v1 …, vk ⟩ is the sum of the weights of its constituent edges:

$$w(p)= \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

The weight of a shortest path from u to v is defined by δ (u, v) =min{w(p): p is a path from u to v}

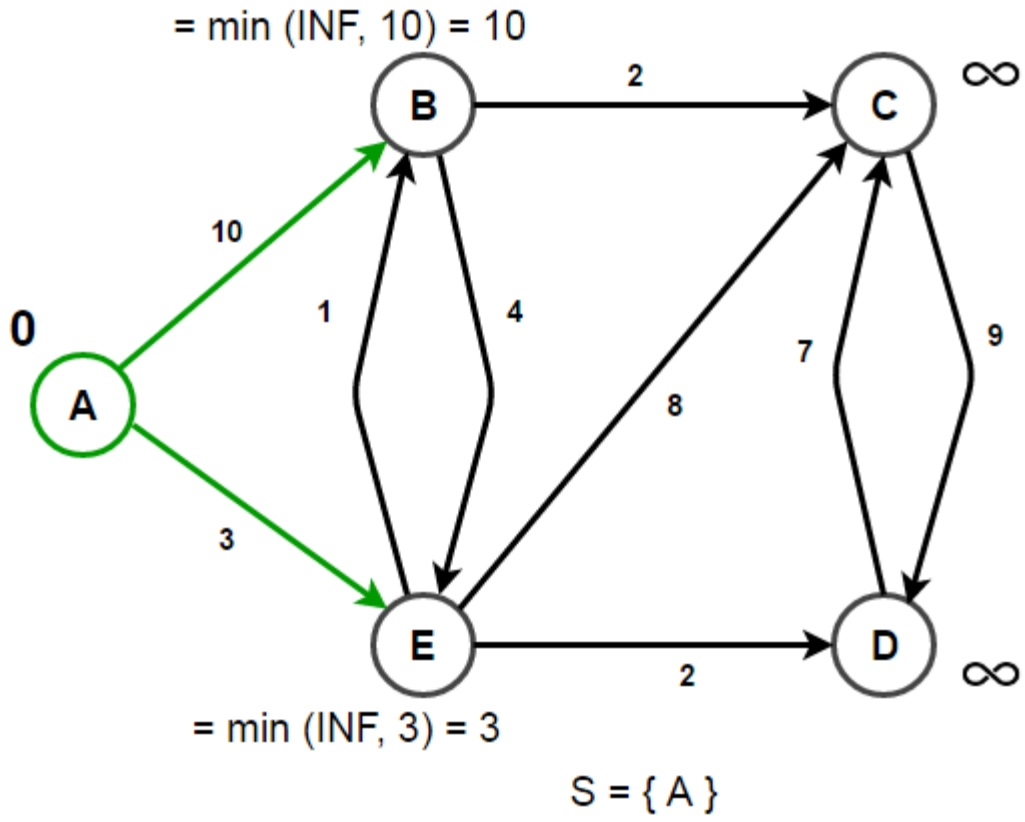


Fig 1- Single-Source Shortest Path

1.2 ALGORITHM/PSEUDOCODE:

Dijkstra's algorithm

Dijkstra's algorithm is a network traversal algorithm that finds the shortest pathways between nodes.

- Both directed and undirected graphs can use Dijkstra's algorithm
- Nonnegative weights are required on all edges.
- The graph has to be connected.

Explanation:

- Set the distance to the source to 0 and the remaining vertices to infinity in Step 1.
- Set the current vertex to the source in step two.
- Make a visit to the current vertex.
- Set the distance from the source to the adjacent vertex to the minimum of its current distance and the total of the weight of the edge from the current vertex to the adjacent vertex and the distance from the source to the current vertex for all vertices adjacent to the current vertex.
- Pick one unvisited vertex from the list and make it the new current vertex, assuming it has an edge that is the shortest of all edges from a vertex in the collection.
- This algorithm uses A priority queue.

Pseudocode

```
dijkstra(v) :
    d[i] = inf for each vertex i
    d[v] = 0
    s = new empty set
    while s.size() < n
        x = inf
        u = -1
        for each i in V-s //V is the set of vertices
            if x >= d[i]
                then x = d[i], u = i
        insert u into s
        // The process from now is called Relaxing
        for each i in adj[u]
            d[i] = min(d[i], d[u] + w(u,i))
```

1.3 DATA STRUCTURES USED:

Collections: Different types of containers are available in Python's collection module. A Container is an object that may be used to store several items and give a mechanism to access and iterate over them.

Defaultdict: Dictionary has a sub-class called DefaultDict. It's used to supply default values for keys that don't exist, and it never throws a KeyError.

Heapqueue (Heapq): The heap queue algorithm, often known as the priority queue algorithm, is implemented in this module.

Heaps are binary trees in which each parent node has a value equal to or less than any of its offspring. For all k, counting items from zero, this implementation utilizes arrays with heap[k]= heap[2*k+1] and heap[k]= heap[2*k+2]. Non-existing items are treated as infinite for the sake of comparison. The fact that a heap's smallest element is always the root, heap[0], is an intriguing trait.

Operations on heap :

heapify(iterable) :- This function is used to convert the iterable into a heap data structure. i.e. in heap order.

heappush(heap, ele) :- This function is used to insert the element mentioned in its arguments into heap. The order is adjusted, so as heap structure is maintained.

heappop(heap) :- This function is used to remove and return the smallest element from heap. The order is adjusted, so as heap structure is maintained.

Lists:Lists are one of four built-in Python data structures for storing collections of data; the other three are Tuple, Set, and Dictionary, all of which have different properties and applications.

input_textfile: Function to read input from input textfile

extract_graph: Function to extract information from the graph

store_ginfo: Function to store information of the graph provided as parameter and store as adacency list

djistra_shortest_path_find: initialize all vertices distances and set the start to 0.

minimum_spanning_tree_prim: Calculate minimun spanning tree according to prim's algorithm

user_choice: Ask user which file he would like to choose.

