May1618

Graphalyzer

Graph Visualization and Analysis Tool

Final Document

Team:

- Andrew Bowler - Webmaster
- Alberto Gomez-Estrada - Communications
- Michael Sgroi - Key Concept Holder
- Taylor Welter - Team Lead
- Richard White - Key Concept Holder

Advisor: Simanta Mitra

Client: Workiva

Table of Contents

1. Introduction

1.1 Problem Statement

Companies are constantly dealing with Big Data. Big Data is very difficult to understand or interpret, and analysis of the data is far from trivial. Sometimes data is often structured such that pieces of the data have relationships with one another, further increasing the complexity. An example would be a social media company that wants to analyze how interconnected a group of friends are, where that group could be hundreds of thousands of different people who might be friends with one another. How can we analyze all of the connections buried within this data?

1.2 Project Goals

Graphalyzer's solution is to provide users with an interactive visual graph of the data. The user can explore the graph and perform analysis on the data through searching, filtering, and drawing of subgraphs. Graphs make this type of data easier to understand because they highlight the structural behavior of the data, and make it easy to find members of high importance, such as interconnectivity or having specific properties.

Definitions of Common Terms and Abbreviations

Term	**Definition**
Big Data	Extremely large or complex datasets that render traditional data processing methods inadequate
Edge	Foundational unit of a graph; used to associate a two nodes
Graph	An abstract data type comprised of a set of nodes and edges, used to analyze relationships within sets of data
HTTP	Hypertext Transfer Protocol; foundation for communication within the World Wide Web
JSON	JavaScript Object Notation
Neo4j	An open-source graph database system written in Java
NGINX	A reverse proxy server for Hypertext Transfer Protocol (HTTP)
Node	Foundational unit of a graph; used to

	represent irreducible set of data
REST	Short for **Re**presentational **S**tate **T**ransfer, a stateless communication protocol that allows efficient client-server interaction
TCP	Transmission Control Protocol, a protocol for sending data over a network
TravisCI	Continuous Integration Server for testing
URL	**U**niform **R**esource **L**ocator, format for web addresses
WebSocket	Is a protocol for bidirectional communication over Transmission Control Protocol (TCP)

Table 1: Common Terms and Abbreviations

2. Deliverables

The deliverables are limited to one Linux virtual machine (VM) containing:

1. The Graphalyzer web client
2. A Neo4j database
3. A REST service for adding data to the database
4. A WebSocket server for communication between client and server
5. An NGINX web server
6. An open source GitHub repository containing all of the source code for the project

3. Project Design

3.1. User Requirements

The web application must be able to facilitate the analysis of large volumes of data, as well as extracting fragments of data that are most relevant to a client, according to their needs. The web application must be easily accessible to the client, and be fast and efficient. Lastly, because of the volumes of data and the potential for critical data to be analyzed using the application, the application must be thoroughly tested and free of fatal error or complications.

3.2. System Requirements

The web application must be deployed on a Linux virtual machine that can be remotely accessed by clients and other users. The application must also process and index large volumes of data efficiently and accurately. The technologies used shall preferably be compatible with technology that is currently in use at Workiva.

3.3. Backend Functional Requirements

3.3.1. The product shall allow for the upload of arbitrary graph files for analysis via REST

3.3.2. The product shall index incoming information associated with nodes and edges

3.3.3. The product shall allow for the searching of nodes by name

3.3.4. The product shall process the graph to determine:

- Depth
- Breadth
- Interconnectedness
- Impact

3.4. Frontend Functional Requirements

3.4.1. The product shall be able to visualize the graph such that:

- The graph's visualization shall clearly distinguish different nodes and their connections
- The visualization shall clearly distinguish properties of the graph through color and shapes, such as impact and interconnectedness

