ALGORITHM VISUALIZATION

A graduate project submitted in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering

by

Armine Khachatryan

December 2019
The graduate project of Armine Khachatryan is approved:

_____________________________    ______________________
Kyle Dewey, Ph.D.                  Date

_____________________________    ______________________
Katya Mkrtchyan, Ph.D.             Date

_____________________________    ______________________
John Noga, Ph.D., Chair            Date

California State University, Northridge
Dedication

I dedicate this paper to my family, who continuously supported me every step of the way. Without all of your support and encouragement, this would not have been possible.
# Table of Contents

Copyright ........................................ ii

Signature page ................................ iii

Dedication .......................................... iv

Abstract ........................................... viii

1 Introduction ...................................... 1
   1.1 Background ................................... 1
      1.1.1 Algorithms ............................... 1
      1.1.2 Styles of Learning ....................... 1
   1.2 Motivation ................................... 2
   1.3 Similar Work .................................. 3
   1.4 My Approach .................................. 4

2 Algorithmic Choices .............................. 6
   2.1 Counting Inversions ........................... 6
      2.1.1 Formal Definition ......................... 7
      2.1.2 Limitations ............................... 7
   2.2 Kruskal’s Minimum Spanning Tree ............ 7
      2.2.1 Formal Definition ......................... 7
      2.2.2 Limitations ............................... 7
   2.3 Steiner Tree ................................ 8
      2.3.1 Formal Definition ......................... 8
      2.3.2 Limitations ............................... 8

3 Technology Choices ............................. 9
ABSTRACT

ALGORITHM VISUALIZATION

By

Armine Khachatryan

Master of Science in Software Engineering

Algorithms are an important topic in both computer science and software engineering because they form the foundation of creating efficient programs and applications.

I created an algorithm visualization (AV) tool for CSUN faculty members, students, and anyone else who is studying algorithms to use. This AV tool helps instructors focus less on drawing complicated diagrams or graphs in class and focus more on explaining the actual logic behind the algorithm.

AVs are more helpful when they are user-friendly and interactive. I came across several AV tools that were not very effective in my opinion because they used a more technical approach, so I decided to recreate some of them. The algorithms I created AVs of are Counting Inversions, Kruskal’s Minimum/Maximum Spanning Tree, and Steiner Tree.
Chapter 1

Introduction

1.1 Background

1.1.1 Algorithms

Algorithms are an important topic in both computer science and software engineering because they form the foundation of creating efficient applications. Algorithm Visualization (AV) refers to diagrams, graphs, animations, and visuals created by software to help make learning algorithms fairly easy.

Although there are a lot of AV tools for beginner-level algorithms, there are fairly few for intermediate-level. Some examples of beginner-level algorithms are "basic sorting, searching, and data structure traversal and retrieval" [11]. Examples of intermediate-level algorithms include "trees, graphs, simple greedy, and divide and conquer" [11].

1.1.2 Styles of Learning

According to an article called “Understanding The 7 Types Of Learning Styles” published by Cheyenne Diaz, although every person is unique and processes information differently, there are seven common types of categories everyone uses [7]. The seven styles of learning are visual, aural, verbal, physical, logical, social, and solitary [7].

Visual learners are people who learn best when they have images or diagrams to assist them to understand the information [7]. Aural learners respond more to sound, so this type of learning is mostly done in music classes [7]. Those who are verbal learners process information better through writing and verbal instruction [7]. Physical learners notice small details of the world around them, and they enjoy solving jigsaw puzzles, playing sports, and exercising [7]. Logical learners are those who try to understand the reasoning or logic
behind everything [7]. These types of learners usually major in science, mathematics, or engineering [7]. Social learners are those who enjoy engaging with others or studying in groups [7]. On the contrary, solitary learners learn best when they study on their own [7].

While it may seem like my project targets only visual learners, CSUN faculty are going to use these AVs during their lectures, which will also target verbal learners. Because computer science combines both math and engineering, logical learners will also benefit from using this tool. Students may use this tool both in groups or individually, which will target social and solitary learners, too. The more distinct ways someone approaches teaching algorithms, the more students they will benefit.

1.2 Motivation

When I was pursuing a bachelor’s degree in computer science, I worked as a Mathematics tutor at the Center for Academic Success in Los Angeles Pierce College and the Learning Resource Center in the Oviatt Library at CSUN, where I would help students struggling in pre-algebra, algebra, trigonometry, business calculus, and calculus I.

Even though I never had trouble explaining mathematical concepts to individuals, when I took upper division computer science courses at CSUN, I began to wonder how someone would explain complex computer science concepts to others, such as depth-first search. Depth-first search is a little harder to visualize because it uses recursion. In computer science, recursion is a method where the answer depends on the answers to smaller cases of that same problem.

When I took MATH 482 at CSUN, which is the math equivalent of the algorithms course COMP 482, the professor was great at explaining the concepts, but there were not a lot of visualizations of the algorithms that were being explained, so they were a little challenging and difficult to understand when I was studying them.
1.3 Similar Work

Before I chose the topic for this project, my classmate showed me a very interesting AV tool called “Fun With Algorithms” [6], which was developed by a CSUN graduate student named Armen Ourfalian for his graduate project. After doing some research on other AV tools, I noticed that not all of them are user-friendly.

Many AV tools have been implemented in the past, but there are significantly less for intermediate-level algorithms. Even though there are particularly more AV tools for beginner-level algorithms, some are not straight-forward, such as the “Kruskal MST Visualization” [4] and “Divide and Conquer - Merge Sort” [5].

The “Kruskal MST Visualization” [4] is difficult for the user to understand and follow along because it uses a more technical approach. The visualization itself assumes the user has a good understanding of disjoint sets. I have worked with disjoint sets in the past, but when I first came across this visualization, I was a little confused, too. In my opinion, if a visualization approach is more technical, it could scare off the user by tricking them into thinking the algorithm is more complicated than it actually is. I also noticed that when the algorithm chooses an edge, it changes the color of that edge to red, and after it adds that edge to the Kruskal graph, it changes the edge width. I personally believe that users notice colors more than they notice sizes, so if we change the color of the edge to blue when it is being considered and then change it to red when after it was added to the Kruskal graph, it will help the user understand how the algorithm works.

