Developer Guide

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

CSC 492 Team 37

***Sponsor: Dr. Matthas Stallmann***

North Carolina State University

Department of Computer Science

4/28/23

# 

# **Contents**

[**Contents 2**](#_wbvva8qw0b6p)

[**Overview 3**](#_h8ffrgmtkmwx)

[**Design 4**](#_ailzemrqw6tm)

[High-Level 4](#_ruix5f3tj21j)

[Low-Level 5](#_ll3cbyl5sqa2)

[Frontend 5](#_oe6s0k98ash4)

[Backend 5](#_4dmj415pu6ju)

[**Frontend 6**](#_dvysuudaoap7)

[**Graph 6**](#_guhfb5y99huo)

[GraphContext 6](#_aechkau38nn)

[GraphInput 6](#_bjja0oyth0y)

[GraphViewer 7](#_x7nsyp9h1t1i)

[**Algorithm 7**](#_qfhkw466odmy)

[AlgorithmConsole 7](#_v5t061r8yl98)

[AlgorithmControls 8](#_m08wvl1z32fa)

[AlgorithmInput 8](#_a0t3hhh9zhqk)

[**Backend 9**](#_u5kxv77tg7va)

[**Algorithm 9**](#_rd9vz9298s0d)

[AlgorithmHandler 9](#_c7dj2palqgao)

[StepHandler 9](#_b39ahnpw2uo5)

[ThreadHandler 10](#_lz5ku71o6ot2)

[Thread 11](#_868o4zcn6ibo)

[**Tests 13**](#_9wpoihrteklg)

[/src/\_\_tests\_\_/ 13](#_b70ginqfjcx5)

[test\_AlgorithmHandler.js 13](#_dzqv77boblsa)

[test\_Converter.js 13](#_wekxdowy5c0p)

[test\_FileToPredicate.js 14](#_fumhmf22gml3)

[test\_InputGraph.js 14](#_mmq5jbb0r706)

[test\_StepHandler.js 14](#_ap7l28ilyamc)

[/src/frontend\_tests/ 14](#_7m5i80jyaue6)

[AlgorithmTest.jsx 15](#_5qo5pbtnlrq6)

[GraphInputTest.jsx 15](#_f8ss7424j757)

[GraphViewerTest.jsx 15](#_u4qr08ev8fro)

[PromptTest.jsx 15](#_wdgo9kgql314)

[StepTest.jsx 15](#_pt0w4ojtr3mq)

[**Future Work 16**](#_bb4pp86vtxyx)

[Additional Features 16](#_clrpzalhd6hu)

[More API Methods 16](#_bgto8ae45das)

[Preferences 16](#_hzlo0f1mdfmn)

[Error Handling 16](#_mt9wrp8jo6kk)

[Performance 16](#_3oxqneodc0y2)

# **Overview**

The development of Web-Based Galant was built using React.js as a framework with many different technologies. The actual web page is hosted through NC State’s Virtual Machines. The structure of our project is a basic React.js file structure with all of our code inside of the src folder. All images and files that a user will see or use is hosted in the public folder. The files themselves are mostly for testing purposes for our testing page, which will be covered further in this document.

The src folder itself holds both of our “backend” and frontend code. While we reference code as backend, no user input is ever sent to the server to be processed. All files are hosted through the client; all the server does is send those files. The backend folder holds all files that process the user input that is not shown to the user. The frontend folder holds all files that generate html for the user to see.

