PROJECT DOCUMENTATION REPORT

ON

Graph Algorithms: Single-source Shortest Path and Minimum Spanning Tree (MST)

PROGRAMMING PROJECT 2
ITCS 6114 – Algorithm and Data Structures

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO

Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Sahil Sood

801082267

Table of Contents

SINGLE-SOURCE SHORTEST PATH

1.1 Introduction

Problem 1: Find shortest paths 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

In graph theory, the Single-source Shortest Path problem is the problem in which we have to find shortest paths from a source vertex v to all other vertices in the graph. Given a graph G = (V,E), with non-negative costs on each edge, and a selected source node v in V, for all w in V, we have to find the cost of the least cost path from v to w. The *cost* of a path is simply the sum of the costs on the edges traversed by the path.

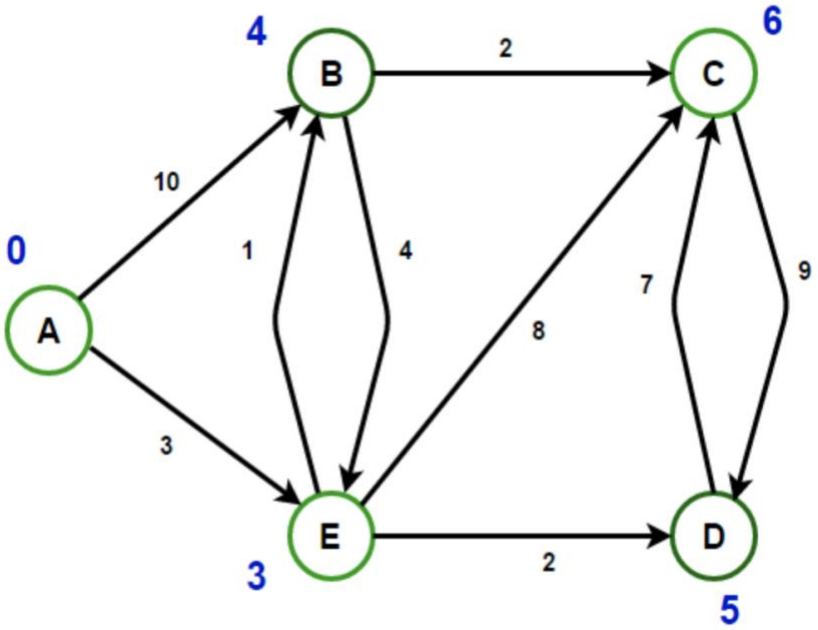
For example,

Path from vertex A to vertex B has min cost of 4 & the route is [A -> E -> B]

Path from vertex A to vertex C has min cost of 6 & the route is [A -> E -> B -> C]

Path from vertex A to vertex D has min cost of 5 & the route is [A -> E -> D]

Path from vertex A to vertex E has min cost of 3 & the route is [A -> E]



1.2 Algorithm

Dijkstra's Algorithm is an algorithm for finding the shortest paths between nodes in a graph. For a given source node in the graph, the algorithm finds the shortest path between that node and every other node. It can be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Dijkstra's Algorithm is based on the principle of relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until shortest distance is reached. The approximation distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value with the length of a newly found path. It uses a priority queue to greedily select the closest vertex that has not yet been processed and performs the relaxation process on all of its outgoing edges.

Dijkstra's algorithm is a greedy algorithm for the SSSP problem. A "greedy" algorithm always makes the locally optimal choice under the assumption that this will lead to an optimal solution overall.

```
1) Initialize distances of all vertices as infinite.

2) Create an empty priority queue pq.  Every item of pq is a pair (weight,
vertex). Weight (or distance) is used as first item  of pair as first item is by
default used to compare two pairs

3) Insert source vertex into pq and make its distance as 0.

4) While either pq doesn't become empty
   a) Extract minimum distance vertex from pq.
      Let the extracted vertex be u.
   b) Loop through all adjacent of u and do
      following for every vertex v.

         // If there is a shorter path to v
         // through u.
         If dist[v] > dist[u] + weight(u, v)

            (i) Update distance of v, i.e., do
                dist[v] = dist[u] + weight(u, v)
            (ii) Insert v into the pq (Even if v is
                already there)

5) Print distance array dist[] to print all shortest paths.

   End
```

1.3 Data Structure Used

1. **Collections:** Python ships with a module that contains a number of container data types called Collections. This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple. The collection datatype used in the code is called defaultdict.