When I came across the “Divide and Conquer - Merge Sort” [5] visualization, I thought it was a great idea to show the user how the merging phase works in breadth-first order. When there are too many steps to follow along to and memorize, it becomes difficult to understand how the algorithm works. However, the merging part was a little unclear because the entire array that is currently being merged was highlighted one color. In my opinion,
if the array that is being merged was highlighted two colors, left half is red and right half is blue, it would tell the user that two arrays are being merged, rather than one array being sorted. The purpose of merge sort is to sort the numbers as it is merging them. The visualization also displays two arrays, one is the original, and the other is the merged one. It displays the array merging in place, which can get confusing to follow.

Unlike the other AV tools I came across, Armen Ourfalian’s graduate project called “Fun with Algorithms” is an effective AV tool because it is user-friendly. The application itself is interactive, colorful, and the user-interface (UI) is not crowded. An interactive AV makes learning easier for the user because when they input the data, they become familiar with the interface. If the interface is not crowded, the user gets familiar with the tool quicker because there are not a lot of features to use. The application is also consistent. For example, every AV has the **Slower**, **Run Algorithm**, and **Faster** buttons, and each button that has the same functionality also has the same color. The **Slower** button is blue, **Run Algorithm** button is green, and **Faster** button is red. I think the color green was a good choice for the **Run Algorithm** button because users normally think of green as a symbol for “Go”, especially while driving. I used a similar approach for this project.

1.4 My Approach

The purpose of this project is to create an algorithm visualization tool for the faculty of CSUN to use when they are teaching COMP 482. After seeing how effective Armen’s approach was, I decided to adopt some of his UI choices to recreate some beginner and intermediate-level algorithms that do not have helpful AVs. Although there are a lot more AVs targeted towards beginner-level algorithms than intermediate-level, some are not as effective, as I discussed earlier. My approach is to create AVs for one beginner-level algorithm, Kruskal’s Minimum Spanning Tree; and two intermediate-level algorithms, Counting Inversions and Steiner Tree.
I decided to make a web application because it is easily accessible from any device. Users won’t have to use up hard drive storage and wait for the application to download in order to use it. They can just open up their browser and visit the website.

My overall intention for this project is to keep the UI simple and not add too many features that would complicate it, so the users can easily learn how to use it.
Chapter 2

Algorithmic Choices

Because the goal of this project is to create a visualization of different algorithms for students and faculty at CSUN to use, I chose a couple of algorithms from the textbook my MATH 482 professor used when teaching. The name of the textbook is “Algorithm Design”, written by Jon Kleinberg and Éva Tardos [12]. I chose the algorithms that I thought students would experience the most difficulty understanding, which were Merge Sort, Kruskal’s Minimum Spanning Tree, and Huffman Coding. I thought the AV tools for Merge Sort were not very user-friendly, as I mentioned before, so I decided to create my own. However, I changed Merge Sort into Counting Inversions and Huffman Coding into Steiner Tree.

2.1 Counting Inversions

I planned to use Merge Sort as my first algorithm, but I changed it to Counting Inversions because I couldn’t find an efficient AV tool for Counting Inversions. Some tools do exist, but they are still limited and may not match the learning style of some students. One student might find an existing Counting Inversions AV to be useful, whereas another student may not. Another reason why I switched to Counting Inversions is because, after I programmed the AV for Merge Sort, I noticed how similar it is to Counting inversions. In fact, Counting Inversions uses Merge Sort in the implementation, which I will discuss more about in Chapter 4. The Counting Inversions problem not only sorts an array, but it also solves for the inversion count, which specifies the number of values that need to be moved in order to sort the array.
2.1.1 Formal Definition

Given an array \( n \) of unsorted numbers, find the inversion count \([9]\). If the \( a \) is already sorted, then the inversion count is zero \([9]\). The inversion count is maximized if \( a \) is the reverse of the sorted array of \( a \) \([9]\).

2.1.2 Limitations

My Counting Inversions AV allows the user to input at most twenty numbers. Each number must fall in the range \([-999, 999]\). These two limitations will ensure that the AV will not overflow or slow down the page.

2.2 Kruskal’s Minimum Spanning Tree

Kruskal’s Minimum Spanning Tree finds a tree that spans all of the vertices and has a minimum overall tree weight. Even though the original problem is to find the minimum spanning tree, I also allowed the user to have the option of finding the maximum spanning tree. This will show them that the algorithm is just as simple to find the maximum spanning tree as it is to find the minimum.

2.2.1 Formal Definition

Given an undirected and connected graph \( g \), find the minimum spanning tree \([8]\). A minimum spanning tree is a spanning tree that has a weight less than or equal to every other spanning tree for \( g \) \([8]\). The weight of a spanning tree is the sum of all the edge weights in that tree \([8]\). The minimum spanning tree has \( V-1 \) edges, where \( V \) is the number of vertices in \( g \) \([8]\).

2.2.2 Limitations

My AV for Kruskal’s Minimum/Maximum Spanning Tree limits the number of vertices the user can input from \( V = 3 \) to \( V = 8 \). The reason why three is the minimum amount of
vertices required for the graph is because it is a directed graph. There must be at least three vertices to have a cycle in the graph. If the minimum was two, then the spanning tree would either be that one edge connecting the two vertices, or would not exist if that one edge also does not exist. The reason why I chose eight to be the maximum amount of vertices in the graph is because the graph gets overloaded with information and becomes hard to see the edges and weights.

2.3 Steiner Tree

My initial plan was to use Huffman Coding as the third algorithm, but I changed it to Steiner Tree because Huffman Coding already had some AVs, whereas I was only able to find one AV for Steiner Tree. Another reason why I switched to Steiner Tree is because Kruskal’s Minimum Spanning Tree and Steiner Tree have a similar logic. Kruskal’s Minimum Spanning Tree finds the spanning tree with a minimum tree weight, and the algorithm for Steiner Tree uses many minimum spanning trees to find the optimal one for a given set of vertices. After a student uses the Kruskal AV, they will have the background knowledge needed to understand Steiner Tree.

2.3.1 Formal Definition

Given a graph $g$ and a subset of vertices $s$ in $g$, find the Steiner Tree. The Steiner Tree is a minimum weight tree that spans all of $s$ [10]. The Steiner Tree may contain vertices that are not in $s$ [10].

