CS 428/429

CGC

Final Documentation

Table of Contents

Project Description

Canvas Graph Creator is a browser-based graph editor. It utilizes HTML5 canvas to allow its user to create and edit graphs. Customization options exist for both nodes and edges, giving them colors, labels, and different styles. Algorithms such as the breadth-first traversal can be run on user-created graphs. Users are able to save and load their creations using JSON. And once a graph is complete, users can save their creation as an image.

The catalyst for this project stems from a need that we CS students have noticed is unfilled as of now: a time-efficient way of creating graphs. Current solutions to the problem involve using PowerPoint, or the more specific draw.io, which, while proper solutions to the problem, are general purpose enough to significantly encumber the process of making a graph. This project fills several niches that have once not been filled, including, but not limited to, homework submissions, interactive student learning and lecture supplements.

Software Process

Iterative Development

Our development cycle operated using iterations, spanning roughly 2 weeks per iteration give or take depending on open slots for available meetings at the end of each cycle. In the course of each iteration, there were mandatory team meetings every wednesday at 8pm using google hangouts to mediate. Failure to attend these meetings without proper excuse delivered ahead of time was grounds to be forced to buy the whole team as well as the TA pizza at the next graded meeting.

Each iteration followed a typical progression. At the beginning we would overview our current user stories, pick those we thought most prudent to work with, and then the team would discuss overarching architecture and how things might best fit together. Once our approach was decided, we would split into groups for each task, more on collaborative development will be discussed later.

Refactoring

Javascript is unique in how it is written, conventions usually taken as law in languages like Java and c++ are weak or nonexistent in Javascript. Things such as classes, scope and variable types exist in Javascript but are vague and circumventable or easy made out as inconsequential. This usually makes writing Javascript easier and faster, but when working with a team there are some downsides to this apparent developmental ease.

Due to the fact that writing Javascript can be done so quickly, and because strict scope and typing are not enforced, it is very easy to make code that becomes complicated very quickly. This is especially true when working with multiple people who may have different visions or expectations. Even a minor difference in the perception of how to best solve a problem can result in drastically different approaches to coding.

The best way to tackle the knots of code that inevitably form in such an environment is of course with a high frequency of refactoring. Ensuring that every member was responsible for some refactoring every iteration ensured that we kept our code base from becoming too convoluted.

Aside from condensing and simplifying code, there were three other types of common refactorings of import. These were, namely, renaming classes/variables, code extraction, and of course commenting. Renaming classes and variables is very important, as even with well worded and permeative code comments understanding what is going on in a program can be very difficult if the names of objects are too similar to each other or too nondescriptive. Proper naming conventions take only a small amount of effort and produce great results.

Code extraction is perhaps the flagship method of refactoring. Ensuring that functions are short and readable, and that the intention of each function is clear is of utmost importance. Not only does this enable an easier understanding of the program’s flow and goals, but it makes debugging much easier as well. The smaller that each function is, the easier it is to locate the origin of an error.

The last major form of refactoring we rigorously employed was code commenting. While inline comments are usually not necessary due to the verbosity of Javascript, function and class block comments ensure the reader is imbued with an understanding of the grand intentions of the associated code. While making such comments can be tedious, our team found that an addon to the Atom text-editor/IDE called docblockr made our efforts towards this end much easier.

Testing

In addition to the refactoring mentioned above, unit and acceptance testing was heavily incorporated into our development process. By modularizing units of functionality and writing tests for them, future changes to the code can be merged with greater confidence; by simply running all of the unit tests, we can see whether or not the latest changes broke any of the original functionality.

Our project also includes acceptance tests in the form of Cucumber feature files. These tests are written in plain english and are used to make sure our project is implementing all the user stories.

To maximize the benefits of unit testing and to integrate testing into our workflow, Travis CI was introduced. Each commit or pull request to the repository now triggers a build on Travis CI. We configured Travis CI to run unit tests after each project build, and these results are reported back to GitHub and displayed next to each branch or pull request. Not only does this provide test results for every change pushed or merged, but it also makes it easier to determine which particular commit caused tests to fail.

