UNIVERSITY OF CALIFORNIA, IRVINE

Graph Center

Term Project Report

MAZHAR ABBASS
73877540
12/6/2017

This report is on Term project in the course EECS 221. The project was to develop a graph solver to solve a real world problem using graph algorithms. A web application was to built to provide a user interface for the solver

Contents

Acknowledgements

I would like to appreciate Professor Sheu and TA Abhinaya for assigning such an interesting and challenging project to me. This project has definitely improved my understanding of graph theory. It also enhanced my software development skills on back-end as well as front-end side of programming. This was one of the three projects in the course and all the project work truly made learning more fun and practical.

Introduction

In graph theory there are numerous algorithms that come handy in day to day applications. A lot of research goes on in analyzing algorithms to find those which have the potential in solving modern day problems. This research started long ago and still continues. Various problems facing humans have been solved or optimized to say the least. Graph theory plays a major role in disciplines like Medicine, Astronomy, Social Networking, Human behavior etc.

Graphs can be used to model many types of relations and processes in physical, biological, social and information systems. Many practical problems can be represented by graphs. Emphasizing their application to real-world systems, the term network is sometimes defined to mean a graph in which attributes (e.g. names) are associated with the nodes and/or edges.

In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another. A similar approach can be taken to problems in social media, travel, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science. The transformation of graphs is often formalized and represented by graph rewrite systems. Complementary to graph transformation systems focusing on rule-based in-memory manipulation of graphs are graph databases geared towards transaction-safe, persistent storing and querying of graph-structured data.

In this project my aim is to design a graph solver that calculates the center or centers of a graph. An optimized solution of this graph problem can offer huge benefits in real world situations. The task here is Given a strongly connected graph. Find the vertex for which the length of shortest path to the farthest vertex is the smallest. This vertex is called center of the

graph. There can be more than one vertices that have a central location. One area where this could help is, suppose that a hospital, a fire department, or a police department, should be placed in a city so that the farthest point is as close as possible. For example a hospital should be placed in such a way that an ambulance can get as a fast as possible to the farthest situated house (point). Suppose that a new fire department should be constructed in this area of the city. The best location would be the region that is situated as close as possible to the farthest point. A way of finding such a region is by running Graph Center algorithm on selected vertices (regions). This report entails all the details and approaches that were used in the project. The project also involves creating a web application to present the graph solver as a service to clients. It is an interactive app wherein a client provides a graph input and the app returns the centers. The information can be used in whichever way the clients want to.

History

So far there have been quite a few successful and varied attempts to solve this problem . The problem is actually solved using various classic shortest path algorithms and some enhancement of the same. The all pair shortest path algorithm and its flavors have been used to find out the the nodes with central location in a network. A different idea which involves a mathematical approach towards the problem is also present. Trying to find eccentricity , diameter , radius etc of a graph or for each node and then using that to calculate the central vertices is at the core of this idea. Although most of the reported works on the center problems are for trees or for general graphs, more and more attention is being paid for the classes of graphs that are between these two extremes, viz. cactus graphs and partial k-trees. A lot of research still goes on in improving the time and space complexities of the underlying algorithms of the graph center problem.

There is actually a very interesting story of how graph theory became a thing. Königsberg was a city in Russia situated on the Pregel River, which served as the residence of the dukes of Prussia in the 16th century. Today, the city is named Kaliningrad, and is a major industrial and commercial centre of western Russia. The river Pregel flowed through the town, dividing it into four regions. In the eighteenth century, seven bridges connected the four regions. Königsberg people used to take long walks through town on Sundays. They wondered whether it was possible to start at one location in the town, travel across all the bridges without crossing any bridge twice and return to the starting point. This problem was first solved by the prolific Swiss mathematician Leonhard Euler, who, as a consequence of his solution invented the branch of mathematics now known as graph theory. Euler's solution consisted of representing the

problem by a “graph” with the four regions represented by four vertices and the seven bridges by seven edges.

Graph Theory is now a major tool in mathematical research, electrical engineering, computer programming and networking, business administration, sociology, economics, marketing, and communications; the list can go on and on. In particular, many problems can be modelled with paths (see the definition below) formed by traveling along the edges of a certain graph. For instance, problems of efficiently planning routes for mail delivery, garbage pickup, snow removal, diagnostics in computer networks, and others, can be solved using models that involve paths in graphs.

Applications

This problem serves a lot of real world applications. Today's world is a connected world. There are about a billion people on FaceBook connecting and interacting daily. The whole network of social platforms resembles our universe. Thus there is a huge demand for faster and better graph algorithms to solve the upcoming challenges due to the influx of more people into social networks like FaceBook and Twitter.

The graph center problem can cater to other computer science graph problems as well. But this report is about its role in real world applications. For instance, to find closeness centrality in a network, for example a website's closeness to incoming and outgoing nodes, this algorithm can be very effective.

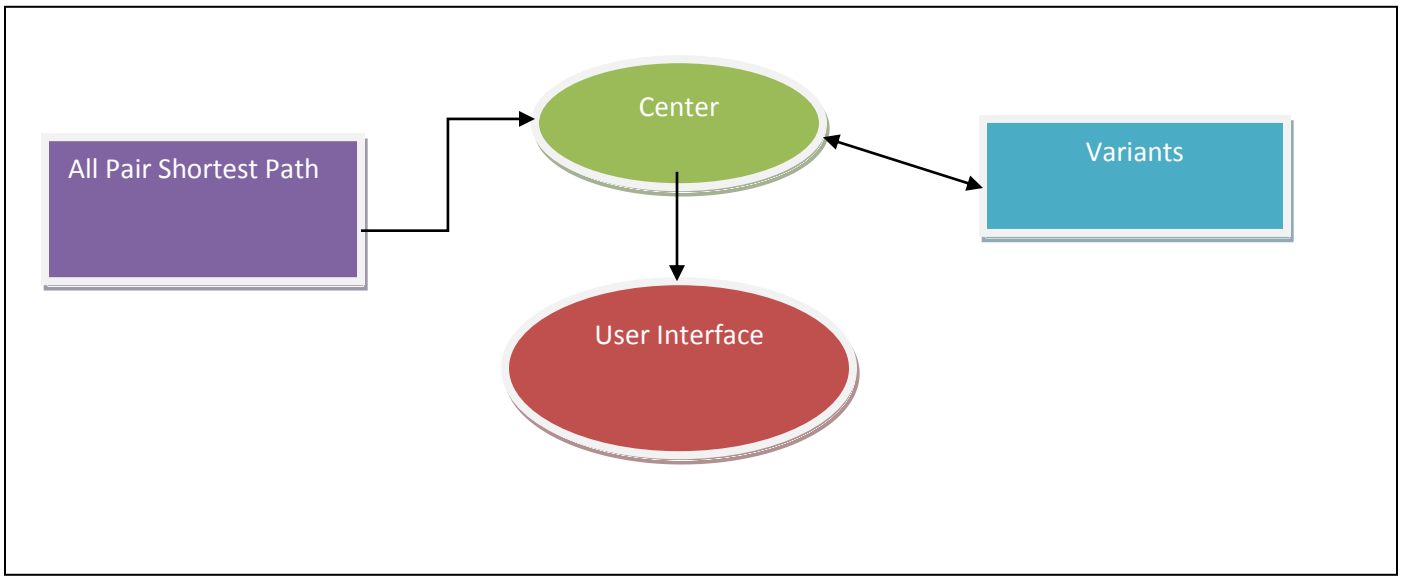
To find the degree of closeness in a payment social network like Venmo. Here there can be positive edge weights denoting the transfer of cash from one client to another where the subject node is having a profit and similarly a negative edge denotes the subject node is having a loss of some amount.