3.4.2. The product shall allow for the exploration of the graph via click or touch

3.4.3. The product shall display the ability to search the graph in the user interface

3.4.4. The product shall display to the user when it is processing a user's search query

3.5. Backend Nonfunctional Requirements

3.5.1. The product shall be able to handle a large number of nodes (approx. 50000) without crashing or hanging

3.5.2. The product shall support different file upload protocols (e.g. REST, FTP, etc.)

3.5.3. The product shall maintain original graph data integrity

3.6. Frontend Nonfunctional Requirements

3.6.1. The product shall be able to process, visualize parts of, and analyze graph files.

3.6.2. The product shall not crash or lose connection to the REST service while in use

3.6.3. The product shall not have any memory leaks or any loss of graph data.

3.6.4. The product shall not voluntarily or involuntarily modify the graph data.

3.7. Design Diagram

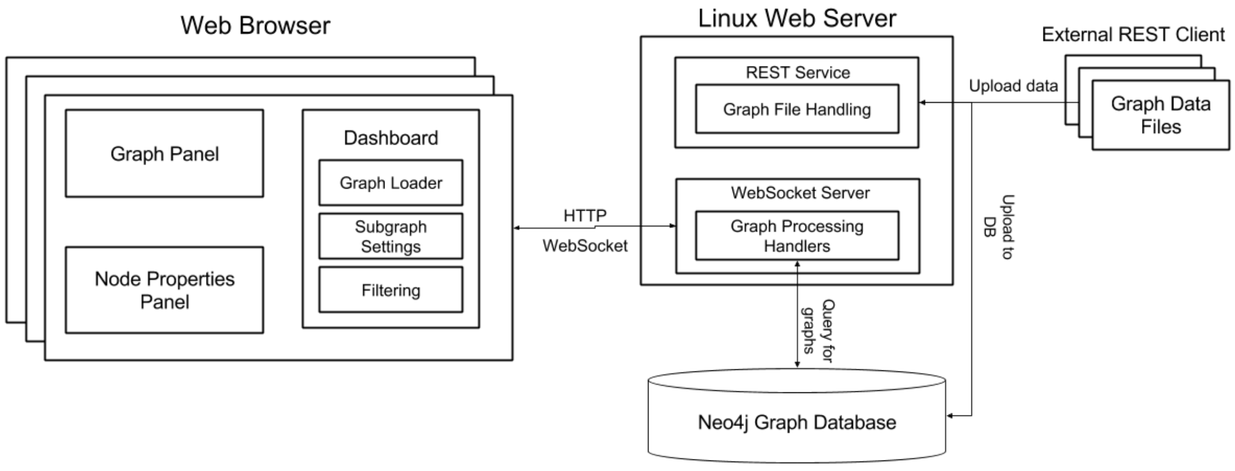


Figure 1: Architecture of Graphalyzer

4. Implementation Details

4.1. Backend

4.1.1. File Upload

The goal of this is to handle CSV file uploads. There are two distinct parts of this service, REST upload and neo4j upload. The goal off this design was to enable uploading by FTP or REST. This was accomplished by dumping REST files to a designated server folder for upload to neo4j.

4.1.1.1. REST Upload

This is responsible for capturing file upload via a REST url and dumping the contents to a file. This file is located inside a temporary folder on the server. After this file is written to the server it is then handled by the neo4j upload service.

4.1.1.2. Neo4j Upload

This is responsible for watching the temporary upload folder for new files. When a new file is found it is uploaded to neo4j. It distinguishes between property and edge files by the name of the uploaded file, e.g. if the file name contains “edge” or “property.” This part of the service also moves the file from the temporary folder to a backup folder on the server in case it is needed for restoration.

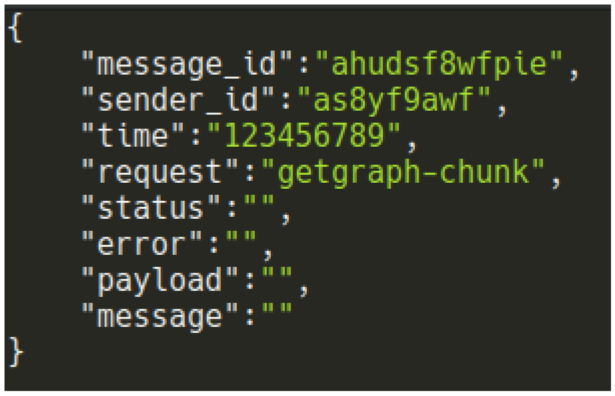
4.1.2. Websocket Server

The communication between the frontend and backend was implemented through a websocket server. This server was written in python3 using Autobahn as the websocket library and py2neo for querying neo4j. The JSON format was used as the message format.

4.1.2.1. Architecture

The websocket server has a basic architecture. There is a main event loop that waits for a client's request. Once a request is received a factor class parses it and determines the type of request. Each request type is represented as a class that knows how to handle that request type.

4.1.2.1.1 JSON Format



```
{
    "message_id":"ahudsf8wfpie",
    "sender_id":"as8yf9awf",
    "time":"123456789",
    "request":"getgraph-chunk",
    "status":"",
    "error":"",
    "payload":"",
    "message":""
}
```

Figure 2: JSON message format

Each client request and server response represented as a JSON string. is The JSON format is as follows.

`message_id` - Is an identifier for each message generated by either side.
`sender_id` - Is an identifier for the client given in the newid request.
`time` - Unix time when the message was sent.
`request` - The type of request. A list of request will be provided below.

`status` - The server will set this to success for failure depending on whether the request succeeded.
`error` - If status if failure this will be the error message.
`payload` - Information to be sent to or from the server, dependent on request type.
`message` - Currently unused. It is there for future use.

Here is a list of the request types with their corresponding request and response payloads:

`getgraph` - Request payload is a graph ID. Response payload is a JSON object with a nodes and edges fields. These fields are JSON arrays with JSON objects with the following formats.

The nodes have an id property, and any amount of additional properties that are properties of the node. An example is:

```
{
 “id”:”someid”,
 “someprop”:”someval”,
 “Anotherprop”:”anotherval”
}
```

Each edges object has an id, to, and from field. The id is the edges id, the from is the id of a node that is the edge’s parent and to is the id of a node that is the edges child. There also can be any number of additional properties that represent the edges properties.

An example is:

```
 {
  “id”:”someid”,
  “to”:”somenodeid”,
  “from”:”anothernodeid”,
  “Someprop”:”someval”
}
```

`listgraphs` - No request payloads. Response payload is JSON array of graph IDs.

`getgraphchunk` - Same as `getgraph` except results will be sent in chunks and the response payload will have an additional `numberofchunks` field that is the number of chunks that will be sent, and `currentchunk` which is the currents chunks number.

`getsubgraph` - Request payload is a JSON object with a graphid,depth,node fields where graph is is the graph you want and the node is the id of the node you want as the root and depth as the depth you want.The response payload is the same as `getgraphchunk`.

`newid` - No request payloads. Response payload is a string that the client can use for its sender_id.

4.1.3. Neo4j

The internal structure of neo4j is essentially schema-less. It holds a list of nodes and edges. Different graphs are distinguished by a graphite property in the nodes and edges. This simple setup allows us to use neo4j without any setup. It also allows us to hold any type of graph.

4.2. Frontend

Here is a screenshot of our entire application.