2. **Defaultdict:** It is a dict subclass that calls a factory function to supply missing values.

3. **Heap queue (or heapq) in Python:** Heap data structure is mainly used to represent a priority queue. In Python, it is available using "heapq" module. The property of this data structure in python is that each time the smallest of heap element is popped(min heap). Whenever elements are pushed or popped, heap structure in maintained. The heap[0] element also returns the smallest element each time.
   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.

4. **Python Lists:** The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

1.4 Run Time of the Code

Input 1:

Program Runtime: 0.0019457340240478516 seconds

Input 2:

Program Runtime: 0.003981113433837891 seconds

Input 3:

Program Runtime: 0.0029959678649902344 seconds

MINIMUM SPANNING TREE (MST)

2.1 Introduction

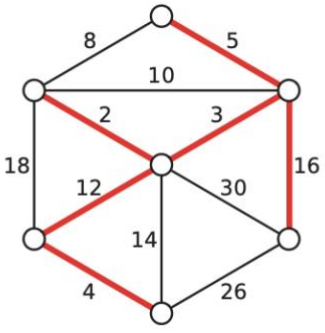
Problem 2: Given a connected, undirected, weighted graph, find a spanning tree using edges that minimizes the total weight $w(T) = \sum_{(u,v)\in T} w(u,v)$. Use Kruskal or Prims algorithm to find Minimum Spanning Tree (MST). You will printout the edges of the tree and the total cost of your answer.

Spanning Trees
In graph theory, a spanning tree T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G, with minimum possible number of edges.

Minimum Spanning Tree
A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

Suppose we are given a connected, undirected, weighted graph. This is a graph G = (V, E) together with a function w: E → R that assigns a real weight w(e) to each edge e, which may be positive, negative, or zero. Our task is to find the minimum spanning tree of G, that is, the spanning tree T that minimizes the function:

$$w(T) = \sum_{e\in T} w(e)$$



A weighted graph and its minimum spanning tree.

2.2 Algorithm

Prim's algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

How Prim's algorithm works

It falls under a class of algorithms called greedy algorithms which find the local optimum in the hopes of finding a global optimum. We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

```
MST-PRIM(G,w, r)
for each u ∈G.V
        u.key = ∞
        u. π = NIL
r.key = 0
Q = G.V
while Q != Θ
        u = EXTRACT_MIN(Q)
        for each v ∈G.Adj[u]
                if v ∈Q and w(u,v) < v.key
                        v. π = u
                        v.key = w(u,v)
```

2.3 Data Structures Used

1. **Collections:** Python ships with a module that contains a number of container data types called Collections. This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple. The collection datatype used in the code is called defaultdict.

2. **Defaultdict:** It is a dict subclass that calls a factory function to supply missing values.

3. **Heap queue (or heapq) in Python:** Heap data structure is mainly used to represent a priority queue. In Python, it is available using "heapq" module. The property of this data structure in python is that each time the smallest of heap element is popped(min heap). Whenever elements are pushed or popped, heap structure in maintained. The heap[0] element also returns the smallest element each time.
   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.

4. **Python Lists:** The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

2.4 Run Time of the Code

Input 1:

Program Runtime: 0.0013079643249511719 seconds

Input 2:

Program Runtime: 0.003484964370727539 seconds

Input 3:

Program Runtime: 0.0030100345611572266 seconds

PROBLEM CODE

3.1 Problem 1

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 09 22:19:31 2018

@author: sahilsood
"""

import collections
import heapq
import time

def shortestPath(edges, source, dnode):
   # The below if-else condition checks whether the input graph is an undirected or directed graph and adds the nodes in the set accordingly
   if graphtype == 'U':
      graph = collections.defaultdict(set)
      for l,r,c in edges:
         graph[l].add((c, r))
         graph[r].add((c, l))

   else:
      graph = collections.defaultdict(set)
      for l,r, c in edges:
         graph[l].add((c, r))