2.3.2 Limitations

My Steiner Tree AV also limits number of vertices the user can input from $V = 3$ to $V = 8$ for the same reason as the Kruskal AV. I would recommend to use at most five vertices. Otherwise, the Steiner graph gets too crowded.
3.1 Angular

Many people often confuse AngularJS and Angular. Even though they are both open-source web application frameworks [1], the big difference is that AngularJS is written in and based on pure JavaScript, whereas Angular is TypeScript-based [1]. AngularJS only refers to Angular 1.0. Angular refers to versions 2.0 and above. AngularJS was the first release of the Angular framework, which was in June of 2012 [14]. Then, Angular 2.0 released in September of 2016 [1]. Angular allows developers to create a single-page application (SPA). The purpose of an SPA is to dynamically update a specific part of the page without loading the entire page.

I chose to code this project using Angular because I am going to deploy it on my personal website, which is also written in Angular. I tried to use the most updated version possible, version 7 at the time. Angular was the first framework I used when I first began web development overall and hadn’t used it since then. I wanted to get more comfortable with using Angular.

3.2 Alternatives to Angular

There are two very popular alternatives to Angular, which are Vue and React [13]. Angular is the oldest framework compared to Vue and React [13]. Vue first released in 2014 by an ex-engineer of Google named Evan You, and React released in 2013 by Facebook [13].

Although one of the similarities between these three frameworks is they all allow developers to create an SPA, the performance of each framework varies [13]. The Document
Object Model (DOM) can be seen as the UI of the application [13]. Every time there is a change in the UI, the DOM updates, which indicates changes made to the application [13].

Angular uses the real DOM, which is very difficult to handle because if the flow is lost, the developer will have to search through the code to find out what the issue is [13]. This can cause a lot of bugs and can be very time-consuming [13]. Usage of the real DOM can slow down the framework’s overall performance [13].

Both Vue and React use the virtual DOM, which is not specific to the browser and is very light in weight [13]. For React, this is included in the React package for free, and it removes majority of the issues for slow performance of the real DOM [13]. Vue used all of the good attributes of the frameworks that released before it. Vue adopted the concept of the usage of the virtual DOM from React, and this makes sure that the performance is bug-free.

When I first started creating web applications overall, the first front-end framework I used was Angular. I thought Angular is fairly complicated for first-time developers, and I noticed its lack of performance. From my personal experience, Vue was a much lighter framework, which would have been a great alternative for this project.

I did not consider to use React because it would be a huge learning curve for me, since I have never used it before. I was still a little rusty when it came to Angular, but I was able to catch on it fairly quickly.

3.3 Git, Github, and Github Pages

Git is a very popular version control system and was developed in 2005 [15]. Git can be installed and used on your local machine [15].

Three popular Git repository management services I have used are Github, Gitlab, and Bitbucket [16]. From my personal experience, all three of these services are great for large
projects, but this project was fairly small. I decided to use Github, which is a “cloud-based hosting service that lets you manage Git repositories”. The reason why I chose Github is because this is the service I was most comfortable with and have used excessively throughout college.

Another reason why I chose Github for this project is so I can deploy it on my personal website using Github Pages. Github Pages allows the developer to host directly from a Github repository for free [17]. The only disadvantage is that the user gets one domain by default, which is why I have to host it on my personal website.

3.4 Cytoscape

Cytoscape.js (Cytoscape) is a graph library that is full-featured and purely written in JavaScript for visualization and analysis [3]. It renders graph images on Node.js and includes layouts to both manually and automatically position the nodes [3]. I used this library to draw the graphs in the Kruskal and Steiner Tree AVs.

3.5 D3

Data-Driven Documents (D3) is a JavaScript library that allows the developer to manipulate documents based on their data [18]. D3 works with HTML, CSS, and SVG [18]. I used D3 to draw and manipulate the array in the Counting Inversions AV. I tried to set it up so that it manipulates an SVG file, but it was more complicated than using plain JavaScript to manipulate the DOM. I assumed it was not a good choice to use SVGs on this project, so I switched the set up so that it would manipulate HTML elements instead. I will explain the changes I made later on in Chapter 4.

3.6 Bootstrap

Bootstrap is an open-source and free CSS library ???. The reason why I chose to use Bootstrap is because this is the CSS library I was most comfortable with when I began
working on this project. Another reason why I chose to use a CSS library is so I can utilize
the grid system it offers.

To demonstrate a basic example of how to use the grid system, the developer will use a
row class with col-\{x\} (column) classes inside [19]. The sum of all the x values within a
row cannot exceed 12 [19]. If it does exceed, then the elements will not stay on the same
row [19]. In the example shown in Figure 3.1, two rows are being displayed [19]. The
first row has two columns, and the second row has three columns [19]. Notice how the
example also has a sm in the column class name [19]. This helps to make the website
mobile-responsive [19]. Mobile-responsive means the layout of the website changes when
the size of the browser window changes [19]. I will not go into too much detail about this
because my project is not mobile-friendly [19].

![Bootstrap's grid system](image)

```html
<div class="row">
  <div class="col-sm-8">col-sm-8</div>
  <div class="col-sm-4">col-sm-4</div>
</div>
```

Figure 3.1: Bootstrap’s grid system

Bootstrap can be used with any framework. However, if I used Vue.js for this project, I
would have used Vuetify, which is a UI library specifically for Vue.js [20]. I do not have a
lot of experience with Vuetify, but one thing I noticed is that the developer can pass a string
representing a color into an attribute called color, for an element. This code is easier to
read because all of the styles are not placed in the class attribute.
Chapter 4

Implementation

4.1 Project Structure

Even though AngularJS uses a Model-View-Controller (MVC) architecture, the architecture for Angular 7 is component-based. I chose to follow the traditional MVC software design pattern [2].

The Model represents all of the data-related logic the user interacts with [2]. For example, in the Kruskal Tree AV, the user clicks two vertices to add or remove an edge with a weight value. Every edge is represented by an Edge model.

The View corresponds to the UI logic of the web application [2]. An example of a View is the Counting Inversions display, which has an input field, buttons, and a drop-down list for ascending (ASC) or descending (DESC) order.

The Controller plays the role of an interface between the View and the Model to handle all of the business logic, manipulate the data using the Model, interact with the Views to display the updated data [2]. On the Steiner Tree page, when the user clicks the **Generate Random Edges** button, the controller generates vertices with random edges (Models) based on the value of V chosen. The controller then updates the graph (View) to display the generated vertices and edges.