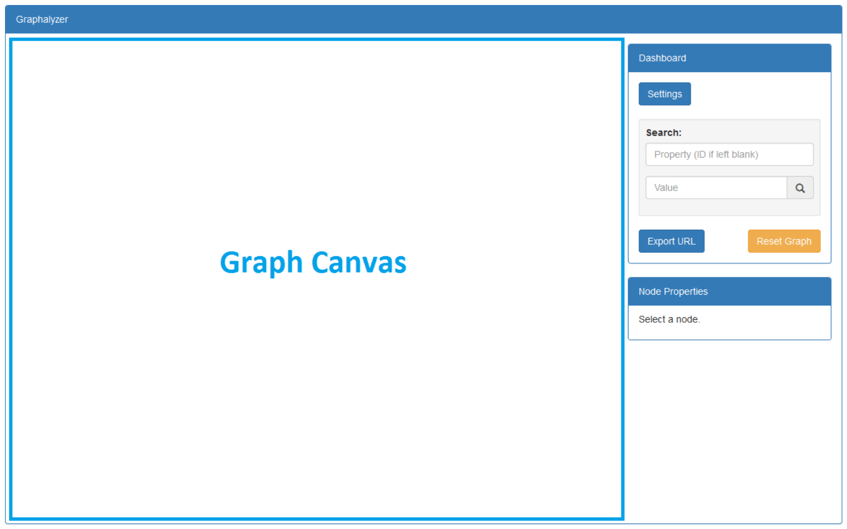


Figure 3: User Interface Screenshot

4.2.1. Graph Canvas

This is where the graph is drawn after receiving it from the database. Users can interact with the graph by clicking on nodes.

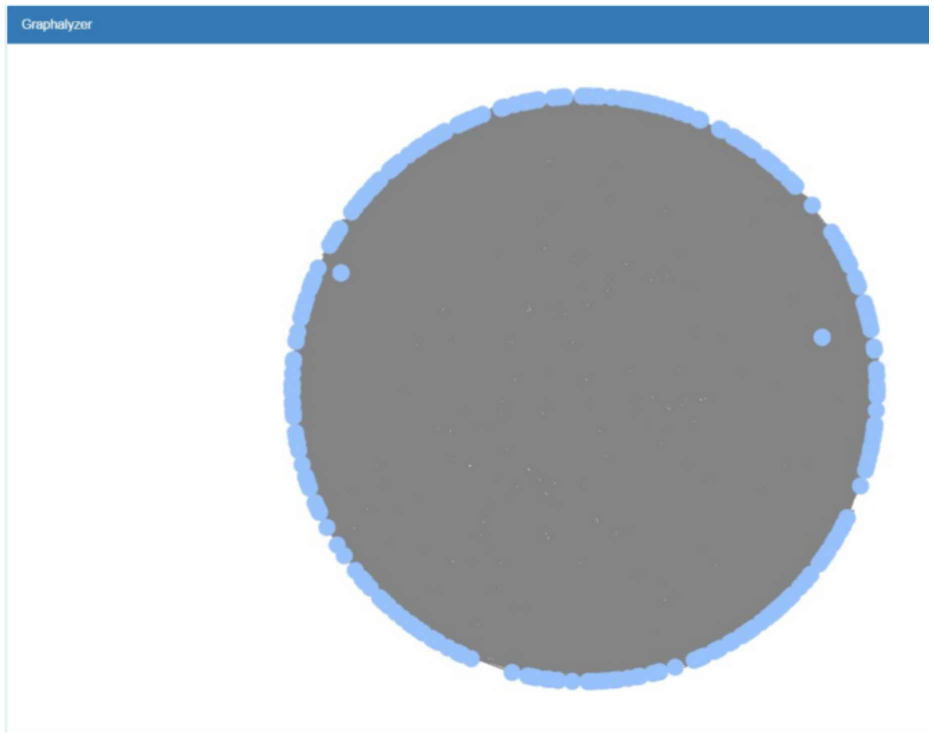


Figure 4: Graph Canvas Screenshot

4.2.2. Dashboard

This is the main UI control for the application. Here users can set parameters for drawing and filtering graphs or subgraphs, and search for nodes. In addition, they can export custom URLs and reset the graph canvas to a blank state.

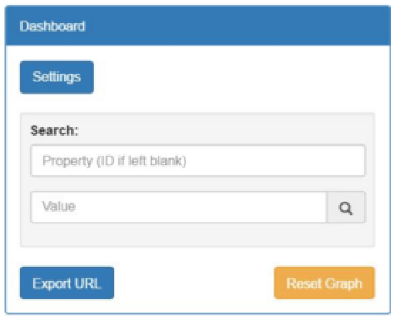


Figure 5: Dashboard Screenshot

4.2.2.1. Settings Modal

This displays a list of settings for users to customize. They can select a graph to load from the database (this list is refreshed in real time), set filtering parameters, and specify options for drawing a subgraph of the selected graph.

