L8 Report

Name: Priyanshu Garg

Enrollment No.: 18114058

Branch: Computer Science and Engineering

Batch: O3

Problem 1:

Implement Dijkstra's algorithm in Java to find all shortest paths between all pair of vertices in a weighted graph. Modify this algorithm to find all shortest paths between two nodes, if more than one occurs. Following this, compute betweenness centrality measure of each node.

Betweenness Centrality of a node/vertex, w is given as, $BC(w) = \sum_{u,v \in V} \frac{\sigma_{uv}(w)}{\sigma_{uv}}$, where, σ_{uv} is the number of all shortest paths between u and v; and $\sigma_{uv}(w)$ is the number of all shortest paths between u and v through w. (https://en.wikipedia.org/wiki/Betweenness_centrality)

Data structure that may be used: List, Set, Map, etc.
Input: A GML (Graph Modeling Language) file as a graph input.
Output: Betweenness Centrality of each node.
Note: Use JGraphT class in java (https://jgrapht.org) for this problem.

Test Case:

Input: P1.gml Adjacency Matrix:

V9	9	0	0	5	8	0	0	3	5	0
V8	1	4	3	0	0	5	0	0	0	
V7	5	2	0	0	0	5	0	0		
V6	7	8	0	0	0	0	0			
V5	0	6	0	0	8	0				
V4	8	0	0	0	0					
V3	0	8	5	0						
V2	0	0	0							
V1	4	0								
V0	0									
	V0	V1	V2	V3	V4	V5	V6	V7	V8	V9

Output:

w	BC(w)	w	BC(w)
V0	7.8333	V5	9.0000
V1	1.3333	V6	0.0000
V2	6.7500	V7	1.7500
V3	2.6667	V8	7.0000
V4	0.0000	V9	6.2500

Algorithm:-

Dijkstra's algorithm, 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.

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While *sptSet* doesn't include all vertices
a) Pick a vertex u which is not there in *sptSet* and has minimum distance value.
b) Include u to *sptSet*.
c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Problem 2:

Create a project/program in Java called Unscramble Word. Given a string of 'N' characters print all the words present in a dictionary of length 'M' such that 3 < M <= N.

Use dictionary present in Linux @ /usr/share/dict/words.

Implement this code in java and the student may use inbuilt data structures such as Maps, Sets, etc. (For fast execution, use of Trie is suggested).

Input: A String
Output: All unscrambled words of given string present in the dictionary categorized by length of word. Also print the total number of words of each length.

Test Case: Input:
"great" Output:

Length: 5	greta, grate, great, retag, targe	Count: 5
Length: 4	ager, gate, gear, geta, grat, rage, rate,	Count:

Algorithm:-

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field is used to distinguish the node as end of word node. A simple structure to represent nodes of the English alphabet can be as following,Inserting a key into Trie is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes

of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth. Searching for a key is similar to insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.