2021

GRAPH PLAYGROUND: A PEDAGOGIC TOOL FOR GRAPH THEORY AND ALGORITHMS

Neeraj Adhikari
University of Rhode Island, neeraj@uri.edu

GRAPH PLAYGROUND: A PEDAGOGIC TOOL FOR GRAPH THEORY AND ALGORITHMS

BY

NEERAJ ADHIKARI

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2021

MASTER OF SCIENCE THESIS

OF

NEERAJ ADHIKARI

APPROVED:

Thesis Committee:

Major Professor Edmund A. Lamagna

Noah M. Daniels

William B. Kinnersley

Brenton DeBoef

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2021

ABSTRACT

A system called *Graph Playground* has been created to enhance the teaching of introductory graph theory and graph algorithms presented in courses like CSC 340, Applied Combinatorics, and CSC 440, Design and Analysis of Algorithms, at the University of Rhode Island. The primary goals of the system are to allow visual creation and manipulation of graphs, and to provide a set of algorithms that can be applied to graphs with intermediate steps shown as visual decorations of the graph. The features and capabilities of the system are described, with demonstrations of the application of algorithms on example graphs. The design and implementation details of the system are also discussed, along with discussion of challenges encountered in the implementation and how they were solved. The system is designed to be easily extensible for new algorithms, and a breakdown of the steps needed to implement a new algorithm is presented.

ACKNOWLEDGMENTS

I would like to extend my deepest gratitude to my major professor, Dr. Edmund A. Lamagna. His help and expert guidance, from the inception to completion of both the thesis and the software, has been truly invaluable. It has been a pleasure to work with him and take his classes at URI.

I am extremely grateful to Dr. Noah Daniels and Dr. Bill Kinnersley for their immensely useful feedback and comments on the thesis and the software. I would also like to thank Dr. Mike Barrus for chairing my thesis committee and helping bring the thesis defense to a successful completion.

Thanks also to my friend Safar Ligal, who contributed the brilliant idea of using generators to decouple the execution aspects of algorithms from their implementation.

I would like to express my appreciation to my parents, Kumud Adhikari and Ranjana Dahal, and my sister, Neha Sharma, for their ever-present love and encouragement. Finally, I would like to thank Aasara Khatiwada for her love, her support during stressful times and for always motivating me to keep working.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES viii

CHAPTER

1 Introduction 1

1.1 Previous Work 2

1.1.1 GraphTea 2

1.1.2 Graph Online 2

1.1.3 Graph Magics 2

1.1.4 D3 Graph Theory 3

1.2 Issues Uncovered 3

1.2.1 User Interface and User Experience 3

1.2.2 Lack of Openness 4

1.2.3 Limited Features 4

1.3 Overview of the Document 5

List of References 5

2 Features of Graph Playground 7

2.1 Deployment and Access 7

2.2 User Interface Overview 8

2.3 Creating and Manipulating Graphs 14

Page

2.3.1 Creating Graphs 14
2.3.2 Moving Vertices and Edges 16
2.3.3 Editing Vertex Labels 17
2.3.4 Editing Weights for Weighted Graphs 17
2.3.5 Deleting Vertices and Edges 18
2.3.6 Saving and Loading Graphs 18
2.3.7 Bookmarking and Opening Bookmarked Graphs 20
2.3.8 Manipulating Tabs 20
2.3.9 Using Auto-Layout Options 21
2.4 Using the Tool to Demonstrate Isomorphism 23
2.5 Using the Tool to Demonstrate Planarity 26
2.6 Algorithms 27
2.6.1 The Algorithm Control Panel 28
2.6.2 Executing Algorithms 30
2.6.3 Minimum Spanning Tree Algorithms 31
2.6.4 Search Algorithms 33
2.6.5 Shortest Path Algorithms 36
2.6.6 Walk-Finding Algorithms 39
2.6.7 Decomposition Algorithms 41
2.6.8 Exact TSP Algorithms 44
2.6.9 TSP Approximation Algorithms 46
2.6.10 Network Flow Algorithms 55
List of References 56

Page

3 Design and Implementation of Graph Playground 59

3.1 Overview of Design 60

3.2 Implementation 63

3.2.1 Core Graph Interface and Classes 63

3.2.2 Drawing Classes 66

3.2.3 Layout Interface and Classes 68

3.2.4 Decorator Interface and Classes 70

3.2.5 Algorithm Interface and Classes 72

3.2.6 UI Handlers and Components 73

3.3 Choice of Tools and Technologies 77

3.3.1 Platform 77

3.3.2 Programming Language 77

3.3.3 Frameworks and Libraries 78

3.4 Design and Implementation Problems Encountered 80

3.4.1 Separation of Rendering and Representation of Graphs . 80

3.4.2 Separation of Algorithm Implementation from Rest of the System 81

3.4.3 Placement of Edge and External Vertex Labels 84

3.5 Algorithm API 86

3.6 Extending the System with a New Algorithm 90

3.6.1 Creating an Algorithm Class 90

3.6.2 Programming the Algorithm 91

3.6.3 Using the Decorator and Yield 92

3.6.4 Registering the Algorithm with the System 95

Page

3.6.5 The Result 95

List of References 97

4 Conclusions 98

4.1 Review of Goals Achieved 98

4.2 Future Work 99

BIBLIOGRAPHY 101

LIST OF FIGURES

Figure		Page
1	A screenshot of the user interface.	9
2	The user interface with the various areas labeled.	10
3	The left and right sidebars	11
4	The new graph dropdown menu	14
5	The 'Graph' dropdown menu in the top bar	19
6	Demonstrating isomorphism with the system	24
6	Demonstrating isomorphism with the system (continued)	25
7	Demonstrating planarity of a graph	27
8	The algorithm control panel, showing Kruskal's algorithm ready to execute	28
9	The algorithm selection dropdown in the top bar	31
10	Kruskal's algorithm working on a 4-vertex graph	32
11	Prim's algorithm working on a 4-vertex graph	34
12	Breadth First Search working on a 4-vertex graph	35
13	Depth First Search working on a 4-vertex graph	37
14	Dijkstra's Algorithm working on a 6-vertex graph	38
15	Fleury's Algorithm finding an Euler Trail	40
16	Bellman-Held-Karp Algorithm finding a Hamilton Circuit	42
16	Bellman-Held-Karp Algorithm finding a Hamilton Circuit (continued)	43
17	The Hopcroft-Tarjan algorithm finding an articulation point and two biconnected components	45

Figure **Page**

18 Bellman-Held-Karp Algorithm Finding the Optimal TSP Tour . 47

19 TSP Approximation with Nearest-Neighbor Heuristic 48

20 TSP Approximation with Nearest-Insert Heuristic 50

21 TSP Approximation with Cheapest-Insert Heuristic 51

22 TSP Approximation with MST-Based Algorithm 52

23 TSP Approximation with Christofides Algorithm 54

24 Edmonds-Karp algorithm for computing network flow 56

25 UML Class Diagram of the Graph Interface 64

26 UML Class Diagram of the Weighted Interface 64

27 UML Class Diagram of the GraphDrawing Class, showing only public methods 67

28 UML Class Diagram of the Layout abstract class 68

29 UML Class Diagram of the Decorator interface 69

30 UML Class Diagrams of the Algorithm Interface and the AlgorithmOutput Interface 72

31 UML Class Diagram of the AlgorithmRunner class, showing only public methods 73

32 Edge labels (weights) and external vertex labels on a graph . . . 85

33 The DFS-based component counting algorithm in action 96

CHAPTER 1

Introduction

This document will describe a new web-based application called *Graph Playground* (henceforth also called *the application* or *the system*) created by the author to facilitate and enhance the pedagogy of classes in which graph theory or algorithms on graphs are taught. The application provides a way to visually create graphs, manipulate created graphs by moving or deleting vertices or edges, save graphs locally and, most importantly, apply a variety of algorithms to graphs along with step-by-step visualizations.

The inspiration for the selection of the problem came from the author's experience with teaching CSC 340, Applied Combinatorics, at the University of Rhode Island during the Fall 2020 semester. The class was taught online and conventional methods of demonstrating concepts like the planarity and isomorphism of graphs were found to be insufficient, especially in view of the support for visualization and interaction that is possible with modern technologies. Similar problems were encountered with the demonstration of graph algorithms.

After initial exploration, the author reached the conclusion that no existing freely available tool met the specific needs of the course. As a result, a basic browser-based application was created using the Cytoscape.js [1] library. The library provided support for drawing vertices, edges and labels of a graph, and exposed vertices and edges as objects in the API. Building on this, the author implemented creation and editing of graphs, storage of the graphs using the LocalStorage API[2] found in modern web browsers, redrawing and automatic labeling. *Graph Playground* is an expansion of the feature set and capabilities of that initial tool, but does not build upon it — it was written from scratch.

1.1 Previous Work

There are a number of tools available that attempt to solve the same problem. Many of them are open-source and available at no cost. This application attempts to be a user-friendly and customizable alternative to those tools while including demonstrations of all the important graph concepts and graph algorithms covered in a typical CSC 340 or CSC 440 class at URI.

A few of the major tools of this type are listed below, along with a short introduction to their features and the shortcomings found in them.

1.1.1 GraphTea

GraphTea [3] is a Java-based desktop application that facilitates teaching, learning and research on graph theory. It was first released in 2014 and it supports a wide variety of graph algorithms and operations along with creation, storage and manipulation of graphs. The application is open-source and the source code is available on GitHub.

1.1.2 Graph Online

Graph Online [4] is a web application that lets users create graphs, either in a WISYWIG manner or using an adjacency matrix. The application allows the user to apply any of a collection of common algorithms to the graphs. It was created by the authors of GraphAnalyzer[5], which is an application that lets users visualize graphs and graph algorithms. Graph Online is open-source and the source code published under the MIT License is available on GitHub.

1.1.3 Graph Magics

Graph Magics [6] is a desktop application that supports graph creation, graph generation and application of 17 different algorithms on graphs. The feature set

appears to be comparable to GraphTea.

1.1.4 D3 Graph Theory

D3 Graph Theory [7] is a web application that serves as an interactive introductory graph theory course. The content is divided into small units and each unit consists of a web page with a text lesson and an interactive area where the learner can draw graphs and get feedback and/or visualizations related to the content of the unit. The user interface is appealing and the interactive parts of the lessons are intuitive. D3 Graph Theory is open-source and the source code is available on GitHub under an MIT License.

Despite the existence of similar systems, the application developed by the author adds value to the space as no comparable tool provides a complete package with a good user interface, easy availability and a rich feature set. Some of the issues uncovered in existing systems and the factors differentiating *Graph Playground* from them are discussed in the next section.

1.2 Issues Uncovered

1.2.1 User Interface and User Experience

An important issue with most existing systems was found to be lack of ease of use and an intuitive user interface.

For instance, with GraphTea, besides having to download and install its desktop application, the user is presented with a complex and clunky user interface. Even though it offers a large number of features with regard to customizability and algorithms, having to download and install executables is a hurdle that many users are unwilling to go through, especially when there are web-based alternatives that are more convenient to access.

Graph Online is superficially similar to *Graph Playground*, but presents a

frustrating experience when creating graphs: users have to go through a dialog box with options for edge weight, label, and directedness to create an edge. This can be fairly time-consuming even for a moderate-sized graph. Users can include undirected or directed edges and weighted or unweighted edges in the same graph, which is a source of confusion.

1.2.2 Lack of Openness

Another issue found with at least one application is lack of openness. Graph Magics is distributed under a proprietary license and is not free of cost. From the author's viewpoint, this is a major shortcoming for a pedagogical tool. *Graph Playground* is open-source (licensed under GNU GPL[8]) and is available free-of-cost, and is designed to be extensible by exposing a well-defined API for users who wish to add their own algorithm implementations.

1.2.3 Limited Features

A third issue found in the existing systems was the lack of certain features desirable for CSC 340 and CSC 440 coursework.