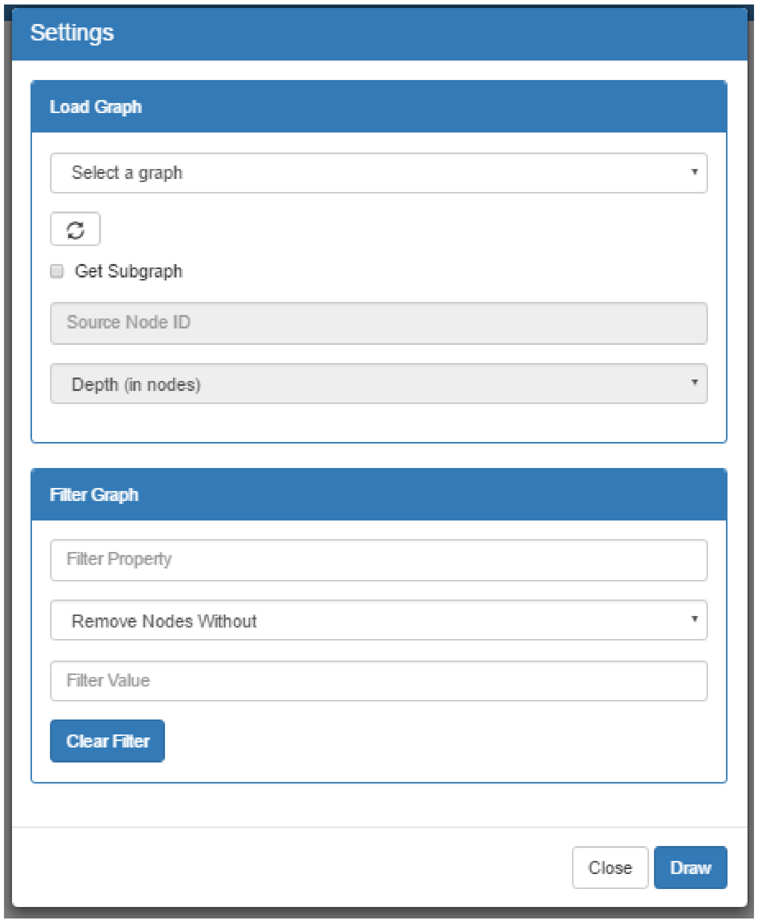


Figure 6: Settings Modal Screenshot

4.2.2.2. Graph Loader
This displays a list of graphs by their names currently stored in the neo4j database. By selecting one the user has defined the graph they want to draw.

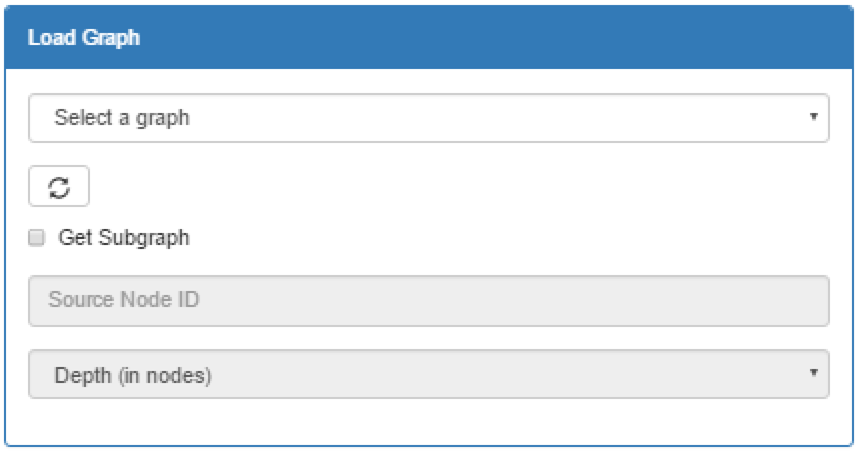


Figure 7: Load Graph Panel Screenshot

4.2.2.3. Subgraph Settings
This displays options for drawing a subgraph of the above selected graph. Users specify the ID of the source node they wish to start from, then a depth ranging from 1-5. This depth translates to the degrees of connectivity from the source node.

4.2.2.4. Filter Panel
This displays options for users to filter their graph. When filtering, all nodes are checked against a user specified filter test. If the node passes the filter test, it is highlighted in the graph canvas. Otherwise (if it fails), it is greyed out but still visible in the graph canvas.

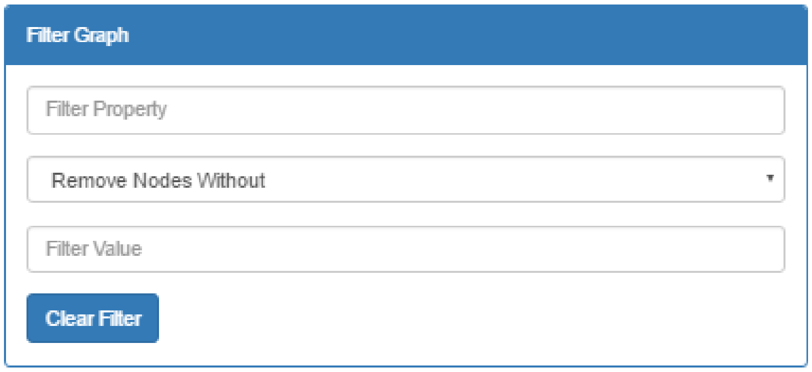


Figure 8: Filter Panel Screenshot

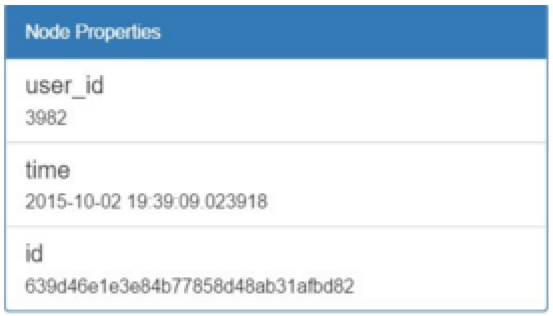
Users can set filtering settings as follows:

Numerical Filtering	Filter nodes based on a specified property, whose value is less than, greater than, or equal to a certain numerical value
Pattern Matching	Filter nodes based on a specified regular expression. If a property is provided as an additional parameter, that property's value is tested against the regular expression, otherwise all of the node's properties are tested against the regular expression.
Remove Nodes Without	Filter nodes solely based on whether or not they have a specified property.