Approach

The approach followed by me in this project can be summarized in following steps

- Understand the problem statement and research works related to it
- Choose an approach to move forward with. In the shortest path approach was the winner
- Compare and then choose the underlying shortest path algorithm
- Build code on top of the underlying algorithm to calculate the central node
- Develop a user interface to demonstrate the solvers' ability to solve the problems
- Create some variations of the algorithm and try to solve some related problems

Block Diagram



Components

Underlying Algorithm

I am using "floyd-warshall" algorithm to find the all pair shortest pair algorithm. The solution is straight-forward: find the lengths of shortest paths between every pair of vertices, and then for each vertex find the length of shortest path to the farthest vertex. Center of the graph is the vertex for which this value is minimal.

For each pair of vertices find the length of the shortest path between them. Find vertex i such that the length of shortest path to the farthest vertex is the smallest. Output vertex i found previously.

Other Algorithms

two of the popular algorithms used to attack this shortest-path problem in graph theory, Floyd's algorithm and Dijkstra's algorithm. I will briefly describe each algorithm, compare the two, and then provide guidelines for choosing between them. In addition, I will describe the results of test cases I ran on both algorithms.

The shortest path problem is the problem of finding a path between two vertices of a graph with a minimum total edge weight. The application of shortest path algorithms is useful in solving many real-world problems. They can be applied to network and telecommunications problems in order to find paths with the lowest delay. Additionally, mapping software utilized these types of algorithms to find the optimum route between two cities.

Both the Floyd's algorithm and Dijkstra's algorithm are examples of dynamic programming. Dynamic programming is a technique for solving problems by breaking them into smaller sub-problems. Each sub-problem is solved only once and the results are stored for later use. Dynamic programming algorithms are more resource intensive than their non-dynamic counterparts, which give them a higher space complexity. However, this higher space complexity should be offset with a lower time complexity. This time complexity reduction will often be linear, and while the theoretical gain is minimal, the actual performance increase may be significant.

Floyd's Algorithm

Stephen Warshall and Robert Floyd independently discovered Floyd's algorithm in 1962. In addition, Bernard Roy discovered this algorithm in 1959. This algorithm is sometimes referred to as the Warshall-Floyd algorithm or the Roy-Floyd algorithm. The algorithm solves a type of problem call the all-pairs shortest-path problem, meaning that it finds the shortest path between all the vertices of a given graph. Actually, the Warshall version of the algorithm finds the transitive closure of a graph but it does not use weights when finding a path. The Floyd algorithm is essentially the same as the Warshall algorithm except it adds weight to the distance calculation.

This algorithm works by estimating the shortest path between two vertices and further improving that estimate until it is optimum. Consider a graph G, with Vertices V numbered 1 to n. The algorithm first finds the shortest path from i to j, using only vertices 1 to k, where k<=n. Next, using the previous result the algorithm finds the shortest path from i to j, using vertices 1 to k+1. We continue using this method until k=n, at which time we have the shortest path between all vertices. This algorithm has a time complexity of O(n3), where n is the number of vertices in the graph. This is noteworthy because we must test up to n2 edge combinations.

Dijkstra's Algorithm

Edsger Dijkstra discovered Dijkstra's algorithm in 1959. This algorithm solves the single-source shortest-path problem by finding the shortest path between a given source vertex and all other vertices. This algorithm works by first finding the path with the lowest total weight between two vertices, j and k. It then uses the fact that a node r, in the path from j to k implies a minimal path from j to r. In the end, we will have the shortest path from a given vertex to all other vertices in the graph. Dijkstra's algorithm belongs to a class of algorithms known as greedy algorithms. A greedy algorithm makes the decision that seems the most promising at a given time and then never reconsiders that decision.

The time complexity of Dijkstra's algorithm is dependent upon the internal data structures used for implementing the queue and representing the graph. When using an adjacency list to represent the graph and an unordered array to implement the queue the time complexity is O(n2), where n is the number of vertices in the graph. However, using an adjacency list to represent the graph and a min-heap to represent the queue the time complexity can go as low as O(e*log n), where e is the number of edges. It is possible to get an even lower time

complexity by using more complicated and memory intensive internal data structures, but that is beyond the scope of this paper.

Comparison

When using a naive implementation of Dijkstra's algorithm the time complexity is quadratic, which is much better that the cubic time complexity of the Floyd's algorithm. However, Dijkstra's algorithm returns only a subset of Floyd's algorithm. Specifically, it returns the shortest path between a given vertex and all other vertices while the Floyd's algorithm returns the shortest path between all vertices. It is interesting to note that if you run Dijkstra's algorithm n times, on n different vertices, you will have a theoretical time complexity of O(n* n2)=O(n3). In other words, if you use Dijkstra's algorithm to find a path from every vertex to every other vertex you will have the same efficiency and result as using Floyd's algorithm.