   # Here we are creating a priority queue and hash set to store visited nodes
   queue, visited = [(0, source, [])], set()
   heapq.heapify(queue)
   # Now we traverse graph with BFS
   while queue:
```

```
        (cost, node, path) = heapq.heappop(queue)
        # Visit the node if it was not visited before
        if node not in visited:
            visited.add(node)
            path = path + [node]
            # Compare the dnode value
            if node == dnode:
                return (cost, path)
            # Visit the neighbours
            for c, neighbour in graph[node]:
                if neighbour not in visited:
                    heapq.heappush(queue, (cost+int(c), neighbour, path))
    return float("inf")



def main():
    # The below line takes the input from the local drive
    inputfile = open("../Problem1/input1.txt", "r")
    lines = inputfile.readlines()
    edges = []
    global graphtype
    for line in lines:
        edges.append(line.split())
    last = list(edges)[-1]
    first = list(edges)[0]
    noofnodes = first[0]
    noofedges = first[1]
    graphtype = first[2]

    edges.pop(0)
    # The below if condition checks if the startnode is present at the last line
    if(len(last)==1):
        startnode=''.join(last)
        edges.pop()
```

```
    # If the start node is not present, the below else condition takes the first node as start node
    else:
        print("\nStart node is not present in the input file. Choosing the first node as Start node")
        firstnode = list(edges)[0]
        startnode = firstnode[0]
    print('\nSource vertex: '+str(startnode)+'\n')
    print('Number of Nodes: '+str(noofnodes)+'\n')
    print('Number of Edges: '+str(noofedges)+'\n')
    if graphtype == 'U':
        print('Graph Type: Undirected\n')
    else:
        print('Graph Type: Directed\n')
    print ("Shortest paths from the source vertex to each vertex in the weighted graphs: ")

    n = [x[0] for x in edges]
    n.extend(y[1] for y in edges)
    destnode = list((set(n)))
    print('----------------------------------')
    for i in destnode:
        print (str(startnode)+" -> "+str(i)+":")
        result = shortestPath(edges, startnode, i)
        print('Path Cost: '+str(result[0]))
        print('Path: '+str(result[1]))
        print('----------------------------------')



if __name__ == "__main__":
    start_time = time.time()
    main()
    print("Program Runtime: %s seconds" % (time.time() - start_time))
```

3.2 Problem 2

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 10 15:24:19 2018

@author: sahilsood
"""

import heapq
from collections import defaultdict
import time



def min_spanning_tree(graph, start_node):
    mst = defaultdict(set)
    global totalcost
    # Creating a set of the visited nodes
    visited = set([start_node])
    edges = [
        (cost, start_node, to)
        for to, cost in graph[start_node].items()
    ]
    heapq.heapify(edges)

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst[frm].add(to)
            totalcost = int(cost)
            print('Path cost from '+str(frm)+' To '+str(to)+' : '+str(cost))
            for to_next, cost in graph[to].items():
                if to_next not in visited:
                    heapq.heappush(edges, (cost, to, to_next))
```

```
    return mst


def main():
    # The below line takes the input from the local drive
    inputfile = open("../Problem2/input1.txt", "r")
    lines = inputfile.readlines()
    edges = []
    for line in lines:
        edges.append(line.split())
    last = list(edges)[-1]
    first = list(edges)[0]
    noofnodes = first[0]
    noofedges = first[1]
    graphtype = first[2]

    edges.pop(0)

    if(len(last)==1):
        edges.pop()
    di = {}
    for li in edges:
        di.setdefault(li[0],{})[li[1]]=li[2]
    firstnode = list(di)[0]
    # The below line takes the first node as start node
    startnode = firstnode[0]
    print('\nSource vertex: '+str(startnode)+'\n')
    print('Number of Nodes: '+str(noofnodes)+'\n')
    print('Number of Edges: '+str(noofedges)+'\n')
    if graphtype == 'U':
        print('Graph Type: Undirected\n')
    else:
        print('Graph Type: Directed\n')
    print('-----------------------------------')
    result = dict(min_spanning_tree(di, startnode))
    print('-----------------------------------\n')
```

```
    print('Edges of the tree for a Minimum Spanning Tree: \n')
    for key in result:
        print(''+str(key)+' -> '+str(result[key]))
        print('-----------------------------------')


if __name__ == "__main__":
    start_time = time.time()
    main()
    print("Program Runtime: %s seconds" % (time.time() - start_time))
```