However, instead of the Controllers directly manipulating the Models and using external libraries, I added another layer of logic called Services. The Controllers will call Services, which will deal with the direct manipulation of Model data and usage of external libraries. This helps to keep the code clean and modular for the next developer to use it.

I am going to discuss the structure of this project in the following order: Models, Algo-
4.1.1 Models

All of the models I used for this project are located in the “src/app/models” directory. The models used in Counting Inversions are Arr, Node, and Element.

The Arr model is used when the user saves and imports arrays, as well as to display example arrays.

Each number that the user inputs is represented by an Element. The entire array is represented by a single Node model, as shown in Figure 4.1, which D3 uses to build out the tree. Every Node model has two Node models attached, left and right. I will discuss more about this in Section 4.1.2.

![Node model code](image)

Figure 4.1: The Node model contains an array of Element models.

The Kruskal and Steiner Tree AVs use the Edge, Vertex, Pair, and Graph models.
The **Pair** contains a key and a value, where the key is a unique number that the Cytoscape library uses to graph. The **id** on the **Vertice** model is of type **Pair**, and the **Edge** model has two values of type **Pair** called source and target. Both source and target refer back to a **Pair** in a **Vertice**.

Both the **Edge** and **Vertice** models are used to enter the user inputted data into a Cytoscape graph. The source and target values on the **Edge** model depend on and represent an **id** on the **Vertice** model.

However, a **Graph** model contains three arrays: **vertices**, **edges**, and **subvertices**. Each element in the **vertices** array is of type **Vertice**, which contains all of the vertices of that graph. Similarly, every element in the **edges** array is of type **Edge** and contains all of the edges of that graph. The **subvertices** is an array of numbers that represents all of the terminals of the Steiner Tree.

### 4.1.2 Services

A Service is an instance of the repository pattern. The repository pattern is normally used on the back-end, but it helps to keep all of the data access logic in one place. For example, CSUN hires a company to build them an application that uses CSUN’s data from a database. UCLA is informed about this application and would also want the same company to build it for them. Instead of the company searching through the entire code base and finding and changing data-access logic to work with UCLA’s database structure, they can simply update the repositories that are responsible for that data access logic. Another benefit of using services is you will not have repetitive code. If I did not use services, then I will have copied and pasted all of the graph-related logic I wrote for the Kruskal AV into Steiner AV.

Rather than allowing the Controller to directly manipulate the models and use the D3 and Cytoscape libraries, I created services to handle that logic. For this project, I created
three services: **D3Service**, **CytoService**, and **FileService**. All of the services are located in the “src/app/services” directory.

The D3Service was responsible for recursively drawing all of the leaves of the tree and removing them all. Even though only the Counting Inversions AV utilized the D3Service, creating the D3Service was beneficial because if anyone else decides to continue this project with implementing more sorting AVs, they won’t have to rewrite all of this logic.

Before the D3Service gets called, a string of numbers that the user inputs gets converted into an array of numbers. Each of these numbers gets placed into an **Element** model. An array of **Element** models gets placed into a **Node** model which becomes the root of the tree that the D3Service is going to draw recursively.

When the array gets split, two new subarrays denoted by L and R are added as children of the **Node**. L and R represent the left and right halves of the array in the **Node** model. Every step gets displayed as the algorithm builds the tree. Figure 4.2 shows an example of what the tree will look like after the tree has been built. After every split, the D3Service only displays the leaves of the tree to the user, so in Figure 4.2, only the green **Nodes** will be shown.

This will display how the array gets sorted, but I also had to keep track of the inversions count throughout each step of the way.

The way I changed Merge Sort into Counting Inversions is I added an **inversions** variable to the **Node** model. This will keep track of the number of inversions every subarray has. After all of the subarrays have been split, and before every merge, I added the inversions count of the left and right subarrays that are being merged. This will be the inversions count of the current array before all elements are merged in. Then, if the first element of the right subarray is merged before the first element of the left subarray, then I add the length of the left subarray to the current inversions count.
The only update I had to make to display the inversion count. I had to insert a new \texttt{div} tag with the inversions count centered inside it. The tag is display right above the array itself, which is the reason why we are inserting the inversions count before the array elements itself. 4.3.

Because both Kruskal and Steiner Tree AVs use Cytoscape, rather than copying and pasting all of the logic directly into both of the Controllers, I was able to just call the CytoService methods to perform the tasks I needed.

The CytoService was used to directly manipulate the Cytoscape graph, such as draw, add or remove vertices and edges, refresh, and retrieve all vertices and edges. I also added some methods that both the Kruskal and Steiner Tree AVs could reuse, such as the user inputting vertices and edges, calculating the total weight of the Kruskal Tree, generating random edges, resetting the graph, and checking whether or not adding an edge would create a cycle in the Kruskal Tree.

CytoService was initially implemented to only work with the Kruskal AV. After I started
creating the Steiner AV, I had to update the `vertexClickEvent` method. For the Kruskal Tree AV, a vertice would only be clicked on if the user was adding an edge, so it would change the color of the vertice to green and save it in memory. When another vertice was clicked, the color of both of the vertices would change back to black, and an edge will be added between those two vertices. However, for the Steiner Tree AV, the user had to be able to choose a subset of vertices by clicking on them, which would not work with the current implementation. Therefore, I added a new variable to the CytoService called `selectSub`. By default, the value of `selectSub` will be false, which allows the user to add an edge and edge weight to the graph. When the value of `selectSub` is true, and the user clicks on a vertex, it would add that vertice to the subset that the Steiner Tree AV will utilize.
Around the end of the development process, I allowed `selectSub` to accept a value of null. If the value of `selectSub` was null, then the graph would be disabled and the user would not be able to make changes to it. The graph would only be in this state after the user has clicked to run the algorithm.

Unlike the D3Service and CytoService, the FileService is fairly more straight-forward. It is in charge of converting a file uploaded string into the required model for the controller to use, or convert the model into a plain string that will be saved or downloaded. FileService also downloads a file with a given string as its content.

All three of these Services will be utilized by the Components.

### 4.1.3 Components

For this project, the components are the controllers. Each component is a TypeScript file that is associated with their own HTML and CSS files in the “app/src/{AV name}” directory. I created one component for every AV.

