Developer’s Guide

Sponsor: Dr. Matthias Stallmann

CSC 492 Team 43

Christina Albores

Vitesh Kambara

Julian Madrigal

Neha Ramesh

Minghong Zou

North Carolina State University

Department of Computer Science

04/22/2024

# Table of Contents

[Table of Contents](#_j68v7sszsfi2)

[SharedWorker/worker.js: Explained](#_jm5lq4nbommp)

[Thread.js: Explained](#_dpmmiw1cyr16)

[Adding Algorithm Resources](#_8yzayueflqky)

[Algorithm.js: Explained](#_9t66jjvmgx9h)

[React Contexts: Explained](#_cjgyl6ni0lk3)

[Graph / Algorithm Examples](#_8du2a144cuuw)

[Adding a new Example](#_fwj8fkietd04)

[Keyboard Shortcuts](#_p4gw2cvdb4dh)

[Testing](#_hg0d0zz3gn79)

[Jest](#_66j25zpqaup)

[Running Tests](#_9kni0tmscm5f)

[Creating/Updating Tests](#_7htn6a9rqo0g)

[Cypress](#_2ljfp42d82c1)

[Running Tests](#_d1lxbs9x6bd8)

[Creating / Updating Tests](#_uhyad1mrnbs3)

[Combined Testing](#_rdzzrolge3qe)

[Running All Tests](#_4uscttinqv5t)

[Total Coverage](#_3jimt1xkxal4)

[Key Use Cases](#_88dz8gityynt)

[Loading a Graph](#_70he06iu2f34)

[Loading an Algorithm](#_164dfp3xv4jz)

[Future Work](#_hnd1dcwz3an7)

# 

# 

# 

# 

# SharedWorker/worker.js: Explained

SharedWorker acts as a central hub for communication among all browser windows. It is located at the following path: 2024SpringTeam43-Stallmann/public/worker.js.

In the worker.js file, each window is represented by a port, and the worker maintains a list of all connected ports. When a new window connects, its port is added to this list.

The event listener listens for incoming messages from the connected ports. If the message is of type **‘graph-init’** or **‘graph-rename’**, it updates the last message accordingly. The message **‘graph-init’** is used when initializing a new graph. The message **‘graph-rename’** is used for when the user saves changes in the graph view. The lastMessage is removed so the user is forced to load the graph in again.

If the incoming message is of type **‘alive’**, it updates the last message to have a type of **‘graph-init’** and then sends this modified message back to the specific port (window) from which the **‘alive’** message was received. This ensures that the last message is sent back to the window that requested it.

If the incoming message is not of type **‘alive’**, meaning it's a regular message from any window, the code iterates through all the connected ports (representing all windows) and sends the incoming message to each of them. This ensures that regular messages are broadcast to all connected windows.

# Thread.js: Explained

Thread.js is the file responsible for defining all the functions and resources available by the algorithm. At the very bottom of the file lies the code for loading and running the algorithm. Since Thread.js is a [web worker](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers), it listens for message **graph/algorithm** to load in the algorithm and graph. It defines the graph, and then immediately calls [eval()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval) on the algorithm code.

## Adding Algorithm Resources

In order to add a new function or resource available to the algorithm, define the function or resource above the **.onmessage** function. Many of the existing functions have the logic placed in **src/utils/Graph.js** as it can be considered a graph utility and be reused elsewhere. If your function is not a **Graph** utility, such as prompting a user for input, then the logic should remain in **Thread.js.**

# Algorithm.js: Explained

Algorithm.js is the class responsible for loading in an algorithm. When creating a new instance of an Algorithm, the constructor expects at least:

new Algorithm(algorithmName, algorithmCode, baseGraph, services, useState());



The algorithm will then:

1. Create an array of steps
2. Add Step 0 as a Snapshot of the base graph.
3. Initialize a StepBuilder
4. Initialize the Configuration
5. Initialize the ThreadHandler
6. Began listening for messages from ThreadHandler

The following message types are supported:

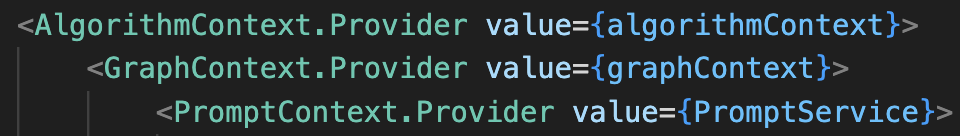
**Table 1: Message types**

|  |  |
| --- | --- |
| **Message Type** | **Purpose** |
| rule | **A Rule / Graph Modification has been made.**  Rule is added to the StepBuilder. |
| step | **A step has been defined.**  Compile previous rules in StepBuilder into a step. |
| error | **Algorithm experienced an error**  Display the error to the user |
| prompt | **Algorithm has requested input from the user**  Prompt user for response |
| config | **Algorithm has defined configuration options**  Apply options to AlgorithmConfiguration |
| complete | **Algorithm has completed**  Terminate the thread and update as completed |
| console | **Algorithm has requested to print a message**  Console.log() message to browser. |

By design, the algorithm will not generate steps until Algorithm.skipToEndStep() or Algorithm.stepForward() has been called. stepForward will then wait 5000ms to expect a response from the thread. If the thread fails to provide a message, then an error is assumed and shown to the user.

# React Contexts: Explained

Our GraphView page contains several React Contexts that allow its descendants access to each context. The use of React Contexts reduces the need for deep prop drilling to pass these values to other components. The graph contexts are included in **src/pages/GraphView/index.jsx**:

*Figure 1: React Context Providers*

Here are some explanations of what each context achieves:

**Table 2: React Context**

|  |  |
| --- | --- |
| **Context** | **Purpose** |
| **AlgorithmContext**  src/pages/GraphView/utils/AlgorithmContext.js | **Allows descendants to access algorithm related resources:**  algorithm  setAlgorithm() |
| **GraphContext**  src/pages/GraphView/utils/GraphContext.js | **Allows descendants to access graph related resources:**  graph / setGraph  baseGraph / setBaseGraph  preferences.style  preferences.setStylePreferences  preferences.layout  preferences.setLayoutPreferences  preferences.cytoscape.instance  preferences.cytoscape.setInstance |
| **PromptContext**  src/pages/GraphView/utils/PromptService.js | **Allows descendants to display a prompt:**  addPrompt |

# Graph / Algorithm Examples

Examples are stored in JSON files as an array of objects with the following structure

 {

"name": "Algorithm/Graph Name",

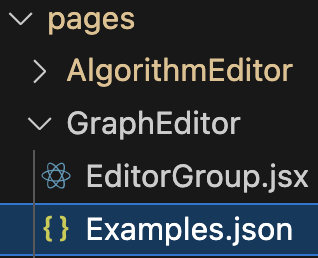
"content": "JSON.stringify(content)"

}



The Algorithm examples can be found at **src/pages/AlgorithmEditor/Examples.json**

The graph examples can be found at **src/pages/GraphEditor/Examples.json**

*Figure 2: Algorithm Examples Figure 3: Graph Examples*

## Adding a new Example

In order to add a new example, **the content must be stringified**. An online tool such as [JSON Stringify](https://jsonformatter.org/json-stringify-online) can be used to easily achieve this. The resulting string can then be used as the content.