Graph Online has some notable omissions in its feature set. For instance there is no support for automatic creation of common graphs like complete graphs, wheel graphs, etc. The user can apply 17 different algorithms to graphs but algorithms for exact and approximate solutions to the Traveling Salesman Problem are missing.

D3 Graph Theory serves as a guided course in graph theory. Consequently, it provides limited control over graph drawing and actions the learner may perform on the graph. In other words, it is not a general-purpose application for graph creation, manipulation and graph operations.

1.3 Overview of the Document

Chapter 1 discusses previous work, i.e. existing systems that try to achieve a similar goal and motivates the need for *Graph Playground*. Additionally it presents an overview of the document.

Chapter 2 is a complete description of the features of the system, describing various parts of the user interface, creation and manipulation of graphs, and execution of algorithms. All algorithms implemented in the system are described with an example execution shown through images.

Chapter 3 describes the design and implementation of the system. A high-level overview and a detailed description of the important modules, interfaces and classes of the system are given. Various design and platform choices are motivated. Challenges encountered and solutions developed are described. The chapter ends with a step-by-step breakdown of how to extend the system with the implementation of a new algorithm.

Chapter 4 is the conclusion, where the work done is reviewed and compared with the project goals, and possible future work is discussed.

List of References

[1] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, "Cytoscape.js: a graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 09 2015. [Online]. Available: https://doi.org/10.1093/bioinformatics/btv557

[2] The Web Hypertext Application Technology Working Group. "HTML Standard web storage." Accessed: 2021-07-02. July 2021. [Online]. Available: https://html.spec.whatwg.org/multipage/webstorage.html

[3] M. A. Rostami, A. Azadi, and M. Seydi, "Graphtea: Interactive graph self-teaching tool," *Communications, Circuits and Educational Technologies*, 01 2014.

[4] "Graph online." Accessed: 2021-07-02. [Online]. Available: http://graphonline.ru/en

[5] "Graphanalyzer." Accessed: 2021-07-02. [Online]. Available: http://grafoanalizator.unick-soft.ru/en/

[6] "Graph magics." Accessed: 2021-07-02. [Online]. Available: http://www.graph-magics.com/index.php

[7] "D3 graph theory - learn graph theory interactively." Accessed: 2021-07-02. [Online]. Available: https://d3gt.com/index.html

[8] Free Software Foundation. "Gnu general public license." [Online]. Available: http://www.gnu.org/licenses/gpl.html

CHAPTER 2

Features of Graph Playground

This chapter will present a detailed overview of how the system is used: from deploying and accessing the software to creating graphs and running algorithms on them.

2.1 Deployment and Access

The system is a frontend-only web application, as discussed in detail in Chapter 3. It uses the npm package manager [1] for package management and the webpack build system [2] for bundling assets together and producing deployable output. Given that the host system has npm installed, one can simply `cd` into the project's root directory in a terminal and use the command `npm install` to install all the dependencies, and then use the command `npm start` to run the build script. The build script runs the typescript compiler, the webpack bundler, runs a local web server to serve the output files and opens the application in the default browser. The same steps can be adapted to deploy the system to any HTTP web server, for example to make the system publicly accessible from the Internet. The system doesn't require any other capability from the server except that it should be able to serve files. Since the system is implemented entirely as a frontend application, no client-server communication happens after the system is loaded into a browser.

2.2 User Interface Overview

The user interface consists of 6 major areas, illustrated in Figure 2. An unlabeled view of the user interface is given in Figure 1. A short description of each area follows:

- **Top Bar:** The top bar contains the name of the system (Graph Playground) on the left side and two drop-down menus to the right of the title. The drop-down menus can be clicked to reveal a list of options. The 'Graph' drop-down menu contains options relating to the actions that may be applied on a graph, like saving the graph, bookmarking the graph, opening a new graph, etc. The 'Algorithm' drop-down menu contains a list of algorithms that can be applied to graphs. The algorithms are described in Section 2.6.

- **Left Sidebar:** The left sidebar contains two major areas: the toolbar and the bookmarks bar. The toolbar contains 'tools' similar to the tools found in image editing applications, which can be used to perform different operations on the graph using a mouse. The use of tools is discussed in Section 2.3. The bookmarks bar shows a list of graphs that have been 'bookmarked' by the user, i.e. saved locally in the browser. Users can open or delete bookmarked graphs through the bookmarks bar. Two bookmarked graphs, one named 'Components Test' and another named 'Graph One' are visible in Figure 1. An image of the just the left sidebar is in Figure 3 (i).

- **Right Sidebar:** The right sidebar contains three areas: auto-layout buttons, the display customizer and the auto-label scheme selector. The auto-layout buttons provide four different ways of laying out the vertices of a graph. Auto-layout options are discussed in more detail in Section 2.3.9. The display customizer contains two elements: the vertex size slider, which

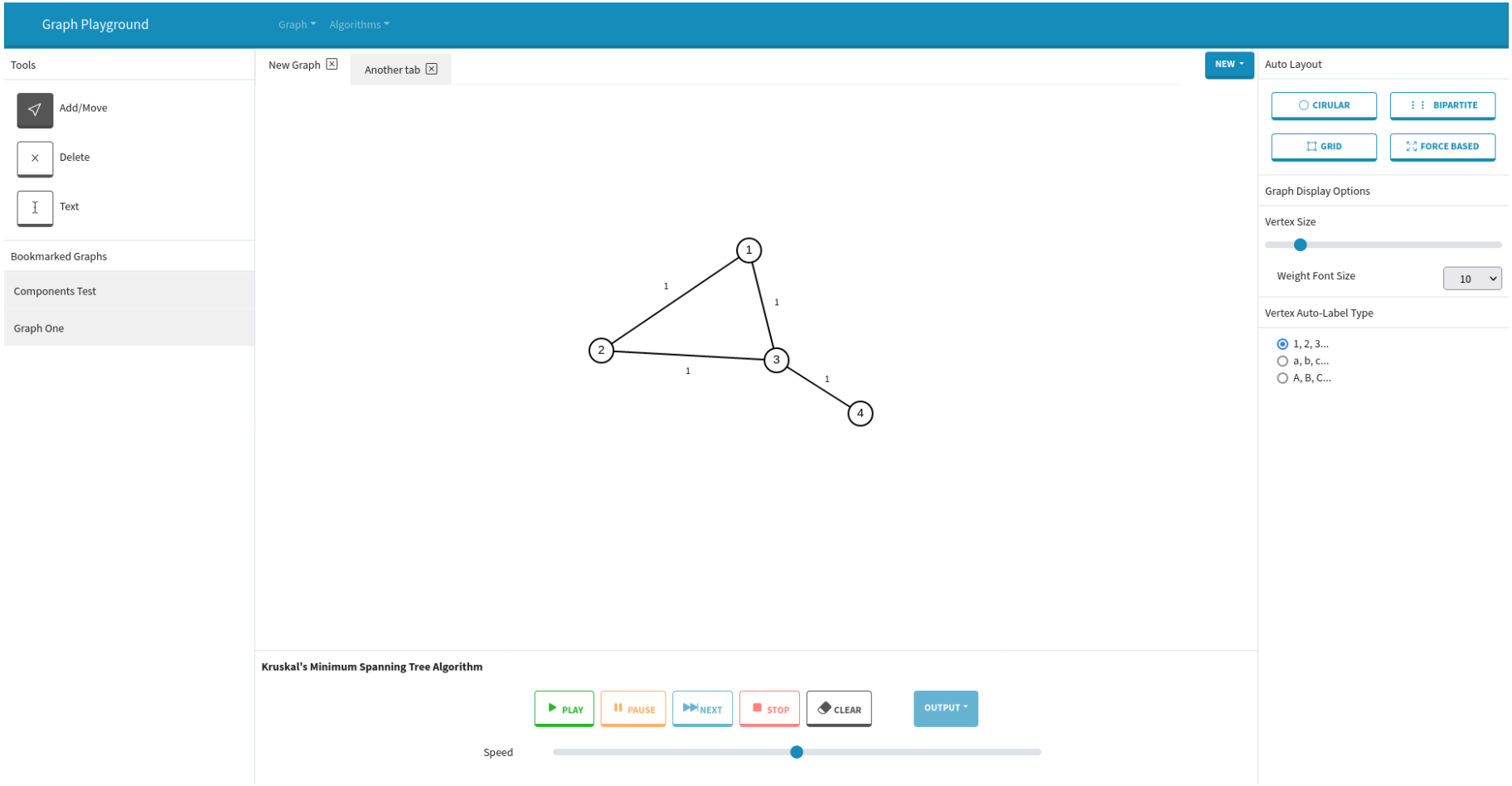


Figure 1: A screenshot of the user interface.

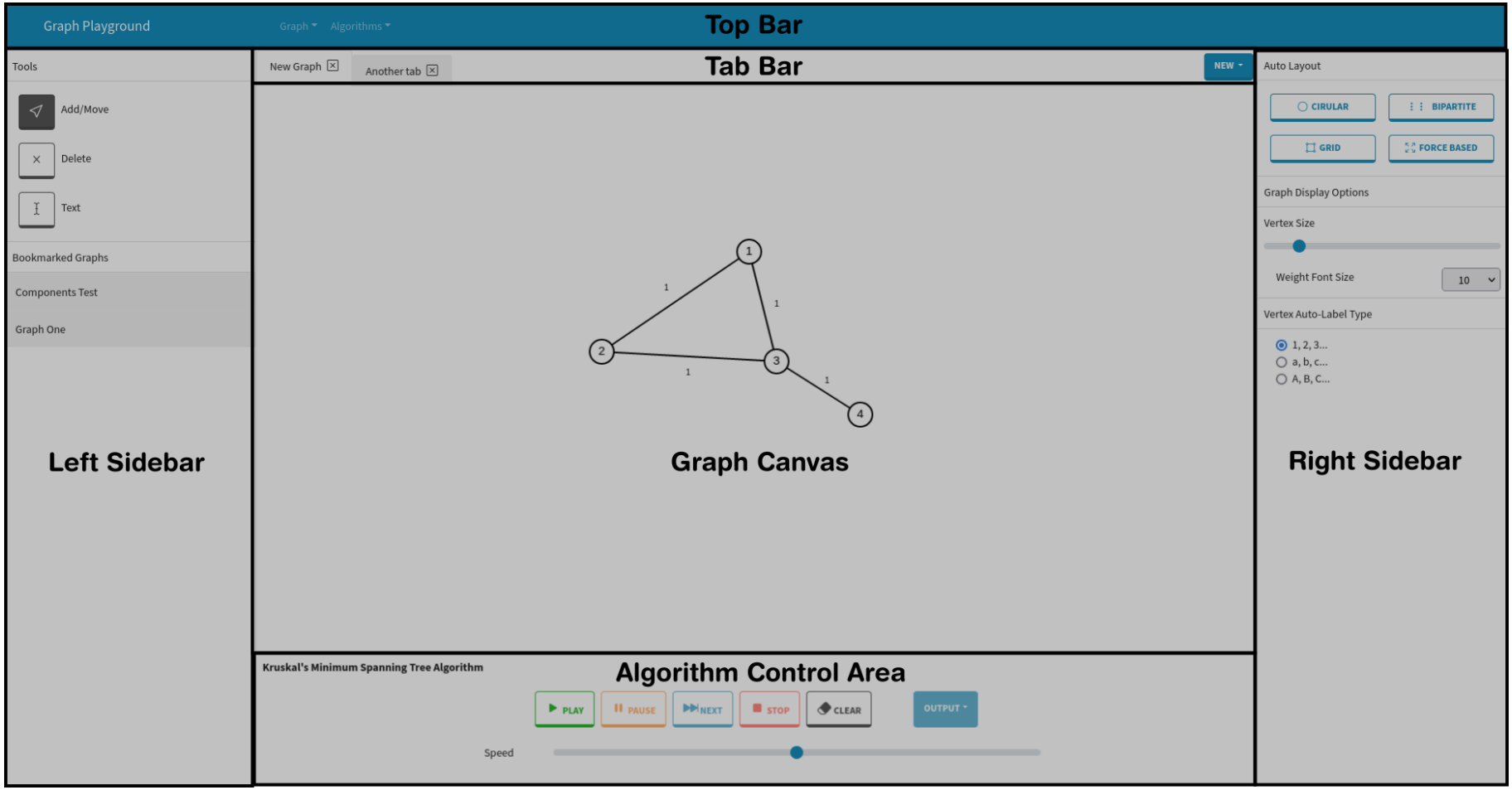
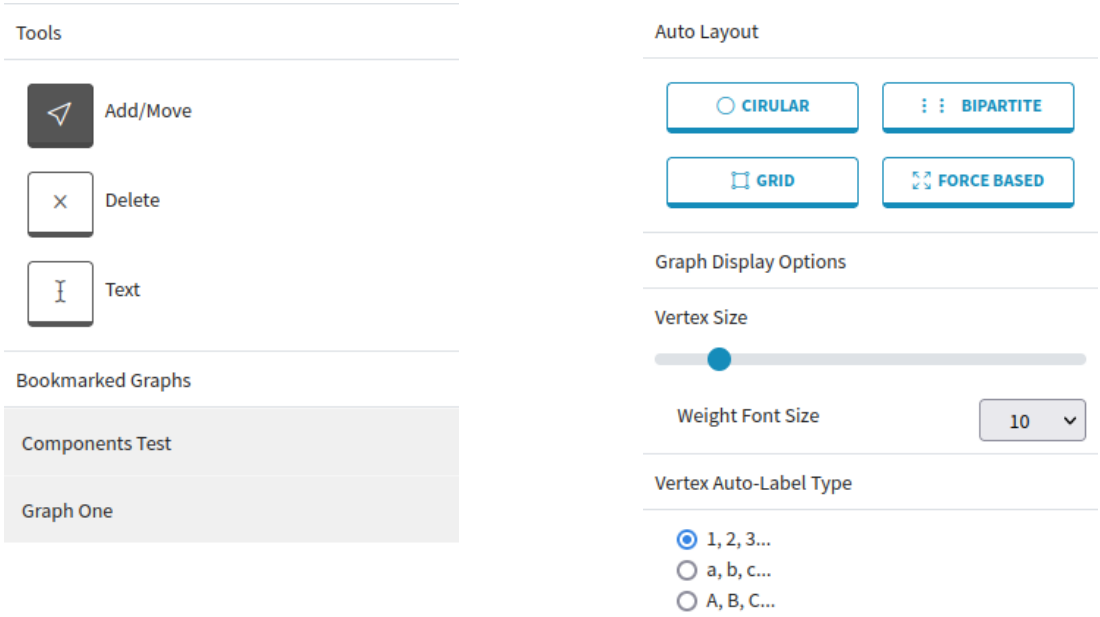


