Web-Based Galant

Interim Project Report

Requirements, Design, Implementation & Testing

Dr. Matthias Stallmann

CSC 492 Team 37

Noah Alexander, Rishabh Karwa, Art Schell, Yuval Sherman, Andrew Watson

North Carolina State University

Department of Computer Science

2/24/23

# Executive Summary *Author(s): Yuval Sherman Reviewer(s)/Editor(s): Noah Alexander*

Dr. Stallmann is a professor in the computer science department at NCSU. He has developed tools for visualization of algorithms and proof techniques, the most recent of which is “Graph algorithm animation tool” or Galant. He teaches CSC505, a graduate level course on design and analysis of algorithms and uses Galant pedagogically in class as well as in a research capacity.

Galant as it is implemented now requires the user to have other software such as git, Apache ant, and a JDK installed on their device. It has also suffered some feature bloat over its 10-year lifespan and has come to be overly complex with many unnecessary features. It is also not fully platform independent, and users may experience different behavior depending on their OS. Galant is implemented primarily in Java and compiles algorithm animation code to Java, which causes errors to be unclear and difficult to debug.

Our proposed solution is to create a web-based version of the core features of Galant implemented in JavaScript. This web application will be usable on Chrome, Firefox, Edge, and Safari without requiring the user to download anything onto their device. By rewriting the algorithm and control flow, we can greatly simplify the implementation by cutting out unnecessary features. Cutting out the step of compiling the algorithm pseudo code to Java will mean that syntax or runtime errors that occur in the execution of the algorithmic steps will be easier to propagate to the user in a way that makes sense to the user without knowledge of the implementation.

We have reviewed requirements with Dr. Stallmann and finalized 5 core use cases. These include being able to access the web app, upload a graph, upload an algorithm, and be able to step forward and backward through that algorithm and see visual updates on the graph. We have completed a high-level design and reviewed it with Dr. Stallmann, who helped us narrow our scope and cut out some steps that were deemed to be unnecessary. We decided on using a predicate-state logic to maintain our algorithm and display states separately as well as allow the user to step forward and backward without issue or increased complexity of code. We are using VSCode as our primary IDE, ReactJS as our frontend framework, Cytoscape.js as our graphing library, and Jest as our testing framework. We have created black box system tests for all our use cases, and are adding more as we go. We have finished our first iteration of development, which includes use cases 0 and 1, access to the web page and uploading a graph respectively. We have begun on iteration 2, which includes being able to upload an algorithm and step through it with console feedback. We have completed a low level design detailing how the algorithm upload will interact with the algorithm runner thread and vice versa, as well as how the thread will interact with the console and the graph visual. We expect this iteration to be completed by spring break and we have split the work into a few independent portions to be worked on concurrently.

# Project Description *Author(s): Yuval Sherman Reviewer(s)/Editor(s): Noah Alexander*

## Sponsor Background

Dr. Stallmann is a professor in the computer science department at NCSU. His primary research interests are in combinatorial optimization with emphasis on dealing with NP-completeness. His current projects include evaluation of algorithms for NP-hard combinatorial optimization problems as well as the theory and practical applications of NP-completeness. He has also developed tools for visualization of algorithms and proof techniques, the most recent of which is “Graph algorithm animation tool” or Galant. He also teaches CSC505, a graduate level course on design and analysis of algorithms. He uses Galant pedagogically in class as well as in a research capacity. Galant has also been used in the past in classes like CSC316.

Matthias Stallmann, Professor of Computer Science, 2302 EB2 890 Oval Dr, Raleigh, NC 27695, 919-515-7978, mfms@ncsu.edu.

## **Problem Description**

Galant as it is implemented now requires the user to have other software such as git, Apache ant, and a JDK installed on their device. It has also suffered some feature bloat over its 10-year lifespan and has come to be overly complex with many unnecessary features. It is also not fully platform independent, and users may experience different behavior depending on if they’re using Mac, Windows, or Linux. Galant is implemented primarily in Java and compiles algorithm animation code to Java, which causes errors to be unclear and require the user to have a working knowledge of the Galant implementation to debug. Dr. Stallmann wants the core features of Galant to be ported to the web to streamline the user experience and simplify the implementation.

## **Proposed Solution & Project Goals/Benefits**

Our proposed solution is to create a web-based version of the core features of Galant implemented in JavaScript. This web application will be usable on Chrome, Firefox, Edge, and Safari without requiring the user to download anything onto their device. By rewriting the algorithm and control flow, we have the opportunity to greatly simplify the implementation by cutting out unnecessary features. Cutting out the step of compiling the algorithm pseudo code to Java will mean that syntax or runtime errors that occur in the execution of the algorithmic steps will be easier to propagate to the user without knowledge of the implementation. These improvements will allow Dr. Stallmann to make better use of the algorithm visualizations in his research and his classes. The streamlined user experience will also mean that it is possible to quickly demo a prepared algorithm on any computer without the hassle of fulfilling the technical requirements to run classical Galant.

# Resources Needed *Author(s): Andrew Watson Editor(s): Art Schell, Yuval Sherman*

| Resource | Purpose | Status | Version | Licensing Info |
| --- | --- | --- | --- | --- |
| Galant | Galant does not exist in our stack, but it is what we are pulling from and rewriting for our website. | Obtained. It can be found on Dr. Stallmann’s [GitHub](https://github.com/mfms-ncsu/galant). | Most recent release is Haydn 1.3, but we are working from Galant’s current state. | Gnu Public License. |
| GitHub | We use this for version control and project management. | Obtained, provided by the teaching staff. | GitHub Enterprise Server 3.7.4 | N/A |
| GitHub Actions | We use this to deploy our project and to manage some of our issue workflows within GitHub. | Obtained, provided by the teaching staff. | N/A | N/A |
| CSC IT Virtual Machine | Virtual machine to host and serve the web pages we create. Will get a permanent domain at galant.csc.ncsu.edu at the end of semester | Obtained, provided by CSC IT | N/A | N/A |
| React | React is a declarative, component- and state-based framework for creating user interfaces. It allows us to control and pull from things in our UI like the graph and algorithm. | Obtained. | v18.2.0 | MIT |
| Cytoscape.js | Package to display the graph and its animation to the user. | Obtained. | 2.0.0 | MIT |
| Jest | Unit Testing | Obtained. | v24 | MIT |

# Risks & Risk Mitigation *Author(s): Andrew Watson Reviewer(s)/Editor(s): Yuval Sherman*

There are a few potential risks we may encounter in the course of this project. First is our choice of framework. The project is unusual compared to others our team has undertaken. We are familiar with such complexity in Java projects, but we don’t have experience with such sprawling, feature-rich JavaScript projects. Based on our initial research, we have chosen React as the framework to use for the user interface. We have chosen it based on its component-based structure which seems to act in a similar fashion to classes. We hope this will allow us to closely mimic the original Galant project, requiring less rewriting on our end. However, it is possible, given our inexperience, that we have made a poor choice and that React is not able to accomplish what we want it to. It is also possible that React can handle what we need, but takes much more adaptation of the original code than it would have been to use vanilla JavaScript.

A risk that we’re facing is that the nature of our development cycle may mean that work is repeated as we get farther into the semester. Our sponsor has made it clear he wants us to get certain core features working now in a simple manner, but this style of implementing things right now just to get them working may mean that later on, changing these things to handle more complex input or different styles of algorithms could mean retracing our steps. We will attempt to create a system that leaves the door open for future iterations and improvements by keeping our design simple and documenting well. The area of development that is most at risk of this is the formalization of the algorithm pseudo code we are expecting. Our sponsor has suggested that we start by accepting “pseudo code” that is very close to JavaScript to ease the gap between pseudo code and actually running the algorithm. However, if we design around the idea that we don’t need to parse the algorithm code very much, it will likely be difficult to change the pseudo code format in the future to be more general and less language-oriented.

# Development Methodology *Author(s): Andrew Watson Reviewer(s)/Editor(s): Rishabh Karwa*

Our development methodology is similar to an iterative process, but distinct in several important ways. In our first stage, we created broad requirements for our vision for the entire project. We then created a high level design for what components would be needed to complete a basic version of the project. We will then go component by component and create a low level design for each. Finally, we will iteratively develop, test, and deploy our project. We do not have a strict iterative schedule, but we plan to move quickly and efficiently to ensure full and robust code that we are confident in.

We have identified small, three week development iterations for which we can design, code and test our project. We are currently finished with our first iteration and beginning our second. In practice, our iterations have bled into each other with bug fixes from the previous iteration being done simultaneously with the design of the next.