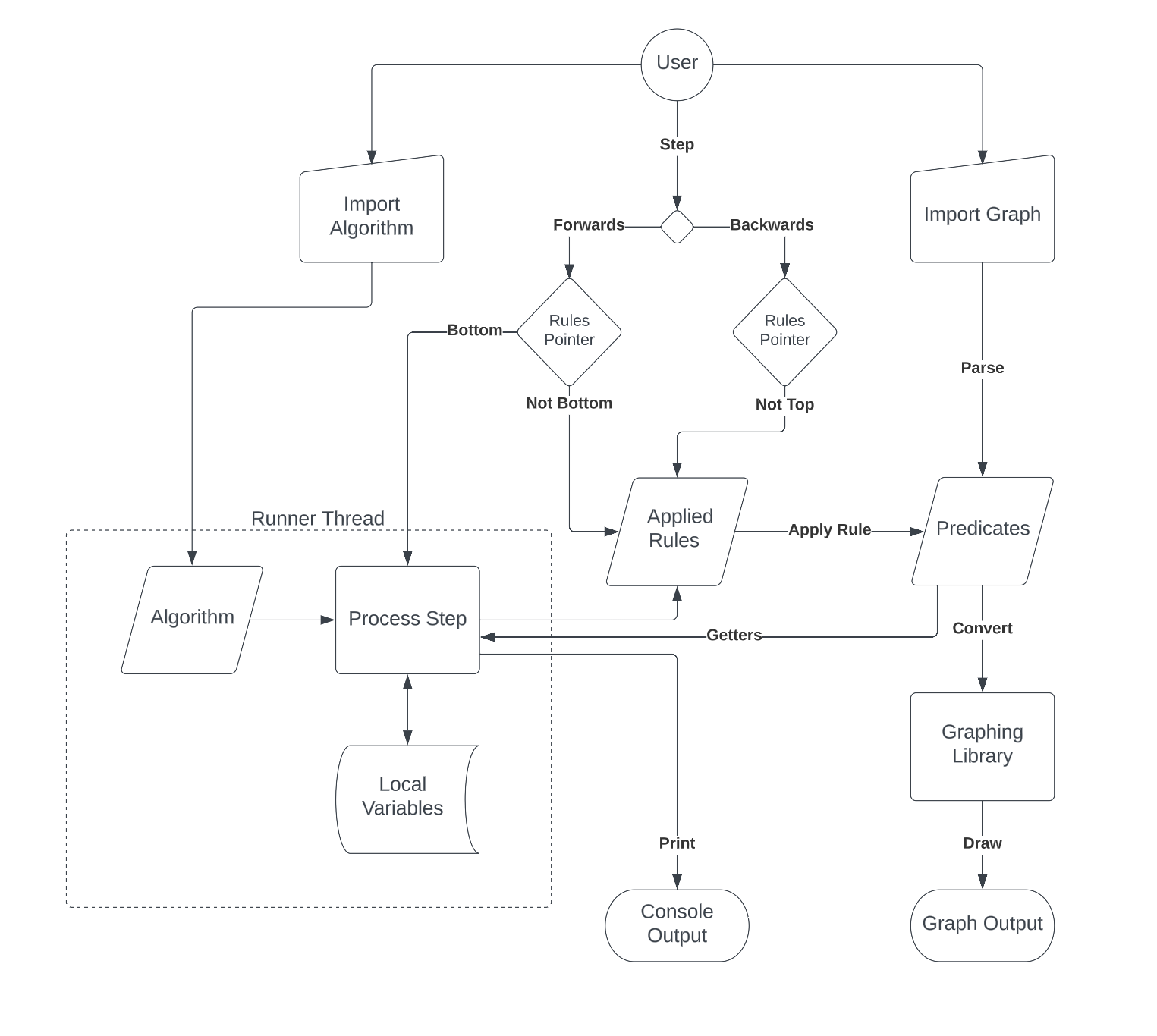
This project used GitHub for version control.

Technologies Used

| Technology | Purpose | Version | Licensing Info |
| --- | --- | --- | --- |
| 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. | v18.2.0 | MIT |
| Cytoscape.js | Package to display the graph and its animation to the user. | 2.0.0 | MIT |
| Immer | Patching Predicates | v9.0.21 | MIT |
| Jest | Unit Testing | v24 | MIT |
| react-router | Client-side Routing | v6 | MIT |

# **Design**

## High-Level



## Low-Level

### Frontend

### Backend

# 

# **Frontend**

# **Graph**

## GraphContext

The GraphContext.jsx file is one of the most important parts of the frontend. It allows all frontend components to share access to the same graph, and manages the updating of that graph. In React, to create a GraphContext, import GraphProvider from the GraphContext file and surround all the frontend components with it.