The Counting Inversions Controller deals with validating the user input, converting the user-inputted string into an array, building a tree by splitting every array into two subarrays, merging two subarrays, traversing the tree, and generating random input.

The Kruskal Controller deals with generating random edges, calling the Cytoservice to convert, save, and/or load the graph, creating example graph instances, getting and display-ing the Kruskal graph step-by-step, and getting the next/previous steps.

The Steiner Tree Controller uses and updates two instances of the CytoService: the current graph and the optimal graph. The controller also creates vertices, adds vertices and edges to the specified graph, allows the user to choose subvertices, runs and pauses the algorithm, updates vertex and edge colors in the the specified graph, generates all possible subsets that the algorithm will use to find the Steiner Tree, and finds the Kruskal Tree for a
given subset of vertices.

All of the controllers can change the speed of the algorithm, as well as pause or reset the instance. They also create example input for the user. The HTML that every TypeScript file is associated with plays the role of the view.

4.1.4 Views

As I mentioned earlier, each view is an HTML file that has their own controller in the same directory. The views are located in the “app/src/{AV name}” directory.

An example of how to display data from a variable to the HTML file is shown in Figure 4.4. First, we check if there are elements in the messages array passing messages.length into *ngIf. If messages does have elements inside, we loop through all of the elements and display them. We reverse the messages array so we display the most recent messages first. This is done by using the reverse() method. To loop through all of the elements, we pass let message of messages.reverse() into *ngFor. I also pass in let i = index because I want to change the color of the most recent message so they will see the step the algorithm took to get to its current state. This is done by conditionally binding the orange-message and dark-orange-message classes. The way to display a message is to place {{message}} in the tag, in this example it is the div tag.

```
<div *ngIf="messages.length" class="message-scroll-panel">
  <div *ngFor="let message of messages.reverse(); let i = index" class="message"
  [ngClass]="'{dark-orange-message': i === 0, 'orange-message': i !== 0}'"
  >{{ message }}<
  </div>
</div>
```

Figure 4.4: Displaying data on the HTML file

Another example of how the view calls the controller is through a button click, as shown in Figure 4.5. Initially, the number of vertices is 3, and {{ numVertices }} displays this to
the user. We bind the function `incrementVertices()` to a click event. The `incrementVertices()` function will increment `numVertices` by one. The value of `numVertices` on the first line of Figure 4.5 will update and show the number four. All three of the AVs use this logic.

![Code snippet](image)

**Figure 4.5: Button click event**

### 4.2 Implementation Issues

I noticed that the more AVs you create, the easier the implementation becomes. Because Steiner Tree was the second graph algorithm that I created an AV of, it was comparatively easier to implement because most of the graphing logic was already done. However, one major issue I encountered was displaying multiple Cytoscape graph instances on a single page.

#### 4.2.1 Multiple Graph Display

During the initial setup, I was dependency injecting two instances of the CytoService, so I can use two graphs, one as the current graph and the other as the optimal graph. However, I ran into an issue where when I updated the current graph, it would also update the optimal graph. This was happening because both of the services were pointing to the same Cytoscape instance as shown in Figure 4.6.

The way I resolved this issue was, instead of dependency injecting two CytoService instances into the constructor, I created two new CytoService instances inside the constructor as shown in Figure 4.7.
4.2.2 Drawing with D3

People normally use D3 to draw an SVG, which is what I initially tried to do. The code for how I set it up to draw the SVG is shown in Figure 4.8.
Figure 4.7: Creating two new instances of CytoService

However, I had to ask one of my classmates for help with drawing the SVG file, and it was going to be a huge learning curve for me, when I could focus all of that time on more important content related to this project. I decided to find a way to draw raw HTML elements instead of SVG, which is shown in Figure 4.9.
4.2.3 Pause and Continue

I added the pause and continue feature on the Kruskal and Steiner Tree AVs before Counting Inversions. The way the pause and continue feature works on the Kruskal and Steiner AVs is when the user clicks the **Pause** button, the algorithm saves the current state in memory and breaks out of the function. When the user clicks the **Continue** button, the algorithm re-runs the function from the saved state. Initially, I tried to add the same pause
and continue logic to Counting Inversions AV, but it wouldn’t work as well because I would have to save more information in memory, for the algorithm uses recursion, which is hard to stop and return to later on.
The new approach was to pause the algorithm without breaking out of the function. The way I did this was, when the user clicked the **Pause** button, the value of a variable called `paused` would toggle. If `paused` is set to true, then the algorithm will sleep infinitely until the user clicks the **Continue** button, as shown in Figure 4.10 by the `sleepWhilePaused()` helper method on line 138.

```
async sleepWhilePaused() {
    if(this.paused) {
        while(this.paused) {
            await this.d3Service.sleep(this.mergeSleepTime);
            if(!this.solving) {
                return;
            }
        }
    } else {
        await this.d3Service.sleep(this.mergeSleepTime);
    }
}
```

```
async merge(leftNode, rightNode) {
    let result = [];
    this.afterArray = [];
    this.displayBefore = false;
    this.displayAfter = false;

    await this.sleepWhilePaused();
    if(!this.solving) {
        return;
    }
}
```

Figure 4.10: Algorithm sleeps infinitely until the user clicks the **Continue** button

### 4.2.4 Refactor Reset

The user is allowed to reset the algorithm only when it is paused. If they do pause it, the algorithm is still running. The reason why is because the way the pause functionality works
for the Counting Inversions AV is that it does not break out of the function. The algorithm just sleeps infinitely until the user clicks the Continue button. This causes issues because when the user resets it and clicks Run Algorithm again, it will still run the old instance. I had to add some logic that will break out of the algorithm when the user clicks the Reset button. The way the algorithm stops and breaks out of the function is shown by lines 496 to 498 in Figure 4.11.

![Image of code snippet](image)

Figure 4.11: Using a helper method to exit a function

After every call to the sleepWhilePaused() function in Figure 4.10, I added those three lines to break out of the algorithm when the user clicks to reset.

I tried to refactor this logic and move it into the exit() function to avoid repetitive code. However, this caused issues because the “return;” only exits out of the current function it is
4.2.5 Graph Display

The initial plan for displaying the Kruskal Tree AV was to show two graphs, the original graph that the user has inputted and the graph containing the minimum-spanning tree (MST). After the user has inputted vertices and edges with weights for the graph and clicked Run Algorithm, the original graph will remain the same. The MST graph will initially have all of the vertices and no edges. As the algorithm runs, it will add edges to the MST graph. Once the algorithm has completed, the user can compare the original graph with the MST graph to see which edges in the original graph are missing in the MST graph.