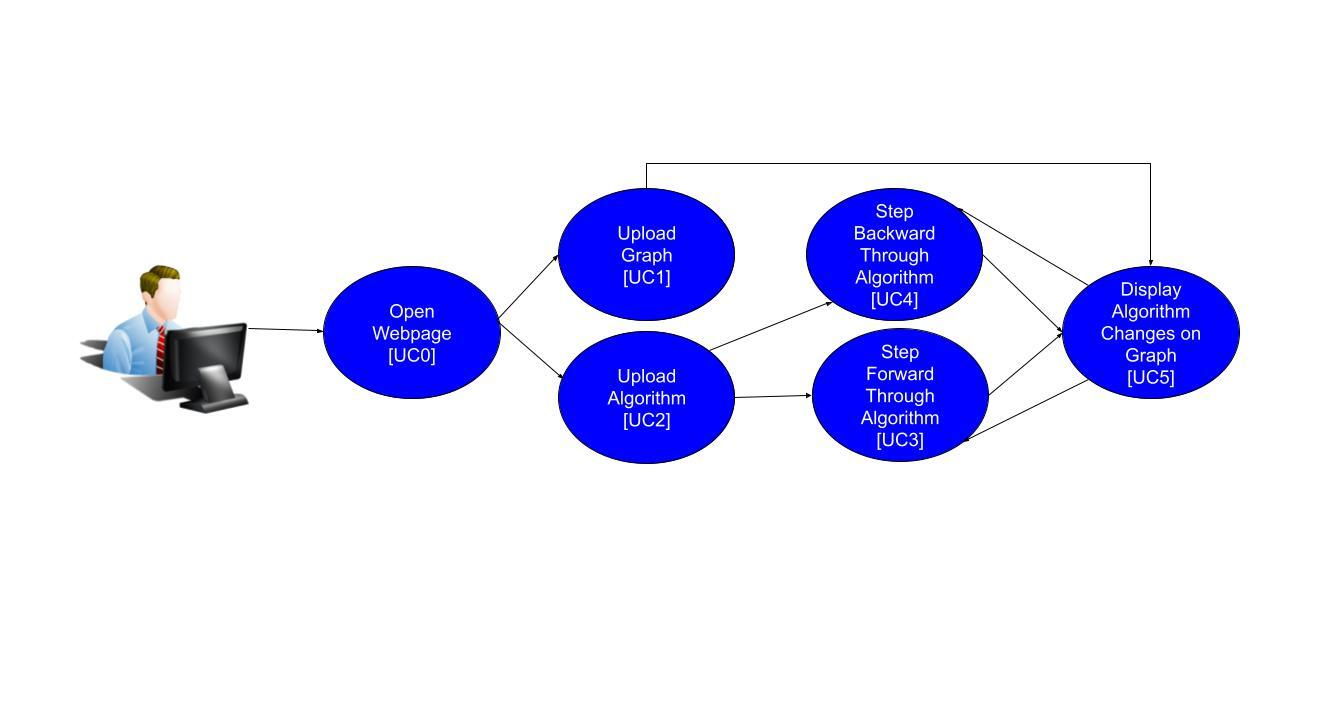
We also plan to rotate the roles we as a team take on (such as Team Leader or Technical Lead). These will rotate approximately every two weeks. More details can be found in the Task Plan section below.

We are utilizing most of the features of GitHub for our project management. We are keeping our requirement, design, and system test documentation in the repository’s wiki. We are using GitHub Issues to track bugs and feature improvements, and the Projects page to track the progress of issues. We are using GitHub Actions to run test and test coverage tools to verify changes before we merge branches.

# System Requirements

*Author(s): Rishabh Karwa*

*Editor(s): Art Schell, Yuval Sherman*

After the first meeting with Dr. Stallman, we broke the project down into functional requirements, non-functional requirements, and constraints. The functional requirements consisted of 8 use cases. After our second meeting, we simplified much of the scope of our project, reducing this down to 5 use cases. We may expand on these requirements if we have time to add more functionality to the project after iteration 2 (implementing functionality to upload and step forwards/backwards through an algorithm with the changes displayed on the graph).

## Functional Requirements

| **0** | ID | **[UC0] Galant Webpage** |
| --- | --- | --- |
| Description | A user should be able to navigate to the web-based Galant webpage. |
| **0.1** | Preconditions | None |
| **0.2** | Main Flow | User navigates to the Galant webpage **[E1]**. User has the following options:   * upload graph files **[UC1]** * upload algorithm file **[UC2]** |
| **0.3** | Sub-flows | None |
| **0.4** | Alternative Flows | **[E1]** The webpage shows an appropriate 404 page if an incorrect link is used. |
| **0.5** | Data Format | None |

| **1** | ID | **[UC1] Upload a Graph** |
| --- | --- | --- |
| Description | A user should be able to load a graph file in a specified format and see it displayed on the page |
| **1.1** | Preconditions | A user is able to access the web-based-galant webpage **[UC0]** |
| **1.2** | Main Flow | The user chooses to upload a file and is prompted to choose a file from their device **[E1][E2]**.  The graph information in the file is displayed to the user on the page **[S1][S2].** |
| **1.3** | Sub-flows | **[S1]** Nodes and Edges appear as specified in the input file  **[S2]** The graph coordinates should scale squarely within the window dimensions so that the entire graph is visible |
| **1.4** | Alternative Flows | **[E1]** The format of the input file does not match the specified format in section **1.5**. The graph will not load and the user will see an error message saying that the input file was not valid.  **[E2]** The input file matches the specified format in section **1.5** but is missing required information or another syntax error. The graph will not load and the user will see an error message with the line number and description of the error |
| **1.5** | Data Format | An empty line is ignored.  A line beginning with ‘c’ is ignored, as a comment.  A line beginning with ‘n’ defines a node. After this, values are separated by spaces.   * The first value is a string representing the node’s id. * The next two values are numbers representing the x,y coordinates of the node. * If the next value does not contain a colon, it will specify the weight of the node. Otherwise, the weight will be null. * All future values must contain a colon. These may be used in the future to add more properties. By default, the keys and values will be interpreted as strings.   A line beginning with ‘e’ defines an undirected edge. After this, values are separated by spaces.   * The first two values are strings representing the ids of the connected nodes, in the order source -> target * If the next value does not contain a colon, it will specify the weight of the edge. Otherwise, the weight will be null. * All future values must contain a colon. These may be used in the future to add more properties. By default, the keys and values will be interpreted as strings.   A line beginning with ‘d’ defines a directed edge. After this, values are separated by spaces.   * The first two values are strings representing the ids of the connected nodes, in the order source -> target. * If the next value does not contain a colon, it will specify the weight of the edge. Otherwise, the weight will be null. * All future values must contain a colon. These may be used in the future to add more properties. By default, the keys and values will be interpreted as strings. |

| **2** | ID | **[UC2] Upload an Algorithm** |
| --- | --- | --- |
| Description | A user should be able to upload an algorithm file in JavaScript format. |
| **2.1** | Preconditions | A user is able to access the web-based-galant webpage **[UC0]**. |
| **2.2** | Main Flow | The user chooses to upload a file and is prompted to choose a file from their device **[E1][E2]**.  The algorithm in the file is displayed to the user on the page. |
| **2.3** | Sub-flows | None |
| **2.4** | Alternative Flows | **[E1]** The selected file is not a JavaScript file. The algorithm will not load and the user will see an error message saying that the input file was not valid. |
| **2.5** | Data Format | The algorithms will be expected to be pure JavaScript code, with some additional built-in functions that allow the user to interface with the graph visualization system. These are discussed further in **[UC3]** and **[UC4]**. |

| **3** | ID | **[UC3] Step Forward Through an Algorithm** |
| --- | --- | --- |
| Description | A user should be able to navigate through a loaded algorithm step by step. |
| **3.1** | Preconditions | A user is able to access the web-based Galant webpage **[UC0]**.  A user is able to upload an algorithm file from their device **[UC2]**. |
| **3.2** | Main Flow | User attempts to step forward in the algorithm **[E1][E2][E3][E4][E5]**. The algorithm stops executing code after it applies a rule that changes the graph via a built-in function, and prints any messages or errors to the console **[S2]**. The result is displayed in a console. |
| **3.3** | Sub-flows | **[S1]** The code contains standard, single-threaded JavaScript code which does not interface with any external APIs, be these libraries, the browser, the OS, or the web.   * This code should all be runnable with no issues, even if it is complex, using advanced features like object orientation, data structures, and so on. This should be a pure implementation of JavaScript.   **[S2]** The code contains builtin function calls that access the console.   * The code should be able to print any value to the console. |
| **3.4** | Alternative Flows | **[E1]** The user tries to run an algorithm, but it has a syntax error.   * The syntax error is reported to the user. The location of the error should be clearly identified.   **[E2]** The user tries to run an algorithm, but it has a runtime exception.   * The exception is reported to the user. The location and reason of the exception should be clearly identified.   **[E3]** The user steps forward, but the algorithm has finished.   * Nothing is executed.   **[E4]** The user steps forward, and the algorithm takes a significant amount of time to execute a step.   * The algorithm being slow or being caught in a loop should not halt the rest of the browser-based UI. It should be displayed to the user that the algorithm is still processing a step.   **[E5]** The user attempts to call functions or access web APIs that are not permitted.   * The algorithm does not permit the user to step forward, and the user is alerted that their code is not allowed to be run |
| **3.5** | Security Requirements | **[SEC1]** The algorithm must be unable to interface with the browser. This means it must not be able to access the document or window objects, and should have no access to the browser API functions.  **[SEC2]** The algorithm must be unable to run threads. This means it must not be able to access Web Workers or similar APIs.  **[SEC3]** The algorithm must be unable to access the Internet. This means it must not be able to access HTTP requests, socket libraries, or similar APIs.  **[SEC4]** The algorithm must be unable to access external variables in other parts of Galant. The only variables that should be available to the algorithm are its own local variables, as well as Galant builtin functions and the values they return. |