Both Floyd's and Dijkstra's algorithm may be used for finding the shortest path between vertices. The biggest difference is that Floyd's algorithm finds the shortest path between all vertices and Dijkstra's algorithm finds the shortest path between a single vertex and all other vertices. The space overhead for Dijkstra's algorithm is considerably more than that for Floyd's algorithm. In addition, Floyd's algorithm is much easier to implement. In most cases, for a small values number of vertices, the savings of using Dijkstra's algorithm are negligible and probably not worth the effort and overhead required. However, when the number of vertices increases the performance of Floyd's algorithm drops quickly. Therefore, the use of Dijkstra's algorithm can provide a solution when performance is a factor. On the other hand, if you will need the shortest path between several vertices on the same graph you may want to consider Dijkstra's algorithm. In the test case, running the algorithm for more than ¼ of the vertices decreased performance below that of running Floyd's algorithm.

Johnson's Algorithm

While Floyd-Warshall is efficient for dense graphs, if the graph is sparse then an alternative all pairs shortest path strategy known as Johnson's algorithm can be used. This algorithm basically uses Bellman-Ford to detect any negative weight cycles and then employs the technique of reweighting the edges to allow Dijkstra's algorithm to find the shortest paths. This algorithm can be made to run in O(V2 lg V + VE). Bellman-Ford is another example of a single-source shortest-path algorithm, like Dijkstra. Bellman-Ford and Floyd-Warshall are similar—for example, they're both dynamic programming algorithms—but Floyd-Warshall is not the same algorithm as "for each node v, run Bellman-Ford with v as the source node". In particular, Floyd-

Warshall runs in O(V3)O(V3) time, while repeated-Bellman-Ford runs in O(V2E)O(V2E) time (O(VE)O(VE) time for each source vertex).

Terms

Graph Centrality

In graph theory and network analysis, indicators of centrality identify the most important vertices within a graph. Applications include identifying the most influential person(s) in a social network, key infrastructure nodes in the Internet or urban networks, and super-spreaders of disease. Centrality concepts were first developed in social network analysis, and many of the terms used to measure centrality reflect their sociological origin.[1] They should not be confused with node influence metrics, which seek to quantify the influence of every node in the network.

Degree Centrality

Historically first and conceptually simplest is degree centrality, which is defined as the number of links incident upon a node (i.e., the number of ties that a node has). The degree can be interpreted in terms of the immediate risk of a node for catching whatever is flowing through the network (such as a virus, or some information). In the case of a directed network (where ties have direction), we usually define two separate measures of degree centrality, namely indegree and outdegree. Accordingly, indegree is a count of the number of ties directed to the node and outdegree is the number of ties that the node directs to others. When ties are associated to some positive aspects such as friendship or collaboration, indegree is often interpreted as a form of popularity, and outdegree as gregariousness.

Closeness Centrality

In a connected graph, the normalized closeness centrality (or closeness) of a node is the average length of the shortest path between the node and all other nodes in the graph. Thus the more central a node is, the closer it is to all other nodes.

Closeness was defined by Bavelas (1950) as the reciprocal of the farness . However, when speaking of closeness centrality, people usually refer to its normalized form. This adjustment allows comparisons between nodes of graphs of different sizes. Taking distances from or to all other nodes is irrelevant in undirected graphs, whereas it can produce totally different results in directed graphs (e.g. a website can have a high closeness centrality from outgoing link, but low closeness centrality from incoming links).

Harmonic Centrality

In a (not necessarily connected) graph, the harmonic centrality reverses the sum and reciprocal operations in the definition of closeness centrality.

Betweeness Centrality

Betweenness is a centrality measure of a vertex within a graph (there is also edge betweenness, which is not discussed here). Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between two other nodes. It was introduced as a measure for quantifying the control of a human on the communication between other humans in a social network by Linton Freeman[17] In his conception, vertices that have a high probability to occur on a randomly chosen shortest path between two randomly chosen vertices have a high betweenness.

Pseudo - Code

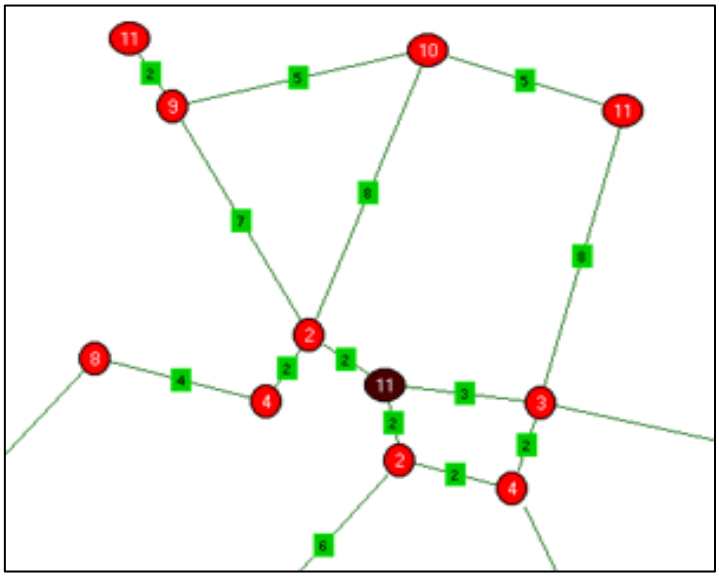
I have used floyd-warshall algorithm to find the all pair shortest path and then I calculate the ones which have the lowest value for the farthest vertex. Here is a code snippet

```
# N - number of vertices
# A(i,j) - length of the shortest path from vertex i to j
Compute all elements of A (by the help of Floyd-Warshall algorithm)
# dist(i) - length of shortest path from i to its farthest vertex

For i = 1 to N
   dist(i) = 0
   For j = 1 to N
      If ( i != j AND A(i,j) > dist(i) ) Then dist(i)=A(i,j)
End For i

center = 1
For i = 1 to N
   If dist(i) < dist(center) Then center = i

Output center
```

Complexity of this algorithm is determined by the part that finds the lengths of shortest paths between every pair of vertices. Thus its complexity is O($\mathbf{N^3}$).

Center of Graph

The center of Central location means that the length of the shortest path to the farthest vertex is minimal. Here vertex colored in dark-red represents the **center** of the graph
- number written on a vertex represents the length of the shortest path from center to that vertex
- number written on graph center vertex represents the length of shortest path to its farthest vertex



Tech Stack

The following technologies were used in the project.