Collaborative Development

As stated earlier, at the beginning of each iteration and after our next set of user stories had been decided upon, we would split into groups for each task. This process is more commonly called ‘pair programming,’ and although the name implies groups of two, we took a more fluid approach. Depending on the needs of the iteration and it’s various tasks, we broke into groups of two, three, or sometimes even worked independently.

Once broken into groups the approach adopted by each sub-team was also subject to a ‘whatever works best’ basis. Sometimes the members would adopt a typical pair programming approach and work together in person, one person programming at a time. Other times, the group members would split work and work independently, or perhaps code in unison in person or over google hangouts.

While we did not follow strict pair programming paradigm, there were a few very important procedures that were essential to our procedure. These were collective code ownership, peer review, and the git branch workflow. Even if working independently in a sub group, all members discussed and explained their intentions and understanding to their teammates, to ensure collaborative work stayed collaborative. Our regular wednesday meetings ensured that this happened on at least a semi regular basis.

More importantly perhaps, was our unfailing practice of code review. At the end of each iteration when it came time to tag our code, as well as many times along the way, our code was always peer reviewed to guarantee as strongly as possible it’s quality. By maintaining a consistent habit of code peer review we greatly reduced the risk of failing code come the end of the iteration.

Finally, aiding our procedural collaborative development was the git workflow. Although the git workflow has a wide array of tools in it’s arsenal, just using separate branches aided our development considerably. Having so many people work on a single code base at once is liable to get confusing fast, with frequently broken code besides. Each group was mandated to create a branch for their own task, and at the end of each iteration after rigorous code review had been applied, a pull request would be made and so merged into master.

Requirements & Specifications

User Stories

Add Node: As a user, I want to be able to add nodes to the graph.
Add Edge: As a user, I want to be able to add edges to the graph.
Remove Node/Edge: As a user, I want to be able to erase elements from the graph.
Edit Node/Edge Data: As a user, I want to be able to edit node and edge data.
Toolbar: As a user, I want to be able to choose tools based to perform actions on the graph.
Pan & Zoom: As a user, I want to be able to pan and zoom about the graph.
Node/Edge Labels: As a user, I want to be able to create labels on nodes and edges.
Context Menu: As a user, I want to be able to easily perform different actions via a context menu.
Sidebar: As a user, I want to be able to view all data associated with a component of the graph.
Algorithms: As a user, I want to be able to run algorithms upon a graph.
Import/Export: As a user, I want to be able to import and export a graph.

Use Cases

Actor	Goal	Brief
User	Add a Node	User switches to ‘add node’ tool. User clicks on canvas. System adds a node to the canvas.
User	Add an Edge	User switches to ‘add edge’ tool. User selects one node and then selects another node. System adds an edge from the first node to the second node.
User	Remove an Element	User switches to ‘eraser’ tool. User selects a graph element. System removes the element (and its dependent edges in the Node case) from the graph.
User	Move a Node	User switches to ‘select object’ tool. User clicks and drags a node. System moves the node to wherever the user clicks.
User	Edit graph element values	User switches to ‘select object’ tool. User clicks on a graph element. System shows element’s data on the sidebar. User edits data element. System updates element.
User	Run algorithm upon a graph.	User switches to ‘algorithm’ tool. User chooses algorithm. User selects source. User clicks generate results. System processes results and then steps through the result.
User	Save graph as Image	User right clicks. User chooses ‘Save as Image’ option. Browser downloads an image of the current canvas.
User	Import Graph	User clicks on the File Load option. User chooses a json file. System loads the file and displays the graph.

User	Export Graph	User clicks on the File Save option. System generates a json file for the graph. Browser downloads the json.

Architecture & Design

Top-down Architecture Overview

The overall architecture of the Canvas Graph Creator is well described by our file structure. First to notice is the file ‘app.js’, this is the entry point of the whole application. It is not responsible for much aside from initiating the graph and the UI, as well as requesting the loop that continuously redraws the canvas.

The ui folder holds all the code responsible for creating, managing, and interacting with the HTML front-end of the webpage, minus the canvas itself. There are three primary interfaces, the first is the toolbar - located to the left of the screen and containing various tools or modes of interaction available to use on the canvas. The second is the sidebar, located to the right side of the screen that holds contextual buttons sensitive to the currently selected tool and the context of user action. The last is the topbar which performs much as the sidebar, but holds more auxiliary controls and is present mostly for organizational purposes.