| **4** | ID | **[UC4] Step Backward Through an Algorithm** |
| --- | --- | --- |
| Description | The user should be able to step backwards through an algorithm. |
| **4.1** | Preconditions | A user is able to access the web-based Galant webpage **[UC0]**.  A user is able to upload an algorithm file from their device **[UC2]**.  A user is able to step forward through an algorithm **[UC3]**. |
| **4.2** | Main Flow | The user steps backward in the algorithm **[E1]**. Any results from the last step forward **[UC3]** are undone. Stepping forward again will re-do what was just undone, rather than executing a new step. |
| **4.3** | Sub-flows | None |
| **4.4** | Alternative Flows | **[E1]** The user steps backward, but everything has already been undone. Nothing is undone, it is displayed to the user somehow that the algorithm is not in the middle of execution. |
| **4.5** | Security Requirements | The security requirements are the same as **[UC3].** |

| **5** | ID | **[UC5] Update the visuals of the displayed graph with the algorithm** |
| --- | --- | --- |
| Description | The visual representation of the graph should be updated as the user steps forwards and backwards. |
| **5.1** | Preconditions | A user is able to access the web-based Galant webpage **[UC0].**  The graph is loaded and displayed **[UC1]**.  An algorithm is loaded **[UC2]**.  The user is stepping through the algorithm **[UC3]**. |
| **5.2** | Main Flow | The user steps through the loaded algorithm (this is a modification of **UC3**). Rather than reaching a print statement, the algorithm can also find a result from a built-in function call that affects the graph. As the graph-state changes from the algorithm, the visual display of the graph is updated. **[E1][E2]** |
| **5.3** | Sub-flows | None |
| **5.4** | Alternative Flows | **[E1]** The user attempts to step forwards at the end of the algorithm (or backwards at the beginning). Nothing should change in the visual representation of the graph. |
| **5.5** | Security Requirements | None |

## Non-Functional Requirements

[NFR1] System **should** be intuitive to use for new users

* The Galant website will have a link to a user guide specifying how they can input graph and algorithm files into Galant and run the program as needed. This will include the graph text file format to create graphs.

[NFR2] System **should** have a programmer guide that helps algorithm writers understand the algorithm functionality.

* The programmer guide will specify all built-in methods that will be used for an algorithm JavaScript file, so that they can input their own algorithm files in Galant. It will also contain examples of algorithm files.

[NFR3] System **should** have a developer guide that helps algorithm writers understand the software.

* The developer guide will contain comprehensive information about the entire codebase, such as the dependencies between files, and how they interact with each other. This will allow for future iterations of the project to continue without major roadblocks.

[NFR4] System **must** run in the browser without downloading source code or having any tools locally installed.

* The system **must** run in Chrome, Firefox, Edge, and Safari

## Constraints

[C1] The system’s animation and graph reading **must** be implemented in JavaScript.

[C2] The system **must** implement graph files as text files and algorithm files as JavaScript files.

[C3] System **must** only run algorithm code as it is stepped through, not all at once.

[C4] The system may use any JavaScript libraries or extensions.

[C5] System **must** accept input graphs in the format defined insection 1.5

# Design *Author(s): Art Schell, Yuval Sherman Reviewer(s)/Editor(s): Rishabh Karwa*

## **High-Level Design**

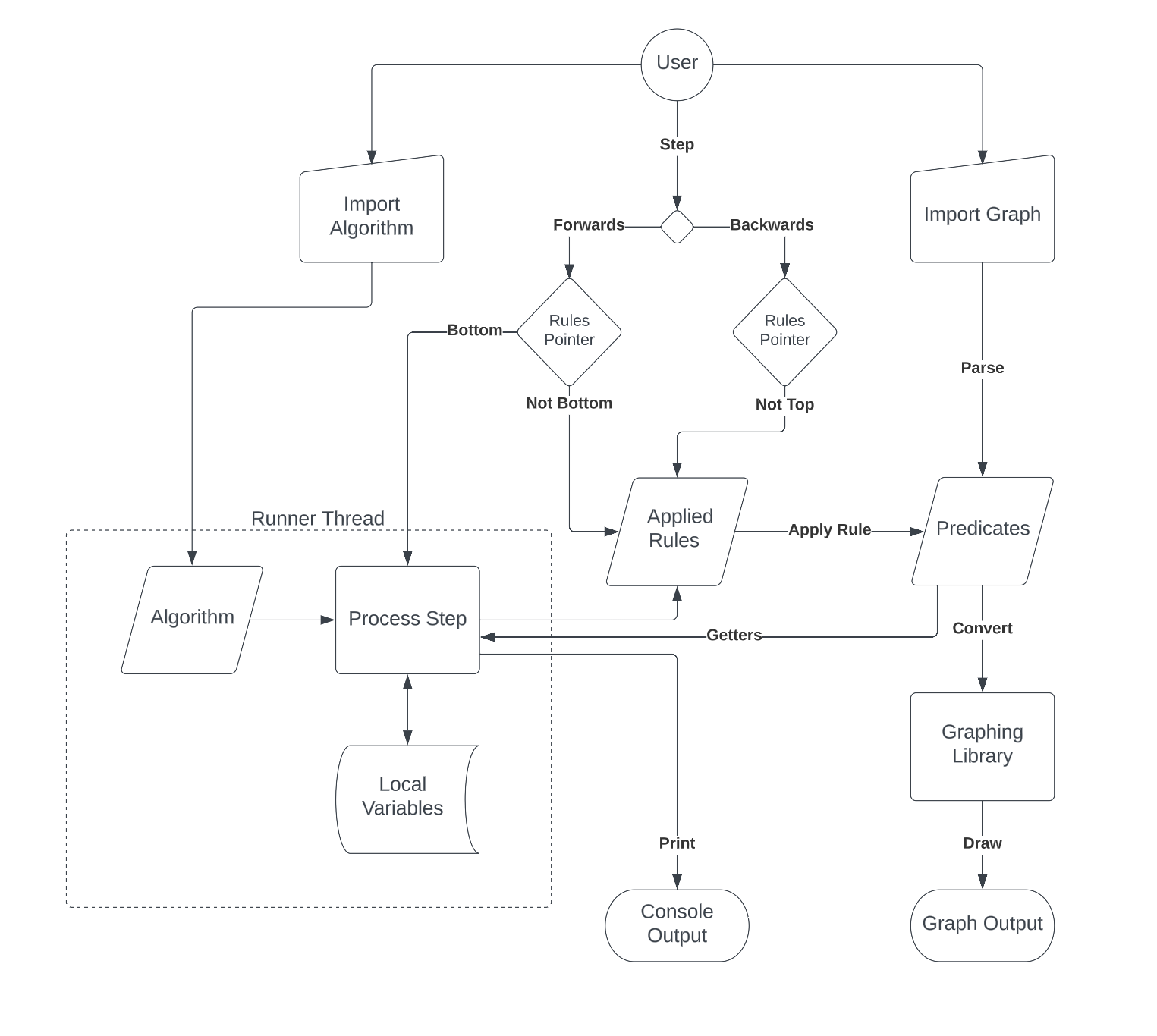
This is a diagram of our high-level design.

Figure 1. High Level Design

On the graph side of things, graphs will be uploaded by a user and parsed into an internal representation based on predicates that define the nodes and edges. This representation will be able to interface with “rules” used by the algorithm that have pre-defined inversions. This way a rule’s effect on the predicates can be reliably undone when the user steps backward.

To display the graph, the internal graph format will be converted into a format recognized by Cytoscape, a graph visualization library we are using to display the graph.

On the algorithm side of things, the algorithm will be uploaded and passed to a thread. Reuploading a graph or algorithm during execution will reset execution of the algorithm. The algorithm will be compiled and parsed into the required format, though we hope that by implementing the algorithm in JavaScript we can minimize this process, limiting it to creating a secure context in which to run the algorithm, injecting built-in functions, and finding points at which the algorithm should pause execution.

As the algorithm is executed by the user, any results produced by built-in functions will be added to a list as rules to apply. Undoing steps of the algorithm will keep the algorithm in the same place, but undo the effects of the applied rules to revert the visual state of the graph. When redoing, if there are any rules in the list yet to be executed, these will be executed before resuming execution of the algorithm. Algorithm outputs will also be able to interface with the console.

