WEB INTELLIGENCE PROJECT

ICS 2205

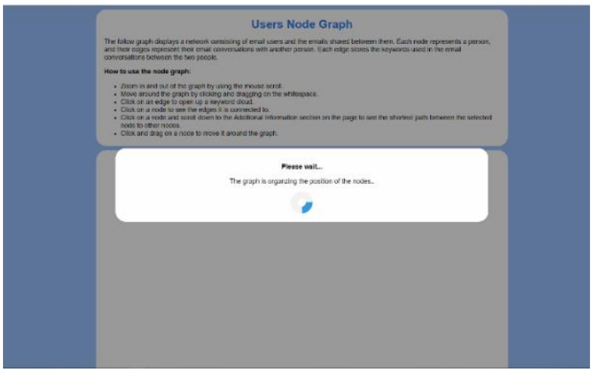
VALERIA HOLOMJOVA	ID:126623 A
GABRIEL CAMILLERI	ID:21299 M

14/01/2019

Table of Content

Description

Overview



Users Node Graph

The follow graph displays a network consisting of email users and the emails shared between them. Each node represents a person, and their edges represent their email conversations with another person. Each edge stores the keywords used in the email conversations between the two people.

How to use the node graph:

- Zoom in and out of the graph by using the mouse scroll.
- Move around the graph by clicking and dragging on the whitespace.
- Click on an edge to open up a keyword cloud.
- Click on a node to see the edges it is connected to.
- Click on a node and scroll down to the Additional Information section on the page to see the shortest path between the selected node to other nodes.
- Click and drag on a node to move it around the graph.

Please wait...

The graph is organizing the position of the nodes...

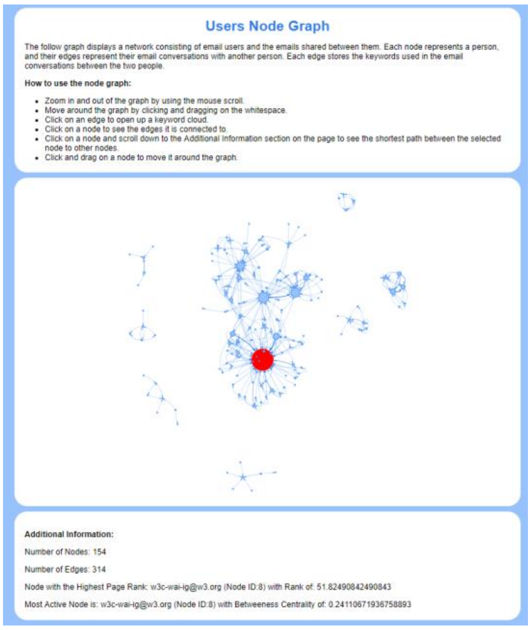
(Screenshot illustrating Loading Box)

Our design goal was to create a highly interactive node graph and user-friendly interface. We achieved this by creating a node graph that can easily be manipulated by users – each node in the graph is draggable and the user is able to pan around the graph. When the user opens the index of the project, a small loading screen appears for a few seconds to allow the graph to place the nodes in a non-overlapping manner.

Once the graph is loaded, the loading box disappears, and the user can now interact with the node graph. At the top of the page there is a description of the graph and instructions on how the user can interact with the graph. The user can then scroll to zoom in/out the graph and click and drag the mouse to pan around the graph. It is important to note that some nodes may still overlap others because we tried to decrease the loading time of the node graph. However, users can drag nodes to their liking which also highlights all the edges attached to the node.

Things to note about the node graph:

- The most active node in the network is coloured in red.
- The sizes of the nodes are calculated based on the number of emails sent and received by the node.
- The node graph is directed and people who send an email to themselves are included.
- The weight of the edges depends on the number of emails sent between the distinct sender and receiver.
- All information on the graph is found in the Additional Information section under the graph.



Users Node Graph

The follow graph displays a network consisting of email users and the emails shared between them. Each node represents a person, and their edges represent their email conversations with another person. Each edge stores the keywords used in the email conversations between the two people.

How to use the node graph:

- Zoom in and out of the graph by using the mouse scroll.
- Move around the graph by clicking and dragging on the whitespace.
- Click on an edge to open up a keyword cloud.
- Click on a node to see the edges it is connected to.
- Click on a node and scroll down to the Additional Information section on the page to see the shortest path between the selected node to other nodes.
- Click and drag on a node to move it around the graph.