- Java
- Servlets
- JSP
- HTML/CSS
- JavaScript
- Tomcat

The back end code was done in java. For displaying the graph in a UI I used a javascript library dedicated for such tasks. It is **sigmaJS**. It is an open source library. Sigma is a JavaScript library dedicated to graph drawing. It makes easy to publish networks on Web pages, and allows developers to integrate network exploration in rich Web applications. Sigma provides a lot of built-in features, such as Canvas and WebGL renderers or mouse and touch support, to make networks manipulation on Web pages smooth and fast for the user. The default configuration of sigma deals with mouse and touch support, refreshing and rescaling when the container's

size changes, rendering on WebGL if the browser supports it and Canvas else, recentering the graph and adapting the nodes and edges sizes to the screen. Sigma provides a lot of different settings to make it easy to customize how to draw and interact with networks. And you can also directly add your own functions to your scripts to render nodes and edges the exact way you want. Sigma is a rendering engine, and it is up to you to add all the interactivity you want. The public API makes it possible to modify the data, move the camera, refresh the rendering, listen to events.

JavaServer Pages (JSP) is a technology that helps software developers create dynamically generated web pages based on HTML, XML, or other document types. Released in 1999 by Sun Microsystems, JSP is similar to PHP and ASP, but it uses the Java programming language.

To deploy and run JavaServer Pages, a compatible web server with a servlet container, such as Apache Tomcat or Jetty, is required. Architecturally, JSP may be viewed as a high-level abstraction of Java servlets. JSPs are translated into servlets at runtime, therefore JSP is a Servlet; each JSP servlet is cached and re-used until the original JSP is modified. JSP can be used independently or as the view component of a server-side model–view–controller design, normally with JavaBeans as the model and Java servlets (or a framework such as Apache Struts) as the controller. This is a type of Model 2 architecture.

JSP allows Java code and certain pre-defined actions to be interleaved with static web markup content, such as HTML, with the resulting page being compiled and executed on the server to deliver a document. The compiled pages, as well as any dependent Java libraries, contain Java bytecode rather than machine code. Like any other Java program, they must be executed within a Java virtual machine (JVM) that interacts with the server's host operating system to provide an abstract, platform-neutral environment. JSPs are usually used to deliver HTML and XML documents, but through the use of OutputStream, they can deliver other types of data as well.

The Web container creates JSP implicit objects like request, response, session, application, config, page, pageContext, out and exception. JSP Engine creates these objects during translation phase.

Tomcat 7: The Apache Tomcat® software is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. The Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket specifications are developed under the Java Community Process. For this Project I used tomcat7.

Servlets: Java servlet is a Java program that extends the capabilities of a server. Although servlets can respond to any types of requests, they most commonly implement applications hosted on Web

servers.[1] Such Web servlets are the Java counterpart to other dynamic Web content technologies such as PHP and ASP.NET. A Java servlet processes or stores a Java class in Java EE that conforms to the Java Servlet API,[2] a standard for implementing Java classes that respond to requests. Servlets could in principle communicate over any client–server protocol, but they are most often used with the HTTP protocol. Thus "servlet" is often used as shorthand for "HTTP servlet".[3] Thus, a software developer may use a servlet to add dynamic content to a web server using the Java platform. The generated content is commonly HTML, but may be other data such as XML. Servlets can maintain state in session variables across many server transactions by using HTTP cookies, or URL rewriting.

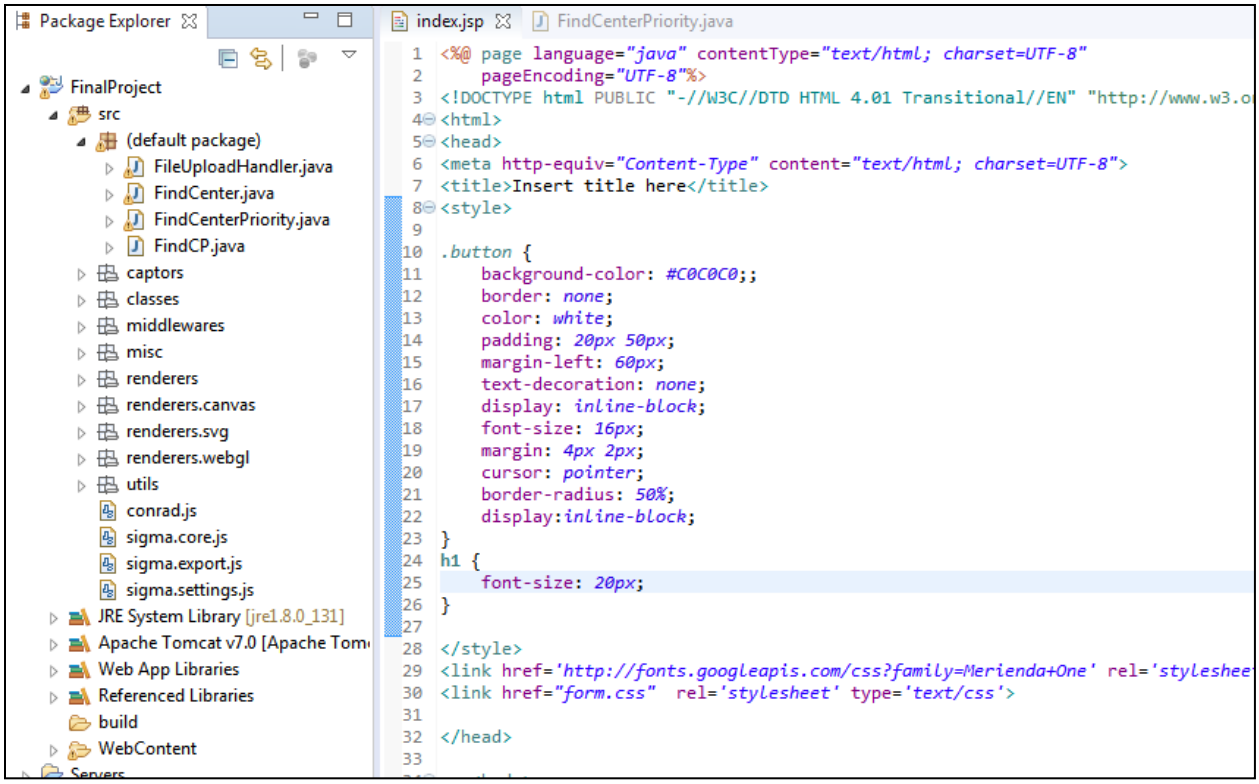
To deploy and run a servlet, a web container must be used. A web container (also known as a servlet container) is essentially the component of a web server that interacts with the servlets. The web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights.