SAMPLE INPUTS

4.1 Problem 1

9 16 D
A B 2
A D 1
B D 3
B E 10
C A 4
C F 5
D C 2
D F 8
D E 2
D G 4
E G 1
F H 2
G F 1
G I 1
H G 1
I H 3
A

4.2 Problem 2

6 10 U
A B 1
A C 2
B C 1
B D 3
B E 2
C D 1
C E 2
D E 4
D F 3
E F 3
F F 0

SAMPLE OUTPUTS

5.1 Problem 1

```
Source vertex: A
Number of Nodes: 9
Number of Edges: 16
Graph Type: Directed

Shortest paths from the source vertex to each vertex in the weighted graphs:
-----------------------------------
A -> I:
Path Cost: 5
Path: ['A', 'D', 'E', 'G', 'I']
-----------------------------------
A -> B:
Path Cost: 2
Path: ['A', 'B']
-----------------------------------
A -> A:
Path Cost: 0
Path: ['A']
-----------------------------------
A -> F:
Path Cost: 5
Path: ['A', 'D', 'E', 'G', 'F']
-----------------------------------
A -> D:
Path Cost: 1
Path: ['A', 'D']
-----------------------------------
A -> C:
Path Cost: 3
Path: ['A', 'D', 'C']
-----------------------------------
A -> G:
Path Cost: 4
Path: ['A', 'D', 'E', 'G']
-----------------------------------
A -> H:
Path Cost: 7
Path: ['A', 'D', 'E', 'G', 'F', 'H']
-----------------------------------
A -> E:
Path Cost: 3
Path: ['A', 'D', 'E']
-----------------------------------
Program Runtime: 0.0060999393463134766 seconds
```

5.2 Problem 2

```
Source vertex: A

Number of Nodes: 6

Number of Edges: 10

Graph Type: Undirected

---------------------------------
Path cost from A To B : 1
Path cost from B To C : 1
Path cost from C To D : 1
Path cost from B To E : 2
Path cost from D To F : 3
---------------------------------

Edges of the tree for a Minimum Spanning Tree:

A -> {'B'}
---------------------------------
B -> {'E', 'C'}
---------------------------------
C -> {'D'}
---------------------------------
D -> {'F'}
---------------------------------
Program Runtime: 0.0010743141174316406 seconds
```

INSTRUCTIONS TO RUN THE PROGRAM

1. The project has been bundled into a zip file which contains two folders of the respective problems as well as the Project file.
2. The two given problems – Problem 1 and Problem 2 has been kept in two separate folders. For the execution of each problems, select the respective folders and import the program code into any IDE of your choice.
3. For the ease of execution, each problems has been written as 3 separate program codes named after the 3 different input files. The code in all the 3 programs are the same apart from the input path of the input file that has been modified to run each input file separately.
4. The path of the input file can be changed in any program code so as to analyse the different input files in a single execution.

CONCLUSION

In this project, for Problem 1, we aim to find the Single source shortest path in both directed and undirected weighted graphs for a given source vertex assuming there is no negative edge in our graph. We use Dijkstra's algorithm to solve the given problem 1. Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

For the Problem 2, we use Prim's algorithm to find a Minimum Spanning tree using edges that minimizes the total weight $w(T) = \sum (u,v) \in T\ w(u,v)$. The algorithm It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

REFERENCES

[1] Shortest path problem, Dijkstra's Algorithm, Wikipedia. Accessed: November 2018.

[2] Dijkstra's Shortest Path Algorithm using priority queue of STL, Geeks for Geeks Accessed: November 2018.

[3] Prim's Spanning tree algorithm, Tutorials Point. Accessed: November 2018.