Table 2: Filtering Methods

4.2.3. Node Properties Panel

This displays a list of properties of a node when it selected (clicked) by the user.



Node Properties

user_id
3982

time
2015-10-02 19:39:09.023918

id
639d46e1e3e84b77858d48ab31afbd82

Figure 9: Node Properties Panel Screenshot

4.2.4. Deep Linking

4.2.4.1. Export URL

This was implemented by taking graph state variables and converting them to JSON, appending them to the URL as parameters, and copying the newly constructed URL to the user's clipboard. This means that users can share their application states with other people to show their own graph analysis.

4.2.4.2. Import URL

This was implemented using a regex parser that reads variables from the URL and sets the corresponding React.js state variables. It the attempts to show the graph associated with these variables if it is the first time loading the page and the variables have been set.

5. Testing

5.1 Test Environment

5.1.1 Testing Tools

After very careful consideration and experimentation with several testing tools, we decided to use Jest, React.js's built-in testing framework, and PyUnit, Python's unit testing framework, for our unit testing. We were limited in scope, regarding what we could test, and due to that, we limited our testing to unit testing. Some of the reasons for this decision are discussed below.

Our tests are designed to be run upon push to our GitHub repository using TravisCI.

5.1.2 Testing Challenges

The first challenge we encountered was in identifying a testing framework that could be easily integrated into React.js and that we could learn to use and adapt for our purposes. We attempted to use Jasmine, Mocha, and encountered some difficulties using Jest. Because we were using React.js as our Javascript framework, as opposed to using plain Javascript, the testing frameworks required more involved and specialized setup than we anticipated. After consulting the documentation for each of the aforementioned frameworks, and reaching out the open-source community for advice and guidance, we decided upon using Jest as our testing framework for the front-end. The back-end Python server was united tested using Python's built-in testing framework, PyUnit, which was thoroughly documented and, since we were using plain Python and not a framework, like Django, was relatively easy to set up.

The second challenge we encountered when implementing our testing plan was one that prevented us from successfully implementing system testing. As a requirement set by the client, our application was deployed on servers that are owned by the client and require IP addresses to be part of a whitelist. Because of this, the instance of our project that is created by TravisCI is unable to contact our Python server or our database. We, therefore, limited our testing to unit testing, with complete system testing as a possible future approach once the barriers are removed.

5.2 Test Plan

5.2.1 Test Objectives

For the purposes of this project, we limited our testing to some of the main use cases, i.e. drawing the graph, resetting the graph panel, getting a graph list, and sending a request to the server. Our intent was to ensure that the application was functional, that all elements rendered correctly, and that all components interacted correctly, without testing non-functional requirements.

We did not follow test-driven development practices, and our tests were derived from the requirements put in place by the client.

5.2.2 Test Cases

5.2.2.1 Front-End:

A. Dashboard renders correctly
B. Settings panel opens correctly when appropriate button is clicked
C. Export URL button copies the correct URL to the clipboard

5.2.2.2 Back-End:

A. Server instantiates correctly
B. Server accepts correctly formatted requests
C. Server rejects incorrectly formatted requests

5.2.3 Test Results

5.2.3.1 Front-End:

A. We were able to confirm that dashboard renders correctly
B. Settings panel opens correctly when appropriate button is clicked
C. Did not test copying to clipboard, but correct URL is generated

5.2.3.2 Back-End:

A. Server instantiation test passed
B. Correctly formatted requests pass
C. Incorrectly formatted requests are rejected

5.3 Future Work

Due to time constraints and technical limitations, there were some features that were not tested. As the project evolved in complexity and scope, it became clear that testing React.js components was rather involved, and was something that we did not have much experience doing. Because of this, we limited the features we attempted to test. In addition, as indicated in the previous sections, we also limited our testing to basic unit testing. For future iterations of this project, the following would remain to be implemented:

- Testing mocked and dynamically rendered React components
- System and integration testing, so that end-to-end functionality can be tested
- Test front-end and back-end functionality simultaneously.

Appendix I. Operations Manual

A1.1. Setup

Before doing anything, clone the Git repository at
https://github.com/rwhite226/Graphalyzer.git
Next, install Node.js and the Node Package Manager (npm)

A1.1.1. Backend Server Setup (Linux only):

To run the python backend you will need to do the following:

1. Have a dedicated Linux server.
2. Install python (v3.4 minimum), neo4j (v2.3 minimum), pip, zip.
3. Start the neo4j database `neo4j start`.
4. Where the project is located go to `graphalyzer-server/` directory.
5. Run the `package.sh` script `sh ./package.sh`.
6. This should create a file in this directory called `ServerApp`.
7. Run `python ServerApp` (depending on the os this may be python3, python3.4, python3.5, etc).

Note port 1618 will need to be open on the server, if another port is desired edit `graphalyzer-server/src/ServerConfig.py` and go to step 4.

A1.1.2. Frontend Installation/Running (Linux/Mac/Windows):

1. Go to `graph_ui/`.
2. `npm install -g browserify gulp` to install Browserify and Gulp globally on your machine (skip this if you have done so before).
3. `npm install` to load all packages from `package.json`.
4. Run `gulp` to compile the JavaScript for development environment, or `gulp production` for minified code.
5. For local development: open up `index.html` locally on your machine.

A1.2. Using Graphalyzer

Our primary use cases are walked through explicitly on our project website with examples, but here we will walk through using everything in the application in a basic order. If any terms of the UI aren't clear, see the screenshots of the UI in the Frontend Implementation Details (**section 4.2**).

A1.2.1. Uploading Files

REST files are to be uploaded via the URL : http://52.3.104.50/upload/

1. There is a sample REST upload html file inside the sample-configs/rest folder in the repository.
2. In order to use the REST html file, simply open the file in your browser.
3. Click “Choose File” and select the CSV file you wish to upload.
4. Click “Upload.”