Figure 2: The user interface with the various areas labeled.



(i) The left sidebar (ii) The right sidebar

Figure 3: The left and right sidebars

allows users to select the size of the vertices from a fixed range, and a weight font size selector which allows user to set the font size of edge weights for weighted graphs. The auto-label selector allows users to select one of three auto-labeling schemes: numeric (1,2,3...), uppercase alphabetical (A,B,C...) and lowercase alphabetical (a,b,c...). Auto-labeling schemes are discussed in Section 2.3.1. An image of the the right sidebar is in Figure 3 (ii).

- **Graph Canvas:** The graph canvas is the central area of the user interface, and the area where most user interaction takes place. It is the area where graphs are displayed and where graphs can be created, manipulated and edited.

  The graph canvas is seen displaying a weighted graph with 4 vertices and all edge weights equal to 1 in Figure 1.

The graph canvas additionally contains two user interface areas that are overlaid on it. Towards the bottom left of the graph canvas is the notification area, where notification boxes are shown to the user for messages that are typically generated by algorithms (for instance about the success or failure of algorithm executions). The notification area is used for messages and alerts that are lower-priority than messages shown using the browser's alert dialog feature. The other area overlaid on top of the graph canvas is the status line, which is at the bottom center of the canvas. This area is used to display text indicating the 'status' of the algorithm currently running, and can contain arbitrary text set by the algorithm.

- **Tab Bar:** The tab bar is the thin strip above the graph canvas that contains tabs currently open in the system. The system provides users the ability to multi-task by having more than one graph open, and by quickly being able to switch between the tabs. Only one tab can be active at a time and the contents of the active tab is displayed in the graph canvas. At the right end of the tab bar, there is a small 'NEW' drop-down button that can be clicked to reveal a list of options the user can select to create new tabs with various different kinds of empty graphs. The 'NEW' menu is described further in Section 2.3.1.

  The tab bar in Figure 1 can be seen displaying two open tabs, titled 'New Graph' and 'Another tab' respectively. The tab titled 'New Graph' is the currently active one.

- **Algorithm Control Area** The algorithm control area is the area below the graph canvas. Initially, the area is empty. When the user selects an algorithm from the menu in the top bar, the algorithm control area is populated with a control panel that provides a set of buttons and sliders which can be

used to control the execution of the algorithm on the graph. Executing and controlling the execution of algorithms is discussed in Section 2.6.

In Figure 1, the algorithm control area can be seen containing the control panel for Kruskal's Minimum Spanning Tree algorithm.

2.3 Creating and Manipulating Graphs

2.3.1 Creating Graphs

New graph types

To create a graph, the user can click on the 'NEW' button just above the top-right corner of the graph canvas, i.e. at the rightmost end of the tab bar. The new graph menu contains options for 5 different types of empty graphs that can be created, as shown in Figure 4.

- Undirected unweighted graph
- Undirected weighted graph
- Directed unweighted graph
- Directed weighted graph
- Euclidean graph

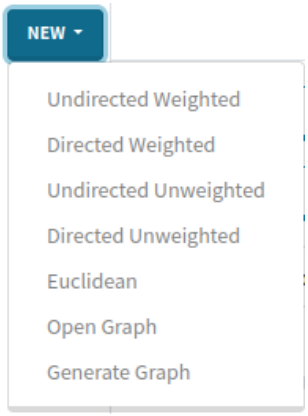


Figure 4: The new graph dropdown menu

The first four items in the above list are fairly self-explanatory. They provide options to create graphs that have either directed or undirected edges, and either weighted or unweighted edges. Currently, the system does not support creation of multigraphs and graphs with loop edges. The fifth item in the above list warrants

some discussion: it is a graph where the edges are implicit and only vertices are displayed. The 'edge weights' are the distances between the corresponding vertices. Certain algorithms in the system, like the algorithms for solving the Traveling Salesman Problem, only operate on Euclidean Graphs.

Once a user clicks on one of the new graph menu items, a new tab with the title 'New Graph' and an empty graph is created and activated.

Adding vertices

Once there is a graph active in the graph canvas, the user can start adding vertices. To add a vertex, it is important that the 'Add/Move' tool be selected from the toolbar, which is located in the left sidebar. After the Add/Move tool is selected, the user can click on any point inside the graph canvas to add a vertex there. The added vertex will be labeled by a default label that follows the selected auto-label scheme. For instance, if the numeric auto-label scheme is selected, the first vertex added will have a label of '1'. The second will have a label of '2', the third '3' and so on. When the user switches to a different auto-labeling scheme, the vertices added will follow that scheme, adding a vertex with the 'earliest' label that does not already exist in the graph.

Connecting vertices to create edges

The vertices that are already present can be connected to form edges in the graph. To connect two previously non-adjacent edges, the user needs to:

(i) Make sure the Add/Move tool is selected in the toolbar

(ii) Click on the source vertex. The vertex will be highlighted in blue to indicate that the vertex is in 'edge add' mode.

(iii) Click on the destination vertex.

After this the graph should have an edge between the two vertices. Of course, the distinction between 'source vertex' and 'destination vertex' only matters for directed graphs. If the graph is a weighted graph, then a default weight of 1 will be assigned.

By default, a newly added edge is displayed as a straight line between its two incident vertices. Curved edges are sometimes desirable, and moving edges to make them curved is described in Section 2.3.2.

2.3.2 Moving Vertices and Edges

Moving vertices can be simply accomplished by clicking on a vertex and moving the mouse without releasing the mouse click, (i.e., dragging the mouse). Again, the 'Add/Move' tool needs to be selected in the toolbar for vertex moving to be possible. When a vertex is moved, all the edges incident to the moved vertex move with the vertex, as might be expected by the user.

As mentioned in the previous section, edges are by default drawn as straight lines. However, curved edges can be important in several cases, like when a planar graph such as K_4 needs to be drawn as a plane graph. To move an edge (while keeping the incident vertices stationary), the user needs to:

(i) Make sure that the Add/Move tool is selected in the toolbar.

(ii) Click on the edge to add a 'curve point' on the edge. The curve point will be displayed as a small white circle that is visible when the mouse pointer is directly over the edge.

(iii) Drag the curve point using the mouse. The drawing library used by the system (KonvaJS[3]) creates a spline that runs through the curve point.

2.3.3 Editing Vertex Labels

As discussed earlier, vertices are automatically labeled according to the selected auto-label scheme, but vertex labels can be edited after a vertex is created. To edit a vertex label, the user needs to:

(i) Make sure that the Text tool is selected in the toolbar. Selecting the text tool should change the cursor from the default arrowhead shaped pointer to a 'text-edit cursor' used in most text-editing applications.

(ii) Double-click on the vertex label. Doing this should select the label text and enable it for editing.

(iii) Input the text for the new label and hit the Enter key.

An important point to note about vertex labels is that while the auto-label schemes create unique vertex labels by ensuring an already-used vertex label is not used again for a newly created vertex, the user is free to create multiple vertices with the same label by editing vertex labels. This could potentially be a source of confusion, so it is advised that the user exercise care when intentionally creating duplicate labels.

2.3.4 Editing Weights for Weighted Graphs

In weighted graphs, the edge weight is displayed just beside the mid-point of the edge. The edge weight can be edited by the user almost exactly like the vertex labels. Specifically, to edit an edge weight, the user needs to:

(i) Make sure that either the Add/Move tool or the Text tool is selected in the toolbar. In either case, a 'text edit cursor' will be displayed instead of the normal cursor when the user hovers the mouse on top of the edge weight.

(ii) Double-click on the edge weight. Doing this should select the weight text and enable it for editing.

(iii) Input the new weight and hit the Enter key. Since the weight can only take on numeric values (either integers or floating point values), an error will be shown if the entered weight does not represent a valid numeric value.

2.3.5 Deleting Vertices and Edges

Vertices and edges in the graph can be deleted by using the Delete tool from the toolbar. To delete a vertex or an edge, the user needs to:

(i) Make that the Delete tool is selected in the toolbar. The cursor should take the shape of a cross when it is inside the graph canvas.

(ii) Click on the object to be deleted. For vertices, this means clicking inside the vertex's circular area. For edges, the width of the region where the edge detects clicks is slightly larger than the width of the edge drawn. So the edge will be deleted if the user clicks at a point sufficiently near the edge's line.

When a vertex is deleted, all of the edges incident to it are deleted as well.

2.3.6 Saving and Loading Graphs

The ability to save graphs as files and open saved graphs is an integral feature of the system. Since the system is entirely frontend-based and doesn't have the ability to save graphs 'in the cloud', saving graphs as files on the local computer is really the only permanent way to save graphs. Graphs can also be 'bookmarked' as described in Section 2.3.7, but bookmarking graphs is a less permanent way to save them 2.3.7.