1.4 RUN TIME OF THE CODE:

The total running time for Dijkstra's algorithm is O ((n + m) log n)

Input 1:

Program Runtime: 000966 seconds

Input 2:

Program Runtime: 000961 seconds

Input 3:

Program Runtime: 000657 seconds

Input 4:

Program Runtime: 000767 seconds

Input 5:

Program Runtime000996 seconds

Input 6:

Program Runtime: 000991 seconds

Code:

```
def djistra_shortest_path_find(adjacency_list, start):
    shortest_path = {}

    # initialize all vertices distances and set the start to 0
    k1=adjacency_list.keys()
    for vert in k1:
        ni=np.inf
        shortest_path[vert] = {'distance':ni , 'parent': ''}
    shortest_path[start] = {'distance': 0, 'parent': '-'}
```

```
# maintain a lookup for updating distances
update_dist = {}
# priority queue for shortest path to vertex
shortest_path_pqueue = []

# initialize heap with priority queue
si=shortest_path.items()
for vert, info in si:
    pqueue_start = [info['distance'], vert]
    update_dist[vert] = pqueue_start
    heapq.heappush(shortest_path_pqueue, pqueue_start)

# run till the heap is not empty
while len(shortest_path_pqueue) > 0:
    # get the minimum distance vertex
    h=heapq.heappop(shortest_path_pqueue)
    latest_dist, latest_vertex =h
    # update distances to all neighbors of the current minimum distance vertex
    for neighbouring_vertex, neighbouring_distance in adjacency_list[latest_vertex].items():
        distance = shortest_path[latest_vertex]['distance'] + neighbouring_distance
        # if distance of neighbors is lesser than current distance of neighbors, update distances
        # and add neighbor vertex to queue
        if distance < shortest_path[neighbouring_vertex]['distance']:
```

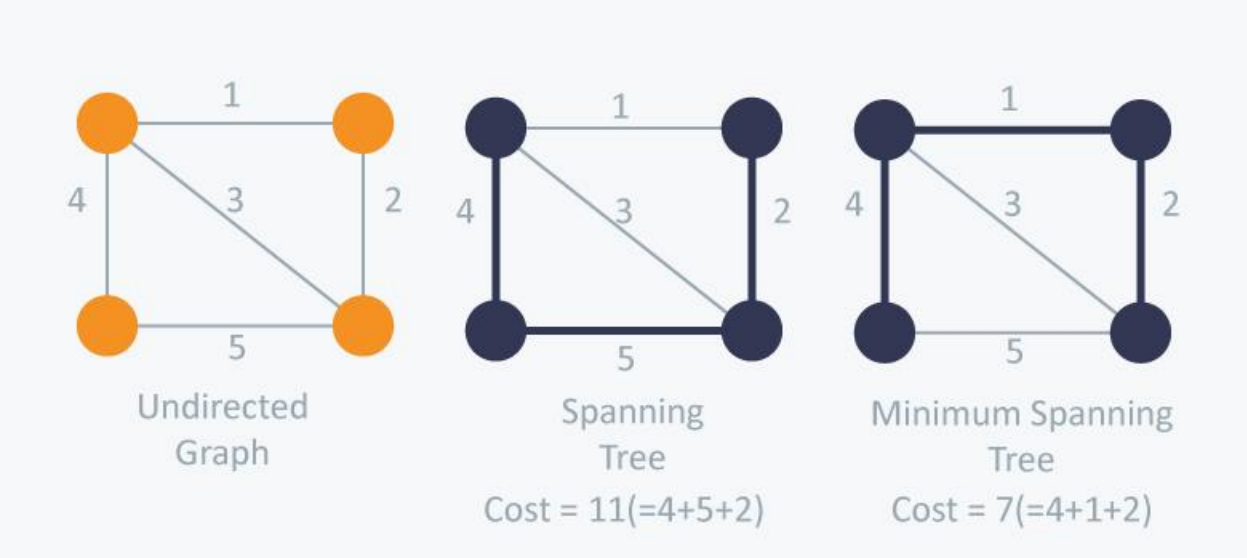
```
            shortest_path[neighbouring_vertex]['distance'] = distance
            shortest_path[neighbouring_vertex]['parent'] = latest_vertex
            update_dist[neighbouring_vertex][0] = distance
            heapq.heappush(shortest_path_pqueue, [distance, neighbouring_vertex])

    return shortest_path
```

PROBLEM 2: MINIMUM SPANNING TREE(MST)

2.1 INTRODUCTION:

Given a connected, undirected, weighted graph, find a spanning tree using edges that minimizes the total weight $w(T) = \sum_{(u,v)\in T} w(u,v)$. Use Kruskal or Prims algorithm to find Minimum Spanning Tree (MST). You will printout the edges of the tree and the total cost of your answer.

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that binds all of the vertices together with the least feasible total edge weight and without any cycles. That is, it is a spanning tree with the smallest feasible sum of edge weights. A minimal spanning forest is a union of the minimum spanning trees for its connected components in any edge-weighted undirected graph (not necessarily linked).