Additional Information:

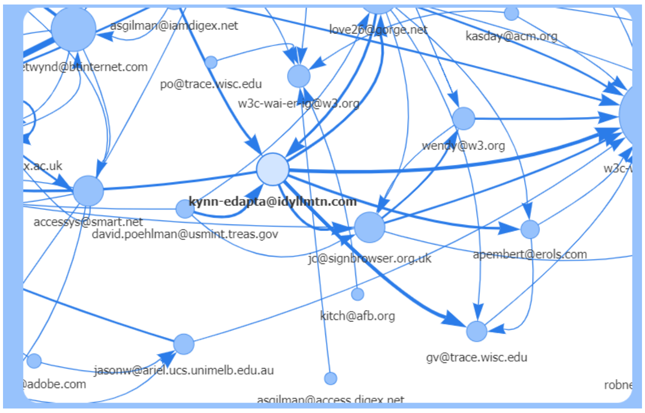
Number of Nodes: 154

Number of Edges: 314

Node with the Highest Page Rank: w3c-wai-ig@w3.org (Node ID:8) with Rank of: 51.82490842490843

Most Active Node is: w3c-wai-ig@w3.org (Node ID:8) with Betweeness Centrality of: 0.24110671936758893

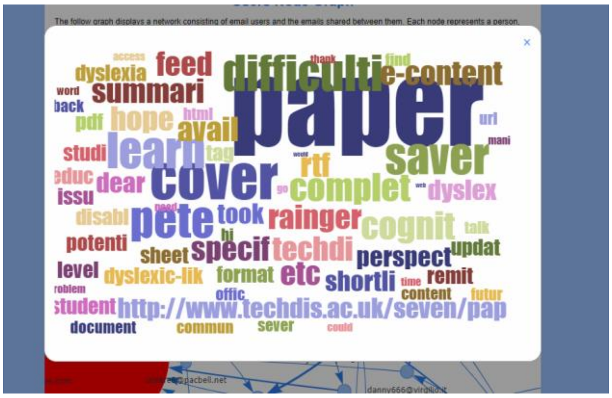
(Screenshot of fully loaded Front Page)



(Screenshot of dragging a Node)

Output from Interacting with the Graph:

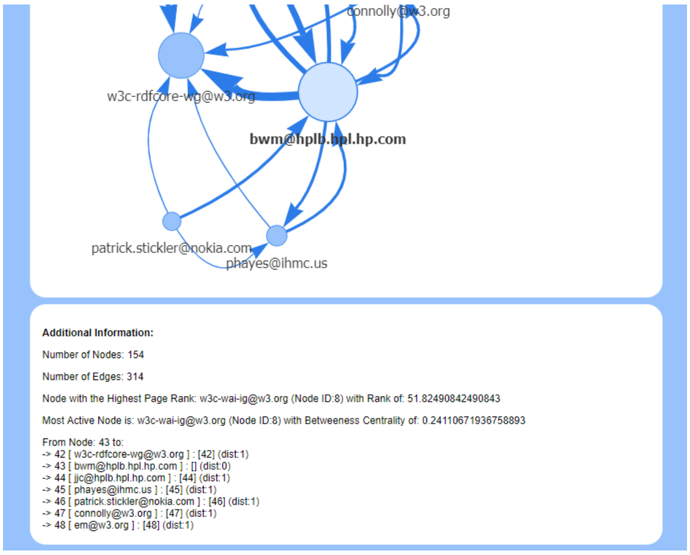
When the edges between two nodes is clicked on, a modal box appears with a keyword cloud represnting the keywords extracted from all the emails between the distinct two recipients. It must be noted that once an edge is selected and the keyword cloud popped up and was closed, clicking on the same selected edge right away will not cause the keyword cloud to pop up, the user will have to click somewhere else (like the blank space or another edge) and then select the edge for the keyword to pop up once more.



The follow graph displays a network consisting of email users and the emails shared between them. Each node represents a person.

(Screenshot of Keyword Cloud pop up)

When a node is clicked on, the shortest path between the selected node and all the other nodes is displayed in the Additional Information section (Dijkstra's Algorithm). It must be noted that there's a bug we encountered that when a node is clicked, sometimes a keyword cloud pops up. We have not found a way to prevent this, but the user can simply close the modal box pop up and the shortest paths of the node will still be displayed in the Additional Information.



```
Additional Information:

Number of Nodes: 154

Number of Edges: 314

Node with the Highest Page Rank: w3c-wai-ig@w3.org (Node ID:8) with Rank of: 51.82490842490843

Most Active Node is: w3c-wai-ig@w3.org (Node ID:8) with Betweeness Centrality of: 0.24110671936758893

From Node: 43 to:
-> 42 [ w3c-rdfcore-wg@w3.org ] : [42] (dist:1)
-> 43 [ bwm@hplb.hpl.hp.com ] : [] (dist:0)
-> 44 [ jjc@hplb.hpl.hp.com ] : [44] (dist:1)
-> 45 [ phayes@ihmc.us ] : [45] (dist:1)
-> 46 [ patrick.stickler@nokia.com ] : [46] (dist:1)
-> 47 [ connolly@w3.org ] : [47] (dist:1)
-> 48 [ em@w3.org ] : [48] (dist:1)
```

(Screenshot of Dijkstra's Algorithm output when click on the bwm@hplb.hpl.hp.com

Completion

All parts of the project were completed.