To save the graph that is currently active on the canvas, the user needs to click the 'Graph' dropdown menu in the top bar and then click 'Save' in the menu

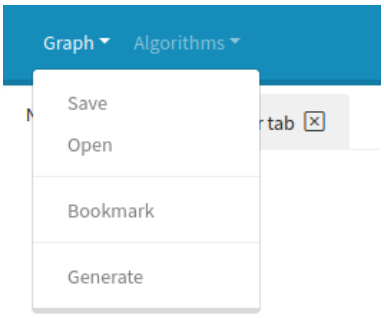


Figure 5: The 'Graph' dropdown menu in the top bar

that drops down. This dropdown menu is shown in Figure 5. Clicking the 'Save' button will open a dialog box where the user can enter a file name for the graph file, which by default is the name of the tab the active graph is in. The system appends a `.json` extension to the provided file name, as the graph is stored in the popular JavaScript Object Notation (JSON)[4] format. When the user clicks on the 'Save' button in the dialog box, the output file is presented as a download in the browser. From that point browser-specific polices about handling downloads apply, common ones being saving the file to the user's Downloads folder or asking the user for a folder to store the file.

To open a saved graph, the user needs to click the 'Graph' dropdown menu in the top bar and then click 'Open' in the menu that drops down. This will cause the browser to open the default interface for file uploads for web pages, which usually involves showing a file picker to select a file. Once the user selects a file, the file is read and, if it is in the correct format, it is interpreted as a graph. A new tab is created with the title extracted from the file name and the graph is displayed in the tab.

2.3.7 Bookmarking and Opening Bookmarked Graphs

The system provides a 'bookmark' feature that allows users to save graphs directly on the browser. Bookmarked graphs are saved in the browser locally using the Local Storage API[5]. This means that while bookmarking graphs is faster and more convenient than saving the graph as a file, bookmarks can be lost if the user clears the browser's history. Furthermore, graphs bookmarked on one browser are not available on another browser.

To bookmark the currently active graph, the user needs to click the 'Graph' dropdown menu in the top bar and then click 'Bookmark' in the menu that drops down. This will cause the graph to be added as a bookmarked graph, and displayed as an item in the bookmarks bar in the left sidebar.

The bookmarks bar displays a list of all currently bookmarked graphs. The name used in the bookmarks bar is the title of the graph's tab when the user bookmarked the graph. Tabs and tab titles are discussed further in Section 2.3.8. The user can click on an item in the bookmarks bar to open the bookmarked graph. A new tab will be created with the bookmarked graph and the new tab will be activated. Hovering the mouse on an item in the bookmarks bar will also display a 'DELETE' button towards the right of the bookmarked graph's name, which can be clicked to delete a bookmark. Once deleted, the bookmarked graph is lost and cannot be recovered, unless the user has the graph already open in a tab or has saved the graph elsewhere.

2.3.8 Manipulating Tabs

The system enables multi-tasking with the help of tabs, similar to the tab interface found in most modern web browsers. Each tab has a graph associated with it, and the currently open tabs are displayed in the tab bar. The user can click on any tab to activate that tab, which will cause the associated graph to be

displayed in the graph canvas. A tab can be closed by clicking on the small cross button displayed on the right side of the tab.

The tab bar also allows the editing of tab titles. The user can double-click on the tab's title to send it into 'edit mode', which means that the text of the title will be highlighted and will accept input from the keyboard. After entering the new title, the user can hit the Enter key to confirm the title.

The user can also drag and drop the tab to place it in a different position with respect to other tabs.

2.3.9 Using Auto-Layout Options

The system provides four auto-layout options for graphs. The buttons for applying the auto-layout options can be found in the right sidebar, as shown in Figure 3 (ii). Once a button is clicked with a graph active in the graph canvas, the system will examine the graph to compute vertex positions and assign the positions to the graph. The four auto-layout options are:

Circular Layout

This auto-layout option lays out the vertices in a circle at equally spaced points along the circumference. The sequence used for laying out the vertices is based on the internal identifiers of the vertices, which corresponds to the order the vertices were created. The vertices are positioned in clockwise fashion starting from the rightmost end of the circle.

Bipartite Layout

The bipartite layout option places the vertices on two vertical columns. If the graph is bipartite, the layout will be properly bipartite, i.e. vertices will be placed in such a way that there are no edges between any two vertices in a column.

Grid Layout

The grid layout positions the vertices on a rectangular grid, trying to achieve a square shape when possible.

Force-Based Layout

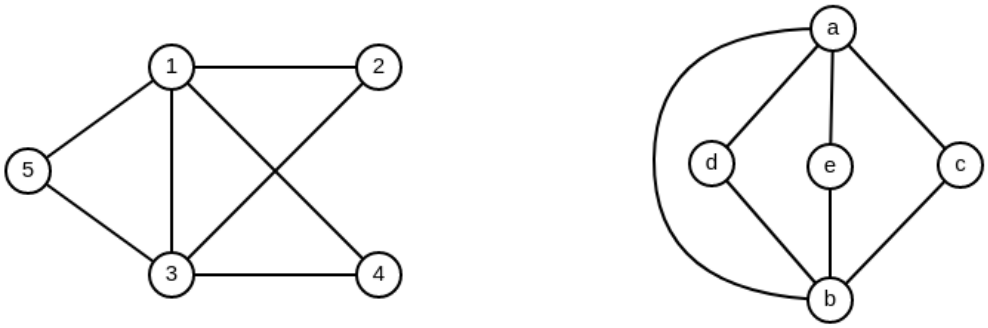
The force-based continuous layout is an implementation of an algorithm by Eades [6]. The algorithm was implemented from the description presented in [7]. In general terms, the algorithm works by assigning attractive (spring) forces between adjacent vertices and repulsive vertices between non-adjacent vertices. The algorithm iteratively re-computes and applies forces on the vertices, and the updated positions are shown dynamically in the graph canvas. Changing the graph by adding or removing vertices or edges will cause the positions of the vertices to change as expected.

2.4 Using the Tool to Demonstrate Isomorphism

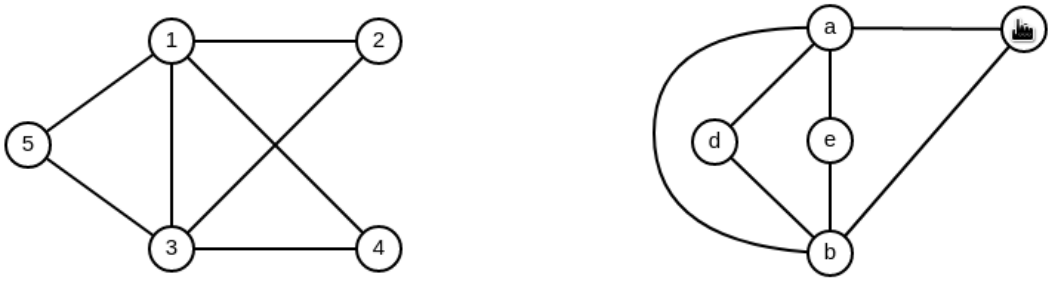
Isomorphism is one of the first topics taught in a course or unit on graph theory like CSC 340. Demonstrating isomorphism was also one of the primary motivations for the tool that was the precursor to the system. In this section, a description of the process that can be used to demonstrate that two graphs are isomorphic is presented. In this context, 'demonstrating isomorphism' should be interpreted to mean the process of manipulating the drawing of one of two graphs to make the manipulated graph's drawing visually identical to the other graph's, convincing the viewer—or at least making it easier to verify—that the two graphs are indeed equivalent.

Since the system only supports displaying one graph at a time, we will demonstrate isomorphism between two different connected components of a single graph, considering them to be different graphs for the purposes of the presentation.

Figure 6 displays the steps for an example where two pre-created graphs are demonstrated to be isomorphic. Figure 6 (i) shows two graphs side-by-side and, even though the graphs have the same number of vertices and the same number of edges, it might not be obvious they are isomorphic. In Figure 6 (ii), vertex c of the right-side graph is dragged to the upper right, to place it at a position analogous to vertex 2. In Figure 6 (iii), vertex e is dragged towards the bottom right corner, placing it at a position similar to vertex 4 in the left graph. Then in Figure 6 (iv), the edge (a, b) is grabbed by its curve point, and the figure includes the depiction of a 'grabbing' mouse pointer. Finally, in Figure 6 (v), the edge (a, b)'s curve point is moved to make a straight line between vertices a and b, and the two graphs have an identical depiction, disregarding the vertex names. Thus we have demonstrated the two graphs are isomorphic.

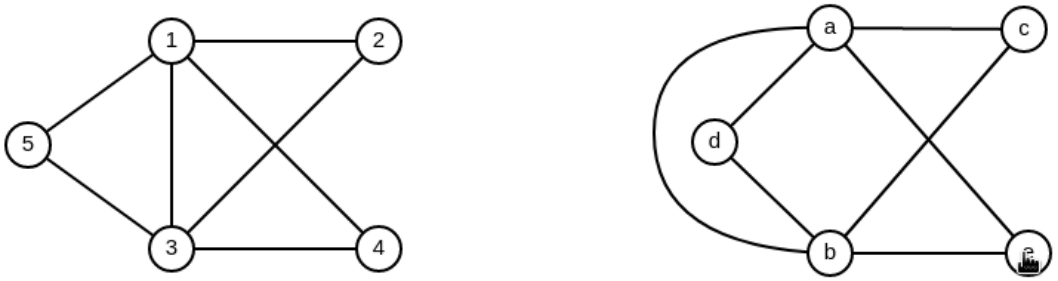


(i) Two graphs for which isomorphism is not readily obvious

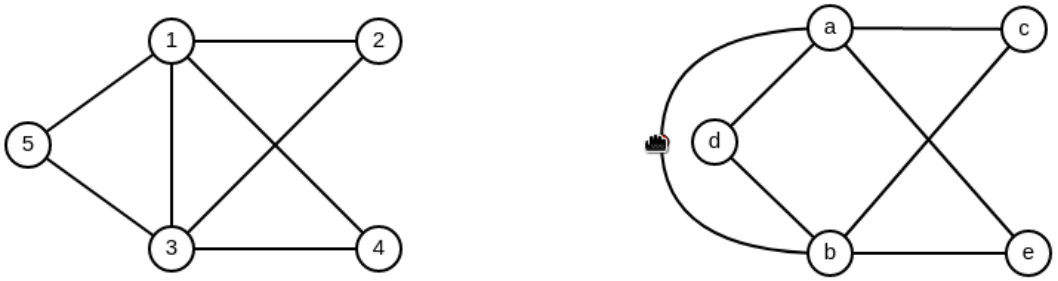


(ii) Moving vertex c to the top right

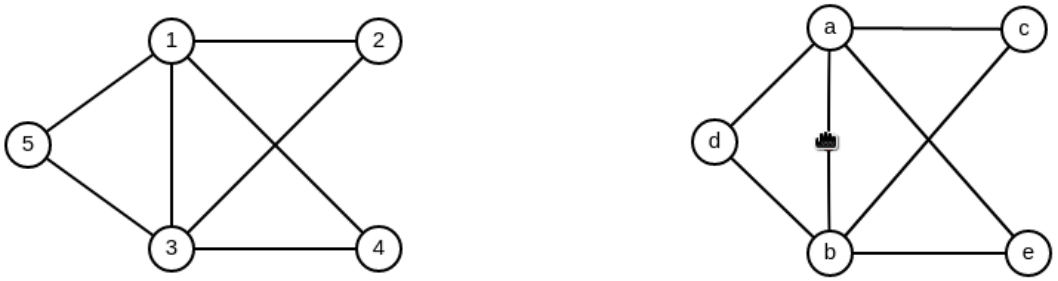
Figure 6: Demonstrating isomorphism with the system



(iii) Moving vertex e to the bottom right



(iv) Grabbing edge a-b by its curve point



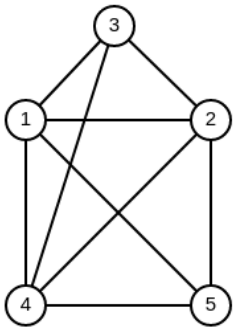
(v) Moving edge a-b to make it a straight line

Figure 6: Demonstrating isomorphism with the system (continued)

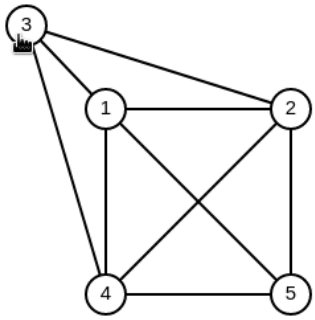
2.5 Using the Tool to Demonstrate Planarity

Planarity is another important topic in graph theory, and one that is introduced soon after isomorphism in a course like CSC 340. In this section, a description of the process that can be used to demonstrate that a graph is planar is presented. In this context, 'demonstrating planarity' is interpreted to mean the process of manipulating a non-planar depiction of a graph initially containing edge crossings to obtain a graph depiction with no edge crossings If we don't add or remove vertices or edges during the process, the transformation of the drawing into one without edge crossings should convince the viewer that the graph is planar.