# Keyboard Shortcuts

**Table 3: Keyboard Shortcuts**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keyboard Shortcut** | **Action** | **Description** | **File Path** | **Line Number** |
| A | Opening Editors | Open Algorithm Editor from Main Page | src/pages/GraphView/HeaderComponent.jsx | 31 |
| G | Opening Editors | Open Graph Editor from Main Page | src/pages/GraphView/HeaderComponent.jsx | 31 |
| Cmd-S  Ctrl-S | In-Editor Controls  The Cmd/Ctrl shortcuts do not work as expected. | Download Current Graph/Algorithm File in the Editor | src/components/Editor/EditorOverlay.jsx | 34 |
| Cmd-L  Ctrl-L | In-Editor Controls | Load Current Graph/Algorithm File in the Editor | 1. (Loading in Graph Files) src/pages/GraphEditor/EditorGroup.jsx 2. (Loading in Algorithm Files) src/pages/AlgorithmEditor/EditorGroup.jsx | 1. 33 2. 26 |
| Cmd-N  Ctrl-N | In-Editor Controls | Choose New Graph/Algorithm File in the Editor | src/components/Editor/NewButtonComponent.jsx | 28 |
| Z | Edit Mode | Undo Changes | src/pages/GraphView/GraphEditOverlay/GraphEditOverlay.jsx | 57 |
| Y | Edit Mode | Redo Changes | src/pages/GraphView/GraphEditOverlay/GraphEditOverlay.jsx | 57 |
| R | Edit Mode | Revert Graph back to original state | src/pages/GraphView/GraphEditOverlay/GraphEditOverlay.jsx | 57 |
| S | Edit Mode | Save Changes | src/pages/GraphView/GraphEditOverlay/GraphEditOverlay.jsx | 57 |
| Left Arrow | Algorithm Controls | Step Back | src/pages/GraphView/GraphOverlay/AlgorithmControls/AlgorithmControls.jsx | 88 |
| Right Arrow | Algorithm Controls | Step Forward | src/pages/GraphView/GraphOverlay/AlgorithmControls/AlgorithmControls.jsx | 90 |
| X | Algorithm Controls | Terminate Algorithm | src/pages/GraphView/GraphOverlay/AlgorithmControls/AlgorithmControls.jsx | 92 |
| S | Algorithm Controls | Export Graph | src/pages/GraphView/GraphOverlay/AlgorithmControls/AlgorithmControls.jsx | 94 |
| Cmd/Win + Right Arrow | Algorithm Controls | Skip to the End of the Algorithm | src/pages/GraphView/GraphOverlay/AlgorithmControls/AlgorithmControls.jsx | 96 |
| C | Control Settings Popover | Open popover for Layout Controls | src/pages/GraphView/GraphOverlay/GraphControls/Components/ControlSettingsPopover.jsx | 17 |
| N | Node Settings Popover | Open popover button for Node Settings | src/pages/GraphView/GraphOverlay/GraphControls/Components/NodeSettingsPopover.jsx | 48 |
| E | Edge Settings Popover | Open popover button for Edge Settings | src/pages/GraphView/GraphOverlay/GraphControls/Components/EdgeSettingsPopover.jsx | 47 |
| P | Preference Popover | Open popover button for Preference Panel | src/pages/GraphView/GraphOverlay/GraphControls/Components/PreferencesPopover.jsx | 16 |
| H | Help | Open help button | src/pages/GraphView/HeaderComponent.jsx | 44 |

# 

# Testing

GalantJS uses [Jest](https://jestjs.io/) for unit testing and recently implemented [Cypress](https://www.cypress.io/) for automated testing. The package.json of the project contains all the scripts that can be called from the terminal using:

npm run [scriptName]





*Figure 4: NPM Scripts*

Many of the tests, however, are not meant to be called directly but are automatically called based on the invocation of other scripts. For example, the pretest and posttest run before and after the execution of **npm run test.**

## Jest

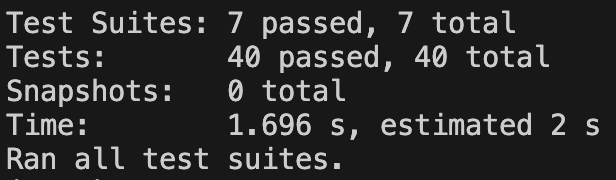
### Running Tests

Jest is used for unit testing, and its tests can be run independently by executing commands.

npm run test:jest



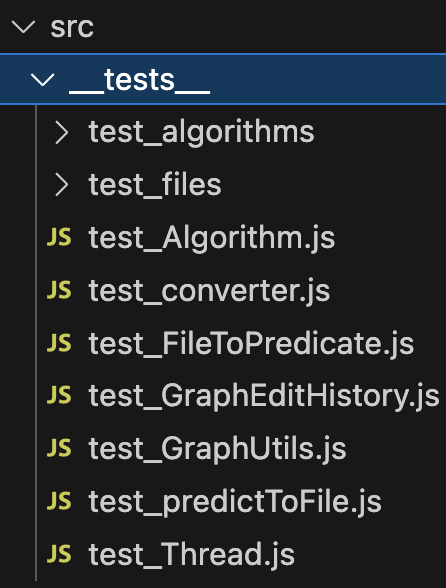
This will run your tests and output the number of passing test suites:



*Figure 5: Example Jest Test Results*

Furthermore, it will also create a coverage directory ‘**coverage’** in the root directory containing the coverage from **only the jest unit tests:**

### Creating/Updating Tests

All of the unit tests for Jest are located in the directory **‘src/\_\_tests\_\_’.** Any .js file created under this directory will be executed during the test command. For more information about how to write Jest tests, refer to existing tests and [Jest Documentation](https://jestjs.io/docs/getting-started).

*Figure 6: src/\_\_tests\_\_ directory*

## Cypress

### Running Tests

Running Cypress is a bit less straightforward compared to running Jest. Firstly, it requires [instrumentation of the code to enable code coverage](https://docs.cypress.io/guides/tooling/code-coverage). Next, it requires that the project be running and accessible on port 3000. There are a few different ways to run cypress as well. For example, ‘**npx cypress run’** runs the tests in the CLI. Meanwhile ‘**npx cypress open’** opens a GUI that allows you to run and visualize individual tests and debug issues.

In most scenarios, you’re more interested in running Cypress and generating reports and coverage. In this instance, you can simply run the command:

npm run test:cy

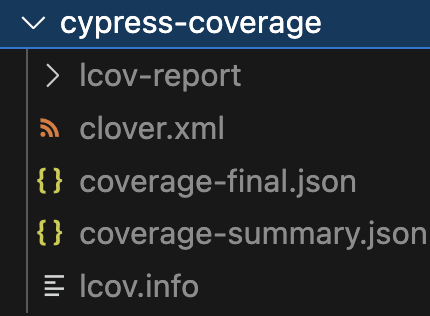


This will:

1. Instrument your code via **npm run instrument**
2. Run **cypress run**
3. Kill the process after **cypress run**

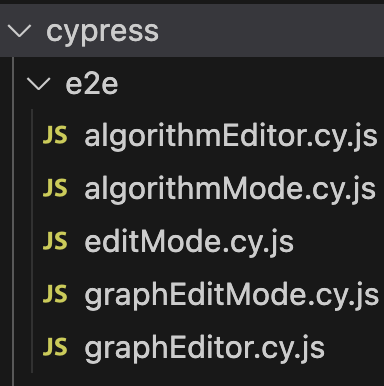
**NOTE: This may fail to kill the process and may require manually killing the process afterward.**

This will also create your cypress coverage under the directory cypress-coverage

*Figure 7: Cypress coverage directory*

**For more information about what cypress commands and options can be used, refer** [**here**](https://docs.cypress.io/guides/guides/command-line#Commands)**.**

### Creating / Updating Tests

All of the cypress tests are found under the directory **‘cypress/e2e’** and end with extension .cy.js. For information on creating tests, refer to existing tests or [cypress documentation.](https://docs.cypress.io/guides/end-to-end-testing/writing-your-first-end-to-end-test)

*Figure 8: cypress/e2e directory*

## Combined Testing

### Running All Tests

In situations where you are interested in running *all* tests, this can be achieved by running:

npm run test

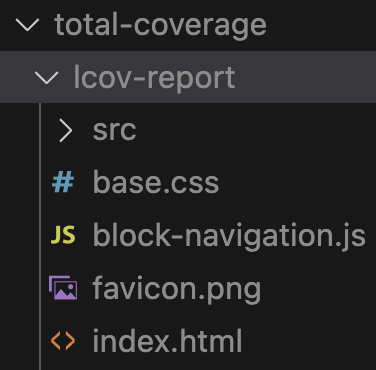


This script does the following:

1. Executes **npm run test:jest**
2. Executes **npm run test:cy**
3. Combines coverage from **/coverage** and /**cypress-coverage**
4. Create a new coverage report under directory **/total-coverage**

### Total Coverage

Results are found under the **/total-coverage** directory.