File uploading to neo4j is accomplished by an automated script that uploads files to neo4j. All that is necessary is to drop the CSV file into the temporary folder set inside the python configuration file.

A1.2.2. Drawing A Graph

1. The graph canvas at the moment is empty. We need to draw a graph. On the Dashboard, go to the Settings Modal.
2. In the Graph Loader, there is a dropdown with a list of graphs currently stored in the server’s neo4j database. Select the graph you wish to draw.
3. If you wish to draw a subgraph of your graph, check the “Is Subgraph” checkbox. Input the text ID of the source node you wish to start your subgraph from. Select a depth of your subgraph, ranging from 1-5. Typically, 1 or 2 are selected for easy analysis.
4. If you wish to perform filtering, in the Filter Graph Panel, set your filter settings. More details of how to filter with Graphalyzer are in **section 4.2.2.4**.

A1.2.3. Interacting with the Graph

After drawing a graph, there will be a brief loading bar (time dependent on the size of the data) before the graph is displayed on the graph canvas.

1. To select nodes from the graph, simply click on them. In the Node Properties Panel, all of the properties and their values of the selected node are listed in a table.
2. To search for nodes in the graph, in the Dashboard, there is a small Search Panel. Input the property you want to search all nodes for. If no property is provided, the id property is searched for by default. Input the value of the property you wish to search for, and click the search icon. The graph canvas will focus in on the node you searched for (provided it exists) and automatically display its properties in the Node Properties Panel.
3. If you wish to reset the graph canvas to a blank state, then click the Reset Graph button in the Dashboard.

A1.2.4. Sharing Your Graph Analysis

Graphalyzer allows users to export their graph visualization and analysis to share with other users of the tool. To do this, simply click on the Export URL button in the Dashboard. This will copy the entire state of your browser's instance of Graphalyzer to your computer's clipboard.

The user you share this URL with must paste this URL in their browser, and upon load, Graphalyzer will read the URL parameters and load the graph to the state you saved, including filtering and subgraph settings.

Appendix II. Previous Designs

A2.1. September-October 2015

Here we designed our very initial form of our UI. This was written using Angular.js as a frontend framework. We scrapped this design in favor of using React.js as a frontend framework because it was easier to use and more compatible with other libraries.

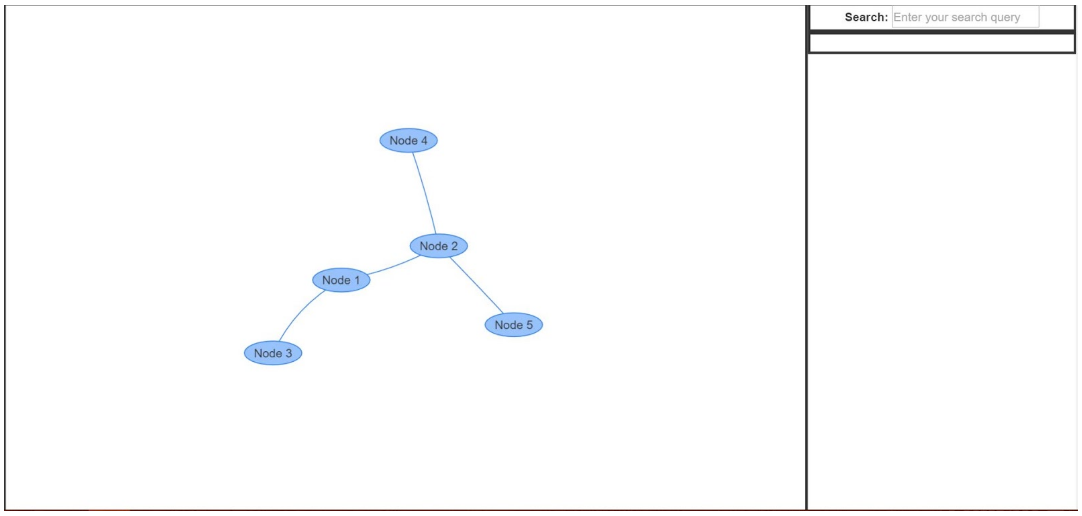
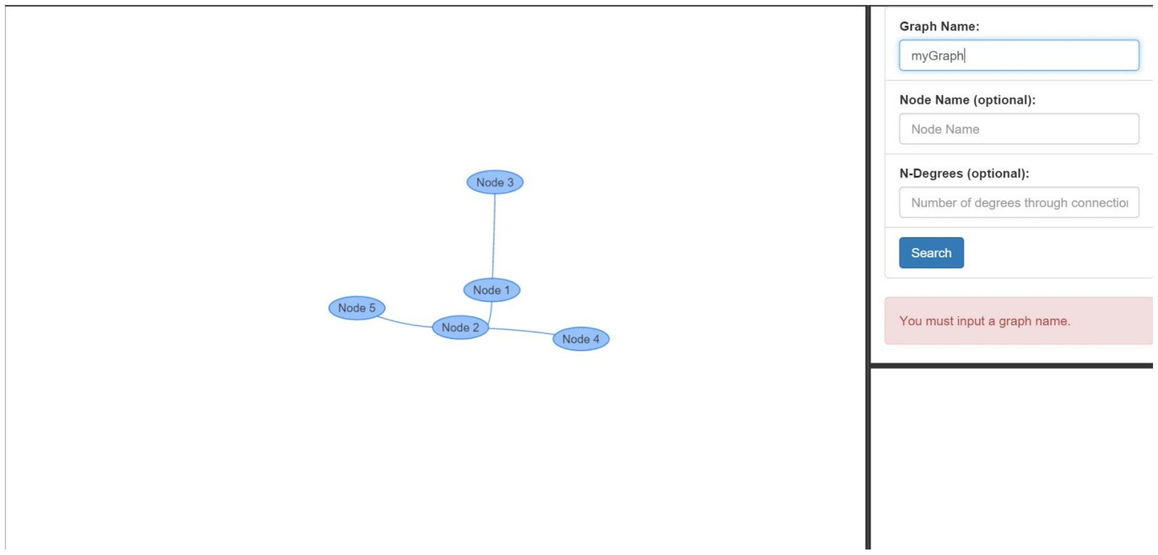


Figure 10: Initial UI design #1



You must input a graph name.