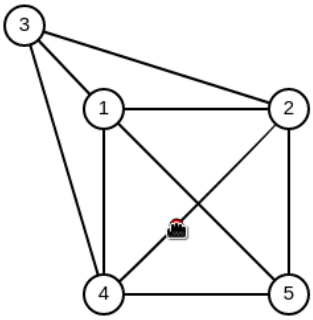
Figure 7 shows the steps for an example where a graph is demonstrated to be planar. The initial drawing of the graph contains 3 edge crossings and is depicted in Figure 7 (i). In the next step, we drag vertex 3 towards the left, as shown in Figure 7 (ii). Now just one edge crossing remains, and we remove the crossing by first creating a curve point on edge $(2, 4)$ by clicking on the edge, as shown in Figure 7 (iii), and then dragging the curve point outside the graph as shown in Figure 7 (iv). Now the graph drawing has no crossings and we have demonstrated the graph is planar.



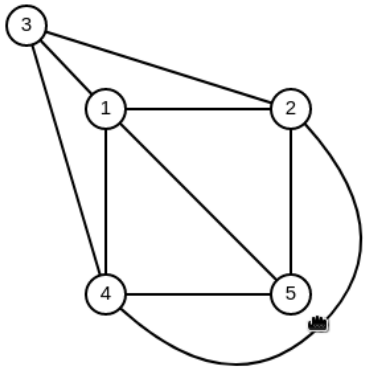
(i) A graph with a non-planar depiction



(ii) Moving vertex 3 to the top left



(iii) Clicking on edge 2-4 to add a curve point



(iv) Dragging egde 2-4 to remove crossing with edge 1-5

Figure 7: Demonstrating planarity of a graph

2.6 Algorithms

The central feature of the system is its ability to run algorithms on graphs and visualize the intermediate steps and the results. This section presents overviews of the application of all the algorithms implemented in the system, describing the visualisations produced by the algorithms and the output they generate. Before delving into the descriptions of the specific algorithms, an overview of the algorithm control panel is presented in Section 2.6.1, since the algorithm control panel is used

to control the execution of all algorithms.

2.6.1 The Algorithm Control Panel

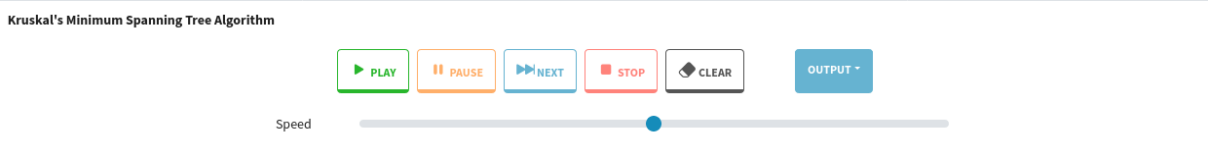


Figure 8: The algorithm control panel, showing Kruskal's algorithm ready to execute

The algorithm control panel appears just below the graph canvas when the user clicks on a name of an algorithm from the dropdown menu in the top bar. The algorithm's name is displayed on the top left of the algorithm control panel. An image of the algorithm control panel is shown in Figure 8. The control panel contains the following controls.

The Play Button

The play button is the leftmost button in the button row of the control panel. If the execution of the algorithm has not yet started, clicking on the play button starts the algorithm. If the execution of the algorithm has been paused, clicking on the play button resumes the execution.

The Pause Button

The pause button is located to the right of the play button on the button row. Clicking the pause button while the algorithm is executing will pause the execution of the algorithm. When the algorithm is not executing, the pause button is disabled and does not respond to clicks.

The Next Button

The next button is located to the right of the pause button on the button row. Clicking on the next button while the algorithm is paused will move the execution of the algorithm forward by one step. When the algorithm is running or stopped, the next button is disabled and does not respond to clicks.

The Stop Button

The stop button is located fourth from left in the button row and it stops the execution of the algorithm when clicked. The stop button can be used to stop the execution of the algorithm when the algorithm is either running or paused. After an algorithm execution is stopped, clicking play will start it from the beginning.

The Clear Button

The clear button is located fifth from left in the button row. It is enabled only after the execution of the algorithm has ended, either by itself or after the user has clicked the stop button. When it is enabled, clicking on it causes any changes on the graph's appearance made by the algorithm to be removed and the graph is restored to its default appearance.

The Output Button

The output button is the rightmost button in the button row in the algorithm control panel. It is enabled when the algorithm finishes running. For algorithms that produce an output graph, the output button can be used to open the output in a new tab or save the output graph as a file. The button is a drop-down button that exposes the two previously mentioned options for extracting the output of an algorithm.

If the user clicks on the 'New Tab' option from the drop-down, the output

graph will be opened in a new tab. Clicking on the 'Save as File' option causes the save file dialog to open, which allows the user to save the output graph as a file, similar to the process described in Section 2.3.6.

The Speed Slider

The speed slider is a horizontal HTML range input [8] that allows users to drag a circle along the horizontal axis to set the speed the algorithm executes. The specifics of how the algorithm speed is set and how the 'stepping' of the algorithm is implemented is discussed in Section 3.5. In general terms, setting the slider to its rightmost position corresponds to a delay of approximately 1 millisecond between the iteration steps of the algorithm, and setting the slider to its leftmost position corresponds to a delay of approximately 2 seconds. The range is linear, so setting it halfway sets a delay of about 1.0005 seconds.

With the algorithm control panel introduced, we can now proceed to a discussion of how algorithms can be executed on the system.

2.6.2 Executing Algorithms

To execute an algorithm on a graph, the user needs to first select the algorithm by opening the dropdown menu labeled 'Algorithms' in the top bar and clicking on the name of an algorithm. The dropdown menu is shown in Figure 9. Provided that a graph is active in the graph canvas, a control panel for the corresponding algorithm will appear in the algorithm control area below the canvas. Algorithm control panels attach to specific graphs and a control panel opened for a graph will not stay in the control area when the user switches to a different tab. If the just-activated graph has a control panel associated with it, that control panel will be displayed in the control area.

Once the algorithm control panel is visible, the user simply clicks the 'Play'

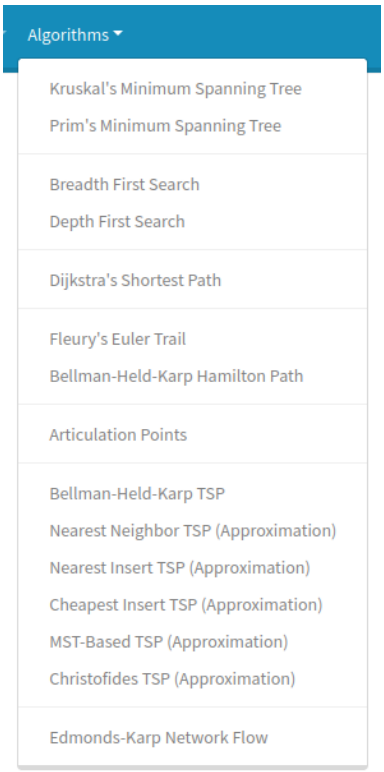


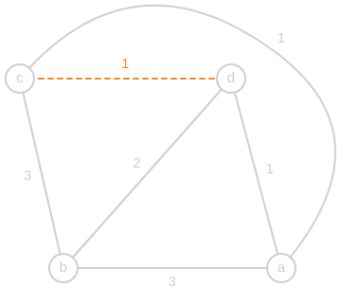
Figure 9: The algorithm selection dropdown in the top bar

button to start the algorithm. Algorithms that do not take any input (except the graph itself) will start executing immediately. Algorithms that take one or more vertices as inputs (for example, Breadth First Search) will ask the user to click on the starting vertex and then will begin execution. In the demonstrations of the implemented algorithms that follow, mention of starting the algorithm will be omitted and only the algorithm's effects on the graph will be displayed.

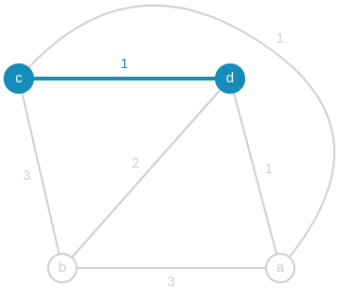
2.6.3 Minimum Spanning Tree Algorithms

Kruskal's Algorithm

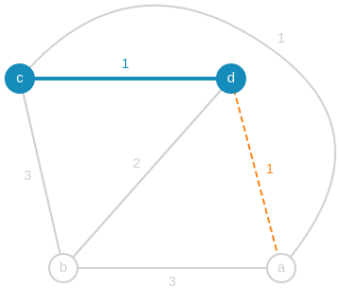
Kruskal's Algorithm [9] is an algorithm for finding the minimum-weight spanning tree in an undirected, weighted graph. It is a greedy algorithm that works by iterating through the edges in ascending order of weight. The algorithm starts by creating a forest with each vertex of the graph in a separate tree by itself, and



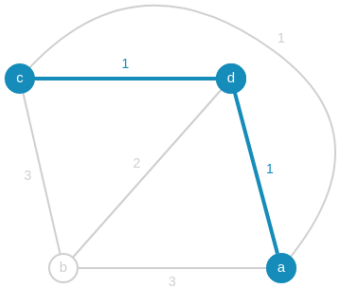
(i) Examining edge (c, d)



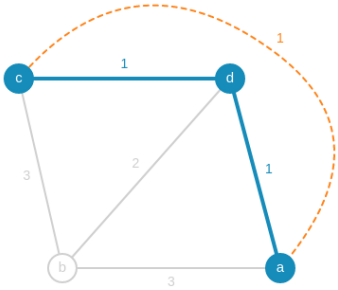
(ii) Add edge (c, d) to tree



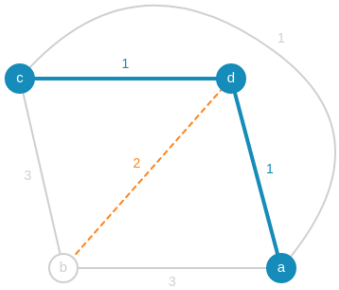
(iii) Examine edge (d, a)



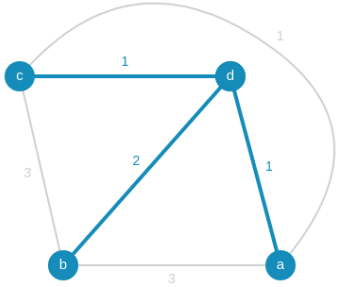
(iv) Added edge (d, a) to tree



(v) Examine edge (c, a). Creates circuit.



(vi) Examine edge (d, b)



(vii) Add edge (d, b) to tree. MST complete

Figure 10: Kruskal's algorithm working on a 4-vertex graph

single tree in the forest—the minimum-weight spanning tree.

A demonstration of the execution of Kruskal's Algorithm on a small graph with 4 vertices and 6 edges is presented in Figure 10. Vertices and edges added to

the spanning tree are set to the `SELECTED` state, which is displayed by coloring the vertices and edges teal. In the four-vertex graph shown in the figure, the minimum spanning tree we obtain contains edges (c, d), (d, b) and (d, a), and the total weight is 4.