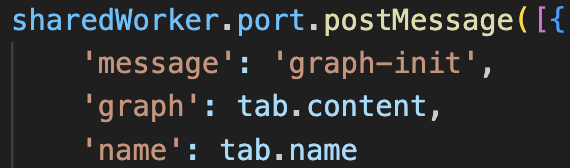
*Figure 9: total-coverage directory*

# 

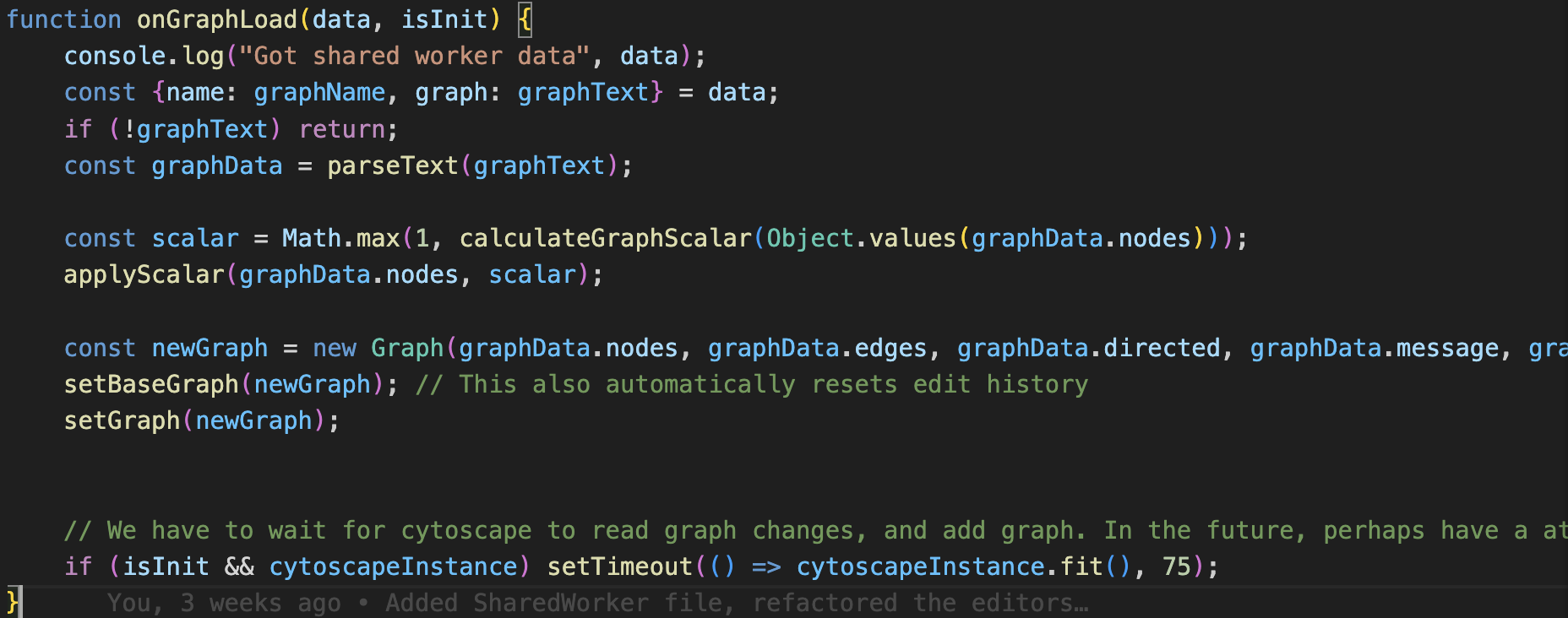
# Key Use Cases

## Loading a Graph

Loading a graph begins in the editor in the file **src/pages/GraphEditor/EditorGroup.jsx**. When a user requests a graph to be loaded, the function **loadGraph** is run. This function uses the Worker API to send a message ‘graph-init’ along with the graph text.