Distribution of Work

Valerija Holomjova was allocated D2 of the assignment and Gabriel Camilleri was allocated D3 of the assignment. Both partners were also allocated D1 and D4 of the assignment. Although individual roles were allocated, both partners would help each other with the other's sections when problems or difficulties were encountered.

Third Party Libraries/Tools

The following third-party libraries have been used in the project:

- Porter Stemmer Algorithm from https://tartarus.org/martin/PorterStemmer/php.txt (the code can be found in stemmer.php) – this includes functions for the Porter stemming algorithm in order to obtain the stem of a given keyword in functions.php.
- Stop words from https://gist.github.com/sebleier/554280 (these were declared in functions.php) – this includes an array of stop words that were used to filter keywords in functions.php.
- Vis.js from https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.js (included in header.php) - functions used to create and manipulate the node graph which is used in nodegraph.js.
- D3 from https://d3js.org/d3.v3.min.js (included in header.php) - needed to create the keyword cloud graphs.
- D3 Word Cloud Layout from https://github.com/jasondavies/d3-cloud/blob/master/build/d3.layout.cloud.js (the code can be found in d3.layout.cloud.js) - needed to create the keyword cloud graphs.
- ConThread-BC3 Corpus from http://mostafadehghani.com/conthread-bc3-v1-0-conversation-threads-annotated-bc3-email-dataset/ (the code can be found in data.xml) – the dataset.

Future Work

How can this work be extended? Provide 2 possible improvements.

The first improvement could be made to the functionality of the word cloud. For instance, users could have the ability to mouse over a keyword and see which other user conversations had the keyword. It also would be interesting if a semantic meaning could be derived from the keyword based on how it was used in the e-mail. This would give more context to the conversation data. For instance, the word "cover" was displayed in the keyword cloud screenshot found in the overview section of the documentation. This keyword could either be interpreted as a noun or a verb and could give a better understanding of what the conversations are about. This could be achieved through the use of natural language processing techniques.

The second improvement could be made to the visualization of the node graph. For instance, the nodes in the graph could be visually distinguished from each other based on their page rank score. (For example, darker nodes have higher page rank scores). Nodes could also carry more data on the user such as who they send/receive emails from frequently or which words do they use the most when sending e-mails. The nodes could also be placed in better positions so that they do not overlap.

Where and how do you envision the use of this work?

This kind of work could be used in catching illegal activity. For instance, the code can be adapted to search for terrorist recruitment or schemes by analysing the keywords in the conversation threads. This could help prevent potential world-wide terrorist incidents. It could also be used in politics when analysing the e-mails sent or received by government officials to determine whether any of them are corrupt.

This kind of work could also be used in consumerism. For instance, if the nodes were set as social media accounts and the edges were the keywords derived from the messages between two users, the keywords could be extracted to determine the potential tastes or interests of the users. For instance, if a user mentions her dog to her friend in a message, a social media site could try and show more advertisements of dog products that could be of interest to the user.

Source Code

Functions.php

The following file includes all the functions used to parse the XML document of emails into an array of document objects. The Porter Stemmer library and stop words were included at the top of the page to be used to obtain the keywords.

Document class

The following object stores a distinct sender and receiver, the number of emails that were passed between the two recipients and 3 different types of keywords array. The first array ($keywords) stores a collection of keywords that are used in all the emails between the two recipients. The second array ($keywordsFreq) stores the keywords and the frequency in which they occur in the $keywords array. The third array ($keywordsWeights) stores the keywords and their TF-IDF weights.

```
//Object storing the recipients, keywords and number of emails.
class Document{
    public $to=[];
    public $from=[];
    public $keywords=[];
    public $keywordsFreqs=[];
    public $keywordsWeights=[];
    public $emailNo=0;
}
```

Keyword class

The following object stores the name of a keyword and it's TF-IDF weight. It is used as an element in the $keywordsWeights array in a single Document object.

```
//Object storing the name and weight of a keyword.
class Keyword{
    public $word;
    public $weight;
}
```

getEmail() function

The following function returns an email address from a given string. It starts by tokenizing the string using the 'multiexplode()' function. It then loops through each token and checks whether a token has an '@' character and is not surround by quotation marks. If a token matches these conditions, the token is converted to a lower-case string, trimmed for excess characters and then returned. If no tokens match the given conditions, an empty string is returned.

getEmails() function

The following function works by looping through an array of strings and calling the 'getEmail()' function to obtain the email address in each string. Each email address is pushed into an array which is then returned. If one of the strings has no email address (the 'getEmail()' function returns an empty string when an email address isn't found), it is not added into the final array.