A Servlet is an object that receives a request and generates a response based on that request. The basic Servlet package defines Java objects to represent servlet requests and responses, as well as objects to reflect the servlet's configuration parameters and execution environment. The package javax.servlet.http defines HTTP-specific subclasses of the generic servlet elements, including session management objects that track multiple requests and responses between the web server and a client. Servlets may be packaged in a WAR file as a web application.

Web Application

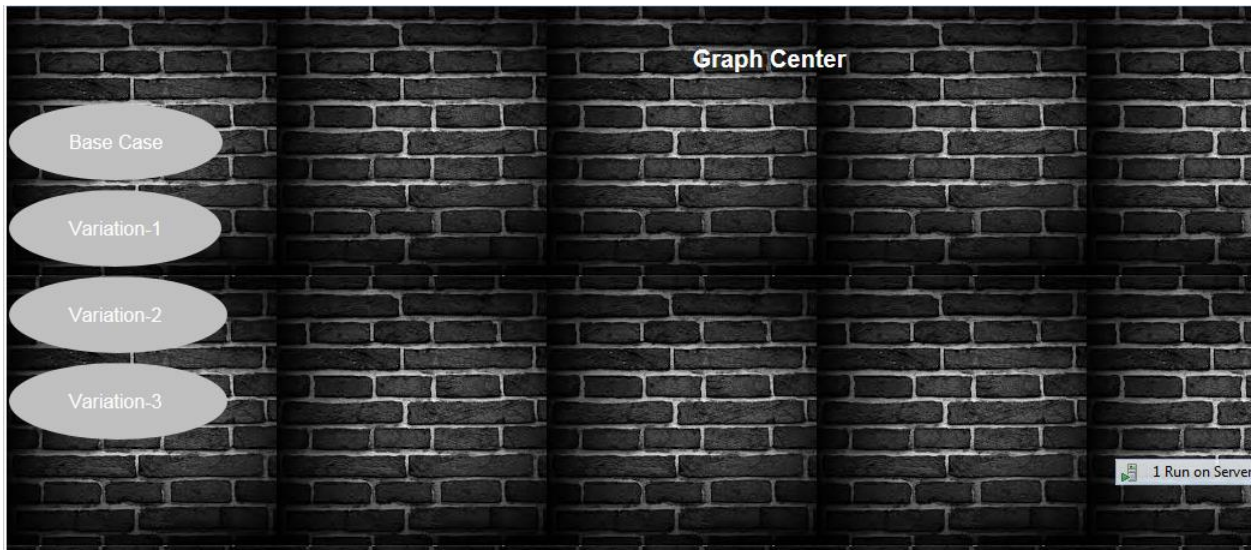
The web app was designed in eclipse EE. As mentioned earlier it was done using JSP/Servet technology along with a major part in SigmaJS library from Javascript. The web application's entry point is index.jsp. It contains links to other pages. It has 4 buttons which redirect the client to other feature pages of the app. The base case generates a directed graph with positive edges and calculates the center. Other three pages are mentioned as variations. The back end is handled by servlets. There is one servlet to parse csv and generate the graph. The other servlet is to calculate the center. There are two additional servlets for handling the variations.

Here is the project structure.



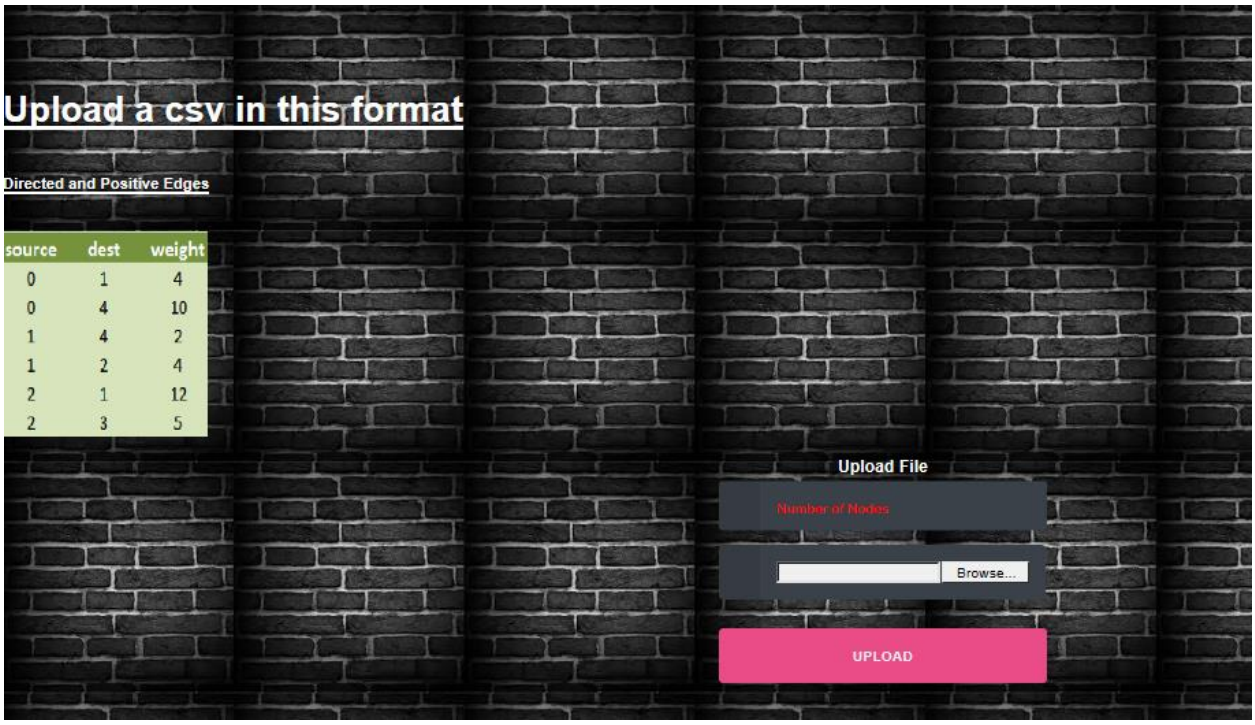
- index page

  The index page looks something like this. The oval buttons are links to other pages named clearly on each buttons thus maintaining user friendly design style.