*Figure 10: SharedWorker API postMessage function*

The graph-view window will listen for the graph-init message, and handle it appropriately. This is done in the file **src/pages/GraphView/utils/SharedWorker.js.** This file simply abstracts the communication with SharedWorkerAPI and centralizes the code for receiving and sending messages. The handler for graph-init is contained in the file **src/pages/GraphView/index.jsx** in the function **onGraphLoad.**

*Figure 11: onGraphLoad() function*

**onGraphLoad** will then handle the message by parsing the graph text, calculating a scalar, and then applying the scalar. A new Graph instance will be created from **src/utils/Graph.js** and set to the base graph, which is the graph referenced by Algorithm and GraphEditHistory

## 

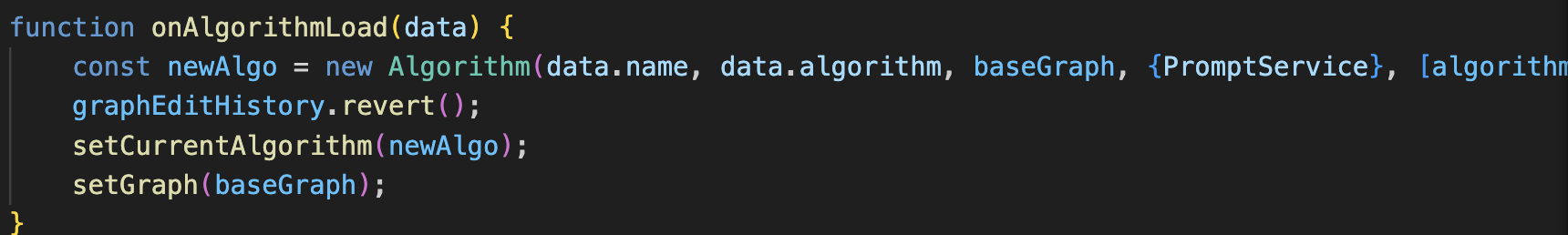
## Loading an Algorithm

Loading an algorithm begins in the editor, in the file **src/pages/AlgorithmEditor/EditorGroup.jsx** when a user requests an algorithm to be loaded, the function **loadAlgorithm** is called**.**