The data folder holds the code responsible for storing the logical state of the graph, as well as the code for drawing the visual elements on the canvas. It holds classes for labels, nodes, and edges, as well as various subclasses for each. There are also various forms of customizations available, including but not limited to color, size, shape, line styles, etc.

The algorithm folder contains the code for the algorithm emulation. Once a user has created a graph, they may then switch to the algorithm tool in the sidebar and choose one of many selectable algorithms to run. The algorithm emulator features playing, pausing, and stepping as well as speed controls.

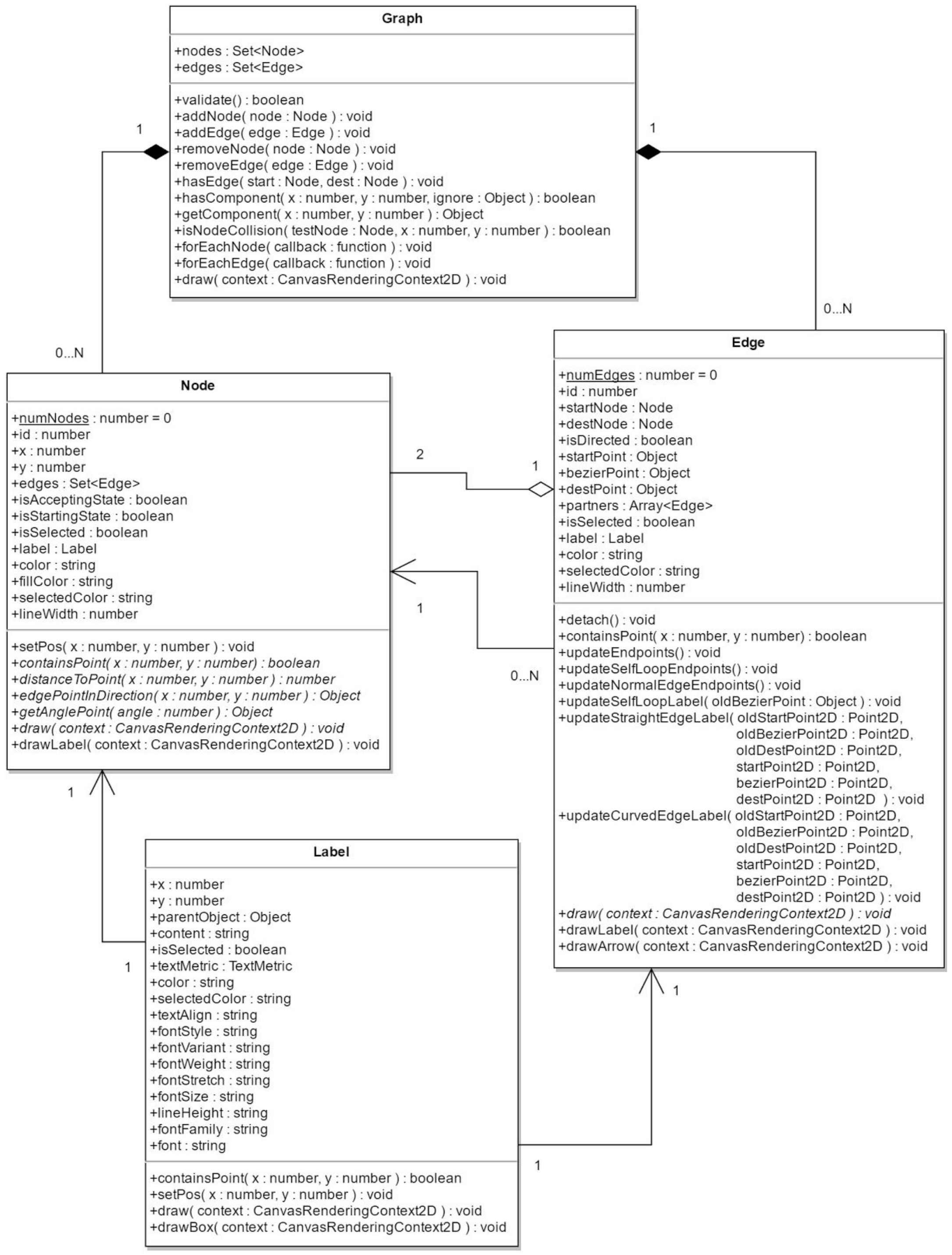
The tool folder contains one class for each tool in the application sidebar. When the tool is selected, it modifies the interactions that occur with the graph, as well as the contents of the top and sidebar.