Prim's Algorithm

Prim's Algorithm [10] is another procedure for finding the minimum-weight spanning tree in an undirected, weighted graph. Like Kruskal's algorithm, it is also a greedy method. It works by starting with a one-vertex tree and successively adds vertices until we have a tree containing all of the vertices. An arbitrary vertex can be selected as the starting point (the user clicks on this in our implementation). At each step, the procedure adds the least-weight edge (and the corresponding vertex) joining a vertex already in the tree with one not yet included. When all vertices have been added, we are left with the minimum spanning tree—if the graph does not have more than one component.

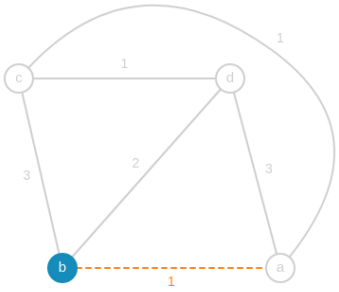
A demonstration of the execution of Prim's Algorithm on a small example graph is presented in Figure 11, which shows the visual decorations the algorithm makes to the graph. The algorithms starts at an arbitrarily selected vertex (b in this case) and keeps on adding vertices to the partial tree. This graph varies from the graph of Figure 10 in its edge weights, and we can see in Figure 11 (vi) that a minimum spanning tree is highlighted, this time with a total weigh of 3.

2.6.4 Search Algorithms

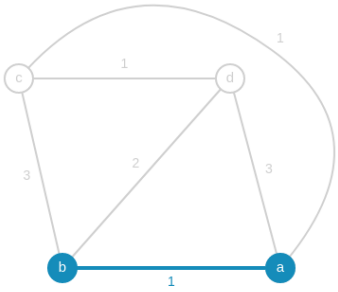
Breadth First Search

Breadth First Search is a simple and popular algorithm for searching for nodes in order of increasing depth from the root node of a tree.

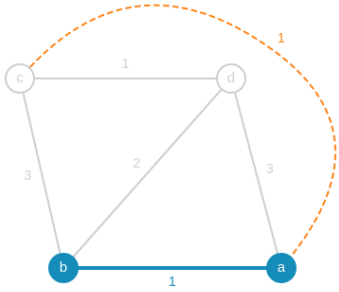
In the case of general graphs, Breadth First Search can be applied to create



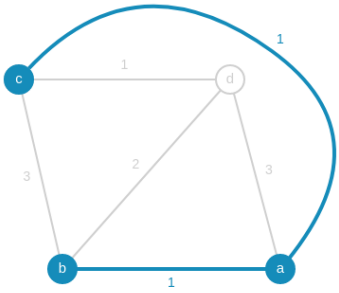
(i) Start with vertex b, examine edge (b, a)



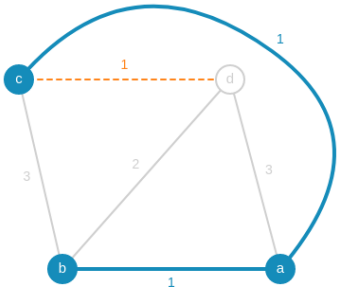
(ii) Add (b, a) to tree



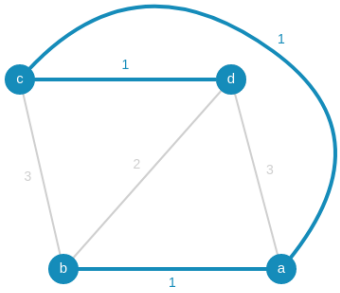
(iii) Examine edge (c, a)



(iv) Add edge (c, a) to tree



(v) Examine edge (c, d)



(vi) Add edge (c, d) to tree. MST complete.

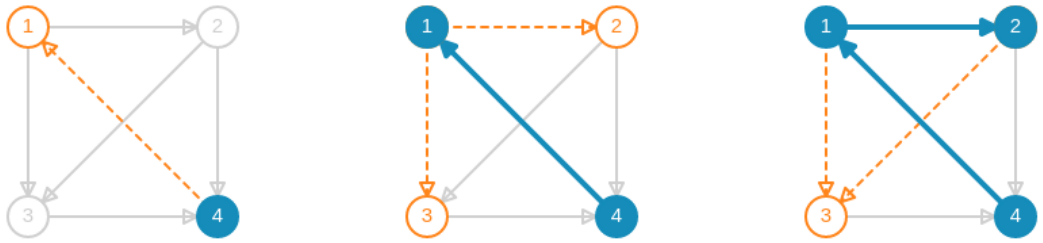
Figure 11: Prim's algorithm working on a 4-vertex graph

a Breadth First Search or Spanning Tree of the graph, which is a tree rooted at a starting vertex and with paths along the tree representing the shortest paths to other vertices from the starting vertex.

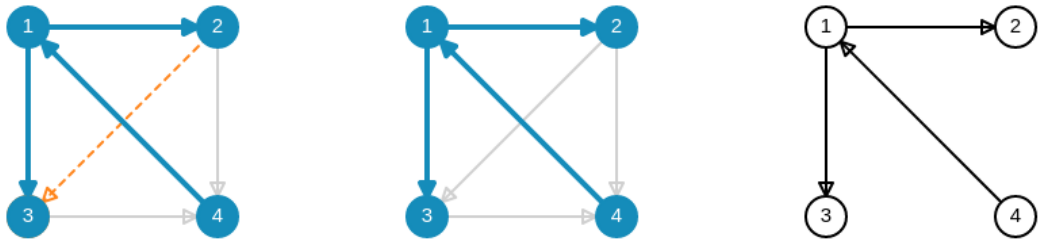
Here, 'shortest path' between a pair of vertices refers to the minimum number of edge traversals required to travel from one vertex to the other. That is, for weighted graphs, the edge weights are not considered.

Breadth First Search works by using a FIFO queue data structure: a vertex is removed from the queue to be explored, and all of its children not already in the tree are inserted into the queue. This ensures that all vertices at a given distance d from the start vertex are explored before any vertex at distance $d + 1$.

A demonstration of the execution of Breadth First Search Algorithm on a small example graph is presented in Figure 12, along with the visual decorations the algorithm makes to the graph. In this case the user has selected vertex 4 as the start vertex, and the algorithm proceeds by adding vertices to the search tree in order of their distance from the start vertex.



(i) Start with 4, add 1 to queue (ii) Explore 1, add 2 and 3 to queue (iii) Explore 2, add 3 to queue



(iv) Explore 3, add nothing to queue (v) BFS exploration complete (vi) The search tree output by BFS

Figure 12: Breadth First Search working on a 4-vertex graph

Depth First Search

Depth First Search is another simple and popular algorithm for searching the nodes of a graph. Implementation-wise, the only significant difference from Breadth First Search is that Depth First search uses a LIFO stack data instead of

a FIFO queue.

Doing so alters the vertex traversal order in such a way that the algorithm explores as far as possible along a single branch before backtracking and exploring other branches. The application of Depth First Search creates a Depth First Search or Spanning Tree of the graph.

A demonstration of the execution of Depth First Search Algorithm on a small example graph with 4 vertices is presented in Figure 13, along with the visual decorations the algorithm makes to the graph. In this case the start vertex provided by the user is vertex 2, and the traversal proceeds as we would expect. When the algorithm finishes we end up producing a search tree of depth 3, which is not surprising since depth first search maxes out the depth it can reach before backtracking to traverse other branches. In this case, there are no other branches and the tree is a path graph.

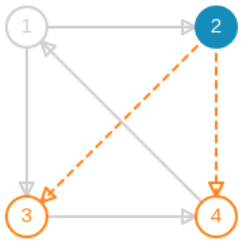
2.6.5 Shortest Path Algorithms

Dijkstra's Algorithm

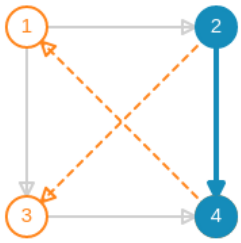
Dijkstra's Algorithm [11] is an procedure for finding the shortest path between vertices in a weighted graph where weights represent the distance between two vertices. More specifically, Dijkstra's algorithm finds the distance from a given starting vertex (called the *source*) to all other vertices in the graph.

The algorithm keeps track of all vertices that have not been fully explored and explores the one with the least known distance from the source. 'Exploring' a vertex involves using the distance from source to that vertex (and the edge weights) to update the currently known shortest distances of its neighbors. Once a vertex has been explored, it is not explored again and its distance from the source does not change. The algorithm ends when all vertices have been explored.

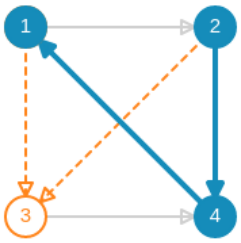
A demonstration of Dijkstra's Shortest Path algorithm running on an example



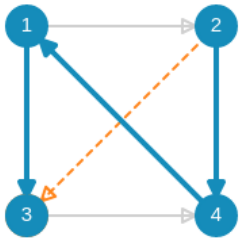
(i) Start with 2. Push 3 and 4 to stack.



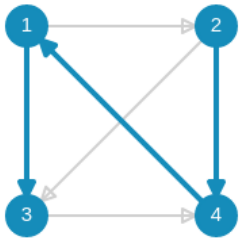
(ii) Explore 4. Push 1 to stack.



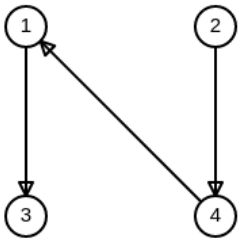
(iii) Explore 1



(iv) Explore 3



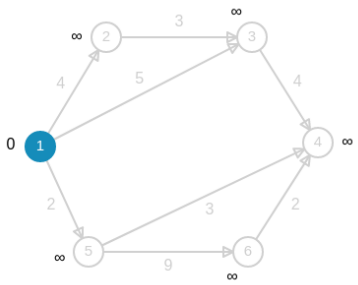
(v) DFS exploration complete



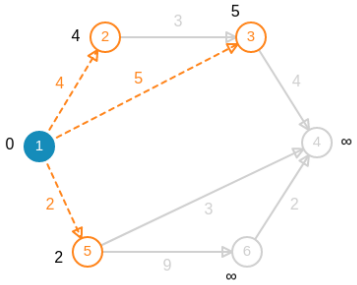
(vi) Search tree output by DFS

Figure 13: Depth First Search working on a 4-vertex graph

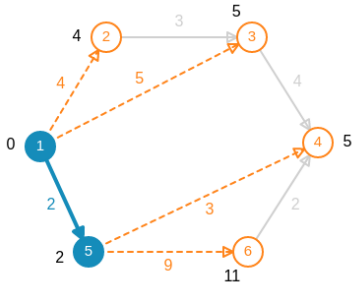
graph with six vertices is shown in Figure 14. The user has selected vertex 1 as the start vertex. Initially distances to all vertices (which can be seen as external labels on vertices) except for the source are infinity. At each step, the algorithm updates the distances as it discovers better paths. In the end we compute the shortest distance to each vertex and produce a shortest-path search or spanning tree of the graph.