## Frontend Design

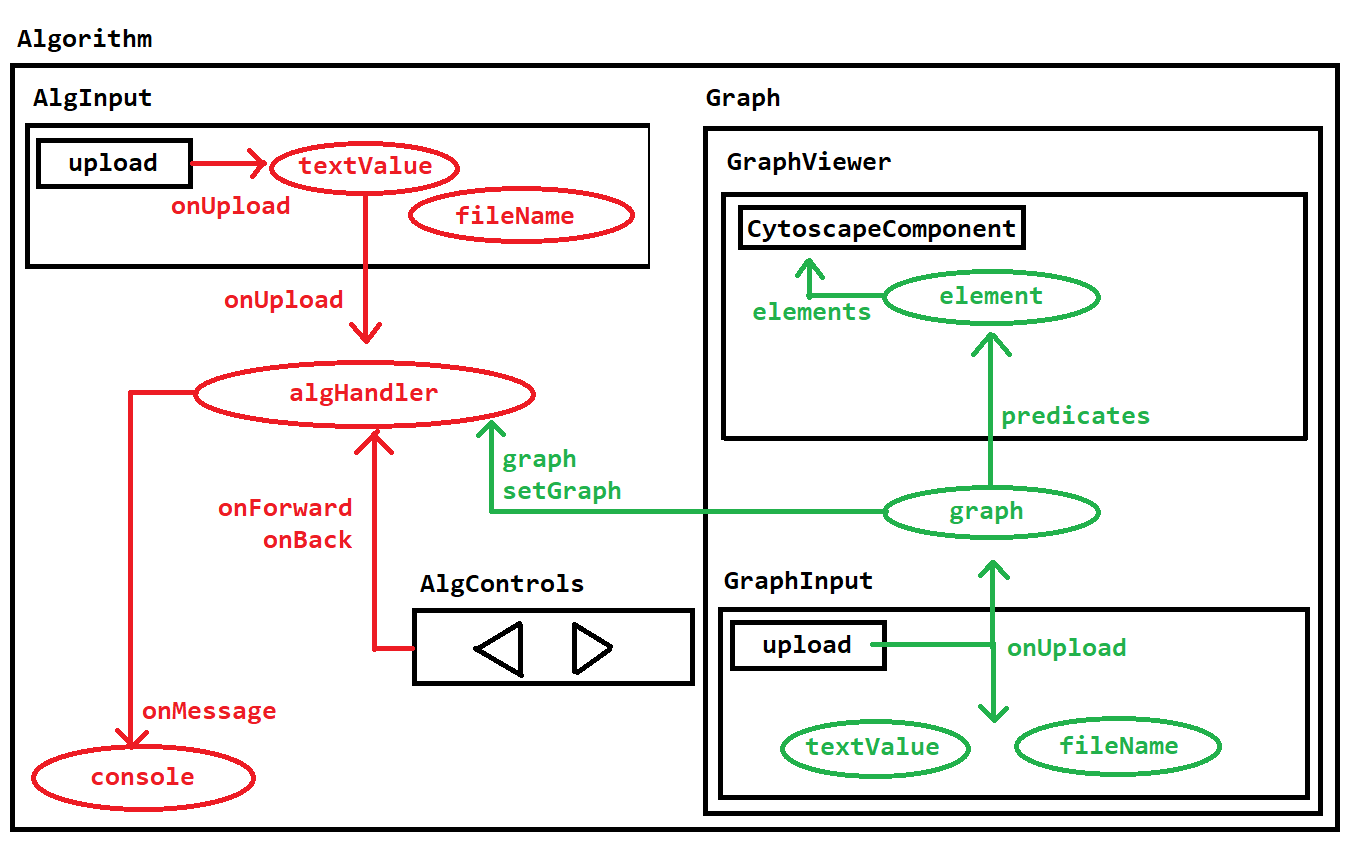
This is a diagram of our front-end design using React, also displaying data flow between components utilizing properties and callback functions.

Figure 2. Frontend Component Design

The Graph component receives input from GraphInput which handles prompting the user for a file and displaying the uploaded file. This will convert and pass the graph outwards via a callback function where it is stored in the Graph component. The graph will then be displayed in the GraphViewer which converts it into a form Cytoscape understands.

On the Algorithm side, an AlgInput functions similarly to the GraphInput, receiving files from the user. Uploading an algorithm will reset the current state of the executing algorithm stored in AlgHandler. This will also happen when you upload a graph, as both the new graph and the function that updates it will be passed out of the Graph component.

Users will control the algorithm via forward and back buttons in AlgControls, which will call methods of the AlgHandler. The AlgHandler will also output any console data that the algorithm produces to be displayed.

## Backend Design

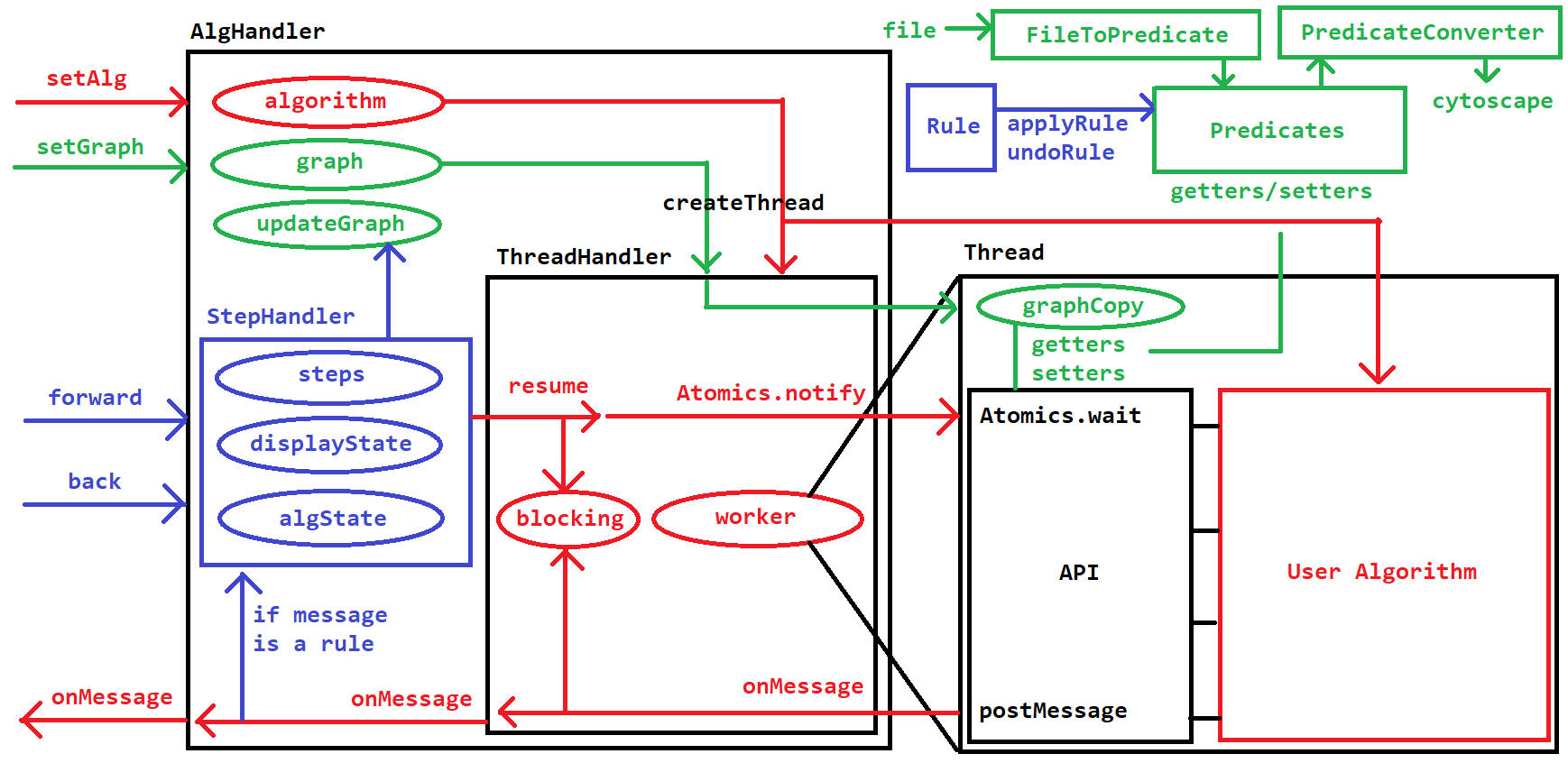


Figure 3. Backend Sequence Flow

The graph side of things in the backend is very simple. FileToPredicate and PredicateConverter convert the graph from the file format to the predicate format to the Cytoscape format. The predicate format is defined as follows:

{  
 “nodes”: {  
 string (id): {“x”: number, “y”: number, “weight”: number,  
 “color”: string, “marked”: bool, “selected”: bool}, …more nodes

}  
 “undirected”: {  
 number (id): {“source”: string, “target”: string, “weight”: number,  
 “color”: string, “selected”: bool}, …more edges  
 }  
 “directed”: {same format as undirected}  
}

The algorithm side is more involved. The AlgHandler is the interface into the algorithm execution that lives in the frontend. It is given the algorithm and graph via setters, and once it has both it can create a ThreadHandler which manages a thread. This communicates with the thread via Atomics.notify, allowing the thread to pause execution with Atomics.wait until notified by the ThreadHandler. The thread communicates back with postMessage, and these messages are received and parsed. The ThreadHandler will also manage whether the thread is currently executing and further instructions need to be blocked.