However, I thought it would take the user longer to differentiate the missing edges, so I decided to use one graph for displaying both the original and MST edges. After the user clicks, Run Algorithm, the new plan is to display the original graph with the black vertices and black dashed edges with weights. As the algorithm starts running, and it is checking whether or not that edge creates a cycle, the styling of that edge will be solid and blue. If the edge creates a cycle, then we do not add it to the MST, so the styling of that edge will be dashed and gray. If the edge does not create a cycle, then we add it to the MST, and the styling of that edge will be solid and red, and the vertices the edge is connecting will turn red. Once the algorithm has finished running, the user will see one graph with dashed gray or solid red edges and red vertices.

4.2.6 Queue Usage for Next/Previous Step

When I was taking into consideration to allow the user to click the Previous and Next steps, the initial plan was to use two priority queues and set them up so they sort the edges based on the weight in ascending or descending order, depending on if the user wants to find the minimum or maximum spanning tree. The first queue will contain all the edges that have not been attempted to add to the MST graph, and the second queue will contain
all of the edges that have been added to the MST graph. However, the issue with this logic was that if many edges have the same weight, it will not maintain their order because it is a priority queue, which is set up to sort the edges within it. This issue occurred when the user clicked the Previous or Next buttons because the algorithm will remove an edge from one queue and add it to the next, which will not maintain the order.

The new plan is to use one priority queue, which is only be used to sort the edges, and two arrays. After the user clicks Run Algorithm, all of the edges in the queue will be pushed into the first array, so the first array will contain all edges that have not been attempted to add to the MST graph. After they have been attempted to be added, the edges will be placed in the second array. In this case, if the user clicks the Previous button, the algorithm will pop off the last edge from the second array and place it into a temporary variable, then that edge will turn solid blue in the display. If the user clicks Previous again, it will place that edge into the beginning of the first array. If the user clicks the Next or Run Algorithm button, the first edge at the beginning of the first array will be popped off and placed into the temporary variable and the display will update to show that edge as solid blue. When the user clicks Next again, the temporary edge will be placed to the end of the second array.

4.2.7 Acyclic Graph Check

In my data structures course, one of the projects we were given was, given a set of vertices and edges, implement Kruskal’s Minimum Spanning Tree. Before adding an edge to the Kruskal graph, you must check whether or not that edge would create a cycle. My initial plan was to use a similar implementation as the one I made in the data structures class, where an adjacency list will be used. An adjacency list is “an array A of separate lists” [17]. Every element in array A, denoted as Ai, is a list containing all of the vertices that are adjacent to vertex i [17]. Upon adding an edge to the Kruskal graph, I tried to use an adjacency list to determine whether or not that edge will create a cycle. However, this
implementation was a little complicated and took up at most $V^2$ amount of space, where $V$ is the number of vertices in the original graph.

After speaking with Professor Noga, he suggested to use an array of arrays, which would also seem to take up $V^2$ amount of space, but instead of adding to the array, we will be removing and combining elements of the array. The way this implementation would help determine if adding an edge will create a cycle in the Kruskal graph is, you would start off with an array $A$ of size $V$. Each element in $A$ will be an array, $A_i$, containing exactly one element, which will be the id of that Vertice. Upon inserting an edge $E$ to the Kruskal graph, combine the two elements in $A$ containing the ids of the vertices that $E$ is connecting. Using this strategy, the way to determine whether or not adding an edge will create a cycle is to check if the two vertices the edge is connecting are in the same element of $A$. The downside of this technique is that it is impossible to update $A$ after removing a Kruskal edge. However, a simple solution to this issue is to keep a history of changes for array $A$.

4.2.8 Updating Cytoscape Graph

When I first set-up Cytoscape for this project, after a vertex or edge was added to the graph, I would replace the graph with a new instance so that the graph would update the display. I tried to use the Cytoscape library methods to add an edge or vertex, but they didn’t initially work, so I didn’t prioritize this. I realized that by continuously recreating a Cytoscape instance, a lot of data had to be stored in memory in the CytoService to maintain the graph, such as all of the vertices and edges. After I changed the graph to update the initial instance, rather than delete and create a new instance, I was able to eliminate a lot of data that was stored in memory.
Chapter 5

Validation

Because this AV tool will be used by faculty and students of CSUN in the classroom, I wanted to ensure it would be an effective tool. The way I did so is I demoed the website to Professor Noga’s COMP 482 class and received their feedback. I also showed the project to my sister and asked for her feedback because she is also studying computer science at CSUN.

When I demoed the application to Professor Noga’s COMP 482 class, I received the following feedback:

• Display a history of messages for each step rather than one message describing the current step

• When the user has clicked on an edge to update the weight, change the line style instead of only the color to accommodate color blindness

• When the user has clicked on a vertex, change the shape of the vertex to accommodate color blindness

• Make the two Steiner Tree graphs a little bigger

• Name the optimal graph “Possible Optimal Graph”, and when the algorithm has finished running, change the name to “Final Optimal Graph”

• Move the messages to the left of the page

• Display what the colors and shapes signify for each of the AVs

Due to time constraint, I prioritized the list and made the following changes to the project:
• Display what the colors and shapes signify for each of the AVs

• When the user has clicked on a vertex, change the shape of the vertex to accommodate color blindness

• When the user has clicked on an edge to update the weight, change the line style instead of only the color to accommodate color blindness

• Name the optimal graph “Possible Optimal Graph”, and when the algorithm has finished running, change the name to “Final Optimal Graph”

• Display a history of messages for each step rather than one message describing the current step

From the list above, all of the changes were successful except the third bullet point. I was able to change the line style to dotted when an edge weight was being edited or an edge was being checked to be added to the Kruskal graph. However, on the Cytoscape graph, solid and dotted lines look very similar as shown in Figure 5.1. I also tried to make the line style of the edge be double instead of dotted, but double does not exist in the Cytoscape library.