<GraphProvider>

<Algorithm />

<Graph />

</GraphProvider>

To use the graph in a component, use the React useContext() function like so:  
let [graph, //The current value of the graph

startGraph, //The starting value of the graph, before the algorithm began

loadGraph, //A function which loads a new graph

updateGraph, //A function that updates the graph with a list of patches.

registerOnLoad, //A function which registers a function to be called whenever a new graph is loaded. The registered function will not be called when the graph changes from an algorithm, and so is useful for initialization behavior like resetting the graph camera or restarting the algorithm.

] = useContext(GraphContext);

## GraphInput

The GraphInput.jsx file returns an “Upload Graph” button which, when clicked, opens the file system and allows the user to pick a graph file to be uploaded. The text from the file is parsed into a predicate presentation of the graph, with the line:

predicates = parseText(file);

The predicates are passed to the GraphContext.jsx file, which then calls the GraphViewer.jsx to load the graph onto the screen, by using a prop:

props.onUpload(predicates);

where, onUpload() with the function loadGraph passed to it in Graph.jsx:

<GraphInput onUpload={loadGraph}/>

loadGraph is declared in GraphContext.jsx, and used in GraphViewer.jsx to change the graph.

The graph file is also displayed in the text area below the “Upload Graph” button.

## GraphViewer

The GraphViewer.jsx file returns the Cytoscape Component to display the graph to the user. This file also contains a Cystoscape Stylesheet to change the way in which nodes and edges are displayed to the user. It also includes buttons to change the layout of the graph (“Auto-layout”), buttons to zoom into the graph (“Auto-camera”), and buttons to hide/show node weights/labels and edge weights/labels. These buttons call PredicateConverter.jsx to dynamically update the predicates for the graph. This is done with the code:

useEffect(() => {

...

let newElements = predicateConverter(graph, nodeWeights, nodeLabels, edgeWeights, edgeLabels);

...

}, [graph, nodeWeights, nodeLabels, edgeWeights, edgeLabels])

useEffect() is called whenever graph, nodeWeights, nodeLabels, edgeWeights, or edgeLabels change in value.

The button that is used to change edges between directed and undirected, resets the algorithm, by calling GraphContext.jsx to set the start graph to the new graph.

This is done through the shared React Context between GraphContext.jsx and GraphViewer.jsx.

const [graph, startGraph, loadGraph, updateGraph, registerOnLoad] = useContext(GraphContext);

# **Algorithm**

## AlgorithmConsole

The AlgorithmConsole.jsx file returns a console that is dynamically updated with messages or errors while running the algorithm. The Algorithm.jsx file passes new messages to the AlorithmConsole, which appends it to the list of messages in the console. This is done through:

props.onSetupConsole(addNewMessage);

onSetupConsole() is used in Algorithm.jsx as a property passed to AlgorithmHandler to send messages to AlgorithmConsole.

## AlgorithmControls

The AlgorithmControls.jsx file is responsible for displaying the front and back arrows, which when clicked, are passed to Algorithm.jsx, then to AlgorithmHandler.js, then to StepHandler.js to check if the Algorithm can move forward or backward, and then updating the graph as necessary for the changes. The AlgorithmControls.jsx file also uses the useEffect dependency to see if the right or left arrow keys are held down, which will have the same effect as clicking the forward and backward arrow. This is done through the code:

useEffect(() => {

document.addEventListener('keydown', handleKeyPress, true)}, [])

## AlgorithmInput

The AlgorithmInput.jsx file returns an “Upload Algorithm” button which, when clicked, opens the file system and allows the user to pick an algorithm file to be uploaded. The text from the file is passed to the Algorithm.jsx file which dynamically updates the AlgorithmHandler component, which updates its constructor with the new algorithm. The algorithm file is also displayed in the text area below the “Upload Algorithm” button.

The following line:

props.onUpload(file);

passes the onUpload() function to AlgorithmInput.jsx as a prop, where is responsible for for helping set the current Algorithm through the following code in Algorithm.jsx:

<AlgorithmInput onUpload={(alg) => {

algHandler.setAlgorithm(alg);

setAlgorithm(alg); }}/>

# **Backend**

# **Algorithm**

## AlgorithmHandler

The AlgorithmHandler is the starting point for the Algorithm backend. It can be stepped forward and back, and manages the messages that the ThreadHandler creates via several callback functions. It also ensures the algorithm is timed out if it runs for too long.