Figure 11: Initial UI design #2

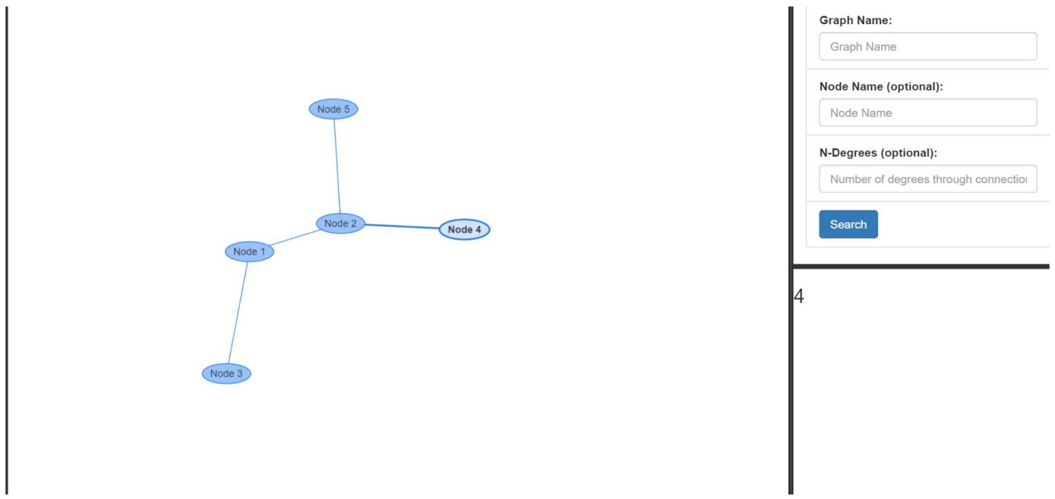
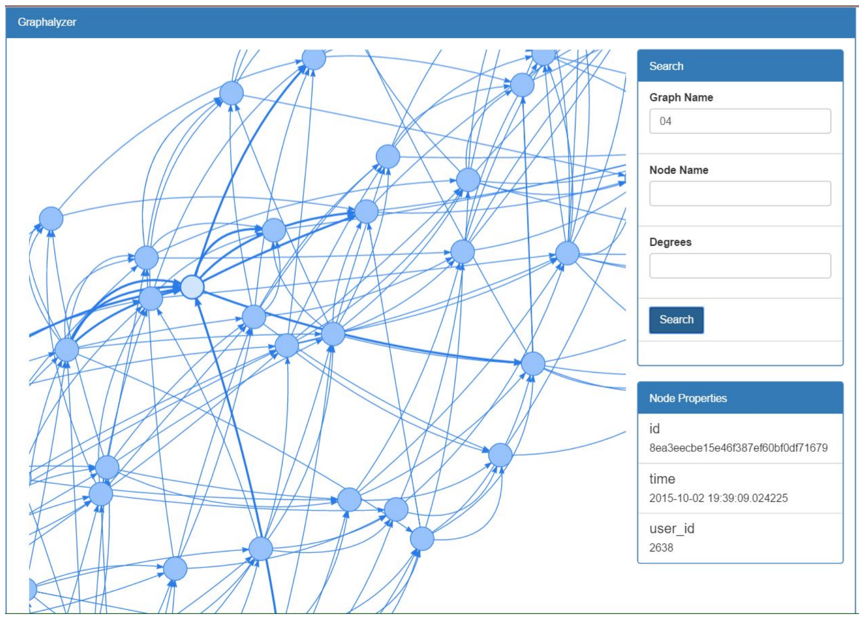


Figure 12: Initial UI design #3

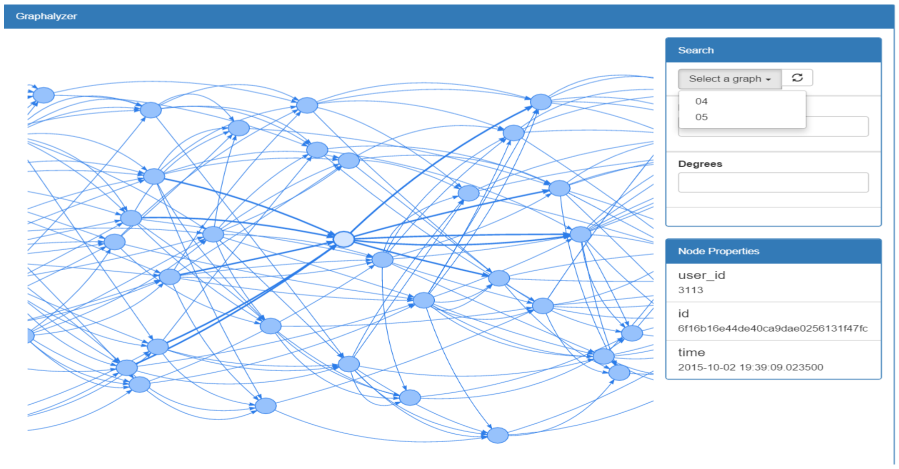
A2.2. October 2015 - January 2016

This was our initial design using React.js as our frontend framework. We used libraries such as react-bootstrap to make the design more intuitive to the user. The problems with this design was lack of implementation for filtering or subgraphs, and the configurations we had set for our graphs were detrimental to performance, such as having “smooth and curvy” edges.