Inside the thread, the user algorithm communicates with the graph and outside program via the defined API. This API wraps the getters and setters of the Predicates into functions exposed to the user program, and ensures that any modifications to the graph are reported to the outside via postMessage. The API connects to a copy of the graph which is used to receive information about the graph without relying on the frontend’s copy, which may change due to undoing and redoing steps.

Messages from the thread are categorized into two types. Basic messages are just forwarded to the console as messages to display. Error messages also fall under this category. Rule messages report rules that have been executed, which are handled by the StepHandler. This object keeps track of the list of executed rules and receives commands from the forward and back buttons. It uses the existing rules when the display state is less than the algorithm state, or calls on the thread to resume execution if necessary.

Rules are executed via applyRule and undoRule methods of the Predicates. The rules are stored in the following format:

{  
 “type”: string (the type of the modified predicate, for instance node or edge)  
 “id”: string/number (the id of the modified predicate)  
 “property”: string (the property of the modified predicate, for instance weight or color)  
 “old”: [value] (the old value of the property, to return to after an undo  
 “new”: [value] (the new value of the property, to set after an application  
}

## GUI Design

These wireframes have been reviewed and approved by Dr. Stallmann. They are subject to minor changes based on implementation decisions. The graph-visual and graph-text portion have been implemented and tested as shown below. The algorithm text and console portion are to be implemented in the ongoing iteration.

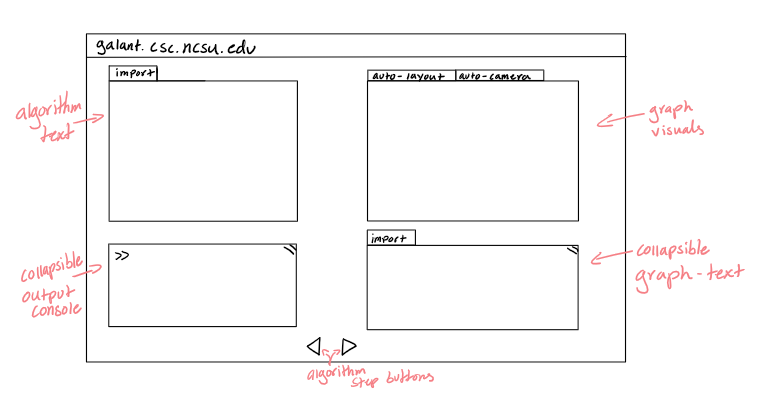
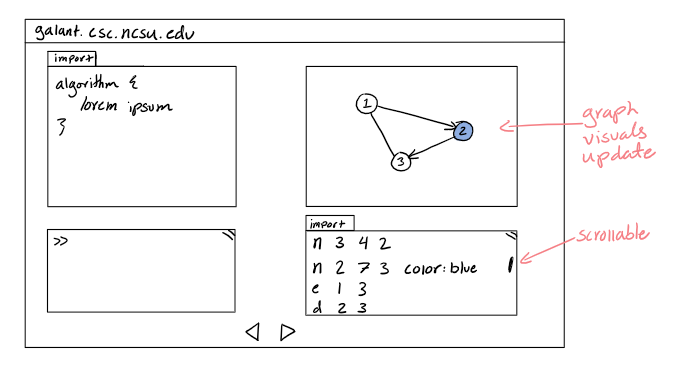
This is the view that the user will see upon first reaching the page. There are options for the user to import an algorithm and a graph.

Figure 4. Frontend On-Load Wireframe

Once a graph is imported, its text representation and visual representation are both visible to the user. When an algorithm is imported, the text is visible on screen. The user has the option to “lock” the algorithm and put it in a state in which it can be stepped through.

Figure 5. Frontend In-Use Wireframe

As the user steps through the algorithm, they can see changes to the graph such as colors appear on the visual tab. Changes to the text representation of the graph are also visible.

# Implementation *Author(s): Art Schell Reviewer(s)/Editor(s): Yuval Sherman*

## Iteration Definition & Current Status

**Iteration 1**: Create the Galant webpage (UC0), set up required dependencies, and complete the graph feature, from uploading a graph to displaying it (UC1).

**Iteration 2**: Create the algorithm system, allowing the user to run a basic algorithm that colors all graph nodes, and having the ability to step forward and back. (UC2, UC3, UC4, and bare minimum for UC5)

**Iteration 3**: Create the algorithm’s full API, with access to all features for visually displaying things on the graph, data structures, and other important algorithm functions.

Further iterations will build in additional features and functionality depending on the time available.

Our current status is that we have finished Iteration 1 with the full graph workflow, from upload to display. This functionality works well, and by integrating with an existing graph visualization library we were able to implement a few bonus features such as automatic graph layout and camera control. We will want to add a few additional features to this use case moving forward, for instance the ability to hide graph weights and labels, a feature that exists in the current algorithm API for Java Galant. We have also done the design work for Iteration 2 and are ready to begin implementation of this iteration’s features in the coming weeks.

## Security Considerations

Our primary threat model is the user’s input of algorithm files. We must ensure that our algorithm code is properly sandboxed to prevent users from being tricked into running code that interferes with their privacy, by reading information from their computer (such as cookies), or their integrity of actions they perform on the Web, by making unauthorized requests to other sites. Thus, we have included requirements to ensure these actions are not permitted in algorithm code.

To limit the possible security risks of a web application, we have decided to implement all functionality client-side, with the web server’s only function being to host static HTML and JS files. This limits the amount of potential attack targets of our system, as we have no account system or private information. This means issues of confidentiality, integrity, privacy, and authentication are minimized. The issue of availability does still act as a potential target, as a DDOS attack or similar could still be a threat. We will ensure our server has protections and logging of requests to ensure this by using a standard web server program with built-in security measures. This will be discussed with NCSU IT.

## Project Folder Structure

Generally, we have our project folder structure laid out into two main parts, Frontend and Backend. The Frontend folder contains all of our React components, nested in the same way they are on the page. The Backend folder contains classes and functions used in the internal logic of the system, for instance conversion between file or object types and the algorithm system. Our automatically run unit tests are in a \_\_tests\_\_ folder provided by Jest, and manual frontend tests are in the relevant frontend folders.

src  
 \_\_tests\_\_  
 (all unit tests)  
 backend  
 FileToPredicate.js  
 PredicateConverter.js  
 frontend  
 App.jsx  
 App.scss  
 Graph  
 Graph.jsx  
 Graph.scss  
 GraphInput  
 GraphInput.jsx  
 GraphInput.scss   
 GraphViewer  
 GraphInput.jsx  
 GraphInput.scss  
 GraphViewerTest  
 GraphViewerTest.jsx  
 GraphViewerTest.scss

# Project Configuration/Settings A*uthor(s): Andrew Watson Reviewer(s)/Editor(s): Yuval Sherman*

We conversed with CSC IT about obtaining a virtual machine to stand up our drafts of our websites. It currently can be found at *sd-vm41.csc.ncsu.edu*. We have an Apache server running on it which points to the build folder of a copy of our project. This allows for quick manual deployment by pulling the latest copy and creating a production build on the virtual machine. In the future, once we are satisfied with the state of our project, CSC IT will give us access to *galant.csc.ncsu.edu* and we will do the same process. Currently we are just using HTTP. Eventually, we will need to figure out obtaining an SSL certificate so that we can support HTTPS.

# Testing Author(s): Noah Alexander Reviewer(s)/Editor(s): Rishabh Karwa

## Overall View

For this project testing will be done via system tests as well as unit testing. Our project involves converting much of the implementation from Java into JavaScript. After doing some research we have found some frameworks that allow for unit testing with JavaScript. As our project continues to grow more and more, information will be displayed to the user and as such our system tests will be used to make sure that the output is what it needs to be. We will develop test data to be used for both the system and unit tests. Both test data for unit testing as well as system testing will be discussed further in each section.