parseDocument() function

The following function creates documents using a SimpleXMLElement object containing all the data from the XML email file. It does this by iterating through each thread and email in the data and obtaining the email content (to be used as keywords) as well as the recipients. The email content is converted to an array of valid keywords using the ‘cleanUpMsg()’ function. Next the sender is extracted from the email and all the participants the email was sent to (including the people in the CC), are merged into an array. Note that only the emails of the recipients are extracted by calling the ‘getEmails()’ function in the case of receivers or the ‘getEmail()’ function in the case of a sender. The function then calls the ‘createDoc()’ function to use the sender, recipients and keywords to create a document (or append the email data to an existing one) that will be stored in the global array of documents.

```
function parseDocument($xml){
    //Iterating through each thread.
    foreach($xml->thread as $thread) {
        //Iterating through each email.
        foreach($thread->DOC as $email){
            //Obtains the messages in each email - and cleans it up into an array of keywords.
            $text = $email->Text->content;
            $keywords = cleanUpMsg($text);

            //Gets the To, From AND CC.
            $from = str_replace(array('>','<'), '', $email->From);
            //Split them by , and strip for ' ' then append them to an array of recievers.
            $to = explode(',',$email->To);
            //If CC exists split and strip the string then merge with the current recievers.
            if(isset($email->Cc)){
                $cc = explode(',',$email->Cc);
                $to = array_merge($to,$cc);
            }
            $to = getEmails($to);
            $from = getEmail($from);
            //Adds a document to the global document array with given keywords and recipients.
            createDoc($from, $to, $keywords);
        }
    }
```

After all the documents are created, the function calls the ‘addFreq()’ function to create an array of keywords and their freuqnecies for each document object. It also calls the ‘addWeights()’ function to create an array of Keyword Objects described above for each document that will store the keyword and its TF-IDF weight.

```
    //Adds keywords with frequencies and then keywords with TF-IDF weights to each document object.
    addFreq();
    addWeights();
}
```

appendDoc() function

The following function is used to create a new document using a given sender, receiver and array of keywords, or else find a document which matches the given sender and receiver and append the given keywords to it. It functions by iterating through each document in the global document array and finding one who’s sender and receiver match the given one. If this condition is met, the given keywords are appended to the found document and the variable representing the number foe mails is incremented. If no document matching the given recipients is found, a new document is created like in the ‘createDoc()’ function and added into the global array of documents.

createDoc() function

The following function is used to create or update a document given a sender, an array of receiver and an array of keywords from an email. First, the function loops through each of the receivers given. If the global documents array is empty, it creates a new document with the given sender, current receiver, array of keywords and increments the number of emails in the new document. It then pushes the new document in to the global array of documents. If the global array wasn’t empty, it calls the ‘appendDoc()’ function to create a new document or append the keywords to an existing one with the given sender, current receiver, array of keywords.

addWeights() function

The following function iterates through each document in the global array of documents and creates an array of the keyword objects for each document. This is done by accessing each keyword from the array containing the keywords and frequencies in each document. The TF-IDF weight of each keyword is then calculated using the following formula which calculates the weight ***W*** for a given a term ***t*** in document ***d***:

$$W = T\log(\frac{N}{D})$$

Where

- W = Weight of term ***t*** in document ***d***
- T = Number of occurrences of term ***t*** in document ***d***
- D = Number of documents containing the term ***t***
- N = Total number of documents

The value of D in the formula is calculated using the ‘getDocNo()’ function which returns the number of documents containing a given keyword. T is obtained using the frequency value of the keyword in the current iteration. N is obtained by using the standard library function ‘sizeof()’ to count the number of document objects in the global document array.

After the TF-IDF weight for a keyword is calculated, the value is rounded so that the weights can be used to calculate a pixel length for the words in the word cloud later. A keyword object is then created with the keyword name and calculated TF-IDF weight and then pushed into the $keywordsWeights array of the current document object.

```
//Adds keywords with weights to each document, the keywords are keyword class objects.
function addWeights(){
    global $documents;
    $totalDoc = sizeof($documents); //Total number of documents.
    foreach($documents as $document){
        //For each word in the document calculate the weights.
        foreach($document->keywordsFreqs as $key => $value){
            //Calculates the TF-IDF weight for a keyword.
            $TFIDF = round($value * log($totalDoc/getDocNo($key)));
            $weighted = new keyWord();
            $weighted->word = $key;
            $weighted->weight = $TFIDF;
            //Adds a keyword and it's TF-IDF weight to the keyword weight array of the current document.
            array_push($document->keywordsWeights,$weighted);
        }
    }
}
```

multiexplode() function

The following function splits a string using several delimiters. It works by replacing all the occurrences of the delimiters in the string with one of the delimiters. And then returning the tokens obtained by splitting the string with that one delimiter.

```
//Splits a string by multiple delimiters.
function multiexplode ($delimiters,$string) {
    $temp = str_replace($delimiters, $delimiters[0], $string);
    return explode($delimiters[0], $temp);
}
```