id
8ea3eecbe15e46f387ef60bf0df71679
time
2015-10-02 19:39:09.024225
user_id
2638

Figure 13: Second UI design #1



user_id
3113

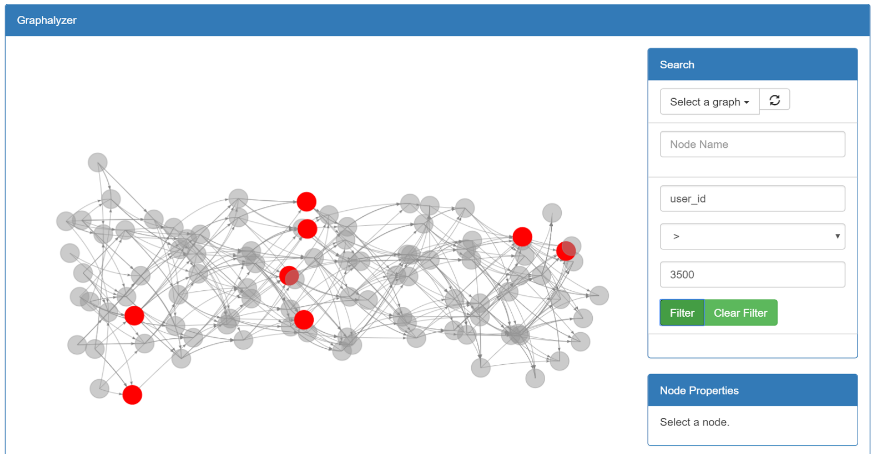
id
6f16b16e44de40ca9dae0256131f47fc

time
2015-10-02 19:39:09.023500

Figure 14: Second UI design #2

A2.3. February-March 2016

This design was used after we implemented filtering and a loading bar for the users. This design was close to our final product, but was still missing other features such as subgraphs and URL exporting. In addition, the Search Panel was getting too cluttered and was no longer "just a search panel".



user_id
>
3500

Select a node.

Figure 15: Third UI design #1

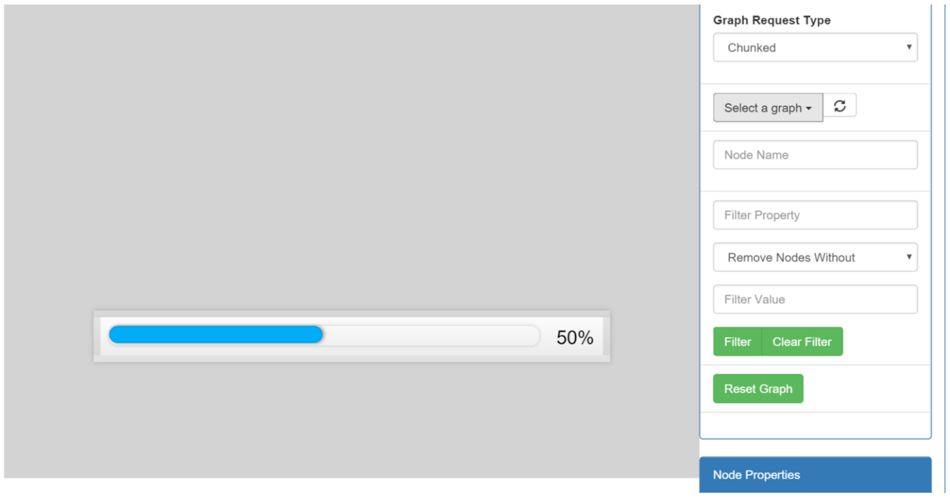


Figure 16: Third UI design #2

Appendix III. Other Considerations

A3.1. Future of Graphalyzer

Graphalyzer is continuing development under Workiva after the conclusion of this semester. They have notified us that they will be getting to work on this, most likely with future interns, and uploading their own data.

We believe that Graphalyzer can be used for many companies or individuals who want to analyze Big Data, so **maintaining the open source status** of this tool is important to us. Workiva has agreed to keep Graphalyzer open source, and they will maintain their own fork of our GitHub repository, and will designate their fork as the "main version".

A3.2. Additional Features

During our weekly meetings throughout the second semester, Workiva began using Graphalyzer more themselves rather than us demonstrating the features for them. This was valuable to our improvement of the application, because they are primary users. They have made notes of features they would like to see added, if we had more time to work on the project.

A3.2.1. Exporting Graphs in SVG Format

Graph canvases could be exported in SVG image format for quick data visualization sharing between users. This cuts out the need to use Graphalyzer in the browser when the only thing needed in this case is a brief visualization.

A3.2.2. Adding Multiple Filters

Graphs could be put through multiple filters, each filter designated with its own color. A use case example would be a graph of documents within a server that were related to each other in some arbitrary way. If the user wanted to filter the documents based on their types and multiple types existed, Graphalyzer could designate each type with its own color, making the data easier to understand.

A3.2.3. Clustering By Zoom

While not a high priority item because of the powerful ability to draw subgraphs, if the user wanted to see an entire graph but only show nodes of high importance, the graph could cluster all "lesser nodes" into one cluster node when zoomed out. Zooming in on the graph canvas would open these clusters and make the data more detailed in appearance.