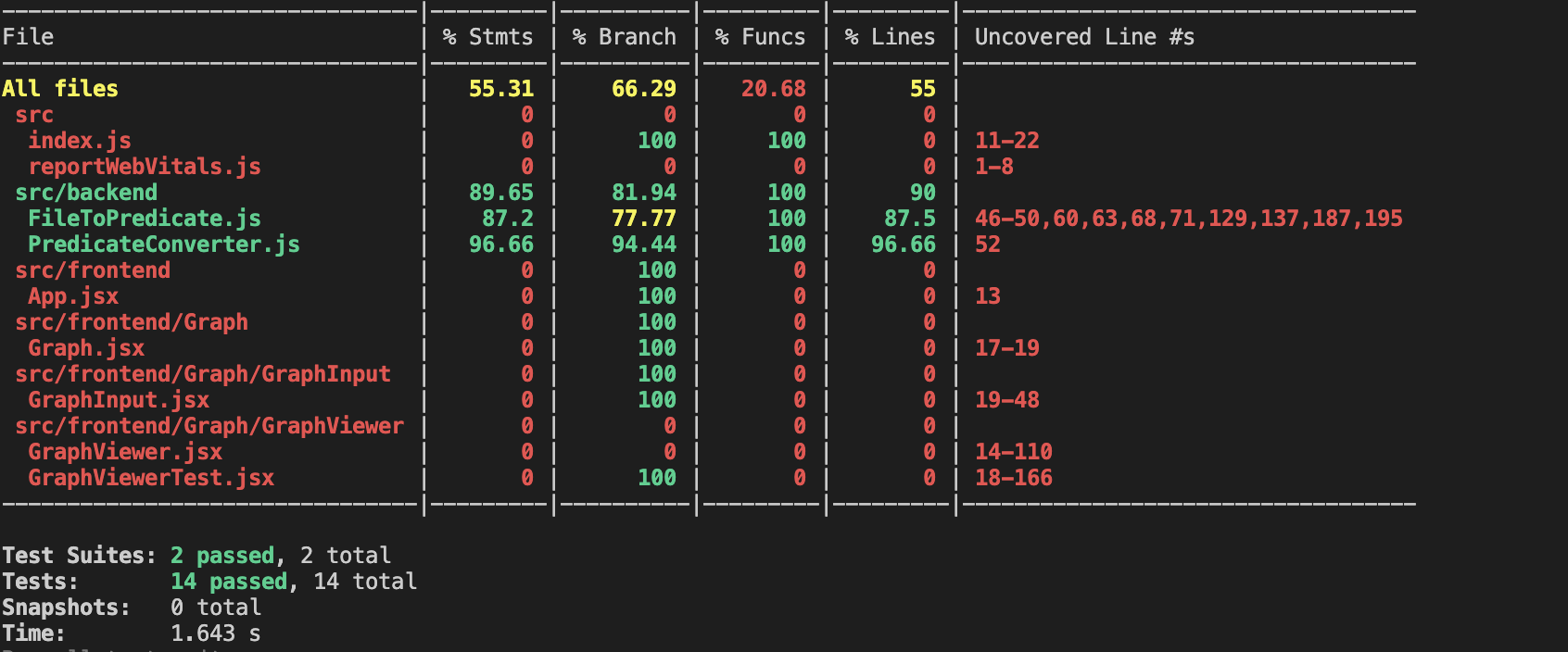
## Unit Testing

We are unit testing with JEST, a JavaScript framework that does unit testing, in order to test our code. Each testing suite file will be tested for at least 70% line coverage and at least 50% branch coverage. The reason for these coverage checks are so that we can be as close to certain as possible that there are no bugs in our project. JEST allows us to create automated unit tests as well as collect test coverage for these tests. We plan on using GitHub actions to automate these coverage checks whenever we push any code. Currently these checks are not in place as we have not pushed to main with these changes. Once there, it will automatically run tests and check for any failing tests as well as line and branch coverages each time someone creates a pull request.

There are currently two unit test classes within our system. These tests reflect the possible paths and scenarios when data is inputted into the program. As we only have two backend components right now, each one has been given its own testing suite. These testing suites test both, error and success routes. As our project grows, more testing suites will be needed and allow us to ensure proper error handling and flow control.

## Unit Test Results

Currently, with our two testing suites there are a combined 14 tests with all tests passing. Since we are using JEST we can find a coverage report of all tests. This figure shows the tests running with a coverage report.



As can be seen both backend files have line coverage greater than 70% and branch coverage greater than 50%.

Our project meets all criteria we set for our unit tests at this portion of the project. We do still have some errors in our frontend that will be covered in the acceptance testing that could potentially be caused by backend code, so even though we have the coverage we still need to keep doing thorough testing for error cases and edge cases.

## Acceptance Testing

For our acceptance tests we have created a system test plan that tests each use case. When testing these use cases we will be creating multiple tests for acceptance and error testing. As much of our project involves displaying a graph to the user as well as running algorithms that change said graph, a main focus of our tests will be making sure that the correct graph states are being displayed. These cannot be tested with unit tests so thorough system tests will need to be done. We will provide sample testing data that is used for our system tests. This section includes our system test cases. Each use case will be tested for error cases as well as acceptance cases. The test id’s for each test case starts with what use case it is testing and then a descriptor of what the test is. There are specific and repeatable tests with expected results and a place to put the actual results of said test. Currently we have tested the first four tests within the system test as these tests deal with loading the webpage and displaying a graph, via a file input. The other tests, which are a majority, involve testing what occurs whenever a graph is displayed and an algorithm is run manipulating this graph. As we have not implemented the algorithm portion, we cannot run these tests.

**Testing files to be used:**

**Bfs.js:**

/\*\*

\* Implementation of breadth-first search. If the graph is undirected, the

\* edges are regarded as being directed both ways.

\*/

function algorithm {

showNodeWeights();

showNodeLabels();

var queue = NodeQueue;

var sequence\_number = 0;

beginStep();

for (var nodes in v) {

setWeight(nodes, INFINITY);

}

var start = getStartNode();

setWeight(start, 0);

highlight(start);

put(start, queue);

label(start, "#" + sequence\_number);

sequence\_number += 1;

endStep();

while ( ! empty(queue) ) {

var next = get(queue);

beginStep();

mark(next);

for (var e in outgoing(other)) {

if ( ! highlighted(other) ) {

highlight(e);

highlight(other);

var distance = weight(next) + 1;

setWeight(other, distance);

put(other, queue);

label(other, "#" + sequence\_number);

sequence\_number += 1;

}

else color(e, BLUE);

}

endStep();