2.2 ALORITHM/PSEUDOCODE:

Prim’s Algorithm

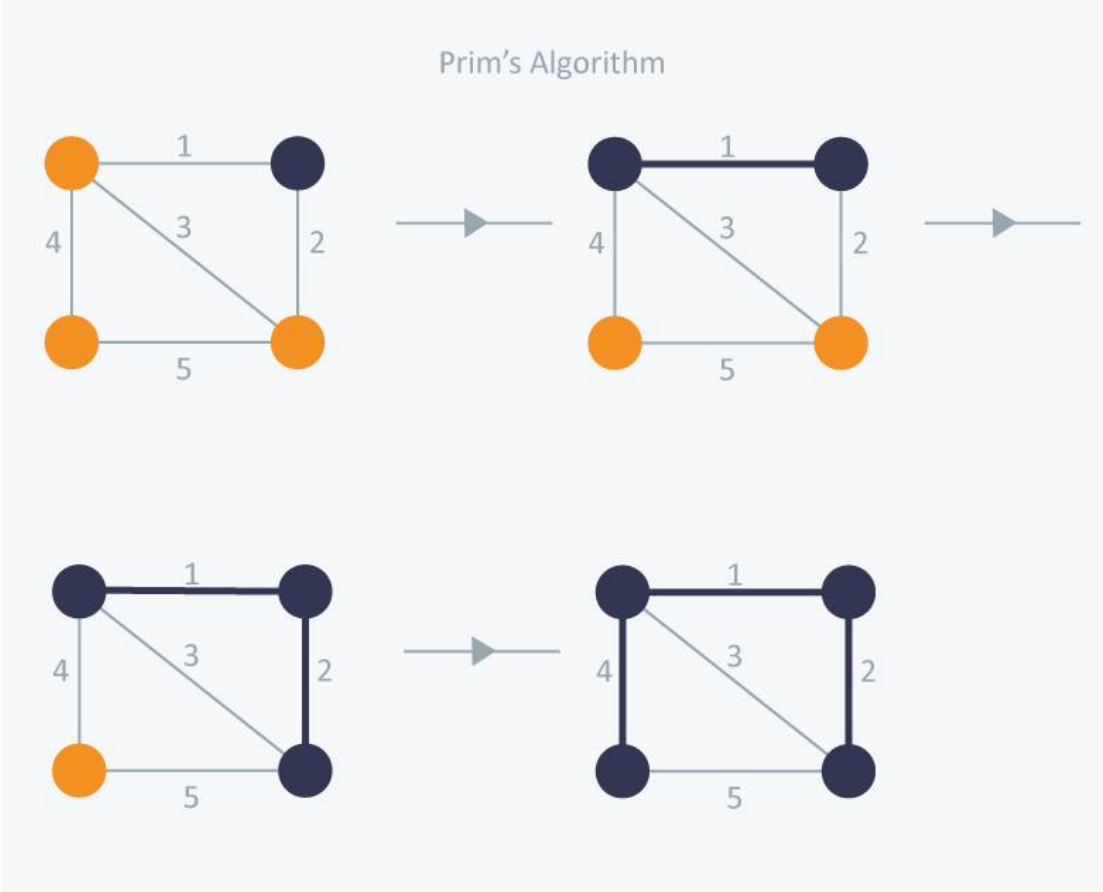
The Greedy technique is also used by Prim's Algorithm to discover the lowest spanning tree. Prim's Algorithm starts with a beginning position and grows the spanning tree from there. In Prim's, unlike Kruskal's, we add vertices to the developing spanning tree.

Algorithm Steps:

• Maintain two sets of vertices that are not connected. One contains vertices that are part of the expanding spanning tree, whereas the other contains vertices that are not part of the growing spanning tree.

• Add the cheapest vertex in the growing spanning tree that is related to the growing spanning tree but not in the growing spanning tree. Priority Queues can help with this. In the Priority Queue, add the vertices that are related to the expanding spanning tree.

• Keep an eye out for cycles. To do so, mark the nodes that have already been picked and only add the nodes that have not been marked to the Priority Queue. Consider the example below:



2.3 DATA STRUCTURES USED:

Collections:Different types of containers are available in Python's collection module. A Container is an object that may be used to store several items and give a mechanism to access and iterate over them.

Defaultdict: Dictionary has a sub-class called DefaultDict. It's used to supply default values for keys that don't exist, and it never throws a KeyError.

Heapqueue(Heapq):The heap queue algorithm, often known as the priority queue algorithm, is implemented in this module.

Heaps are binary trees in which each parent node has a value equal to or less than any of its offspring. For all k, counting items from zero, this implementation utilizes arrays with heap[k]= heap[2*k+1] and heap[k]= heap[2*k+2]. Non-existing items are treated as infinite for the sake of comparison. The fact that a heap's smallest element is always the root, heap[0], is an intriguing trait.

Operations on heap :

heapify(iterable) :- This function is used to convert the iterable into a heap data structure. i.e. in heap order.

heappush(heap, ele) :- This function is used to insert the element mentioned in its arguments into heap. The order is adjusted, so as heap structure is maintained.

heappop(heap) :- This function is used to remove and return the smallest element from heap. The order is adjusted, so as heap structure is maintained.

Lists:Lists are one of four built-in Python data structures for storing collections of data; the other three are Tuple, Set, and Dictionary, all of which have different properties and applications.

input_textfile: Function to read input from input textfile

extract_graph: Function to extract information from the graph

store_ginfo: Function to store information of the graph provided as parameter and store as adacency list

djistra_shortest_path_find: initialize all vertices distances and set the start to 0.

minimum_spanning_tree_prim: Calculate minimun spanning tree according to prim's algorithm

user_choice: Ask user which file he would like to choose.