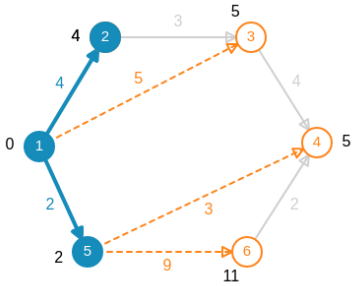
(i) All distances except source ∞



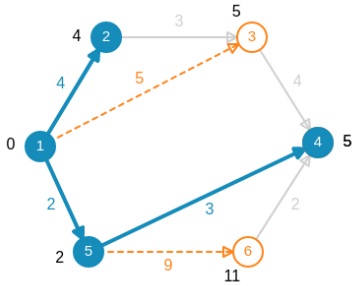
(ii) Explore vertex 1, update distances for 2, 3, 5



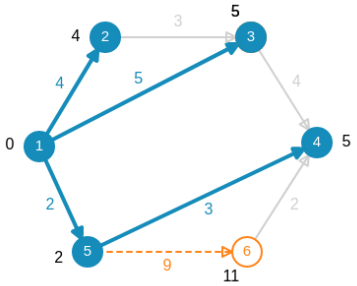
(iii) Explore vertex 2, update distances for 4, 6



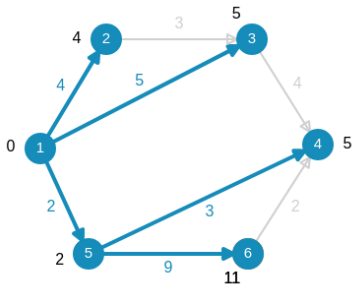
(iv) Explore vertex 4, no updates



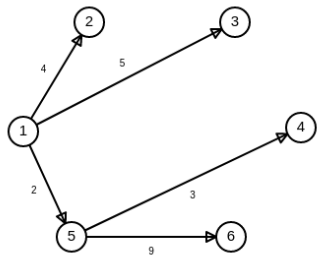
(v) Explore vertex 4, no updates



(vi) Explore vertex 3, no updates



(vii) Explore vertex 6, no updates



(viii) Shortest Path Tree output by Dijkstra's Algorithm

Figure 14: Dijkstra's Algorithm working on a 6-vertex graph

2.6.6 Walk-Finding Algorithms

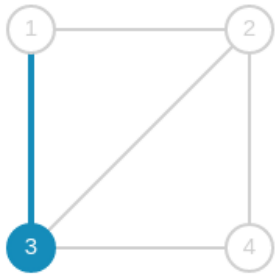
Fleury's Algorithm for Euler Trails and Cycles

Fleury's Algorithm is a straightforward algorithm for finding Euler Cycles or Trails in graphs. For graphs where all vertices are of even degree, the algorithm finds an Euler Cycle and for graphs where all but two vertices are of even degree, the algorithm finds a non-cycle Euler Trail. The algorithm works by starting with an initial vertex and moving from vertex to vertex, while adding edges to the trail. Edges added to the trail are removed from the original graph, and edges are only added to the trail if removing the edge from the graph does not disconnect it (i.e. the edge is not a bridge), or if no non-bridge edge incident to the current vertex can be added. When all edges have been added in this way, the result will be an Euler Cycle (if one exists) or a non-cycle Euler trail.

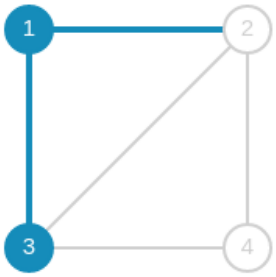
Figure 15 shows an example where Fleury's Algorithm is executed on a graph with four vertices and five edges. The graph has two vertices (3 and 2) which are of odd degree, so only non-cycle trails can be found in this graph. The algorithm starts at one of the odd-degree vertices (3) and ends its traversal at the other odd-degree vertex (2), outputting a graph with the edges marked with their traversal order along the trail.

Bellman-Held-Karp Algorithm for Hamilton Paths and Circuits

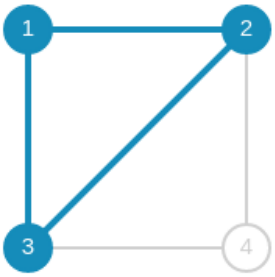
Finding Hamilton Circuits or Paths is a much harder problem than finding Euler cycles. More precisely, it is an NP-complete problem, so no polynomial-time algorithm for solving it is known. The algorithm implemented here is a modified version of Bellman-Held-Karp algorithm [12] [13], also called the Held-Karp algorithm. The algorithm was initially specified for the Traveling Salesman Problem, but works with a small modification for the Hamilton Path/Circuit problem since this problem can be formulated as a special case of the Traveling Salesman



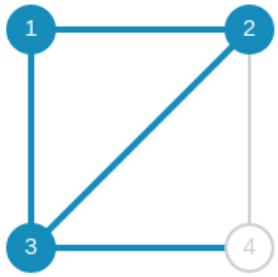
(i) Start with 3, add edge $(3, 1)$



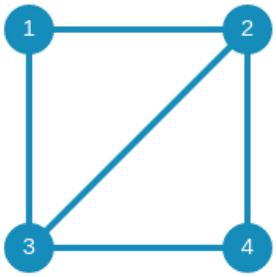
(ii) Move to 1, add edge $(1, 2)$



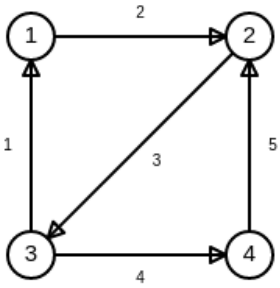
(iii) Move to 2, add edge $(2, 3)$



(iv) Move to 3, add edge $(3, 4)$



(v) Move to 4, add edge $(4, 2)$



(vi) The Euler Trail. Edges labelled with traversal order.

Figure 15: Fleury's Algorithm finding an Euler Trail

Problem.

The procedure has time complexity $O(n^2 2^n)$ and space complexity $O(n2^n)$. The method is a dynamic programming algorithm which is based on the observation that a Hamilton Path ending at a vertex v on (the subgraph induced by) a set of vertices S on the graph can be built by first building a Hamilton Path on the set $S - \{v\}$ ending at vertex w, provided there is an edge between w and v. The algorithm builds Hamilton Paths for progressively larger subsets, storing Hamilton Paths ending at each possible vertex for each subset. At the end, the whole graph is considered and the Hamilton Paths found are used to create a Hamilton Circuit if one exists.

Figure 16 shows a selection of steps for the execution of the algorithm on a four-vertex graph. Even with $n = 4$, $n^2 2^n$ is already comparatively large: 256. Therefore, the steps where the algorithm is considering vertex subsets of size 2 are entirely skipped in Figure 16. In Figures 16 (i) to (xiv), we can see the algorithm considering all subsets of size 3. For each subset of vertices, all possible ending vertices are considered, and for each ending vertex, all the possible paths ending at that vertex and running through the other vertices in the subset are considered.

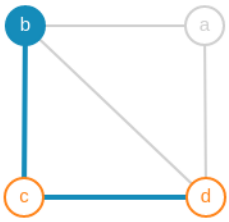
In the figure, subsets being considered are shown with colored edges, with the end vertex being considered set to the `SELECTED` state (colored teal) and the other vertices being set to `CONSIDERING` (colored orange). The interesting steps in the algorithm are Figures 16 (vi), (vii), and (xiii), where the algorithm considers the neighbors of the end vertex but cannot find a path through the remaining vertices. As soon as we find one 4-vertex path with the extreme vertices adjacent (as in Figure 16 (xv)), we have found a Hamilton circuit. If no 4-vertex path has the extremities adjacent, there exists only a non-circuit Hamilton path.

2.6.7 Decomposition Algorithms

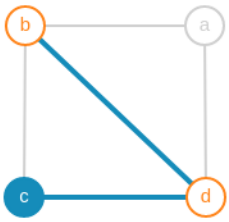
Articulation Points and Biconnected Components

An articulation point in a connected graph is a vertex whose removal would disconnect the graph. A biconnected graph is one in which no articulation points exist. Graphs that are not biconnected can be decomposed into a set of subgraphs that are biconnected. An algorithm for finding the biconnected components can also simultaneously find the articulation points in the graph, and the algorithm implemented here does this.

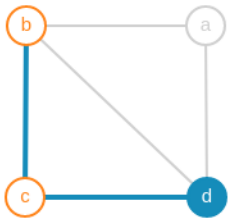
The algorithm is due to Hopcroft and Tarjan [14]. The implementation has been adapted from the one in [15] and works by performing a depth-first traversal of the graph starting from an arbitrary vertex. The procedure keeps track of two



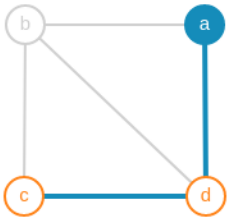
(i) Subpath d-c-b



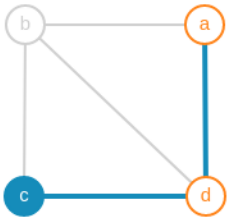
(ii) Subpath b-d-c



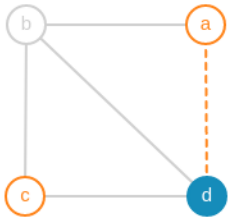
(iii) Subpath b-c-d



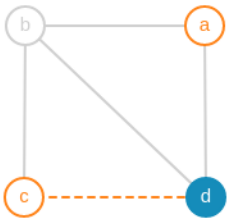
(iv) Subpath c-d-a



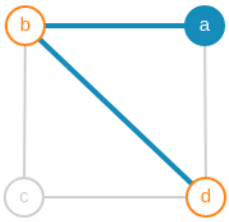
(v) Subpath a-d-c



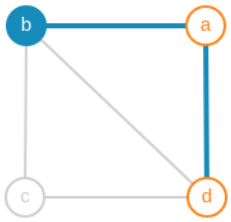
(vi) No subpath c-a-d



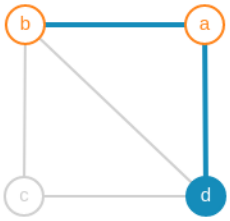
(vii) No subpath a-c-d



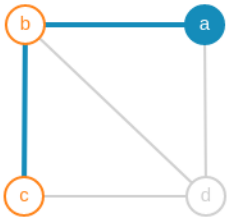
(viii) Subpath d-b-a



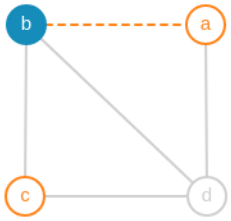
(ix) Subpath d-a-b



(x) Subpath b-a-d

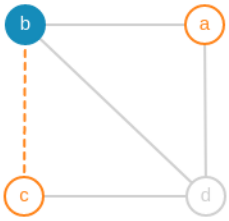


(xi) Subpath c-b-a

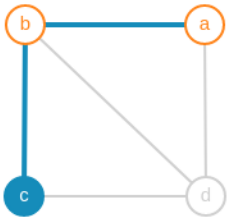


(xii) No subpath c-a-b

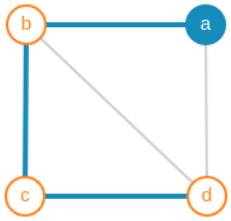
Figure 16: Bellman-Held-Karp Algorithm finding a Hamilton Circuit



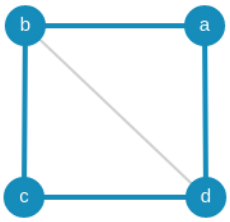
(xiii) No subpath a-c-b



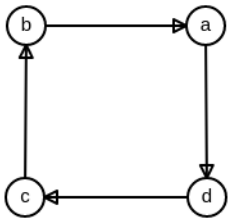
(xiv) Subpath a-b-c



(xv) Subpath d-c-b-a



(xvi) Hamilton path d-c-b-a-d



(xvii) Output as a directed graph

Figure 16: Bellman-Held-Karp Algorithm finding a Hamilton Circuit (continued)

values for each vertex: one called the Depth First Number (the order of the vertex in the depth-first traversal) and the L-value, which is lowest depth-first number that can be reached from that vertex using a path of descendants followed by at most one back edge (an edge which is not a part of the depth-first search tree). The algorithm identifies articulation points (and biconnected components) as it backtracks from the depth-first traversal.

Figure 17 shows the steps for an execution of the algorithm on a 'bow tie' graph with 5 vertices. The graph has one articulation point, namely the vertex 3, and two biconnected components on either side of it. The depth first traversal starts at an arbitrarily selected vertex, in this case vertex 1. As the traversal progresses, vertices are labeled with their depth first number, D. When the algorithm backtracks, it also assigns the L-value which is shown alongside the D-value in the graph. The algorithm uses *auxiliary states* (discussed in Section 3.2.4) to differentiate biconnected components. In this case, the biconnected component consisting of vertices 3, 4 and 5 is colored red and the one consisting of vertices 1, 2 and 3 is colored green.