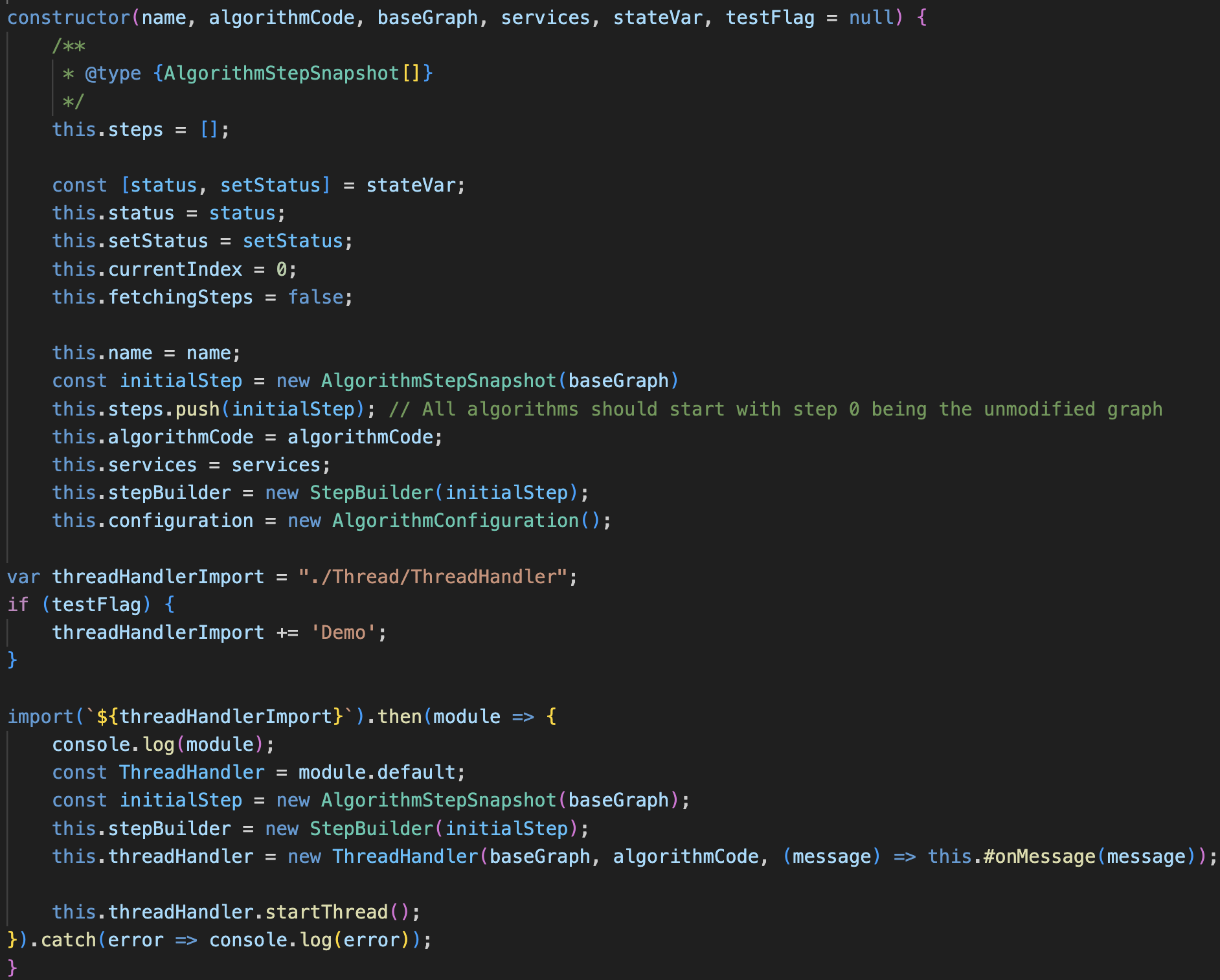
*Figure 12: loadAlgorithm() function*

Load algorithm handles sending the message via the WorkerAPI with message type algo-init and the algorithm text.

The message is then sent to the graph-view page, in which **src/pages/GraphView/utils/SharedWorker.js** will be listening for the file. **SharedWorker.js** simply abstracts the SharedWorker API calls and listeners so that other functions can easily create handlers.

*Figure 13: onAlgorithmLoad() function*

The handler **onAlgorithmLoad** is located inside of **src/pages/GraphView/index.jsx** and creates a new algorithm instance, then reverts the graph edit history in case the user was in edit mode. Then, the algorithm is set to the new algorithm instance and the graph is reset to the base graph.