The util folder contains various miscellaneous classes and files with a wide array of functionality. Contained therein is code for drawing bezier curves, panning the canvas programmatically to a particular element, saving and loading the canvas, and more.

Design

Graph

In order to store the state of the graph as well as other data necessary to display it, classes were created to represent each component of the graph.


Graph
+nodes : Set<Node>
+edges : Set<Edge>
+validate() : boolean
+addNode(node : Node) : void
+addEdge(edge : Edge) : void
+removeNode(node : Node) : void
+removeEdge(edge : Edge) : void
+hasEdge(start : Node, dest : Node) : void
+hasComponent(x : number, y : number, ignore : Object) : boolean
+getComponent(x : number, y : number) : Object
+isNodeCollision(testNode : Node, x : number, y : number) : boolean
+forEachNode(callback : function) : void
+forEachEdge(callback : function) : void
+draw(context : CanvasRenderingContext2D) : void
1
1
0...N
0...N
Node
+numNodes : number = 0
+id : number
+x : number
+y : number
+edges : Set<Edge>
+isAcceptingState : boolean
+isStartingState : boolean
+isSelected : boolean
+label : Label
+color : string
+fillColor : string
+selectedColor : string
+lineWidth : number
+setPos(x : number, y : number) : void
+containsPoint(x : number, y : number) : boolean
+distanceToPoint(x : number, y : number) : number
+edgePointInDirection(x : number, y : number) : Object
+getAnglePoint(angle : number) : Object
+draw(context : CanvasRenderingContext2D) : void
+drawLabel(context : CanvasRenderingContext2D) : void
2
1
1
0...N
Edge
+numEdges : number = 0
+id : number
+startNode : Node
+destNode : Node
+isDirected : boolean
+startPoint : Object
+bezierPoint : Object
+destPoint : Object
+partners : Array<Edge>
+isSelected : boolean
+label : Label
+color : string
+selectedColor : string
+lineWidth : number
+detach() : void
+containsPoint(x : number, y : number) : boolean
+updateEndpoints() : void
+updateSelfLoopEndpoints() : void
+updateNormalEdgeEndpoints() : void
+updateSelfLoopLabel(oldBezierPoint : Object) : void
+updateStraightEdgeLabel(oldStartPoint2D : Point2D, oldBezierPoint2D : Point2D, oldDestPoint2D : Point2D, startPoint2D : Point2D, bezierPoint2D : Point2D, destPoint2D : Point2D) : void
+updateCurvedEdgeLabel(oldStartPoint2D : Point2D, oldBezierPoint2D : Point2D, oldDestPoint2D : Point2D, startPoint2D : Point2D, bezierPoint2D : Point2D, destPoint2D : Point2D) : void
+draw(context : CanvasRenderingContext2D) : void
+drawLabel(context : CanvasRenderingContext2D) : void
+drawArrow(context : CanvasRenderingContext2D) : void
1
1
1
1
Label
+x : number
+y : number
+parentObject : Object
+content : string
+isSelected : boolean
+textMetric : TextMetric
+color : string
+selectedColor : string
+textAlign : string
+fontStyle : string
+fontVariant : string
+fontWeight : string
+fontStretch : string
+fontSize : string
+lineHeight : string
+fontFamily : string
+font : string
+containsPoint(x : number, y : number) : boolean
+setPos(x : number, y : number) : void
+draw(context : CanvasRenderingContext2D) : void
+drawBox(context : CanvasRenderingContext2D) : void

D-0

From the UML class diagram D-0, we can see that the `Graph` object holds a set of `Node` and `Edge` instances. Each `Node` instance has fields for storing its coordinates and state. Edges also have fields for storing similar information; however, instead of coordinates, each edge stores a `startNode` and `destNode`, since the edge's direction and orientation depends on which nodes it is attached to. Nodes also store references to all the edges that they are connected to, though this is simply to avoid checking all of the graph's edges when updating a node's location. The `Label` class is used to represent text that needs to be displayed on the canvas. Both nodes and edges can have labels associated with them, so both have a field for `Label` objects.