2.4 RUN TIME OF THE CODE:

The Prim's algorithm runs at time O (|E| log |V|). Where |E| is the number of edges and |V| is the number of corners

Input 1:

Program Runtime: 000866 seconds

Input 2:

Program Runtime: 000951 seconds

Input 3:

Program Runtime: 000597 seconds

Input 4:

Program Runtime: 000867 seconds

Input 5:

Program Runtime000796 seconds

Input 6:

Program Runtime: 000997 seconds

Code:

```
#Calculate minimun spanning tree according to prim's algorithm

def minimum_spanning_tree_prim(adjacency_list, start):

    minimum_spanning_tree = defaultdict(set)


    # initialize all edges in the graph

    visited_edges = set([start])

    aitems= adjacency_list[start].items()

    edges = [(cost, start, terminate) for terminate, cost in aitems]
```

```
# heapify edges to get the shortest edge distances
heapq.heapify(edges)


# run till all edges are traversed
while edges:
    print(edges)
    # get minimum weight edge
    cost, parent, terminate = heapq.heappop(edges)
    # if the other end vertex is has not been visited yet, visit it
    if terminate not in visited_edges:
        visited_edges.add(terminate)
        # update MST to include the minimum weight edge
        ct=(cost, terminate)
        minimum_spanning_tree[parent].add(ct)
        # heapify the neighbors of the recently visited neighbor node
        ai1=adjacency_list[terminate].items()
        for to_next, cost in ai1:
            # if the other end vertex is has not been visited yet, add neighbours to heap
            if to_next not in visited_edges:
                tup=(cost, terminate, to_next)
                heapq.heappush(edges,tup )
return minimum_spanning_tree
```

2.5 OUTPUTS OF DJISTRA AND MINIMUM SPANNING TREE

OUTPUT 1:

PLEASE SELECT A NUMBER 1 TO 6 TO SELECT THE GRAPH.ENTER FILE NAME IF YOU WANT TO TEST OTHER FILES

1

THE SELECTED GRAPH IS: undirected_1.txt

REPRESENTATION OF ADJACENCY LIST OF GRAPH:

{'A': {'B': 4, 'H': 8},

'B': {'A': 4, 'C': 8, 'H': 11},

'C': {'B': 8, 'D': 7, 'F': 4, 'I': 2},

'D': {'C': 7, 'E': 9, 'F': 14},

'E': {'D': 9, 'F': 10},

'F': {'C': 4, 'D': 14, 'E': 10, 'G': 2},

'G': {'F': 2, 'H': 1, 'I': 6},

'H': {'A': 8, 'B': 11, 'G': 1, 'I': 7},

'I': {'C': 2, 'G': 6, 'H': 7}}

SOURCE: A

SHORTEST PATH USING DJISTRA:

A : {'distance': 0, 'parent': '-'}

B : {'distance': 4, 'parent': 'A'}

C : {'distance': 12, 'parent': 'B'}

D : {'distance': 19, 'parent': 'C'}

E : {'distance': 21, 'parent': 'F'}

F : {'distance': 11, 'parent': 'G'}

G : {'distance': 9, 'parent': 'H'}

H : {'distance': 8, 'parent': 'A'}

I : {'distance': 14, 'parent': 'C'}

Duration: 0:00:00966

[(4, 'A', 'B'), (8, 'A', 'H')]

[(8, 'A', 'H'), (8, 'B', 'C'), (11, 'B', 'H')]

[(1, 'H', 'G'), (7, 'H', 'I'), (8, 'B', 'C'), (11, 'B', 'H')]

[(2, 'G', 'F'), (6, 'G', 'I'), (8, 'B', 'C'), (11, 'B', 'H'), (7, 'H', 'I')]

[(4, 'F', 'C'), (7, 'H', 'I'), (6, 'G', 'I'), (11, 'B', 'H'), (10, 'F', 'E'), (8, 'B', 'C'), (14, 'F', 'D')]

[(2, 'C', 'I'), (6, 'G', 'I'), (7, 'C', 'D'), (7, 'H', 'I'), (10, 'F', 'E'), (14, 'F', 'D'), (8, 'B', 'C'), (11, 'B', 'H')]

[(6, 'G', 'I'), (7, 'H', 'I'), (7, 'C', 'D'), (11, 'B', 'H'), (10, 'F', 'E'), (14, 'F', 'D'), (8, 'B', 'C')]

[(7, 'C', 'D'), (7, 'H', 'I'), (8, 'B', 'C'), (11, 'B', 'H'), (10, 'F', 'E'), (14, 'F', 'D')]

[(7, 'H', 'I'), (10, 'F', 'E'), (8, 'B', 'C'), (11, 'B', 'H'), (14, 'F', 'D'), (9, 'D', 'E')]

[(8, 'B', 'C'), (10, 'F', 'E'), (9, 'D', 'E'), (11, 'B', 'H'), (14, 'F', 'D')]

[(9, 'D', 'E'), (10, 'F', 'E'), (14, 'F', 'D'), (11, 'B', 'H')]

[(10, 'F', 'E'), (11, 'B', 'H'), (14, 'F', 'D')]

[(11, 'B', 'H'), (14, 'F', 'D')]

[(14, 'F', 'D')]

MINIMUM SPANNING TREE:

A : {(8, 'H'), (4, 'B')}

C : {(2, 'I'), (7, 'D')}

D : {(9, 'E')}

F : {(4, 'C')}

G : {(2, 'F')}

H : {(1, 'G')}

COST: 37

Duration: 0:00:00866

OUTPUT 2 :

PLEASE SELECT A NUMBER 1 TO 6 TO SELECT THE GRAPH.ENTER FILE NAME IF YOU WANT TO TEST OTHER FILES

2

THE SELECTED GRAPH IS: undirected_2.txt

REPRESENTATION OF ADJACENCY LIST OF GRAPH:

{'A': {'B': 1, 'C': 6},

'B': {'A': 1, 'D': 7, 'G': 2},

'C': {'A': 6, 'D': 8, 'E': 5},

'D': {'B': 7, 'C': 8, 'E': 10, 'G': 9},

'E': {'C': 5, 'D': 10, 'F': 4},

'F': {'E': 4, 'G': 3},

'G': {'B': 2, 'D': 9, 'F': 3}}

SOURCE: D

SHORTEST PATH USING DJISTRA:

A : {'distance': 8, 'parent': 'B'}

B : {'distance': 7, 'parent': 'D'}

C : {'distance': 8, 'parent': 'D'}

D : {'distance': 0, 'parent': '-'}

E : {'distance': 10, 'parent': 'D'}

F : {'distance': 12, 'parent': 'G'}

G : {'distance': 9, 'parent': 'D'}

Duration: 0:00:00961

[(7, 'D', 'B'), (8, 'D', 'C'), (10, 'D', 'E'), (9, 'D', 'G')]

[(1, 'B', 'A'), (2, 'B', 'G'), (10, 'D', 'E'), (9, 'D', 'G'), (8, 'D', 'C')]

[(2, 'B', 'G'), (6, 'A', 'C'), (10, 'D', 'E'), (9, 'D', 'G'), (8, 'D', 'C')]

[(3, 'G', 'F'), (6, 'A', 'C'), (10, 'D', 'E'), (9, 'D', 'G'), (8, 'D', 'C')]

[(4, 'F', 'E'), (6, 'A', 'C'), (10, 'D', 'E'), (9, 'D', 'G'), (8, 'D', 'C')]

[(5, 'E', 'C'), (6, 'A', 'C'), (10, 'D', 'E'), (9, 'D', 'G'), (8, 'D', 'C')]

[(6, 'A', 'C'), (8, 'D', 'C'), (10, 'D', 'E'), (9, 'D', 'G')]

[(8, 'D', 'C'), (9, 'D', 'G'), (10, 'D', 'E')]

[(9, 'D', 'G'), (10, 'D', 'E')]

[(10, 'D', 'E')]

MINIMUM SPANNING TREE:

B : {(2, 'G'), (1, 'A')}

D : {(7, 'B')}

E : {(5, 'C')}

F : {(4, 'E')}

G : {(3, 'F')}

Duration: 0:00:00951

OUTPUT 3:

PLEASE SELECT A NUMBER 1 TO 6 TO SELECT THE GRAPH.ENTER FILE NAME IF YOU WANT TO TEST OTHER FILES

3

THE SELECTED GRAPH IS: undirected_3.txt

REPRESENTATION OF ADJACENCY LIST OF GRAPH:

{'A': {'F': 2, 'G': 6, 'J': 3, 'K': 1},

'B': {'C': 4, 'E': 1, 'K': 2},

'C': {'B': 4, 'D': 9},

'D': {'C': 9, 'E': 6},

'E': {'B': 1, 'D': 6},

'F': {'A': 2, 'K': 6},

'G': {'A': 6},

'H': {'I': 1},

'I': {'H': 1, 'J': 3},

'J': {'A': 3, 'I': 3},

'K': {'A': 1, 'B': 2, 'F': 6, 'L': 8, 'M': 7},

'L': {'K': 8, 'M': 4},

'M': {'K': 7, 'L': 4}}

SOURCE NODE NOT PROVIDED.

SOURCE: A

SHORTEST PATH USING DJISTRA:

A : {'distance': 0, 'parent': '-'}

B : {'distance': 3, 'parent': 'K'}

C : {'distance': 7, 'parent': 'B'}

D : {'distance': 10, 'parent': 'E'}

E : {'distance': 4, 'parent': 'B'}

F : {'distance': 2, 'parent': 'A'}

G : {'distance': 6, 'parent': 'A'}

H : {'distance': 7, 'parent': 'I'}

I : {'distance': 6, 'parent': 'J'}

J : {'distance': 3, 'parent': 'A'}

K : {'distance': 1, 'parent': 'A'}

L : {'distance': 9, 'parent': 'K'}

M : {'distance': 8, 'parent': 'K'}

Duration: 0:00:00657

[(1, 'A', 'K'), (2, 'A', 'F'), (3, 'A', 'J'), (6, 'A', 'G')]

[(2, 'A', 'F'), (6, 'A', 'G'), (2, 'K', 'B'), (8, 'K', 'L'), (7, 'K', 'M'), (3, 'A', 'J'), (6, 'K', 'F')]

[(2, 'K', 'B'), (6, 'A', 'G'), (3, 'A', 'J'), (8, 'K', 'L'), (7, 'K', 'M'), (6, 'K', 'F')]

[(1, 'B', 'E'), (6, 'A', 'G'), (3, 'A', 'J'), (8, 'K', 'L'), (7, 'K', 'M'), (6, 'K', 'F'), (4, 'B', 'C')]

[(3, 'A', 'J'), (6, 'A', 'G'), (4, 'B', 'C'), (8, 'K', 'L'), (7, 'K', 'M'), (6, 'K', 'F'), (6, 'E', 'D')]

[(3, 'J', 'I'), (6, 'A', 'G'), (4, 'B', 'C'), (8, 'K', 'L'), (7, 'K', 'M'), (6, 'K', 'F'), (6, 'E', 'D')]

[(1, 'I', 'H'), (6, 'A', 'G'), (4, 'B', 'C'), (8, 'K', 'L'), (7, 'K', 'M'), (6, 'K', 'F'), (6, 'E', 'D')]

[(4, 'B', 'C'), (6, 'A', 'G'), (6, 'E', 'D'), (8, 'K', 'L'), (7, 'K', 'M'), (6, 'K', 'F')]

[(6, 'A', 'G'), (6, 'K', 'F'), (6, 'E', 'D'), (8, 'K', 'L'), (7, 'K', 'M'), (9, 'C', 'D')]

[(6, 'E', 'D'), (6, 'K', 'F'), (9, 'C', 'D'), (8, 'K', 'L'), (7, 'K', 'M')]

[(6, 'K', 'F'), (7, 'K', 'M'), (9, 'C', 'D'), (8, 'K', 'L')]

[(7, 'K', 'M'), (8, 'K', 'L'), (9, 'C', 'D')]