I also showed this project to my sister, and she thought it was a very helpful tool. She is currently taking Professor Noga’s COMP 282 course, and he lectured on Kruskal’s Minimum Spanning Tree. After using my AV tool, she said that seeing a visualization of how the algorithm works was very helpful, especially being able to pause and continue whenever you get stuck. She said the colors in the graph helped to indicate what is happening at each step.
Figure 5.1: Solid and dotted edges look very similar in Cytoscape.
In this chapter, I will discuss how to use the application. I tried to keep the UI very simple and intuitive, so the users can easily learn how to use it.

### 6.1 Navigation Bar

The navigation bar has four links: Visual Algos, Counting Inversions, Kruskal Tree, and Steiner Tree. The Visual Algos link takes the user to the home page, and the other three links take the user to the corresponding AV, as shown in Figure 6.1.

![Simple navigation bar](image1.png)

**Figure 6.1: Simple navigation bar**

### 6.2 Menu

As shown in Figure 6.2, every AV has a built-in menu bar with the following buttons: File, Examples, Problem, Pseudo Code, and a question mark.

![Built-in menu buttons](image2.png)

**Figure 6.2: Built-in menu buttons**

Clicking on the File button will open a drop-down list, where the user can either select Save As Text File or Upload Text File, as shown in Figure 6.3.

If the user clicks on Save As Text File from the File drop-down list, a modal will open. Figure 6.4 is an example of the Save As Text File modal for the Steiner Tree AV. The user has the option of either downloading the text file onto their computer or copying the text.
that represents the Steiner graph input.

When the user clicks on the **Upload Text File** option from the **File** drop-down list, another modal will open, as shown in Figure 6.5 for the Steiner Tree AV. The user can either paste a graph instance in the text area or upload a text file. The rules for the input is displayed on the left.

The rules and format for uploading an input for every AV is different. For Counting Inversions, the input must be in the following format: \{0 for ASC or 1 for DESC order\} \{unsigned double numbers separated by exactly one space\}. For Kruskal and Steiner Tree
AVs, the input is very similar. An edge is inputted in the format: \{id of the edge target vertex\} \{id of the edge source vertex\} \{weight of edge\}. The first line of the Kruskal Tree AV must be \{number of vertices\} \{0 for Min or 1 for Max\}. Every line that follows will be an edge. The first line of the Steiner Tree AV must be in the format: \{total number of vertices\} \{subvertices separated by spaces\}. Every line that follows will be an edge.

The Examples drop-down list provides the user with example inputs to select and use, as shown in Figure 6.6 for the Counting Inversions AV. It comes with two examples: Sorted array \((N=5)\) and Maximum number of inversions \((N=6)\). The examples for the Kruskal Tree AV are two graphs: Tree graph \((V=5)\) and Complete graph \((V=4)\). The examples for the Steiner Tree AV include three graphs of sizes 6, 7, and 8. Overall, I tried to create examples for some edge cases, but my AVs do not cover all of them.

The user can also view the formal definition of an algorithm by clicking Problem in
the menu, which will open a different modal. An example of this is shown in Figure 6.7.

Pseudo code is also available for the user to view for every AV. A modal will open displaying it after Pseudo Code is clicked in the Menu. Figure 6.8 shows the pseudo code displayed in the modal for the Kruskal Tree AV.

The last option on the built-in menu is the question mark. The user can click on this if they need hints of how to use the AV. An example is shown in Figure 6.9 for the Counting Inversions AV.

6.3 Home Page

The home page will display a brief summary about the goal of the project and who the audience is, as shown in Figure 6.10. I also set up tabs of the algorithms I made an AV of.
When the user clicks on it, they will be able to watch a tutorial about how to use that AV. I updated the Github repository to have the tutorials, which are located in the “src/assets” directory. However, I did not have enough time to add them to the page.

Every AV has a problem generator and a solver. The problem generator allows the user
to input or select which problem they want the algorithm to solve. The solver solves the problem and displays the steps.

6.4 Counting Inversions

6.4.1 Problem Generator

When the user first visits the Counting Inversions AV, they will see the display in Figure 6.11. The problem generator is where the user inputs the problem they would like the AV to solve. The user can either type in an array, click the **Generate Random Input** button, or select an example from the **Examples** drop-down menu. As the user enters input, a yellow array will get updated at the center of the page as shown in Figure 6.12. If there is an error in the input, a red message will appear below directly below the input. Above every array will be the inversion count, which is zero by default. The user can also choose to either sort the array in ascending or descending order.
The **Run Algorithm** button will remain disabled until the user enters a valid array into the input. The **Slower** (÷2) and **Faster** (×2) buttons will remain disabled until the user clicks **Run Algorithm**. The speed that is directly underneath the **Run Algorithm** button tells the user how slow or fast the algorithm is going to run compared with the default speed of ×1.

### 6.4.2 Solver

The solver is where the AV begins to solve the problem. After the user clicks **Run Algorithm**, the **Run Algorithm** button will toggle into a **Pause Algorithm** that can be used anytime during the solving phase. If the array is sorted, an orange message will appear in the scroll panel on the right saying “Array is already sorted” as shown in Figure 6.13. If the array is not sorted, then the algorithm starts to run.
First, the array will recursively split in half until all of the arrays have only one element. Then, the two arrays that most recently split will change color. Figure ?? displays the left half is red, and the right half is blue.

![Figure 6.14: Changing the color of arrays to be merged to red and blue](image)

After a second, centered below the two halves, a zero will appear. Then, the inversion count of the left half will be added to the zero, followed by that of the right half. Elements will be brought down one by one, and this is the merging stage like the one shown in Figure 6.15. Notice that after the inversion counts are added to the new array, the original counts disappear from the page.

![Figure 6.15: Counting Inversions AV bringing down and merging elements](image)

If an element from the right half gets brought down, then the inversion count of the new array will be incremented by the size of the left half. If it's just the left element brought down, then the inversion count stays the same. Once all elements of the left and right halves have been merged, the new array gets moved up and changes back to yellow, and
this process continues.

The user can pause the algorithm, and a gray **Reset** button will appear. This reset button is only visible as long as the algorithm is paused. Once it continues to run, the reset button disappears again. If they click **Reset**, the AV resets to the inputted problem before it gets solved. At the end of the algorithm, the user can still reset or scroll through the history of messages. The most recent message is displayed at the top of the scroll panel, as shown in Figure 6.16.