- **Base Case**:

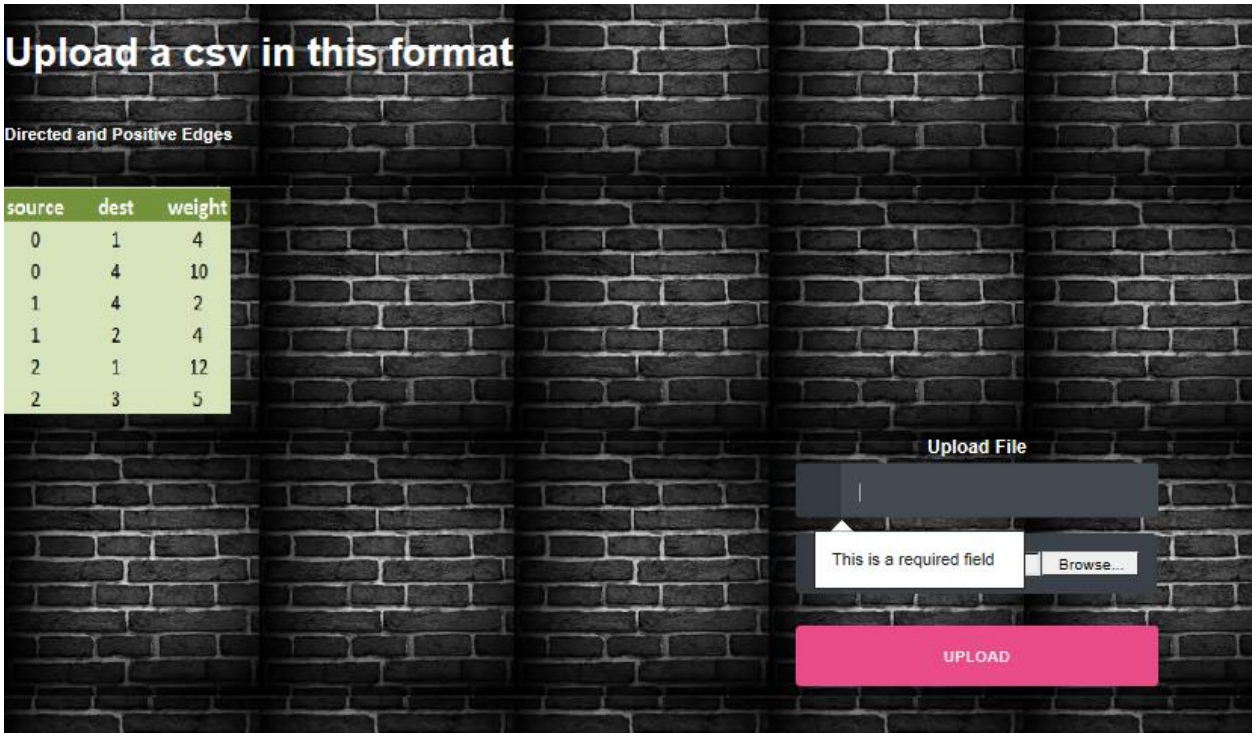
  The base case is the main idea of this project. Central nodes in a directed positive edge weighted graph is calculated. This page prompts the user to upload a csv. Clearly stating the format in a picture so that there is no errors. The number of vertices is also taken as input from the user.



After all inputs are received the user is redirected to the final page called as display.jsp where the user can see the final result. Both the graph and the center nodes.

Error handling:

If the user does not provide any input that state is handled in the front end itself. The user is shown a message to not leave any field empty. Other errors like giving string instead of integer in csv is handled at the back end . Again the user is shown a message to take corresponding action to rectify the error that might be caused because of invalid input.



Variants and Constraints

In order to make it all the more interesting the project also includes one or two variants of the algorithm so that other problems of the same or similar classes and complexity can be solved. The use cases of the variation should also be mentioned along with variant

- **Variation -1**

  The first variation is to run the algorithm on an undirected graph instead of directed graph. Now the city plan for setting up a new hospital is obviously an image of a directed graph because a city has directions from one point to another and maybe different directions on the way back. So it makes sense to put a constraint on the direction and consider it as a separate application. For instance an undirected graph like a social network is a notable use case for this scenario. The center of the social network shows the popularity of a group or individual in the society. This can also be used by ad agencies to see which people or groups are being followed by more people. The can estimate the trending people at a particular point in time. This group can be the next face of their advertisement.

Degree centrality measures might be criticized because they only take into account the immediate ties that an actor has, or the ties of the actor's neighbors, rather than indirect ties to all others. One actor might be tied to a large number of others, but those others might be rather disconnected from the network as a whole. In a case like this, the actor could be quite central, but only in a local neighborhood. Closeness centrality approaches emphasize the distance of an actor to all others in the network by focusing on the distance from each actor to all others. Depending on how one wants to think of what it means to be "close" to others, a number of slightly different measures can be defined.

Here is how it looks in the applications. Note it also puts a constraint on the edges not being negative. Since negative edges don't make sense in a social network.

- **Variation -2**

  The second variation of the project comes from another constraint , this time on the edge weights. This variant allows negative edge weights as well. It should be noted that negative weights were not allowed in the base case the reason being having negative edges denoting distance between nodes in a city is pointless. But a use case for negative edges could be social monetary network like a venmo app where friends exchange cash between one another. Now when we talk about money we have both positive and

negative metrics. Now how finding the center of the network would be helpful, is that you can gauge who among your friends does most transactions with you and will be a potential candidate for asking for a loan.

Venmo is a mobile payment service owned by PayPal. It allows users to transfer money (US only) to one another using a mobile phone app or web interface. enmo was not created for transferring money to people you do not know, but rather it was designed to transfer money between peers who trust each other. Venmo does not have either buyer or seller protection.

Among young people, especially Millennials, "Venmo" is often used as a verb with the meaning, "to send money via the Venmo platform.

- **Variation-3**

The third variation of the project introduces a new parameter. In this variant not only do the edges have a weight but the nodes also have a priority. In the use case where a new hospital is to be set up in a city , suppose two nodes or points in the neighborhood have both got the central location. This is quite possible. Now let's say the points A and B are both the centers of a graph. So as per the base case the hospital should be setup i n either of the two. But consider a hypothetical situation where point A is a university and point B is just a parking lot. So in this case we should very clearly select point A as our ideal location. This means we gave higher priority to node A because of its location.