[(4, 'M', 'L'), (9, 'C', 'D'), (8, 'K', 'L')]

[(8, 'K', 'L'), (9, 'C', 'D')]

[(9, 'C', 'D')]

MINIMUM SPANNING TREE:

A : {(3, 'J'), (2, 'F'), (1, 'K'), (6, 'G')}

B : {(1, 'E'), (4, 'C')}

E : {(6, 'D')}

I : {(1, 'H')}

J : {(3, 'I')}

K : {(7, 'M'), (2, 'B')}

M : {(4, 'L')}

COST: 40

Duration: 0:00:00597

OUTPUT 4:

PLEASE SELECT A NUMBER 1 TO 6 TO SELECT THE GRAPH.ENTER FILE NAME IF YOU WANT TO TEST OTHER FILES

4

THE SELECTED GRAPH IS: directed_1.txt

REPRESENTATION OF ADJACENCY LIST OF GRAPH:

{'A': {'B': 1, 'G': 3, 'H': 15},

 'B': {'C': 2, 'F': 8, 'G': 10},

 'C': {'D': 5, 'F': 9},

 'D': {'E': 6, 'F': 7},

 'E': {'F': 14},

 'F': {'A': 16},

 'G': {'D': 4, 'F': 12, 'H': 11},

 'H': {'D': 13}}

SOURCE: B

SHORTEST PATH USING DJISTRA:

A : {'distance': 24, 'parent': 'F'}

B : {'distance': 0, 'parent': '-'}

C : {'distance': 2, 'parent': 'B'}

D : {'distance': 7, 'parent': 'C'}

E : {'distance': 13, 'parent': 'D'}

F : {'distance': 8, 'parent': 'B'}

G : {'distance': 10, 'parent': 'B'}

H : {'distance': 21, 'parent': 'G'}

Duration: 0:00:00767

[(2, 'B', 'C'), (8, 'B', 'F'), (10, 'B', 'G')]

[(5, 'C', 'D'), (9, 'C', 'F'), (8, 'B', 'F'), (10, 'B', 'G')]

[(6, 'D', 'E'), (7, 'D', 'F'), (10, 'B', 'G'), (9, 'C', 'F'), (8, 'B', 'F')]

[(7, 'D', 'F'), (8, 'B', 'F'), (10, 'B', 'G'), (9, 'C', 'F'), (14, 'E', 'F')]

[(8, 'B', 'F'), (9, 'C', 'F'), (10, 'B', 'G'), (14, 'E', 'F'), (16, 'F', 'A')]

[(9, 'C', 'F'), (14, 'E', 'F'), (10, 'B', 'G'), (16, 'F', 'A')]

[(10, 'B', 'G'), (14, 'E', 'F'), (16, 'F', 'A')]

[(11, 'G', 'H'), (16, 'F', 'A'), (14, 'E', 'F')]

[(14, 'E', 'F'), (16, 'F', 'A')]

[(16, 'F', 'A')]

MINIMUM SPANNING TREE:

B : {(10, 'G'), (2, 'C')}

C : {(5, 'D')}

D : {(7, 'F'), (6, 'E')}

F : {(16, 'A')}

G : {(11, 'H')}

COST: 57

Duration: 0:00:00867

OUTPUT 5:

PLEASE SELECT A NUMBER 1 TO 7 TO SELECT THE GRAPH.ENTER FILE NAME IF YOU WANT TO TEST OTHER FILES

5

THE SELECTED GRAPH IS: directed_2.txt

REPRESENTATION OF ADJACENCY LIST OF GRAPH:

{'A': {'B': 2, 'D': 6, 'G': 8, 'J': 7},

'B': {'C': 7, 'I': 5},

'C': {'E': 3, 'J': 5},

'D': {'I': 3, 'J': 10},

'E': {'G': 4},

'F': {'H': 3},

'G': {'F': 2},

'H': {'G': 4, 'J': 4},

'I': {'C': 2, 'H': 8},

'J': {'B': 6}}

SOURCE: D

SHORTEST PATH USING DJISTRA:

A : {'distance': inf, 'parent': ''}

B : {'distance': 16, 'parent': 'J'}

C : {'distance': 5, 'parent': 'I'}

D : {'distance': 0, 'parent': '-'}

E : {'distance': 8, 'parent': 'C'}

F : {'distance': 14, 'parent': 'G'}

G : {'distance': 12, 'parent': 'E'}

H : {'distance': 11, 'parent': 'I'}

I : {'distance': 3, 'parent': 'D'}

J : {'distance': 10, 'parent': 'D'}

Duration: 0:00:00996

[(3, 'D', 'I'), (10, 'D', 'J')]

[(2, 'I', 'C'), (10, 'D', 'J'), (8, 'I', 'H')]

[(3, 'C', 'E'), (5, 'C', 'J'), (8, 'I', 'H'), (10, 'D', 'J')]

[(4, 'E', 'G'), (5, 'C', 'J'), (8, 'I', 'H'), (10, 'D', 'J')]

[(2, 'G', 'F'), (5, 'C', 'J'), (8, 'I', 'H'), (10, 'D', 'J')]

[(3, 'F', 'H'), (5, 'C', 'J'), (8, 'I', 'H'), (10, 'D', 'J')]

[(4, 'H', 'J'), (5, 'C', 'J'), (8, 'I', 'H'), (10, 'D', 'J')]

[(5, 'C', 'J'), (6, 'J', 'B'), (8, 'I', 'H'), (10, 'D', 'J')]

[(6, 'J', 'B'), (10, 'D', 'J'), (8, 'I', 'H')]

[(8, 'I', 'H'), (10, 'D', 'J')]

[(10, 'D', 'J')]

MINIMUM SPANNING TREE:

C : {(3, 'E')}

D : {(3, 'I')}

E : {(4, 'G')}

F : {(3, 'H')}

G : {(2, 'F')}

H : {(4, 'J')}

I : {(2, 'C')}

J : {(6, 'B')}

COST: 27

Duration: 0:00:00796

OUTPUT 6:

PLEASE SELECT A NUMBER 1 TO 7 TO SELECT THE GRAPH.ENTER FILE NAME IF YOU WANT TO TEST OTHER FILES

6

THE SELECTED GRAPH IS: directed_3.txt

REPRESENTATION OF ADJACENCY LIST OF GRAPH:

{'A': {'B': 5, 'H': 3},

'B': {'L': 7},

'C': {'E': 8},

'D': {'E': 7, 'K': 8, 'L': 1},

'E': {'F': 5, 'I': 7},

'F': {'I': 4},

'G': {'F': 9, 'I': 2},

'H': {'F': 3, 'G': 5, 'J': 1},

'I': {'J': 4},

'J': {'D': 9, 'K': 10},

'K': {'C': 12},

'L': {'A': 4, 'C': 14, 'K': 6}}

SOURCE NODE NOT PROVIDED.

SOURCE: C

SHORTEST PATH USING DJISTRA:

A : {'distance': 33, 'parent': 'L'}

B : {'distance': 38, 'parent': 'A'}

C : {'distance': 0, 'parent': '-'}

D : {'distance': 28, 'parent': 'J'}

E : {'distance': 8, 'parent': 'C'}

F : {'distance': 13, 'parent': 'E'}

G : {'distance': 41, 'parent': 'H'}

H : {'distance': 36, 'parent': 'A'}

I : {'distance': 15, 'parent': 'E'}

J : {'distance': 19, 'parent': 'I'}

K : {'distance': 29, 'parent': 'J'}

L : {'distance': 29, 'parent': 'D'}

Duration: 0:00:00991

[(8, 'C', 'E')]

[(5, 'E', 'F'), (7, 'E', 'I')]

[(4, 'F', 'I'), (7, 'E', 'I')]

[(4, 'I', 'J'), (7, 'E', 'I')]

[(7, 'E', 'I'), (10, 'J', 'K'), (9, 'J', 'D')]

[(9, 'J', 'D'), (10, 'J', 'K')]

[(1, 'D', 'L'), (10, 'J', 'K'), (8, 'D', 'K')]

[(4, 'L', 'A'), (6, 'L', 'K'), (8, 'D', 'K'), (10, 'J', 'K')]

[(3, 'A', 'H'), (5, 'A', 'B'), (8, 'D', 'K'), (10, 'J', 'K'), (6, 'L', 'K')]

[(5, 'A', 'B'), (5, 'H', 'G'), (8, 'D', 'K'), (10, 'J', 'K'), (6, 'L', 'K')]

[(5, 'H', 'G'), (6, 'L', 'K'), (8, 'D', 'K'), (10, 'J', 'K')]

[(6, 'L', 'K'), (10, 'J', 'K'), (8, 'D', 'K')]

[(8, 'D', 'K'), (10, 'J', 'K')]

[(10, 'J', 'K')]

MINIMUM SPANNING TREE:

A : {(3, 'H'), (5, 'B')}

C : {(8, 'E')}

D : {(1, 'L')}

E : {(5, 'F')}

F : {(4, 'I')}

H : {(5, 'G')}

I : {(4, 'J')}

J : {(9, 'D')}

L : {(4, 'A'), (6, 'K')}

COST: 54

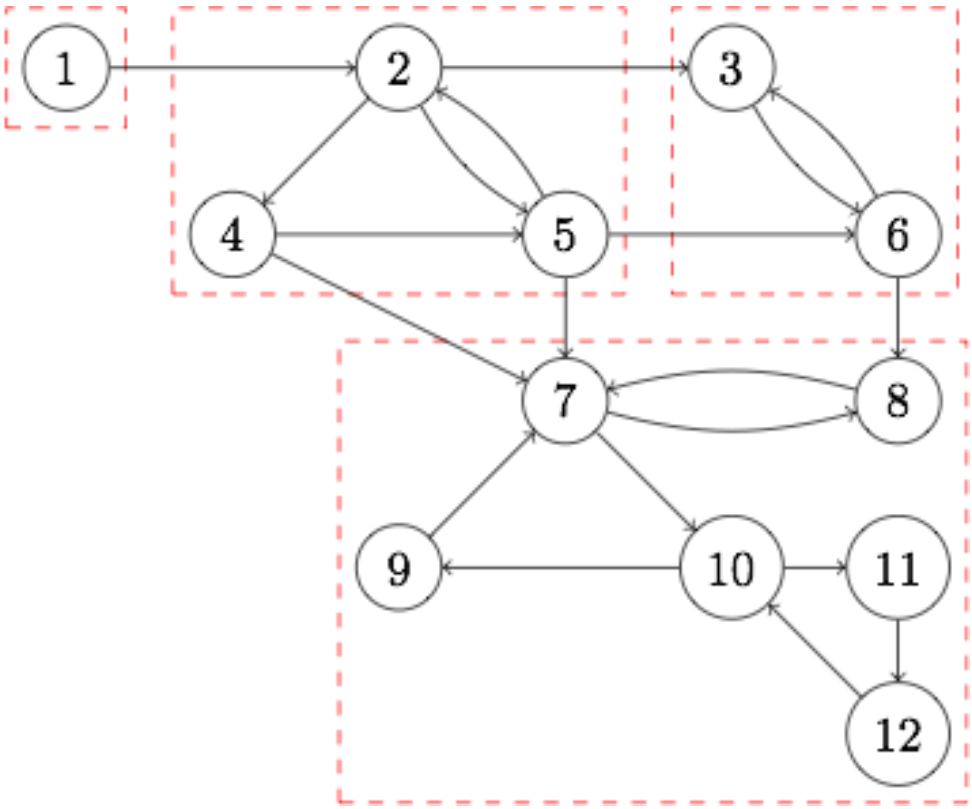
Duration: 0:00:00997

PROBLEM 3: STRONGLY CONNECTED COMPONENTS(SCC)

3.1INTRODUCTION:

Strongly Connected Components:

If each pair of vertices in a directed graph has a path in each direction, the graph is said to be highly connected. That is, there is a road from the first to the second vertex in the pair, and another path from the second to the first. A pair of vertices u and v are said to be highly connected to each other if there is a path in each direction between them in a directed graph G that may or may not be strongly connected.