cleanUpMsg() function

The following function takes the content of an email and parses it into individual keywords. It functions by creating an array to store the keywords. Then, it tokenizes the message by the space, '\n' and '\r' characters. Each of the tokens obtained is converted to lowercase and any html entities found in the token string (such as "&lt;") are converted into characters using the 'htmlspecialchars_decode()' standard library function. The token string is then stripped for all unwanted characters such as (!?#") from both ends of the string. Once this is done, the function checks whether the token string is in the stop words array. If the token string is not in the stop words array and is not empty, the stem is extracted from the token string using the third-party Porter Stemmer library function. The stem is then pushed into the array of keywords. At the end of the function the array of keywords is returned.

```
//Returns an array of clean key words - without stop words, unwanted characters and stemmed.
function cleanUpMsg($message){
    global $stopWords;
    $keywords = [];
    //Tokenizer to parse text.
    $tok = strtok($message, " \n\r");
    while ($tok !== false) {
        //Make word lowercase and strip '.' from end of string!
        $tok = trim(htmlspecialchars_decode(strtolower($tok)),'.,:;\'"()?!-#<>*&%|\/$@~');
        //Check if it is a stop word - if not add to the array!
        if (!in_array($tok, $stopWords) && !empty($tok)) {
            //Porter Stemmer Library - gets the stem of the word.
            $tok = PorterStemmer::Stem($tok);
            array_push($keywords, $tok);
        }
        $tok = strtok(" \n\r");
    }
    return $keywords;
}
```

addFreq() function

The following function iterates through each document in the global array of documents and creates an key-value array of the keywords in the current document and their frequencies using the standard library 'array_count_values()' function. The new array is set as the $keyWordsFreq array for each of the document objects.

getDocNo() function

The following function iterates through each document in the global array of documents and returns the number of documents containing a given keyword. When iterating through a document, if the given keyword is found as one of the keys in the $keywordsFreq array of a document, a variable keeping track of the number of documents containing the word is incremented. At the end of the function this variable is returned.

Header.php

The following file includes all the internal scripts, external scripts, CSS and third-party libraries that were used for the project. The XML file is loaded here and the 'parseDocument' function is called to populate the global array of documents with Document objects from the file 'functions.php'. It also converts all these document objects from a PHP array to a JS array of document objects. It does this by iterating through each PHP document, and required contents of the document such as the keywords (with TF-IDF weights), the recipients and the number of emails into a JS object. It then pushes this object in a JS global array of such objects which will be used to generate the node Graph and word cloud. Note that the function to create a node graph is called by an 'onload()' event found in the opening body tag of the page so that it executes after the page has been loaded so that the data is ready.

Index.php

The following page is where all the content of the project is displayed to the user. It includes text-containers with information for the user on what the node graph is and how to use it. It has a container for the node graph and two invisible modal boxes. One modal box is used to display a keyword cloud when clicking on a link between two nodes. Another modal box is used to display a mini loading screen so that the node graph could try find suitable positions for the nodes which then disappears when the graph is fully loaded and can be used. At the bottom is an Additional Information where all information about the graph (such as the number of nodes, the most active node, etc..) is displayed.

Footer.php

The following page includes the closing tags of the html page and scripts to close the word cloud modal boxes. There are two methods of closing the word cloud modal box – the user can either click on the cross or click anywhere outside the box to close it. Once the box is closed, the html text content of it is wiped so that another graph could be displayed when its opened.

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 [1]

solve() function

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.

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

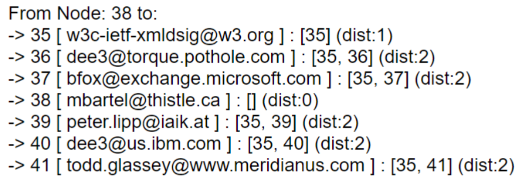
Once exiting the for loop the program checks if there are any solutions anymore if not the program breaks the while loop. However, if not the parent’s solution path is extended with the nearest node and the distance up until the nearest node is set to the accumulated distance. At the very end the solutions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



```
From Node: 38 to:
-> 35 [ w3c-ietf-xmldsig@w3.org ] : [35] (dist:1)
-> 36 [ dee3@torque.pothole.com ] : [35, 36] (dist:2)
-> 37 [ bfox@exchange.microsoft.com ] : [35, 37] (dist:2)
-> 38 [ mbartel@thistle.ca ] : [] (dist:0)
-> 39 [ peter.lipp@iaik.at ] : [35, 39] (dist:2)
-> 40 [ dee3@us.ibm.com ] : [35, 40] (dist:2)
-> 41 [ todd.glassey@www.meridianus.com ] : [35, 41] (dist:2)
```

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.

createKeyCloud() function

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. Whilst the styling of the d3 layout cloud are set.