![Message history](image)

**Figure 6.16:** Most recent message is at the top of the scroll panel

### 6.5 Kruskal Tree

#### 6.5.1 Problem Generator

When the user first visits the Kruskal Tree AV, they will see a display similar to the one in Figure 6.17. By default, the number of vertices are $V=3$, which is the minimum. The user can click the + or - buttons to adjust the number of vertices, but the maximum is $V=8$. Similarly, the algorithm is set up to find the minimum spanning tree by default, denoted by **Min**. The user can click on the **Max** selection, so the algorithm will find the maximum-spanning tree when it starts to solve the graph. The user can click on two vertices to add an edge between them. If an edge already exists, then it will remove that edge. After an edge
is added, they can type in a weight after clicking on an edge. The Backspace key is also set up.

### 6.5.2 Solver

When the algorithm begins to run, all of the edges get displayed below the graph sorted by their weight, as shown in Figure ???. The left-most edge is on the top of the queue and will be used next. When an edge is going to be checked whether or not it will create a cycle in the Kruskal graph, it gets removed from the queue, becomes yellow, and gets placed under **Current Edge**. In Figure 6.18, the color of the edge that is being checked turns blue.

![Figure 6.18: Check if edge creates a cycle in Kruskal Tree](image)

If it does not create a cycle, that edge becomes solid red as in Figure 6.19. If it does
create a cycle, then the edge becomes dashed again, but it grays out.

Figure 6.19: All edges are sorted based on the weight

Similar to Counting Inversions, the user can also reset the algorithm, as well as see the messages in the scroll-panel. They can also skip or go back to a step by clicking Previous and Next.
6.6 Steiner Tree

6.6.1 Problem Generator

When the user first visits the Steiner Tree AV, they will see the display in Figure 6.20. The problem generator for the Steiner Tree AV is very similar to the Kruskal Tree AV. The user can either input their own graph, generate random edges, or choose an example graph. One of the big differences is that they must input a subset of vertices. After they click on the Select SubVertices button and click on some vertices, they will see Figure 6.21. The vertices that are red and shaped like a star are the vertices that were added to the subset. At this state, the user also click Edit Graph to go back and continue to add/remove edges from the graph.

6.6.2 Solver

When the algorithm beings solving, a new graph shows up relatively smaller than the original, which will show the possible Steiner Tree in Figure 6.22. An array of all possible subsets that can span the Steiner Tree get generated and placed in the scroll-panel at the
Figure 6.21: Selecting a subset of vertices bottom right. The subset at the top of the scroll-panel will be added to the Current Graph and checked towards the Possible Steiner Tree graph.

Figure 6.22: Steiner Tree is solving

Once the algorithm finishes solving, the Possible Steiner Tree changes into Final Steiner Tree in Figure 6.23.
Figure 6.23: Steiner Tree is finished
Chapter 7

Future Work

In this chapter, I will discuss how this project can be improved in the future. Due to time constraint, I was not able to add all of the features I envisioned for this project.

This project is strictly developed to work in the browser of a computer or laptop. It would be helpful if students could open this website on their phone or tablets, or be able to download a mobile application of this project. It could also be expanded to cover more algorithms. The ultimate goal would be to create an AV for every algorithm in the “Algorithm Design” textbook, so faculty of CSUN can teach any of them and not feel limited [12].

One of the major issues I ran into and could not fix because it was already too late into the development process is the displacement of the vertices on the graph. Cytoscape simply plots the vertices, rather than placing them in a way that makes reading the graph easy. The only solution I found at the time was to use a different library, but there are not a lot of libraries that can display graphs. An option would have been to use D3 to draw those, but it was going to be very tedious because the developer has to specify where they want the vertex to be plotted. Graph theory itself is very complicated and can be another topic for a graduate project.

The Kruskal AV has the option of going to the previous step or next step, unlike counting inversions and Steiner Tree. If this functionality is to be added to counting inversions, it may be a good idea to break out of the algorithm when the user clicks the Pause Algorithm button. Otherwise, multiple instances will run simultaneously.

After I completed this project, I thought it would have been more beneficial for the user if they were asked to participate in the logic when the algorithm is solving. For example,
if counting inversions is on the merging stage, a message could show up asking the user how many inversions it is to merge a number $m$ before $n$ and display the correct answer afterwards.

It would also be nice to cover more edge cases, such as if a Kruskal or Steiner Tree is disconnected, a message would be displayed after solving the problem. I did not have enough time to implement this feature.

For one of the edge cases in Counting Inversions, if the array is sorted, a message will be displayed saying that the array is already sorted, and the algorithm will not run. I think another way this project can be improved is if the array is sorted, still display a message indicating that it is sorted, but allow the user to choose whether or not they would like to run the algorithm regardless. I think it is more effective this way because the user can see how there will be no inversions if the array is sorted.

When the Counting Inversions AV moves an element down during the merging stage, rather than that element entirely disappearing from the original array, it would fade out. This will help the user remember where that element got merged from and recognize the steps better.

The home page is missing a tutorial video for each AV because I did not have enough time to add them to the page, but they are uploaded to Github, as I have mentioned earlier. It would also be helpful if either the home page or the AV itself had explanations of what every color or shape represented in the AVs. For example, an explanation of what a blue edge or red edges represent in the Kruskal AV.
Chapter 8

Conclusion

Algorithms are an important concept in computer science and software engineering. An AV is an effective tool for teaching algorithms, rather than drawing complex diagrams and graphs on the white board.

The main goal of this project was to create an AV tool for the students and faculty of CSUN to use. I created an AV for three algorithms: Counting Inversions, Kruskal’s Minimum Spanning Tree, and Steiner Tree. Each AV has two modules, the problem generator and the solver. The problem generator allows the user to input to select a problem they would like the algorithm to run. The solver runs the algorithm and displays the changes to the user.

Some of the technologies I used for this project are Angular 7, Github, Cytoscape, and D3. I used Angular 7 and Github, so I can deploy this project on my personal website using Github Pages in the future. Cytoscape was used to draw and manipulate the Kruskal and Steiner Tree graphs. I used D3 to draw the Counting Inversions array, as well as the inversion count.

To make sure this project was working as intended, I gave Professor Noga’s class a live demo and they requested some changes. I prioritized the list of changes and was only able to update some of them, due to time constraint. I also showed my sister this project because she is majoring in computer science at CSUN. She said that Professor Noga lectured on Kruskal’s minimum spanning tree, but she was a little confused because it was hard to visualize.
References