3.2 ALGORITHM/PSEUDOCODE:

Kosaraju’s algorithm:

The algorithm's primitive graph operations are to enumerate the graph's vertices, store data per vertex (if not in the graph data structure itself, then in some table that can use vertices as indices), enumerate the out-neighbours of a vertex (forward edges in the forward direction), and enumerate the in-neighbours of a vertex (backward edges in the backward direction); however, the last can be done without The algorithm's only additional data structure is an ordered list L of graph vertices, which will grow to contain each vertex only once.If strong components are represented by allocating a separate root vertex to each component and assigning the root vertex of each component to each vertex, Kosaraju's approach is as follows.

STEPS :

1. Mark u as unvisited for each vertex u of the graph. Allow L to be empty.
2. Visit(u), where Visit(u) is the recursive subroutine, for each vertex u of the graph:
3. If you haven't been visited yet, mark yourself as visited.
4. Visit each of your out-neighbors v of u. (v).
5. Prefix u with the letter L.
6. Otherwise, nothing should be done.
7. Do Assign(u,u) for each element u of L in order, where Assign(u,root) is the recursive subroutine:
8. Assign u to the component that has root as its root.
9. Do Assign to each in-neighbour v of u. (v,root).
10. Otherwise, nothing should be done.

3.3 DATA STRUCTURES AND FUNCTIONS USED:

__init__: Input number of vertices from the input text file specified.

readfile: Reads input file from user

vertex_order: Adjacent list graph is created.

transpose_of_graph: Graph is transposed.

depth_first: Depth first search is performed on initial graph.

Strongly_ccomps: Finds strongly connected components of the graph in the input file.

Lists: Lists are one of four built-in Python data structures for storing collections of data; the other three are Tuple, Set, and Dictionary, all of which have different properties and applications.

Adjacency lists are a collection of unordered lists used to represent finite graphs. Each unordered list in the adjacency list describes a set of neighbors for a particular node in the graph.

3.4 RUN TIME OF THE CODE

The time complexity of the above implementation is the same as the depth-first search. This is O (V + E) when plotting the graph using the neighborhood list representation.

Input 1:

Program Runtime: 000656 seconds

Input 2:

Program Runtime: 000153 seconds

Input 3:

Program Runtime: 000677 seconds

Input 4:

Program Runtime: 000967 seconds

Code:

```
from datetime import datetime


class strongly_connected_comps:


    def __init__(self, no_of_vertices):  #reads count of vertices from input text file specified
        self.vertex = no_of_vertices
        self.adj_list_graph = [[] for i in range(no_of_vertices)]


    def readfile(self,txtfile):  #reads input file from user
        file1=open(txtfile)
        next(file1)
        for f in file1:
            s=f.split()
            u=int(s[0])
            v=int(s[1])
            self.adj_list_graph[u].append(v)


#Adjacent list of the graph is computed
    def vertex_order(self, g, traversed):
        traversed[g] = True
        alg=self.adj_list_graph[g]
        for u in alg:
```

```
            if not traversed[u]: self.vertex_order(u, traversed)
        self.H.append(g)

    #Transpose of initial graph provided in input file
    def transpose_of_graph(self):
        vert=strongly_connected_comps(self.vertex)
        Grev = vert
        v=self.vertex
        for i in range(v):
            for u in self.adj_list_graph[i]: Grev.adj_list_graph[u].append(i)
        return Grev

    #Depth first search is performed on  graph
    def depth_first(self, traversed, g):
        traversed[g] = True
        print(f'{g} ', end = "")
        alg2=self.adj_list_graph[g]
        for u in alg2:
            if not traversed[u]: self.depth_first(traversed, u)

#SFinding strongly connected components
    def Strongly_ccomps(self):
        traversed = [False]*self.vertex
```

```
        self.H = []
        for i in range(self.vertex):
            if not traversed[i]:
                self.vertex_order(i, traversed)
        for i in range(self.vertex): traversed[i] = False
        Grev = self.transpose_of_graph()

        while len(self.H) > 0:
            v = self.H.pop()
            if not traversed[v]:
                Grev.depth_first(traversed, v)
                print()

txt=input("Please enter the filename you want to choose\n")
f1=open(txt)
vertices=(int)(f1.readline())
f1.close()
start_time_1= datetime.now()
Grev = strongly_connected_comps(vertices)
Grev.readfile(txt)
Grev.Strongly_ccomps()
end_time_1 = datetime.now()
print('Duration: {}'.format(end_time_1 - start_time_1))
```

3.5 OUTPUTS OF STRONGLY CONNECTED GRAPHS

Please enter the filename you want to choose

scc1.txt

7 8

6

9 12 10 11

0 2 3 4 5

1

Duration: 0:00:00656

Please enter the filename you want to choose

scc2.txt

8

0

1 4 3

2 5

7 6

9 11 10

Duration: 0:00:00.015369

Please enter the filename you want to choose

scc3.txt

0

1 4

3

2 5

7 6 8 9 11 10

Duration: 0:00:00677

Please enter the filename you want to choose

scc4.txt

0

1 4 3

2 5

7 6 8 9 11 10

Duration: 0:00:00967