A function called draw() passing the parameters “words” and “bounds” are passed in order to draw the word cloud on screen. Giving the whole thing style font to text and any attributes like width, height translation and transformation of each word.

```
//function to draw the wordcloud
function draw(words, bounds) {
        d3.select(".word-graph").append("svg")
            .attr("width", w)
            .attr("height", h)
            .append("g")
            .attr("transform", "translate(450,300)")
            .selectAll("text")
            .data(words)
            .enter().append("text")
            .style("font-size", function (d) { return d.size + "px"; })
            .style("font-family", "Impact")
            .style("fill", function (d, i) { return fill(i); })
            .attr("text-anchor", "middle")
            .attr("transform", function (d) {
                return "translate(" + [d.x, d.y] + ")rotate(" + d.rotate + ")";
            })
            .text(function (d) { return d.text; }); }
```

Functions.js

The functions JavaScript file contains a number of functionalities each kept within their own function.

getNodes() function

The first function is getNodes() which takes “document” (a list of documents) as a parameter. Two variables are directly initialized; “nodes” which will eventually store a list of nodes and “counter” which will be used to give each node an id. The program loops through every document checks if either the sender or receiver already exist in the “nodes” list, if not the sender/ receiver is pushed with its id and label inside the list and the counter is incremented. At the end the list of nodes is returned.

```
function getNodes(documents) {
   var nodes = [];
   var counter = 1;
   //For each document if sender is not in array send it!
   for (var x = 0; x < documents.length; x++) {
      //For each document check if sender exists in emails, if not, add it.
      if (!ifExists(nodes,documents[x].to)) {
         nodes.push({
            id: counter,
            label: documents[x].to
         }); counter++;
      }
      if (!ifExists(nodes,documents[x].from)) {
         nodes.push({
            id: counter,
            label: documents[x].from
         }); counter++;
      }
   }
   return nodes;
}
```

getIndex() function

The second function is getIndex() passing “nodes” a list of all the nodes and “string” the label of a node as parameters. Within the function the program loops through all the nodes, checks if the string match with any label of a node, once a match is found an index is returned. If not matches are found 0 is returned.

ifExists() function

The third function is ifExists(), passing “nodes” and “string” as parameters. A variable Boolean “found” is directly initialized to false. Then using a for loop the program loops through all the nodes compares the label of a node with the string, if a match if found the variable “found” is set to true and the for loop breaks and finally the variable “found” is returned.

ifExists() function

The fourth method is getEdges() passing two parameters “documents” a list of documents and “nodes” a list of nodes. A variable “edges” is directly initialized to store a list of edges. For each document are pushed a to and from vertices to represent an edge by calling getIndex() and passing the list of nodes and the node id as parameters. As attributes for every edge there exist four; “from” and "to" which will contain node ids , "keywords" a list of all the keywords shared between the two nodes and the "width" which will contain the amount of emails exchanged between the two. Finally the list of edges is returned.

```
    for (var x = 0; x < documents.length; x++) {
        //each edge will contain from to nodes, the keywords between those two nodes and the number of emails sent
        edges.push({
            from: getIndex(nodes, documents[x].from),
            to: getIndex(nodes, documents[x].to),
            keywords: documents[x].keywords,
            width: documents[x].emailNo
        })
    }
    return edges;}
```

getNodeAmount() function

The fifth function is getNodeAmount() passing "nodes" as parameter. The function return the amount of nodes in the list by looping and keeping a counter of all the nodes. Finally that counter is returned. This was used mainly to display the amount of nodes on screen for task D3.

```
//get the amount of nodes
function getNodeAmount(nodes) {
    var nodeAmount = 0;
    for (var x = 0; x < nodes.length; x++) {
        nodeAmount = nodeAmount + 1;
    }
    return nodeAmount;
}
```

getActivity() function

The sixth function is getActivity() passing "temp_edges" a list of all the edges and "temp_nodes" a list of all the nodes as parameters. The variable "activity" is directly initialized to store a list of activities made by two nodes. The function tells how many emails a node sent and received. Finally the function returns the list of active nodes. The width of edges will be based on the number of emails exchanged.

```
    for (var x = 0; x < temp_nodes.length; x++) {
        var total = 0;
        for(var y = 0; y < temp_edges.length; y++){
            if(temp_edges[y].to == temp_nodes[x].id || temp_edges[y].from == temp_nodes[x].id){
                total +=  parseInt(temp_edges[y].width, 10);
            }
        }
        activity.push(total);
    }
    return activity;
}
```

getAveragePaths() function

The seventh function is getAveragePaths() which returns the global variable "averagePaths" which is a list of all short paths from a source node to all possible target nodes.

betweenCentrality() function

The eight function is betweenCentrality() passing "temp_edges" and "temp_nodes" as parameters.

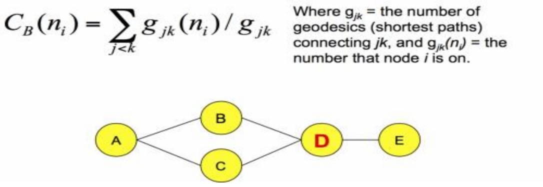
Four variables are directly initialized; “betweenCentrality” which stores the list of nodes containing their own betweenness centrality, “shortPathsPassingThru” which is a list where each node keeps a list of paths that pass through it, “graph” an object which will store the graph and “layout” an object storing the nodes and all adjacent nodes to it.

