## Functions Folder

### Dijkstra.js

The Dijkstra algorithm is a searching algorithm given one has a graph and a source vertex. It finds all the shortest paths from the source to all the vertices within that graph. Each path within the graph from one vertex to another has a cost, so the algorithm has to take into mind the shortest possible path based on the amount of paths taken as well as their respective cost, in order to end up with the least accumulated amount of cost result.

The Dijkstra algorithm consists of the following steps:

1. Initialize the distances according to the algorithm.
2. Pick first node and calculate distances to adjacent nodes
3. Pick next node with minimal distance; repeat adjacent node distance calculations
4. Final result of shortest-path tree

The Dijkstra Algorithm was needed to determine the average shortest path for task D3 ii (b). The file contains a function called solve() that takes two parameters; the graph and the source vertex.

function solve(graph, s)

A variable called “solutions” was created to hold an object. The object is made up of two attributes, “s” while s has an attribute called “dist” which is at first initialized to 0. “dist” will keep a store of the distance taken from the source to the target vertex.

var solutions = {};

solutions[s] = [];

solutions[s].dist = 0;

The program then loops while the condition is true. Three variables are directly initialized; “parent” to null, “nearest” to null and “dist” to Infinity.

var parent = null;

var nearest = null;

var dist = Infinity;

The program then loops for every existing solution or a possible vertex to go to. The program checks if the possible vertex we landed on is the right one, if not the loop continues on to the next iteration. However, if not two variables are directly initialized; “ndistance” which holds the resulting distance of the n number of steps whilst “adjacent” holds the adjacent nodes of the current node the program is on. The program checks if there are any solutions, if so it chooses the nearest node with the lowest total cost and stores the distance and cost inside the variable “d”. The cost of each path is assumed to be 1 in our case. If “d” is less than “dist” in the variable “parent” is stored the reference parent the adjacent node is stored inside “nearest” whilst “dist” is set to “d” which is the accumulated distance so far.

//for each existing solution

for (var n in solutions) {

if (!solutions[n])

continue

var ndistance = solutions[n].dist;

var adjacent = graph[n];

//for each of its adjacent nodes...

for (var a in adjacent) {

//without a solution already...

if (solutions[a])

continue;

//choose nearest node with lowest \*total\* cost

var d = adjacent[a] + ndistance;

if (d < dist) {

//reference parent

parent = solutions[n];

nearest = a;

dist = d;

}

}

}

Once exiting the for loop the program checks if there are any solutions anymore if not the program breaks the while loop. However if not the parent’s solution path is extended with the nearest node and the distance up until the nearest node is set to the accumulated distance. At the very end the solutions are returned.

//no more solutions

if (dist === Infinity) {

break;}

//extend parent's solution path

solutions[nearest] = parent.concat(nearest);

//extend parent's cost

solutions[nearest].dist = dist;

return solutions;

Below is an example of how the output would look like for a given node:



### Keyword.js

This particular JavaScript file sets all the attributes of the D3 Word Cloud and draws it out on the screen. In order for one to see the word cloud the user has to click either on the edges or on the nodes and a Word Cloud of all the important keywords shared between the two vertices is displayed through the cloud. The file contains a function called createKeyCloud() passing the parameter keywords which is a list of all the keywords and the frequency of each word based on how many times it was used.

function createKeyCloud(keywords)

A variable called “frequency\_list” is declared to store a list of the words and their size. The program then puts every single word from the “keywords” inside the “frequency\_list”. Every element inside “frequency\_list” has two attributes; “text” which will store the word and “size” which will store the frequency of the word added with 2 so that the word size on the Word Cloud will be relative to the frequency of a word.

//Turn the KeyWords Into A Frequency List

frequency\_list = [];

for(var i = 0; i < keywords.length; i++) {

frequency\_list.push({

text: keywords[i].word,

size: keywords[i].weight + "2"

})

}

The width and height of the Word Cloud are stored inside the variables “w” and “h” respectively.

//width and height of the wordcloud

var w = 960,

h = 600;