Canvas

In order to draw graphs onto a canvas, a callable function needs to be passed to `window.requestAnimationFrame()`. That function will also need to call `window.requestAnimationFrame()` with itself. In this manner, the function will be called for every frame that the browser requests the canvas to be redrawn (usually as often as the refresh rate of the screen).

To draw the graph, a `draw()` function has been added to `Graph`, `Node`, `Edge`, and `Label` (as can be seen in diagram D-0). The canvas context is needed to make changes to the canvas, so it is passed as the only parameter. The `Graph`'s `draw()` function is called every frame and it in turn calls the `draw()` functions of each `Node` and `Edge`. This way, the objects are able to draw themselves to the canvas every frame.

Tools

Tool classes were added as a way to separate and modularize independent editor functions. In order to enable user interaction with canvas elements, mouse event handling was added to detect user actions. This was necessary because the canvas is essentially a dynamically drawn image; it is a single HTML element and clicking/dragging of content drawn on the canvas is not possible to handle directly with event listeners. To bypass this, we captured mouse events on the canvas element and passed them to to the `MouseHandler`, which differentiates the mouse actions based on graph component locations, mouse position, and mouse click state. For example, if the user clicks on a node, the `MouseHandler` will detect that the mouse position is inside of a node and will call `selectObject()` of the currently selected `Tool`. For dragging, the `MouseHandler` checks if the mouse movement between the `mousedown` and `mouseup` events surpasses a threshold. Each of the `Tool` subclasses then overrides the functions from `Tool` that are called by the `MouseHandler`. This way, the tools can have differing functionality without any duplication of mouse event handling.

User Interface

The design of the user interface was done to emulate the UI paradigms of popular image-editing software such as Photoshop. The user interface has four main components each relating to a major html element upon the front-end. These components are `Canvas`, `Toolbar`, `Top-Bar`, and `Sidebar`. Those four alongside the `ContextMenu` are the components which make up the UI.

`Canvas` is the image itself that is constantly being refreshed. Its main components is a multitude of listeners which search for the different click types. Based on the type of click `Canvas` then calls a `MouseHandler` function which then interacts with the graph, or it calls the `contextmenuEventListener()` which toggles open the `ContextMenu` to be acted upon.

The `ContextMenu` once opened displays a set of options based on the graph element which the mouse was over when the context-click was made. Once an option is chosen the menu's `mouseup` listener calls a `MouseHandler` function which edits the graph accordingly.

The `Toolbar` is the interface for changing tools the tools explained above. It contains `toolMap` which maintains a map between tool names and tool instances. Upon clicking on one of the tools the `selectTool()` function is called which updates both `TopBar` and `Sidebar`.

As mentioned before, the `TopBar` and the `Sidebar` both have unique functions depending on the currently selected tool. Upon the selection of a tool, `selectTool()` calls the `setSidebar()` function of sidebar which selects a class of `Sidebar` and displays it there. These sidebar types include a sidebar for editing objects and one for running algorithms.