The program loops through every node inside “temp_node” and then loops through every edge inside “temp_edge”, it compares the counter “x” if it is either equal to the to node inside the “temp_edge”, else if it is equal to the from node or if it is not the latter two. If it happens to be the first condition the “froms” list is appended with the node from and the list “allPossiblePaths” as well, if second condition the node to is appended inside the lists “tos” and “allPossiblePaths” otherwise the list “allShortPaths” will be pushed an object containing the attributes to and from.

```
for (var y = 0; y < temp_edges.length; y++) {

   if (x == temp_edges[y].to) {
      froms.push(temp_edges[y].from);
      //for dijkstra
      allPossiblePaths.push('' + (temp_edges[y].from));
   }
   else if (x == temp_edges[y].from) {
      tos.push(temp_edges[y].to);
      //for dijkstra
      allPossiblePaths.push('' + (temp_edges[y].to));
   }
   else {
      allShortPaths.push({
         to: temp_edges[y].to, from: temp_edges[y].from});
```

The reason behind this for loop is for two specifically, the first is because in order to calculate the betweenness centrality of each node one must calculate and sum up the number of short paths passing through a node and all the other short paths not passing through them. The formula for this is:



$$C_B(n_i) = \sum_{j<k} g_{jk}(n_i) / g_{jk}$$

Where g_{jk} = the number of geodesics (shortest paths) connecting *jk*, and $g_{jk}(n_i)$ = the number that node *i* is on.

The second reason was to keep a track of all the possible short paths from one node to its adjacent vertices and keep them in a list. This was done in order to calculate using the Dijkstra algorithm all the possible short paths from a source node to all possible reachable nodes. It was initially thought to connect the two reasons together as well to calculate the betweenness centrality. Since, one needs to keep track of all the possible short edges between nodes, however, the gave us problems as it was returning a betweenness centrality of infinity. So, in order to have a result although not an accurate one we decided that instead we keep track of all the possible edges between two nodes since each edge between a node is a short path. However, if one had to make it accurate one would definitely make use of the Dijkstra algorithm. So the algorithm was used to display all the short paths a node can take on screen.

```
    layout[x] = allPossiblePaths;
shortPathsPassingThru.push({
      node: x,
      to: tos,
      from: froms,
      allothers: allShortPaths        });}
```

The program then loops again through all the nodes stores the start node in variable “start” and sends the variables “graph” and “start” as parameters to the method solve in the Dijkstra.js file which returns a list of solutions and keeps it in a variable “solutions”. It then goes through every solution which contains a list of short paths taken from a source node to all reachable nodes and pushes the data in the list “averagePaths”. The list stores objects which contains two attributes a “from” which stores the counter of a for loop and “to” which stores the possible reachable nodes and the distance of the path taken.

```
            averagePaths.push({
                from: x,
  to: " -> " + s + " [ " + label + " ] " + ": [" + solutions[s].join(", ") + "]   (dist:" + solutions[s].dist + ")"
});}}
```

The program then loops to find the betweenness centrality of each node by dividing the number of short paths passing through the node by the number of other paths not passing through. The betweenness centrality is pushed inside the list called “betweenessCentrality” along with the node id. The loop also keeps track of the highest
betweenness centrality so that later the node with the highest one can be highlighted red on the force graph. That node is then returned.

```
    for (var x = 0; x<shortPathsPassingThru.length; x++) {
        var pathsThru = shortPathsPassingThru[x].to.length + shortPathsPassingThru[x].from.length;
        var otherPaths = shortPathsPassingThru[x].allothers.length;
        var betweeness = pathsThru / otherPaths;

        betweenCentrality.push({
            node: shortPathsPassingThru[x].node,
            betweeness: betweeness,
        });

        if (betweeness >= max) {
            max = betweeness;
            activeNode = { node: betweenCentrality[x].node, betweenCentrality:
betweenCentrality[x].betweeness };
        }
```

findPageRankTo() function

The last two method involve calculating the page rank of each node. The first method findPageRankTo() taking “temp_nodes” and “temp_edges” as parameters calculate the page rank given off by a node. That is done by dividing 1 by the number of edges originating from the node. In order to do that the program loops through the “temp_nodes” and then through the “temp_edges”. It takes the first node and tries to find where that nodes has any edges originating from it in the “temp_edges”. If a match is found a list of “tos” meaning the nodes connected to that node is kept so that later the page rank originating from a node is kept by diving by the length of “tos”. Later on that page rank given off by a node is stored in the list “pgRank” where the node id, a list of “tos” and the page rank is stored. The variable pgRank is then returned.

```
    for (var x = 0; x < temp_nodes.length; x++) {
        var tos = [];
        var from = 0;
        var width = 0;
        for (var y = 0; y < temp_edges.length; y++) {
            if (temp_nodes[x].id == temp_edges[y].from) {

                from = temp_edges[y].from;
                tos.push({
```

```
                to: temp_edges[y].to,
                width: temp_edges[y].width
            });
        }
    }
```

However, that is not the page rank of a specific node rather we have found the page rank given by a node depending on the number of edges coming out from a node. Therefore, in order to find the page rank of a node one must find any edges coming into a node and add the page rank given to the node by each edge. The problem that one might find here is, when a node doesn't have any edges coming into it. In order to solve the problem a damping factor was created so that each node does not start off with a page rank of 0.