This whole situation can be mimicked by assigning a priority to each of the nodes and if two or more nodes end up as the centers of the graph, the tie is broken by the priority values.

Here is a sample input for this variant. The priority is added with the node in the adjacency matrix itself and extra precaution is taken while parsing.

Src	Dest	Weight
0	1	3
0(20)	2(60)	8
1(40)	3(30)	1
1	4(35)	7
2	1	4
3	0	2

Source Code

- index

```
    font-size: 16px;
    margin: 4px 2px;
    cursor: pointer;
    border-radius: 50%;
    display:inline-block;
}
h1 {
    font-size: 20px;
}

</style>
<link href='http://fonts.googleapis.com/css?family=Merienda+One' rel='stylesheet' type='text/css'>
<link href="form.css"  rel='stylesheet' type='text/css'>

</head>

   <body>
   <h1 align=center>Graph Center</h1>
     <form action="directed.jsp" >
     <input type="submit" class="button" value="Base Case" >
     </form>
     <form action="Undirected.jsp" >
     <input type="submit" class="button" value="Variation-1" >
     </form>
     <form action="negative.jsp" >
     <input type="submit" class="button" value="Variation-2 " >
     </form>
     <form action="priority.jsp" >
     <input type="submit" class="button" value="Variation-3 " >
     </form>

    </body>

</html>
```

- Find Center

```
List<Integer> getPaths(int[][] graph) {
    int V = graph.length;

    int dist[][] = new int[V][V];

    // Base case
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }

    for (int k = 0; k < V; k++) {

        for (int i = 0; i < V; i++) {

            for (int j = 0; j < V; j++) {

                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }

        }
    }
    return getCenters(dist);
}

List<Integer> getCenters(int dist[][]) {
    int V = dist.length;

    int[] len = new int[V];
    for (int i = 0; i < V; i++) {
        len[i] = 0;
        for (int j = 0; j < V; j++) {
            if (i != j && dist[i][j] > len[i])
                len[i] = dist[i][j];
        }
```

- Parse CSV

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    HttpSession session = request.getSession();
    Part filePart = request.getPart("file");
    String stringLine = null;

    HashSet<String> set = new HashSet<String>();
    String[] arr = new String[3];

    int edge = 0;

    ObjectNode obNode = null;
    int size = 0;
    try {
        size = Integer.parseInt(request.getParameter("maxNodes"));

    } catch (NumberFormatException ne) {

        response.getWriter().print("Value must be an integer");
    } catch (Exception e) {
        response.getWriter().print(e.getMessage());
        return;
    }

    int[][] graph = new int[size + 1][size + 1];

    preProcess(graph);

    BufferedReader bufferedReader;
    CsvMapper csvMapper = new CsvMapper();

    ArrayNode arrayNode = csvMapper.createArrayNode();
    ArrayNode sourceNode = csvMapper.createArrayNode();
```

- Sample CSS

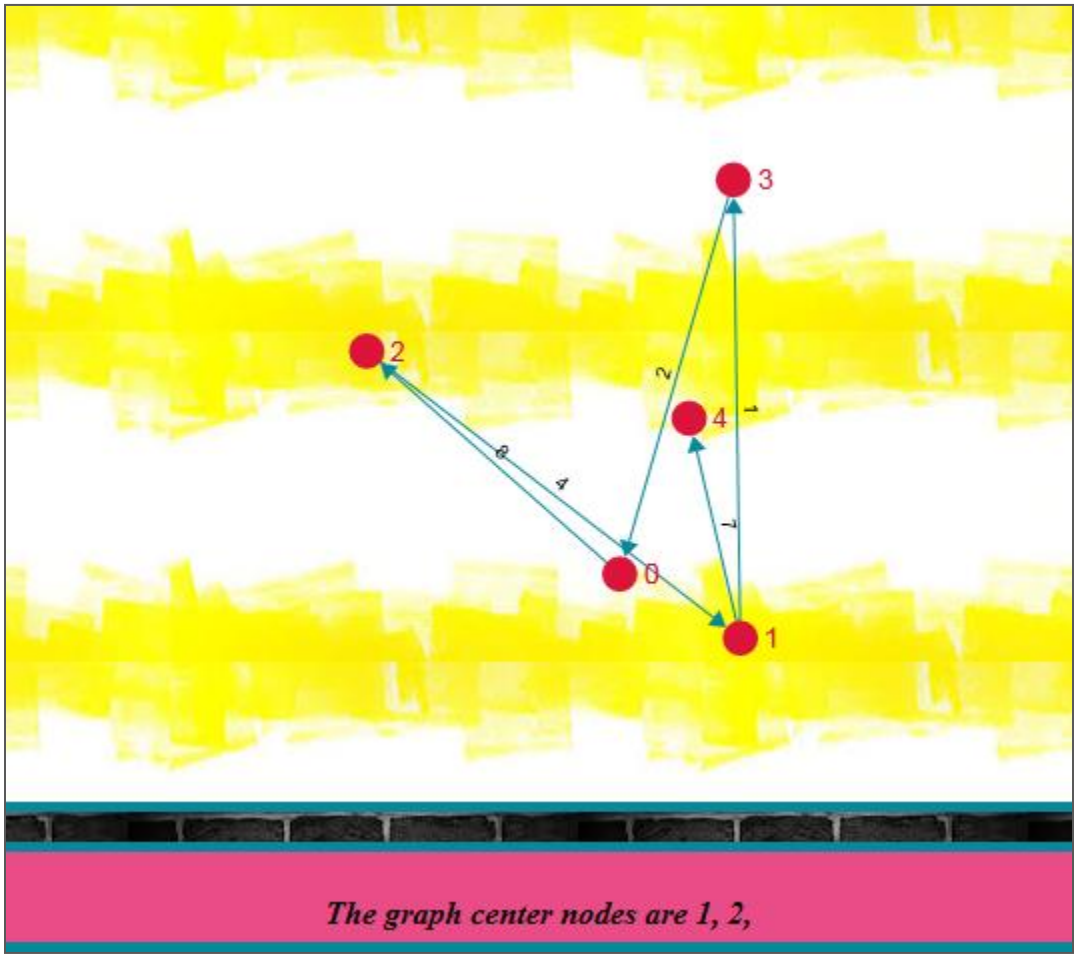
```css
html {
  box-sizing: border-box;
  font-size: 100%;
  height: 100%;
}

body {
  background-color: #2c3338;
  color: #606468;
  font-family: 'Open Sans', sans-serif;
  font-size: 14px;
  font-size: 0.875rem;
  font-weight: 400;
  height: 100%;
  line-height: 1.5;
  margin: 0;
  min-height: 100vh;
  background-image: url("background.png");
}