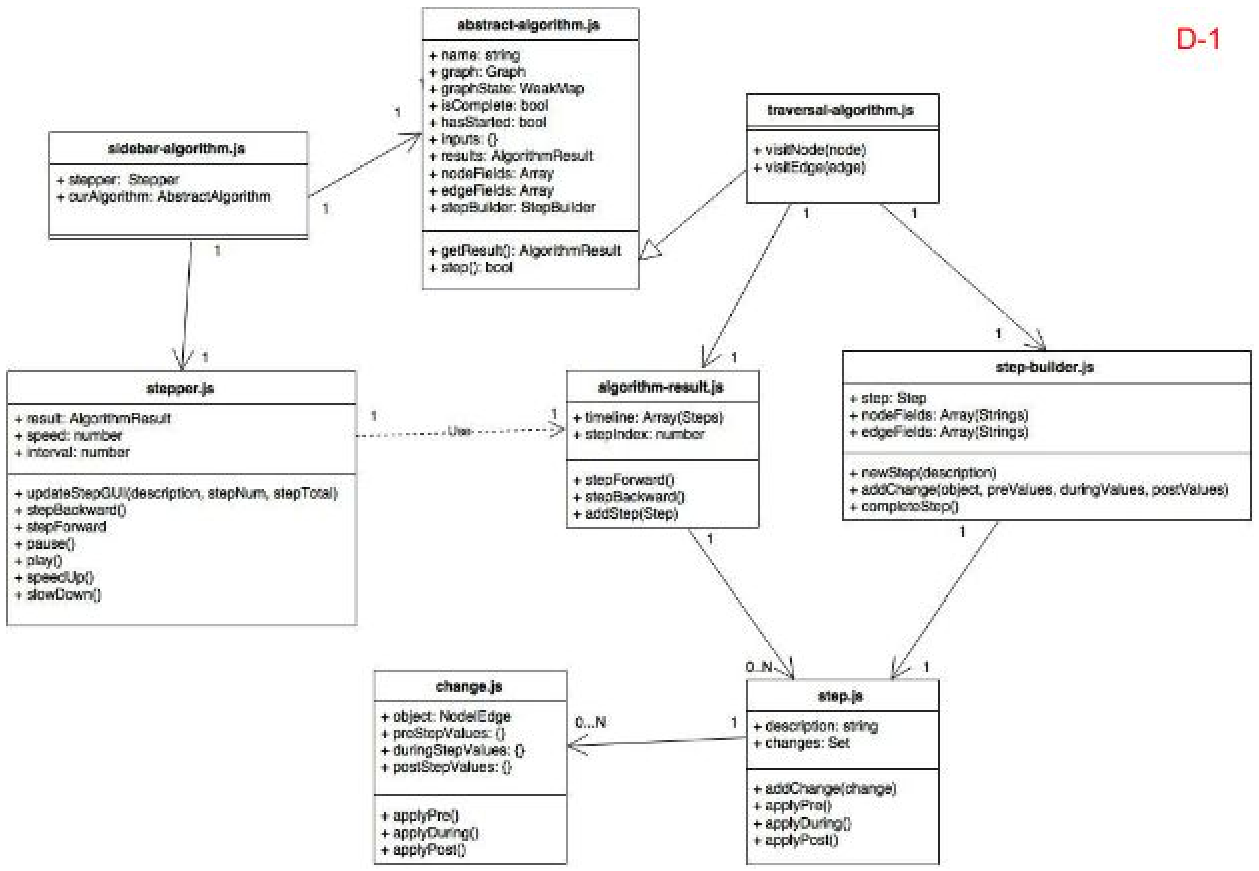
The `TopBar` is also set with function calls of `showModes()`, `showInputs()`, and `showModeInputs()`. The `TopBar` uses a form dependent on the selected tool. It can take input which alters the type of node or edge is being created by one of the add tools. Or it can work in tandem with the `Sidebar` to help run algorithms as explained below.

Algorithms

The algorithm emulation system is robust and easily expandable, built with the ease of future development in mind. The core components are, as seen in diagram D-1 (seen below), the sidebar-algorithm, the stepper, the results, and the algorithm itself. Originally we had planned to keep track of the logical state of the graph as well as the visual state separately, stepping back and forth between them as the algorithm ran. It quickly became clear, however, that this would involve an excessive amount of work and quickly become overly complicated. Our second and current design simplified things significantly, and now we only run the algorithm once and keep track of all of the changes that occur, stepping back and forth between this preemptively generated list. We will proceed to go over all the components of the system and how they work together.

Starting at the bottom of diagram D-1, we have `Change` and `Step`, these are the building blocks that correspond to the graphical changes that the algorithm dictates. A `Change` represents a group of graphical alterations to a single object (a node or an edge) in the graph that occur in a single instance, or step. A `Step` represents a single instant in the timeline of the algorithm, each step may have many changes stored within it.

D-1



Moving upward, we have the algorithm-result and step-builder files, which contain what are essentially just helper classes. `StepBuilder` helps the coder build changes and group them into steps before adding the steps into the algorithm-results. `AlgorithmResult`, as the name implies, is responsible for containing and providing the first interface of access to the results of the algorithm after it has run.