An algorithm handler is created using a constructor.

let handler = new AlgorithmHandler(

graph //The value of the graph

algorithm //The value of the algorithm

updateGraph //A function that updates the graph with a list of patches

onMessage, //A function that is called when a console message is printed

onStatusChanged, //A function that is called when the status is changed. The status consists of displayState and algorithmState (step counter between the buttons), canStepForward and canStepBack (which buttons should be clickable)

setAlgError, //A function that is called when an error occurs.

setAlgPrompt, //A function that is called when a prompt occurs.

testFlag, //When present, runs the handler in “test mode”, removing Workers.

);

To change the graph or algorithm, use the setGraph(graph)and setAlgorithm(algorithm) methods. To use the algorithm, use the stepForward() and stepBack() methods. Updates will be reported back via the callback functions provided when the handler was created.

If a prompt occurs, the algorithm will remain paused until a prompt result is provided. Provide this by calling the enterPromptResult(promptResult) method.

## StepHandler

The StepHandler controls the application and reversion of rules. It stores a list of steps to be executed or undone each time the stepForward()or stepBack()methods are called. Each step consists of a list of individual rules generated by the algorithm.

The algorithm controls the StepHandler by calling several methods. The addRule(rule)method adds a singular rule to the current step, and the completeStep()method completes the current step, executing it if possible.

The output of the StepHandler takes the form of two callback functions passed in on creation. The updateGraph(patches)callback is called each time the graph is to be updated with a list of patches. The onNeedsRules()callback is called each time the algorithm must be unpaused to generate new rules (that is, whenever the graph is updated to the most recent step and must be stepped forward again.)

Rules are represented as an object containing both an apply and revert component, which each contain a list of Patches generated by Immer. These represent individual changes to the Graph object. When a step is applied or reverted, it will gather a list of each patch in the relevant direction and pass this list to the updateGraph callback function. Rule generation by Immer is handled in the Predicates class.

### **ThreadHandler**

This file inside the Algorithm folder is responsible for the creation of Threads, starting and stopping Threads, communication between the Thread, itself, and the AlgorithmHandler, and finally the deletion of the Thread. This is done by creating a class with methods for all of its responsibilities.

ThreadHandler fields:

graph //Cytoscape representation of the graph

algorithm //User derived algorithm

array //Shared array for ThreadHandler and Thread

worker //Worker to create thread for algorithm

onMessage //Function to send message to AlgorithmHandler

ThreadHandler methods:

startThread() //Starts a worker that creates a thread using Thread.js

resumeThread() //Resumes the users algorithm from the thread

killThread() //Kills the currently running work thread

enterPromptResult() //Sends the result of a prompt to the currently running

//worker thread

### **Thread**

The Thread file is where the user's algorithm is actually run and where they can access all of our API. To begin the code starts with a function to receive messages from the ThreadHandler. These beginning messages will give the Thread the shared array, a copy of the graph, and the user’s algorithm. Once the user’s algorithm has been given to the Thread it will then start to run it until it changes some state of the graph and sends a message back to the ThreadHandler.

self.onmessage() //Receives variables to initialize the Thread

Once the user’s algorithm gets run they have access to our built in API that allows them to access our graph. For a full list of our API and how they work, check out the Programmer guide. These API functions have built in methods that allow them to stop the execution of the code and return a message to the ThreadHandler, which will in turn give it to the StepHandler. This is done through the methods wait() and postMessage().

wait()

This method stops the execution of the algorithm by using Atomics. Atomics allows the Thread to continuously look at an index in a shared array until it has been changed. So this method stores a value at a specific index and then waits until that value has been changed.

postMessage()

This method allows the Thread to send a message back to the ThreadHandler. The contents of the message is json with a type and optionally a content. Here is a list of all message options:

{type: "prompt", content: [message, error]} //Prompts the user with a message and also an error message should an error arise

{type: "console", content: message} //Prints the message to the console

{type: "rule", content: rule} //Sends a rule to the ThreadHandler that has changed the graph in some way

At the end of each API call that changes the graph’s state there is an autostep method. This method automatically calls the wait method every time, unless the user has specified that autostep be disabled. If so they will call the step() method that will then run any user code that is within that method and then call wait to stop the execution of the algorithm.