/* modules/anchor.css */

a {
  color: white;
  font-size:30px;
  align:left;

  outline: 0;
    width:200%;
}
```

Results

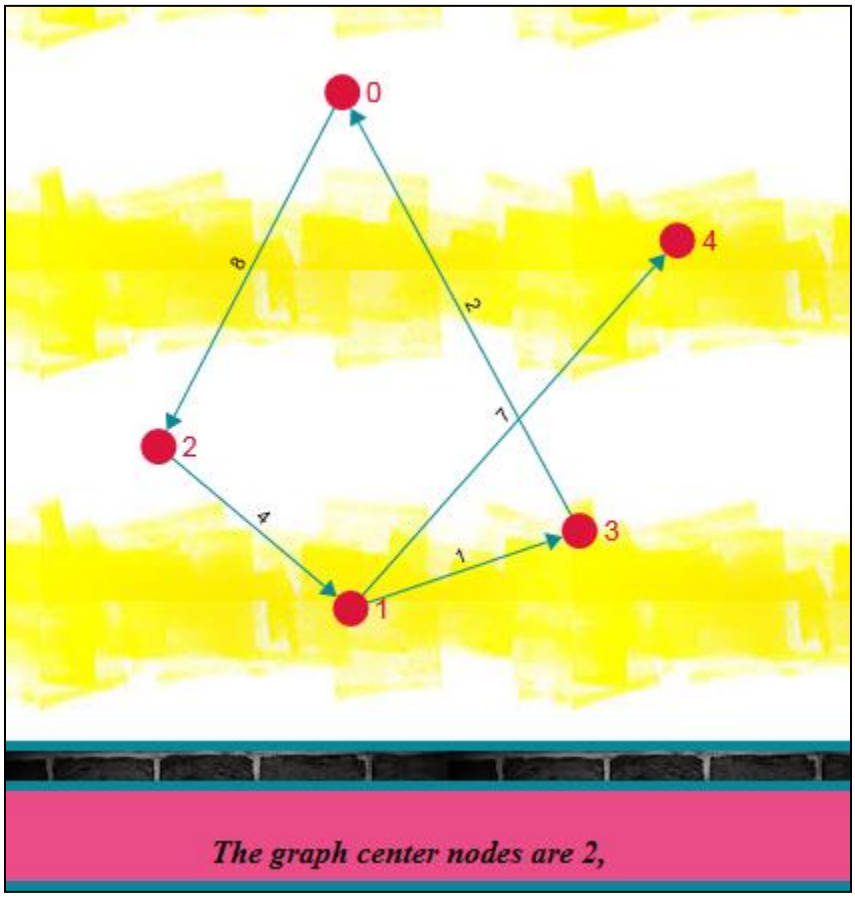
The results obtained from the solver are displayed in the jsp right below the graph. At each step the result was forwarded to next step and finally we get the name of the node. But it starts with parsing a csv and converting it to and adjacency matrix and json at the same time. The matrix is fed to the backend center calculator and json is fed to the front end to display it using javascript libraries. Following results were obtained.

Here is a sample input for base case:

Src	Dest	Weight
0	1	3
0	2	8
1	3	1
1	4	7
2	1	4
3	0	2



The base case results yields the above graph. In the below container the answer is clearly mentioned. In this case the nodes 1 and 2 have central location as per their weights. Similarly the same graph given with some priority on all nodes yields a different answer. Although it will pick any one of the above as the center based on who has more or higher priority.



In this case for the same graph the variation-3 show s 2nd node as the answer that is just because node 2 has higher priority (60) as compared to node1(40) as we can see in the sample input picture above.

Inferences

This whole project provided me with a better understanding of shortest path algorithms and I now have a better perspective of which algorithm to choose for a particular application. Also the different other problems facing human world could be solved by using these algorithms under the hood and customizing them for application specific tasks. For the base case it is clearly understood that the graph would be a dense , directed , no negative weights.

The simplest way to find the center of the graph is to find the all-pairs shortest paths and then picking the vertex where the maximum distance is the smallest. One can find the all pairs shortest path in time O(V3V3) using the Floyd–Warshall algorithm.

It is interesting to note that, one can show a sub-cubic equivalence between finding the center of the graph and the All-Pairs shortest path problem[AGV '15]. This means that to do better than O(V3V3) for the center of the graph, one has to do better than O(V3V3) for the All-Pairs Shortest Path problem.

Challenges

There were quite a few challenges faced in this project. But due to constant efforts I was able to finish the deliverables and meet the deadlines. But it was a very demanding and research based project which will go a long way. Some of the challenges I faced were

- **Front End**: This was one of the major obstacles since developing a UI for a graph is not straight forward. A lot of online resources helped me get some hands on experience with the technology and finally use it for my own piece. Sigma JS is learning curve and to get the best possible design you need to go through a lot of plugins and source code.
- **Variants and Use Cases**: This was a bit tricky too. To think fo what could be a suitable variant of the problem statement and then come up with a use case for the same required a lot of a reading and knowing about other problems that overlap with the graph center problem. One reason for this was that we had to make sure the variant does not overlap with other graph algorithm problems and it had to be somewhat original.

Future Work

Although the set goals were achieved but there is still room for more thought and action. Graph theory is such a vast domain that you never know when a problem may arise that could be solved with graph center approach. So the work ahead would be to optimize the algorithm and look out for more algorithms especially the custom ones instead of the classic ones like floyd-warshall, and use them to drive optimal solutions to this problem. Some amount of work can be done on the UI too in order to make it look better and more user friendly.

References

- http://www.faculty.ucr.edu/~hanneman/nettext/C10_Centrality.html
- https://algs4.cs.princeton.edu/44sp/
- http://www.graph-magics.com/articles/center.php
- https://www.slideshare.net/malinga_perera/floyd-warshallalgorithm
- http://codeforces.com/blog/entry/17974
- www.wikipedia.com
- http://sigmajs.org/
- https://github.com/jacomyal/sigma.js/tree/master/plugins/sigma.renderers.edgeLabels
- https://en.wikipedia.org/wiki/Centrality