The code for the algorithm itself extends a superclass, `AbstractAlgorithm`, filled with code and methods common to all or most algorithm types - or at least, this was the original intention. At the time of the writing of this document, it has proven difficult to isolate such commonalities, and currently the abstract superclass sits quite bare. It is the hope that with a few more algorithms eventually implemented, a proper and useful abstraction will reveal itself.

The `Stepper`, to the left in diagram D-1, steps through the algorithm results one at time. It's main contribution is the management of an interval, a Javascript object that runs code on a timed schedule. It also allows for the alteration of this interval as it runs, enabling the user to speed up or slow down the rate at which the algorithm steps are displayed.

All of this is tied together in the `SidebarAlgorithm`, responsible for creating and managing the sidebar GUI content relevant to the algorithm, as well as running the algorithm generation code. The sidebar for the algorithm emulator displays a playbar similar to that of a music player, as well as an auto scrolling descriptive list of the steps as they run.

Import/Export

The import/export function manifests to the user as a set of buttons on the status bar. Internally, the `Serializer` class is responsible for reading or replacing the graph structure via its `serializeGraph()` and `deserializeGraph()` functions respectively, in response to user actions. These functions are aware of the sub-structures within `Graph`, as defined in D-0, and can convert between one and a stringified JSON object. The “Quick” Load and Save variations simply use a string to store the structure. Regular Load triggers a file selection dialog then asynchronously loads the selection. Regular Save appends the JSON to a Data URI and navigates to it, triggering a download in modern browsers. This is similar to how we save the canvas as an image. The import functions should non-destructively fail on a malformed file.

Acceptance Tests

The Selenium testing framework is composed of several parts: the Cucumber step definitions, the Webdriver Factory, the Image Utilities, and the Page Objects. Most of the implementation for the tests are in `CanvasPage` and `CGCPage.` These classes both follow the Page Object design pattern commonly used in Selenium tests. The Page Objects represent the webpage and have methods that are used to interact with the webpage. These methods are called in the Cucumber Step definition class.

Language, Frameworks, and More

Testing aside, no framework or library was used to develop our project. However, the choice of programming language did influence the design of major components of our system.

ES2015 (also called ES6)

Our choice of programming language was JavaScript. This should come as no surprise because the HTML5 canvas requires the usage of JavaScript to draw. However, we decided early on to use the features of JavaScript as defined in the ECMAScript 2015 Language Specification, which introduced a large set of new features. When our project was first proposed, major browsers had yet to support a significant number of these features (e.g. Chrome 48, released in Jan 2016, only supported ~65% of ES2015). However, there existed compilers to transform code written in ES2015 to JavaScript that followed the ES5 standard, which was already supported by all major browsers. Since ES2015 features were eventually going to be supported by browsers, we decided to use one such compiler, Babel, so that we could use some of the convenient new features.

As a direct result of writing code in ES2015, classes were used in an attempt to organize functionality in an object-oriented way. The same design would likely be possible to implement in ES5 as well; however, the syntax and code would be a lot messier and more confusing due to the prototypical nature of inheritance in JavaScript.

Sass

Using Sass enabled us to write organized, hierarchical, and modular CSS. By making the styles as reusable as possible, we were able to minimize file length and make the code easier to read.

Unit Tests

Mocha

Mocha is our unit testing framework of choice due to its flexibility and usability. Mocha is a popular open source JavaScript testing framework that is able to run tests for both Node.js and the browser. It’s ability to run without opening an HTML file in a browser makes it much easier to run unit tests for our data structures. Mocha also supports usage with any assertion library, so the syntax for the tests can be chosen freely.

Chai

Chai is the assertion library we chose to use for unit tests. Again, this choice was made due to the flexibility of the library. Chai supports both BDD and TDD style syntax for assertions, and is also able to pair with any JavaScript testing framework.

Acceptance Tests

Selenium

Selenium is the software testing framework we chose to write our acceptance and integration tests. We chose to use Selenium because it allows us to make automated tests that are less brittle. Making changes to tests that use Selenium is much easier than record and replay tests. The selenium tests we wrote were in Java since it has the best compatibility with both Selenium and the following softwares.

Cucumber

Cucumber is the software tool we chose to run the automated acceptance tests through Selenium. This choice was made because Cucumber allows us to write naturally phrased tests. This makes the tests both easy to write and easy to understand.