Some of our API calls are not inside of the Thread file, but inside of Graph.js which is a class that holds the graph and holds the methods that manipulate it or return information about it in some way. The way that a user calls these methods without having the Graph object is still through our API, but in turn these API calls generateGetters() and generateSetters().

generateSetters(name)

This method takes a name as a parameter that then calls a method inside of the Graph object that corresponds with that name. This reduces the length of the Thread file and makes it easier to change the state of the graph since they will all be in one place.

generateGetters(name)

This method takes a name as a parameter that then calls a method inside of the Graph object that corresponds with that name. This will retrieve some part of the graph object that the user can then use in their algorithm.

***Note: These methods are not used by the user, but are used to create the link between their API calls and the methods inside of the Graph object.***

# **Tests**

There are two styles of tests present in the codebase, automated unit tests and black box acceptance tests. The automated testing suites are built using Jest and are intended to coverage test the backend functionality of the application, including things like uploading and converting user-created graphs into a predicate form, converting predicates into a Cytoscape.js form, and starting and stepping through an algorithm. The acceptance tests are targeted at the frontend and rely on a tester to verify information visually. This includes things like uploading a graph and seeing how it looks, pressing buttons on the page and verifying the expected visual feedback, and stepping through algorithms. These acceptance tests are available to a tester via the /tests url.

## /src/\_\_tests\_\_/

### test\_AlgorithmHandler.js

Primarily tests src/backend/Algorithm/AlgorithmHandler.js. It defines 4 tests which check that the AlgorithmHandler can be instantiated, initialized with a graph and an algorithm, and stepped through. These tests also test error cases. The graphs which are passed to the algorithm are created manually in these tests, which means that if there are changes to how graphs look or how certain graph attributes are handled, these tests will need to be updated accordingly.

An important thing to note is that when creating objects in Jest unit tests, you will not have access to React context that may be present during the actual execution of your code. AlgorithmHandler expects to be passed some functions that are typically handled by React context, but in this testing environment we have to define our own. The functions updateGraph, addNewMessage, setStatus, setAlgError, setAlgPrompt are defined and mostly just used as setters for local variables, which can be checked for expected value.

### test\_Converter.js

This file primarily tests src/backend/PredicateConverter.js. It consists of 10 tests which create predicate information with various attributes and tests that they are appropriately converted into a set of elements that are Cytoscape.js friendly. The predicate forms as well as the Cytoscape element forms are all manually created as JS objects and lists, so if any changes are made to how data appears or is handled in these forms, these tests will need to be updated.

The predicateConverter function takes 5 arguments but these tests only ever pass the predicates and null for everything else. The other 4 parameters control visibility of labels and weights on the page and are currently not covered.

### test\_FileToPredicate.js

Tests src/backend/FileToPredicate.js. This file creates text representations of graphs as well as graph object representations of the same graph (which matches predicate form). The text is passed to the parseText method from FileToPredicate.js and confirms that the attributes of the resulting graph match the expected values. Note that boolean attributes which are not present in the text representation will still be present in the object form and be set to false. The optional weight value for nodes and edges will also be present in the object form but set to ‘undefined’.

### test\_InputGraph.js

Tests the mechanisms for uploading a graph present in src/frontend/Graph/GraphInput/GraphInput.jsx. In doing so, the graph is also run through FileToPredicate.js. Note that these tests do not test for the actual visual aspects of the graph nor do they test whether or not the graph appears correctly. The extent of these tests is that valid graphs do not throw errors and their text appears on the page appropriately, while invalid graphs throw particular errors which appear on the page. These tests work for valid graphs by having a timeout in which they wait for an error to be thrown, and passing if no error is thrown in that time. These tests will take the longest due to this timeout, which can be adjusted but should remain reasonably long to avoid false-negatives.