}

}

**test\_graph\_1.txt**

n a 400 500 20 label:label\_a

n b 0 500 200 label:label\_b

n c 200 0 15

e a b 20

e a c 15

e b c 10

**error\_graph\_1.txt**

n a 400 500 20 label:label\_a

n b 0 500 200 label:label\_b

n 200 0

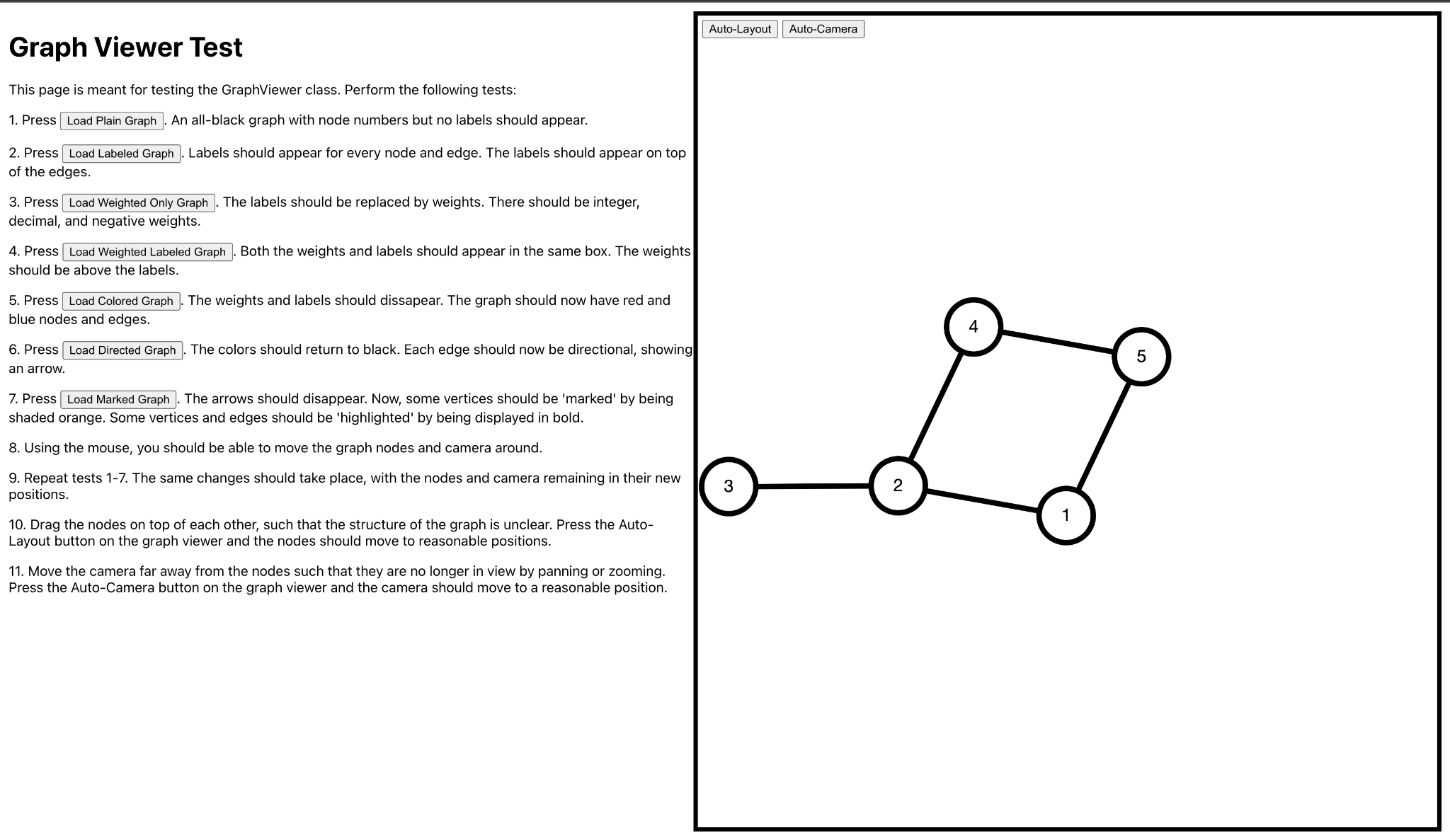
e a b 20

e a c 15

e b c 10

| **Test Name** | **Description** | **Expected Results** | **Actual Results** |
| --- | --- | --- | --- |
| UC0\_View\_Webpage | Preconditions:  None  Steps:   1. Navigate to web-based Galant page [url TBD] | The user is able to view the web-based Galant webpage and sees options to import algorithm files and import graph files. They will see a display for the graph, a space for the predicates to be displayed, and a console. | The webpage is displayed successfully with the graph display block on the top half of the webpage, a button to upload a graph, and a display below it. |
| UC1\_Upload\_Graph\_1 | Preconditions:   1. Web-based Galant web page is available and loaded (UC0)   Steps:   1. Click "import a graph" (or its equivalent) 2. Choose test\_graph\_1.txt from the file selector interface | The graph is displayed in the center of the window display sectioned off for the graph. The predicates are formed and then displayed to the user via text in a window.  Predicates:  n a 400 500 20 label:label\_a  n b 0 500 200 label:label\_b  n c 200 0 15  e a b 20  e a c 15  e b c 10  The graph should have the three nodes with the top id being c, the leftmost one rightmost a, and the leftmost one being b. The id’s are in the center. The ‘a’ node will have a display above it with the text of label\_a and a weight above it of 20. The ‘b’ node will have a display above it with the text of label\_b and a weight above it of 200. The ‘c’ node will have a display above it with the text of 15 for the weight. There should also be an undirected edge going from nodes a to b with a weight of 20 displayed, an undirected edge going from a to c with a weight of 15 displayed, and a final undirected edge going from b to c with a weight of 10 displayed. | The graph is not formatted in the center of the display window. It is to the left and the graph too big to fit on in the display. Nodes b and b are partially visible. Moving it around all data on nodes are there and in the correct spaces. The graph is formatted correctly with edges and nodes formatted correctly. The predicates are displayed correctly in its window. |
| UC1\_Upload\_Error\_Graph\_1 | Preconditions:   1. Web-based Galant web page is available and loaded (UC0)   Steps:   1. Click "import a graph" (or its equivalent) 2. Choose error\_graph\_1.txt from the file selector interface | -The user sees an error message stating that the input file was not valid  -No graph loads and contents of the file do not appear on the page | No error is seen on the page and the screen turns white with nothing being displayed. |
| UC1\_Upload\_Error\_Format\_1 | Preconditions:   1. Web-based Galant web page is available and loaded (UC0)   Steps:   1. Click "import a graph" (or its equivalent) 2. Choose format\_error.pdf from the file selector interface | -The user sees an error message stating that the input file type was not valid  -No graph loads and contents of the file do not appear on the page | A graph is populated on the screen with no error being shown. The predicate display shows the binary of the pdf file. |
| UC3\_Step\_Forward\_1 | Preconditions:   1. The webpage is loaded 2. A valid graph with four nodes and three unweighted edges connecting pairs (0,1), (0,2), and (1,3) is loaded in. 3. A breadth-first search algorithm is in the algorithm box.   Steps:   1. Click "Compile and Run" (or its equivalent) 2. Click the "Step Forward" button. | - After "Compile and Run" is pressed, the algorithm box should be locked from editing. Nothing else should be changed in the graph.  - After "Step Forward" is pressed, all nodes but 0 have a weight of infinity signified, while node 0 has a weight of 0, is highlighted, and is marked with #0. If there is a counter for the state, it displays "1"  Please note, this is based on the behavior of bfs.alg from the original Galant project. If the algorithm is implemented differently, behavior may be different. Please verify that the behavior is consistent with the algorithm code. | Non yet |
| UC3\_Step\_Forward\_2 | Preconditions:   1. UC3\_Step\_Forward\_1 runs correctly   Steps:   1. Click the "Step Forward" button again. | - Node 1 is highlighted (along with the edge to it) and given a weight of 1 and a marking of #1.  - Node 2 is highlighted (along with the edge to it) and given a weight of 1 and a marking of #2.  - If there is a counter for the state, it displays "2". | None yet. |
| UC3\_Step\_Forward\_End | Preconditions:   1. UC3\_Step\_Forward\_2 runs correctly.   Steps:   1. Click "Step Forward" to the end, a total of six times. | - If there is a counter for the state, it displays "7".  - The "Step Forward" button is disabled and cannot be pressed again. | None yet. |
| UC3\_Runtime\_Error | Preconditions:   1. The webpage is loaded. 2. A valid graph is loaded in. 3. A breadth-first search algorithm is in the algorithm box.   Steps:   1. Change the algorithm such that it should throw a *Null Pointer Exception* in the first step when run. This can be done by changing an assignment of an object being used to null. 2. Attempt to run the algorithm as normal. | - When the button is pressed, nothing should change on the graph and an alert of the error should be displayed to the user. | None yet. |
| UC3\_Syntax\_Error | Preconditions:   1. The webpage is loaded. 2. A valid graph is loaded in. 3. A breadth-first search algorithm is in the algorithm box.   Steps:   1. Remove a semicolon (;) from the first line in the algorithm 2. Attempt to run the algorithm as normal. | - Before the "Step Forward" button can even be pressed, nothing should change on the graph and an alert of the error should be displayed to the user, indicating the first line has a syntax error. | None yet. |
| UC4\_Step\_Backwards | Preconditions:-   1. UC3\_Step\_Forwards\_1 has been run successfully.   Steps:   1. Click the "Step Backwards" button. | - The displayed graph returns to its original, un-highlighted form. If there is a counter for the state, it displays "1". | None yet. |
| UC4\_Step\_Back\_Then\_Forwards | Preconditions:   1. UC4\_Step\_Backwards has been runs correctly.   Steps:   1. Click the "Step Forward" button again. | - Everything visually should look the same as it was after the preconditions for UC7\_Step\_Backwards. That is, the expected results are the same as those for UC6\_Step\_Forwards\_1 | None yet. |
| UC4\_Step\_Backwards\_Beginning | Preconditions:   1. UC4\_Step\_Backwards has been run successfully.   Steps:   1. Attempt to click the "Step Backwards" button. | - Because the program is at the beginning of the algorithm, it should be impossible to click the "Step Back" button. It should be disabled. | None yet. |
| UC5\_Valid\_Graph\_Change | Preconditions:   1. A user is able to access the web-based-galant webpage 2. The user has loaded Bfs.alg into the algorithm 3. The user has loaded Test\_graph.txt into the graph, which contains the following text regarding nodes and edges:   “n 5 556 289  n 6 283 269  n 7 192 108  n 8 42 244  n 9 193 368  e 5 6  e 5 7  e 6 7  e 7 8  e 7 9  e 8 9”  Steps:   1. The user steps through the algorithm by clicking the forward button, until all lines of the algorithm have finished and the forward button is grayed out | The algorithm starts with the first rendered node (node with id = 0), and continues the bfs algorithm as needed until the step forward button cannot be clicked. The weights of the edges are displayed on the graph as the following (the position of the points are inaccurate in this picture, due to to the window size): | None yet |
| UC5\_Invalid\_Backward\_Click | Preconditions:   1. A user is able to access the web-based-galant webpage 2. The user has loaded Bfs.alg into the algorithm 3. The user has loaded Test\_graph.txt into the graph, which contains the following text regarding nodes and edges:   “n 5 556 289  n 6 283 269  n 7 192 108  n 8 42 244  n 9 193 368  e 5 6  e 5 7  e 6 7  e 7 8  e 7 9  e 8 9”  Steps:   1. The user clicks the back button to see if anything changes | The back button is grayed out, indicating that it cannot be clicked. When it is clicked, nothing changes for the algorithm or graph. | None yet |
| UC5\_Invalid\_Forward\_Click | Preconditions:   1. A user is able to access the web-based-galant webpage 2. The user has loaded Bfs.alg into the algorithm 3. The user has loaded Test\_graph.txt into the graph   Steps:   1. The user steps through the algorithm by clicking the forward button, until all lines of the algorithm have finished and the forward button is grayed out 2. The user then attempts to click the forward button again | The algorithm and graph have successfully changed as expected. The forward button is now grayed out. Once it is clicked in this state, nothing on the graph or algorithm changes. | None yet |
| UC5\_Error\_During\_Algorithm | Preconditions:   1. A user is able to access the web-based-galant webpage 2. The user has loaded nothing.alg into the algorithm, which is a blank .alg file. 3. The user has loaded Test\_graph.txt into the graph, which contains the following text regarding nodes and edges:   “n 5 556 289  n 6 283 269  n 7 192 108  n 8 42 244  n 9 193 368  e 5 6  e 5 7  e 6 7  e 7 8  e 7 9  e 8 9”  Steps:   1. The user steps forward into the algorithm by pressing the forward key. | The user is met with an error stating that the program cannot find where to begin parsing the algorithm, and the graph state stays the same. | None yet |