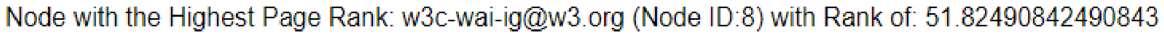
findPageRankFrom() function

The function findPageRankFrom() which takes "pgRank" as a parameter initializes the damping factor to a 0.825. The program then loops through all the "pgRank" list and within the for loop creates another loop to loop again through the list. This is so as to find the sum of page ranks given to a nodes by incoming edges. The result is stored in "rankOfNode" which is then pushed to "pgRank2" taking into account the node id, the list of nodes sending to this particular node and the "rankOfNode" as attributes. The program then finds the node with the highest page rank and stores it in an object called "maximumRankedNode". A calculation is made to solve the previous mentioned problem by using the damping factor, the calculation is finding the difference of 1 from the "damping_factor" and diving it by the length of the list "pgRank2", the result is added to the rank of each node. At the very end the node with highest page rank is returned and displayed on screen.

```
    for (var x = 0; x < pgRank2.length; x++) {
        var y = (1 - damping_factor) / pgRank2.length;

        pgRank2[x].rankOfNode = pgRank2[x].rankOfNode + y;
    }
    return maximumRankedNode;

}
```



```
Node with the Highest Page Rank: w3c-wai-ig@w3.org (Node ID:8) with Rank of: 51.82490842490843
```

Nodegraph.js

createNodeGraph() function

The next and last created JavaScript file is nodegraph.js. Within the file is a function called createNodeGraph() which takes list of documents as a parameter.

```
function createNodeGraph(documents)
```

Within the function are a list of variables directly initialized. The first one is "temp_nodes" which calls getNodes(documents) to store a list of nodes. The second one is "temp_edges" which calls getEdges(documents, temp_nodes) to store a list of edges. Third is "NODES" which stores a list. Fourth is "pgRank" which calls findPageRankTo(temp_nodes, temp_edges) which stores a list of nodes and their page rank they give off themselves. The fifth one is "maximumRankedNode" which stores the node with max rank by calling findPageRankFrom(pgRank). The sixth one is "activeNode" which stores the active node with highest between centrality returned from betweenCentrality(temp_edges, temp_nodes). Seventh variable is pgRank2 which stores the result from getPgRank2() and "activity" which stores a list of nodes with the number of emails each exchanged from getActivity(temp_edges, temp_nodes).

Next the program loops through all the nodes and inside "NODES" pushes the node id, its label, it activity size adding 5 to it (as node size) and the colour red if it is the most active node. If no much found the node is given the same feature without giving it any colour.

```
for (var x = 0; x < temp_nodes.length; x++) {
    //the most active node is coloured red
    if (temp_nodes[x].id == activeNode.node) {
        NODES.push({
            id: temp_nodes[x].id,
            label: temp_nodes[x].label,
            size: activity[x] + 5,
            color: 'red',
        });
    }
    //otherwise push in a normal coloured node
```

Inside the variable "nodes" is stored a list of "NODES" which goes through the process of vis.Dataset() and the same thing with "edges". "Vis.js comes with a flexible DataSet, which can be used to hold and manipulate unstructured data and listen for changes in the data. The DataSet is key/value based. Data items can be added, updated and removed from the DataSet, and one can subscribe to changes in the DataSet. The data in the DataSet can be filtered and ordered, and fields (like dates) can be converted to a specific type. Data can be normalized when appending it to the DataSet as well." [2] The amount of nodes is stored in "nodeAmount" by calling getNodeAmount(nodes).

The program then display all the result from the visualization and reading done from the graph and display it in the correct paragraph containing its respective id.

A network of nodes is created as well as the default options are set within the variable "options". The options are then rendered on each node inside the network.

This was all done using the vis.js library. It was first thought to make use of d3 libraries for the force graph however after some trial and errors it was decided to change to a much simpler library with more features one can use.

Calling the Network() method from vis and passing "container", "data" and the "options" variable the network is set up with a width and height of 1000 and 650 respectively. Whilst turning off the physics of the network and just have a static display, where the user can drag around the edges and nodes.

When the network senses a click on a node, the average shortest paths are displayed beneath the graph. If the properties of that node are larger than 0, the children and the nodes which the child are connected to are displayed in the console.

When the network senses an edge click the program outputs in the console the id and the nodes linked with the edge, as well as displaying the word cloud of the common words shared between the two nodes.

References

[1]: https://brilliant.org/wiki/dijkstras-short-path-finder/ - Examples

[2]: http://visjs.org/docs/data/dataset.html - Overview