JavaCV

JavaCV is an image library we chose to use alongside the testing framework. Our choice of implementing our web application with Canvas meant that traditional Selenium tests could not be used; Selenium can interact with web elements, but it cannot interact with elements on the canvas because the canvas itself is an image. JavaCV was chosen because its contains image recognition software which lets us overcome this issue. Thus it allows us to identify graph and tool elements upon the canvas to verify that our acceptance tests create the graphs that we expect.

Reflection & Lessons Learned

Nicholas James Kelly

The nature of this project was very different from that of last semester 427. As far as actually working on the project, the primary difference came in that this project was much more tightly knit in functionality. Just about everything everyone did was highly tied to everything else, but our process and teamwork mitigated the vast majority of negative effects this might have had. Overall, I learned a lot, especially about testing from Nicholas Lu and Javascript from Allen. I had a great time, and I hope to continue to work on this project after I graduate.

Miles Teh

This project has a few differences from the one from CS 427. Where in CS 427 we built upon an already made plug-in, here we built a program from scratch. Since we were building from scratch many of the initial functions written were intertwined with each other and thus the different user stories required more conversation between groups to be implemented properly, thus teaching us how to work together to build a piece of software from the ground up. I also learned a lot about testing frameworks from Nicholas Lu. It was a good time and a more interesting project than I first thought it would be.

Nicholas Lu

Throughout this project, I learned how to use many frameworks and tools used in a typical production environment. I experienced many of the challenges and benefits of relying on external services like Sauce Labs and Travis. I also learned that features that initially seemed simple could have very challenging issues that have to be solved.

Austin Kramer

The freeform nature of this project, compared to CS 427, led to a much more involved development process. Unlike the pure pair-programming and the building of disjoint features on an existing platform from that class, our group this semester had a much more functionally collaborative team environment, and a coding environment with more intricacies that we all had to cooperatively maintain and expand. In addition, our coding environment, testing framework, and target platform all helped expand my knowledge of web development in general. It was a refreshingly rich learning experience overall.

Matthew Yang

As one of the “co-founders” of the group, I’m really amazed at how our team came together. More than just collaborators on the code, we were a supportive, interactive and communicative group, ready to help each other or share a quick joke at a moment’s notice. It wasn’t always smooth sailing, many of us had to get caught up to speed on a new language, technology or testing framework, but the cohesiveness of our team lead us to fruitful navigation of these barriers. While I personally learned a lot from personal coding experience and the examination of my peers’ implementation, I think the biggest takeaway from this project is a new understanding of the value of proper teamwork.

Allen Yao

Overall, I think that this project was a success. We managed to accomplish what we set out to do: build a graph creator that uses the HTML5 canvas. Our team managed to stay organized and completed tasks and user stories as they were assigned in each iteration. Although we didn’t strictly follow pair programming practices, we were able to review each other’s code as it was committed and merged. Personally, I learned quite a lot through this project. This was the first project using ES2015, Webpack, Selenium, and Cucumber that I have been a part of, and learning how to use these technologies was challenging at times.

Athanasios Sofronis

I consider myself extremely lucky to have been placed in such a successful and motivated group for the project. I believe the project was an enormous success and it was also an excellent exercise in software engineering principles, collaborative design, and teamwork. I am especially impressed by how successfully the new user interface came into play and how intricate, detailed, and structured our testing methods were. This was my first exposure to things like Selenium and Cucumber and I am very glad that I was in such a supportive environment. I was challenged at times to make sure my code integrated well into the entire project and also when it came to testing. I’m glad I had this experience and I’m especially thankful to my teammate without whose help I would have never been able to succeed.

David Shechtman

Working on this project was far and away the most successfully collaborative experience I have had as a student here at the UIUC. A lot of the technologies used were some I was not accustomed to to begin with. I have never done any testing with Cucumber or seen Selenium work. My prior experience with javascript was minimal, and through the avenue of the project I am comfortable with that language now. I also learned a decent amount about how to properly use git and seen the advantages it holds over SVN in practice. Atop of all that, working with various students, all of whom have different strong points in programming was great experience in learning to gel and mesh with various working styles. I want to extend a thank you to all the group members for pulling more than their own weight on multiple occasions and making this 428 semester a blast.