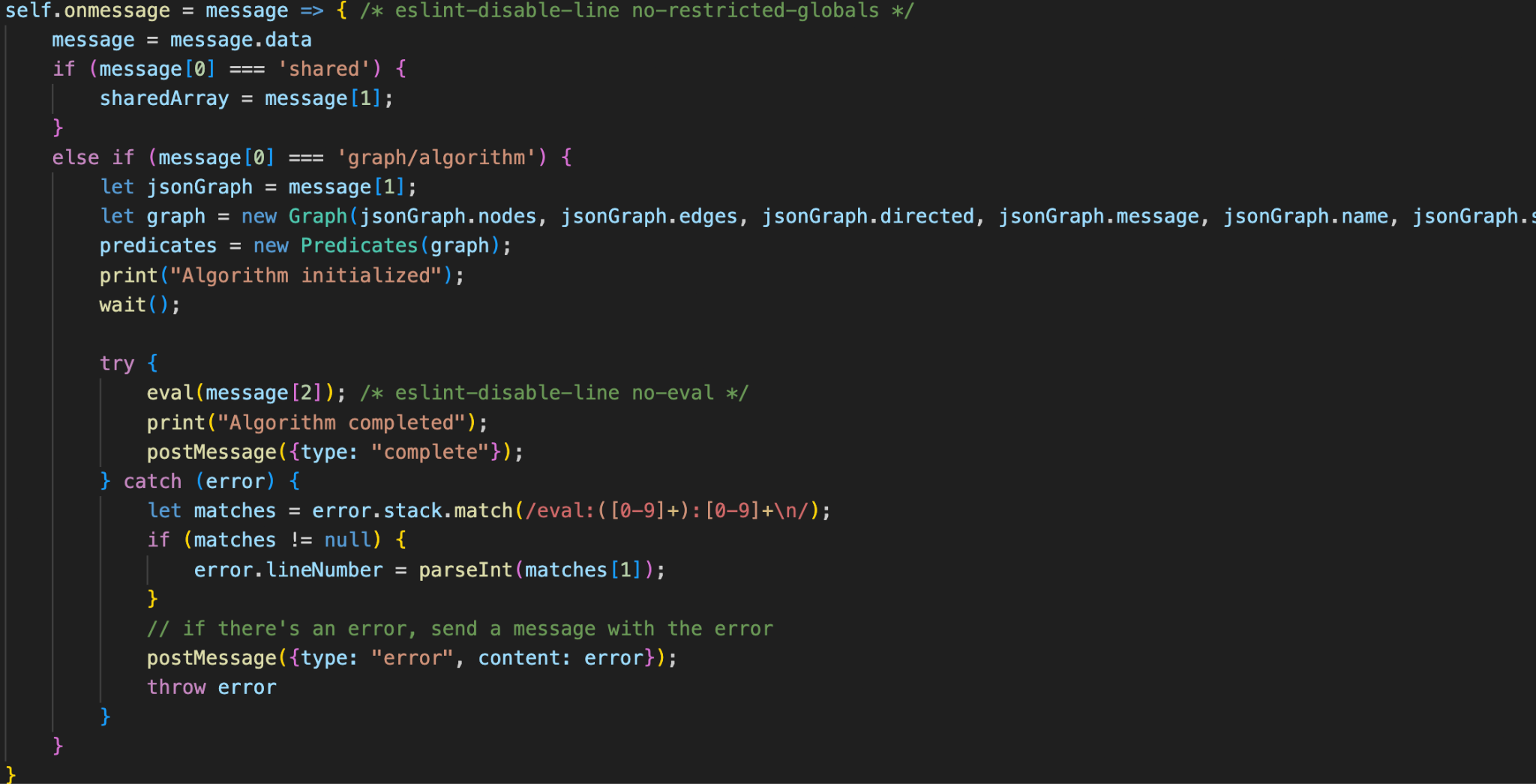
*Figure 14: Algorithm constructor*

During the creation of the Algorithm, several operations are performed which are located in the constructor of the Algorithm class located at **src/utils/Algorithm/Algorithm.js.** In this file, the following attributes are assigned:

**Table 4: Algorithm Attributes**

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| **status/setStatus** | Allows for triggering of useEffect |
| **currentIndex** | the current index of the step |
| **fetchingSteps** | Whether the algorithm is currently fetching a step |
| **name** | Name of the algorithm |
| **steps** | An array of AlgorithmStepSnapshots |
| **algorithmCode** | The algorithm text as a string |
| **services** | Services that the algorithm can call. Currently only has PromptService |
| **stepBuilder** | The step builder contains modifications that are not yet ready to be compiled into a single step |
| **configuration** | All the configuration options for the algorithm. E.g. controlNodePositions. |
| **threadHandler** | The thread handler that executes the code via eval() and sends messages via the WorkerAPI. |

A key line in the constructor is the creation of a ThreadHandler instance. The code for this can be found at **src/utils/Algorithm/Thread/ThreadHandler.js**. The constructor basically communicates with the worker to start, resume, and kill the thread. It does this via a SharedArrayBuffer and Atomics to read from the SharedArrayBuffer. During startThread, it will create a new Worker from **src/utils/Algorithm/Thread/Thread.js** and pass it along the graph and algorithm code.

*Figure 15: Thread.js onMessage handler* 

Thread.js will then execute the code based on eval and begin responding with messages. Thread.js also defines all the functions and resources that the algorithm can utilize.