### test\_StepHandler.js

Tests src/backend/Algorithm/StepHandler.js. This test creates two sets of predicates, an algorithm state and a display state. These two sets are initially equivalent. It then runs through and marks the nodes of the algorithm state, while generating rules that are passed to the StepHandler. The StepHandler is expected to update the display state based on these rules without affecting the algorithm state. Then the StepHandler gets stepped back and forward to see the appropriate state changes be applied/reapplied to the display state.

## /src/frontend\_tests/

All of these are frontend test pages whose purpose is to instruct a user on the page on how to acceptance test the various functionalities of the site. Some of the pages have links for the tester to download particular test files which will match the expected results detailed in the instructions. These test files must be in the /public folder on the project to be available for the user to download. This is unlike the backend test files that are only accessed by the unit tests and therefore hosted in the src/\_\_tests\_\_ folder. Links to these test pages should be added to the index.js file following the format that the existing tests follow.

### AlgorithmTest.jsx

This page describes how to test the algorithm functionality. It has the tester download a graph and a simple algorithm that simple marks all the nodes in the graph. The steps are all clear for the user regarding what they should do as well as what they should expect to happen. Three error algorithms are provided for the user at the end of the steps to see the error response from the site. The file adds the App react element so that both the algorithm and graph elements are present on the page.

### GraphInputTest.jsx

This test page provides both a valid graph file as well as has an image on the page on what the user should expect to see. This test is purely visual for the user. The error graph is expected to error out and not be uploaded. This file only adds the Graph react element since the algorithm portion of the page isn’t necessary.

### GraphViewerTest.jsx

This test page creates a collection of buttons for the user to press and see the results. This is intended to test how the graphs are displayed, not created or uploaded. Therefore, The only react element loaded is GraphViewer. The graphs are manually created when the buttons are pressed. If changes are made to how graph objects are created, these tests will need to be updated.

### PromptTest.jsx

This page describes how to test the algorithm prompt functionality. It has the tester download an algorithm that includes prompts. The steps are all clear for the user regarding what they should do as well as what they should expect to happen. Two test graphs are provided for the user to test the prompt algorithm on. The file adds the App react element so that both the algorithm and graph elements are present on the page.

### StepTest.jsx

This page describes how to test the custom step algorithm functionality. It has the tester download a graph and an algorithm that changes the step protocol as well as defines step functions. The steps are all clear for the user regarding what they should do as well as what they should expect to happen. The file adds the App react element so that both the algorithm and graph elements are present on the page.

# **Future Work**

## Additional Features

### More API Methods

While there are many API methods already created for the use of developing algorithms, there are more that can be added to aid the user. A couple of API methods we think could be useful are:

coordinates(node) //retrieves the x and y coordinates of a node

### Preferences

With default browser settings the website looks good and you can grow the graph by increasing the zoom, but if the text size changes based on browser settings then the whole page looks different. This should be changed so that a user’s preferences change the page such that it does not affect their experience. This could also be changed by adding a preference setting inside of the website itself to allow for text size change or to change the size of the graph, algorithm, console, and graph text. Another change could be to make each window collapsible and allow the others ones to grow based on which windows are collapsed.

### Error Handling

Currently the way errors are handled is based on the browsers stack trace when the errors happen. This is different on every browser. The solution could be to find which browser the user is using and then do error checking based on the browser.

Another error propagation problem is when uploading an algorithm and there are spaces between each identifier in a node or edge it can throw an error. Dr. Stallmann would like it so that any number of spaces can occur between each identifier in a node or edge.

n 1 5 10

n 1 5 10

These two lines should be treated as the same and not throw any errors.

### Performance

The current version of Galant creates two different kinds of labels. One label is in the center of the node, which cytoscape generates, and the other is an html label that we generate and put above each node. This causes a lot of lag when traversing the algorithm after execution, when holding down either forward or backward. Finding an alternative to this html label would help performance.

Another performance issue is when holding down the arrow keys after executing the algorithm results in the graph's updates being loaded in chunks. This is not the intended procedure when holding down the keys. What should happen is that each change should be noticeable and happen one after another, not all at once. The solution could possibly be to have a timer on when these updates occur so that they happen one at a time.