There is also additional testing done via the web page. A GraphViewer test component has been created to test out the functionality of our GraphViewer which is responsible for displaying graphs. This page has preset graph elements embedded in the page and when a user selects a test itt will display the graph to the user and has the expected results of these tests beside it. This allows for easier and faster testing. Going forward we might include our system tests as components to our project so that users can go to different test pages to test out different features. A look at this webpage can be seen here:



Eventually we hope to create more of these pages to test different components and help create an easier way to test our acceptance tests. Currently, to run these tests we have to directly change which component App is utilizing, but in the future this could be dedicated pages on the webpage for ease of use.

# Acceptance Test Results

The first two tests in our system test are the only ones that partially pass. As we have not implemented the full frontend it cannot possibly pass, but for what we have it is expected and within reasonable results. The second test, UC1\_Upload\_Graph\_1, has some visual errors that will need to be resolved, but the functionality is correct. The last two tests that we ran, UC1\_Upload\_Error\_Graph\_1 and UC1\_Upload\_Error\_Format\_1, are currently not running correctly. These are errors that we have not resolved yet, so they do not meet the expected results currently or for the overall project.

The GraphViewer web page tests are working correctly with all tests passing.

Our project meets most of the requirements we set for our acceptance tests for this iteration of our project. Some of our tests are failing currently that shouldn’t be failing. Knowing this could be from the backend or frontend we should be doing more thorough testing and currently for this iteration we just started, we have someone dedicated to fixing these errors.

# Task Plan Author(s): Noah Alexander Reviewer(s)/Editor(s):

For this project we are tasked with taking an already existing Java project and creating a web-based application. The reason for this is to flesh out existing bugs and also make it more user friendly. Currently, to use Galant one has to have the proper tools downloaded and can seem confusing to debug. With a web-based application this will require zero local tools to be downloaded and all be run through the browser. Our running tasks for this project will be constructed based on the iteration. Each iteration will consist of a design, implementation, and testing phase. The tasks are listed below in the table.

Team Member Information:

Noah Alexander: [ngalexa2@ncsu.edu](mailto:ngalexa2@ncsu.edu)

Rishabh Karwa: [rskarwa@ncsu.edu](mailto:rskarwa@ncsu.edu)

Art Schell: [alschell@ncsu.edu](mailto:alschell@ncsu.edu)

Yuval Sherman: [ysherma@ncsu.edu](mailto:ysherman@ncsu.edu)

Andrew Watson: [awatson5@ncsu.edu](mailto:awatson5@ncsu.edu)

| Iteration | Tasks | Assigned | Due Date | Status |
| --- | --- | --- | --- | --- |
| Iteration I | Create Requirements, Design “backend”, Create System-Tests.  Team Roles:   * Team Leader: [Yuval Sherman](mailto:ysherma@ncsu.edu) * Technical Lead: [Andrew Watson](mailto:awatson5@ncsu.edu) * Team Member: [Rishabh Karwa](mailto:rskarwa@ncsu.edu), [Noah Alexander](mailto:ngalexa2@ncsu.edu), Art Schell |  | 1/27 | Finished 1/27 |
|  | Create Requirements: | Entire team | 1/27 | Finished:  1/27 |
|  | * Meet with the sponsor to understand what is needed for the project. | Entire time | 1/17 | Finished:  1/17 |
|  | * Create a traditional set of requirements. | Entire team | 1/19 | Finished:  1/19 |
|  | * + Create use cases from those requirements. | Entire team | 1/26 | Finished 1/26 |
|  | Design “backend”:   * High-end design for the core components. | Entire team | 1/27 | Finished 1/26 |
|  | Create System-Test:   * Create system-tests for each use case. Each use case should have an acceptance and error test at the minimum. | Entire team | 1/27 | Finished 1/27 |
| Iteration II | Design low-level and frontend, implement frontend html webpage, implement graph parser and predicate logic  Team Roles:   * Team Leader: [Rishabh Karwa](mailto:rskarwa@ncsu.edu) * Technical Lead: [Noah Alexander](mailto:ngalexa2@ncsu.edu) * Team Member: Art Schell, Yuval Sherman, [Andrew Watson](mailto:awatson5@ncsu.edu) | Entire team | 2/17 | Finished: 2/17 |
|  | Design: | Entire team | 2/6 | Finished: 2/6 |
|  | * Frontend wireframe of html pages | Yuval | 2/3 | Finished: 2/3 |
|  | * Low-level design of core components |  | 2/3 | Finished: 2/3 |
|  | Implementation | Entire team | 2/17 | Finished: 2/21 |
|  | * Create an html page for Galant website with core components. e.g. section for graph, section for algorithms, buttons for uploading graph and algorithm, etc. | Yuval and Andrew | 2/10 | Finished: 2/10 |
|  | * Create JS file to parse graph files | Rishabh | 2/14 | Finished: 2/14 |
|  | * Create JS file to convert graph “object” into predicates | Noah | 2/14 | Finished: 2/15 |
|  | * Create a file to display the graph to the user | Art | 2/14 | Finished: 2/14 |
|  | * Put all components together |  | 2/17 | Finished: 2/21 |
|  | Testing | Entire team | 2/17 | Finished: 2/17 |
|  | * Test html pages to make sure it is properly displayed in all appropriate web browsers | Yuval and Andrew | 2/17 | Finished: 2/17 |
|  | * Create unit tests to test the graph parser | Rishabh | 2/17 | Finished: 2/16 |
|  | * Createunit tests to test the conversion from graph “object” to predicate logic | Noah | 2/17 | Finished: 2/17 |
|  | * Create tests to test to make sure the graph is being displayed properly based on the cytoscape graph format and our styles | Art | 2/17 | Finished: 2/16 |
| Iteration III | Design a high level diagram for adding algorithms to manipulate the graph. Implement the algorithms such that it can be run with a simple graph and display properly on the screen. All files should be tested.  Team Roles:   * Team Leader: [Rishabh Karwa](mailto:rskarwa@ncsu.edu) * Technical Lead: [Noah Alexander](mailto:ngalexa2@ncsu.edu) * Team Member: Art Schell, Yuval Sherman, [Andrew Watson](mailto:awatson5@ncsu.edu) | Entire team | 3/10 | In Progress |
|  | Design: | Entire team | 2/23 | Finished: 2/23 |
|  | * Design functionality of algorithms at a low level. (components, flows, etc.) | Entire team | 2/23 | Finished: 2/23 |
|  | Implementation: | Entire team | 3/5 | In progress |
|  | * Implement API and Thread java classes as well as the ThreadHandler component | Andrew, Noah | 3/5 | in progress |
|  | * Implement fronted display and backend for algorithm input via file upload and console frontend display | Rishabh | 3/5 | In progress |
|  | * Implement the backend steps when a users step forward or backwards in the algorithm. | Art | 3/5 | In progress |
|  | * Fix bugs from previous iteration   + Only text files should be allowed   + Graph should be display properly, in the center and with enough spacing between nodes   + Errors should stop execution of code and alert properly | Yuval | 3/5 | In progress |
|  | Testing: | Entire team | 3/10 | To do |
|  | Test API and Thread java classes as well as the ThreadHandler component with unit tests | Andrew, Noah, Yuval | 3/10 | To do |
|  | * Implement fronted display and backend for algorithm input via file upload and console frontend display | Rishabh, Yuval | 3/10 | To do |
|  | * Implement the backend steps when a user steps forward or backwards in the algorithm. | Art, Yuval | 3/10 | To do |