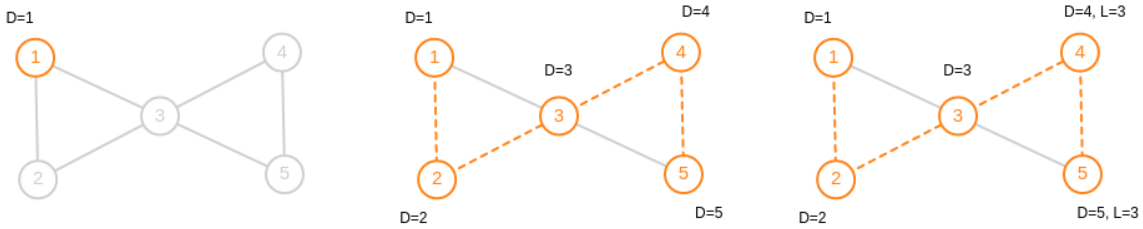
2.6.8 Exact TSP Algorithms

Bellman-Held-Karp Algorithm

The system implements one exact algorithm for the Traveling Salesman Problem. The algorithm implemented is the Bellman-Held-Karp algorithm [12] [13], a variation of which is also implemented for the Hamilton Path problem.

Like the one implemented for the Hamilton Path problem, the algorithm has a time complexity of $O(n^2 2^n)$ and a space complexity of $O(n2^n)$. The algorithm is based on dynamic programming and relies on the fact that a sub-path of an optimal tour is itself optimal.

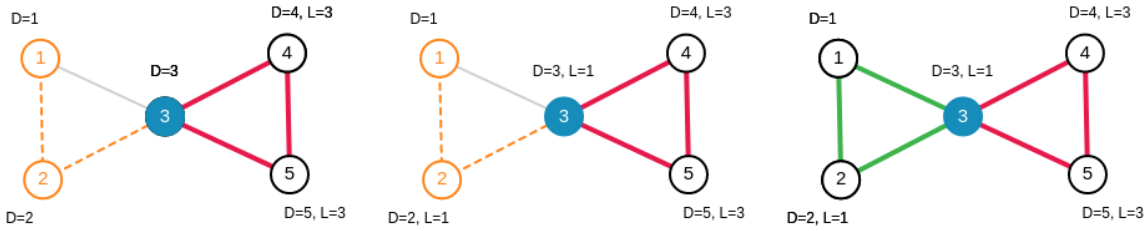
Unlike all other algorithms we have seen, this (and other TSP algorithms)



(i) Start depth first traversal at 1

(ii) Label vertices with depth first number D

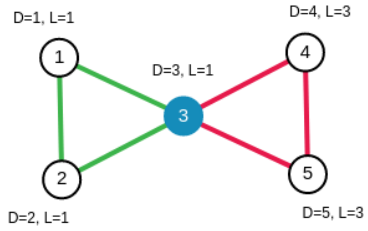
(iii) Backtrack, assigning L-values



(iv) $D_3 \leq L_4$, articulation point found

(v) Backtrack, assigning L-values

(vi) $D_1 \leq L_2$, articulation point found



(vii) Backtrack complete, two biconnected components found

Figure 17: The Hopcroft-Tarjan algorithm finding an articulation point and two biconnected components

operate on Euclidean Graphs. Euclidean graphs are graphs where the edges are implicit, and each vertex is considered to be adjacent to all the others. The weight of an edge is the Euclidean distance between the incident vertices.

Figure 18 shows an execution of the algorithm on a 3-vertex Euclidean graph. The algorithm is almost exactly the same as the Bellman-Held-Karp algorithm for Hamilton paths, but in this case we store the path costs along with the paths and look for the *cheapest* path instead of just *any* path. Since the example graph only has 3 vertices, all decoration steps are shown in the figure. Even though the graph

has exactly three edges and all of them appear in the optimal tour, the figure illustrates the steps the algorithm takes.

2.6.9 TSP Approximation Algorithms

The system implements five approximation algorithms for the Traveling Salesman Problem. Three of them (Nearest Neighbor, Nearest Insert and Cheapest Insert) work by progressively building a tour by inserting a vertex into the current tour based on a heuristic. The fourth (MST Based) algorithm works by creating a minimum spanning tree of the vertices and deriving a tour from the tree. The Christofides algorithm is an improvement of the MST-based approach.

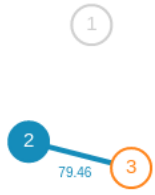
Nearest Neighbor Heuristic

The nearest neighbor heuristic algorithm starts with an initial vertex and adds the nearest vertex to the tour. The newly added vertex becomes the current vertex and the process is repeated until all vertices are added. A circuit is formed by connecting the first and the last vertices. For an n-vertex graph, the nearest-neighbor heuristic creates a tour that is at most $\frac{1}{2}\lceil \lg(n) \rceil + \frac{1}{2}$ times as long as the optimal tour length. [16]

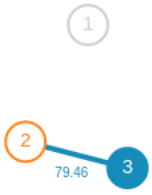
Figure 19 illustrates the nearest-neighbor heuristic algorithm working on a 5-vertex Euclidean graph. The user has selected vertex a as the start vertex and from there, at each step, the algorithm adds the vertex closet to the most recently added vertex to the path. When we have a Hamilton path, the tour is completed by adding an edge to the starting vertex.

Nearest Insert Heuristic

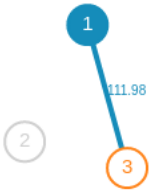
The nearest insert heuristic algorithm is similar to the nearest neighbor heuristic algorithm, but instead of adding the vertex nearest to the current vertex to the



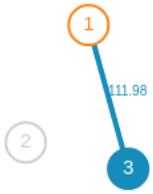
(i) Examine path 3-2



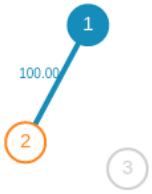
(ii) Examine path 2-3



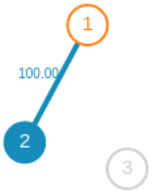
(iii) Examine path 3-1



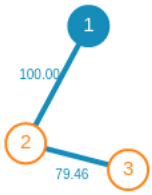
(iv) Examine path 1-3



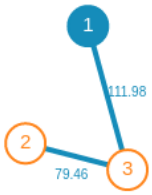
(v) Examine path 2-1



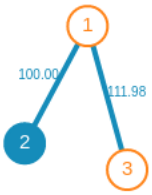
(vi) Examine path 1-2



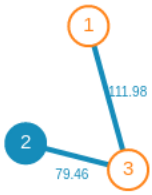
(vii) Examine path 3-2-1



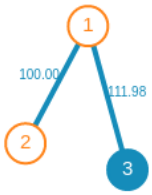
(viii) Examine path 2-3-1



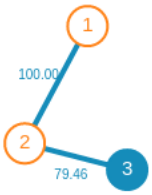
(ix) Examine path 3-1-2



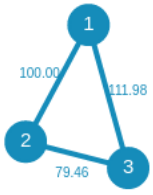
(x) Examine path 1-3-2



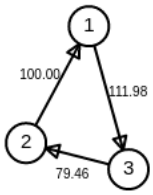
(xi) Examine path 2-1-3



(xii) Examine path 1-2-3

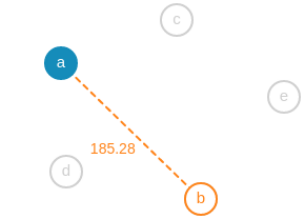


(xiii) Optimal Tour Found

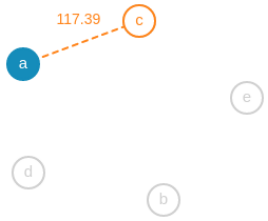


(xiv) Output as a directed graph

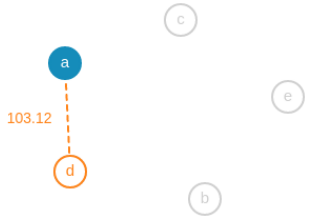
Figure 18: Bellman-Held-Karp Algorithm Finding the Optimal TSP Tour



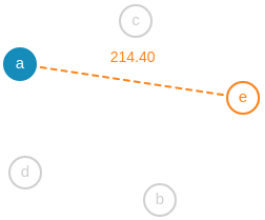
(i) Examine edge a-b



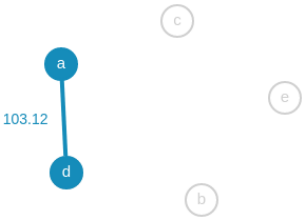
(ii) Examine edge a-c



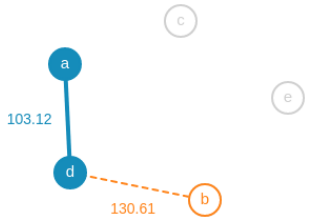
(iii) Examine edge a-d



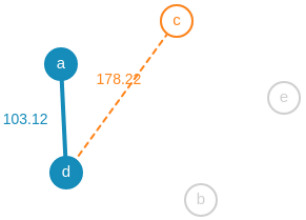
(iv) Examine edge a-e



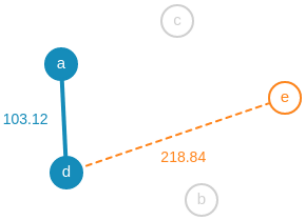
(v) Add edge a-d to path



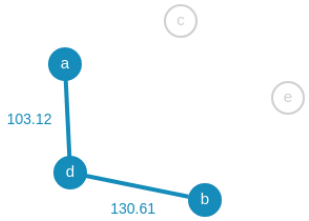
(vi) Examine edge d-b



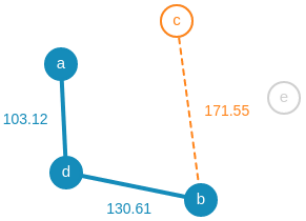
(vii) Examine edge d-c



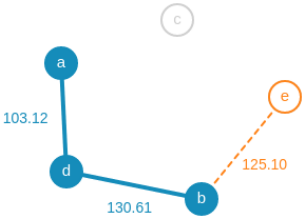
(viii) Examine edge d-e



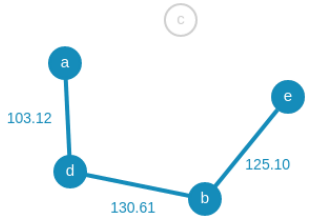
(ix) Add edge d-b to path



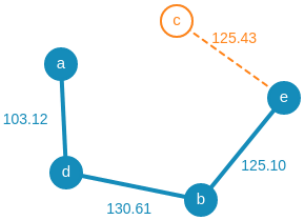
(x) Examine edge b-c



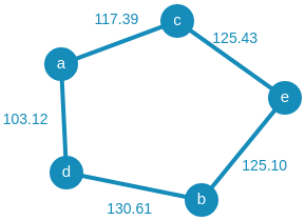
(xi) Examine edge b-e



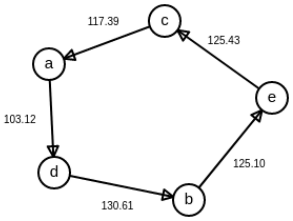
(xii) Add edge b-e to path



(xiii) Examine egde e-c



(xiv) Add e-c to path and complete tour



(xv) Output as a directed graph

Figure 19: TSP Approximation with Nearest-Neighbor Heuristic

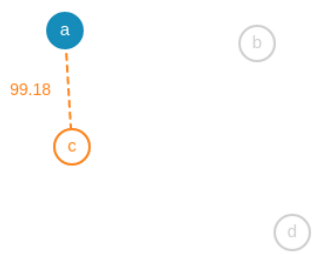
end of the tour, the vertex nearest to the current tour as a whole is inserted into the tour. The insertion is made between that pair of consecutive vertices in a tour which minimizes the difference between the length of the new tour and the length of the previous one. The nearest-insert heuristic produces a tour that is at most twice as long as the optimal traveling salesman tour. [16]

Figure 20 illustrates the nearest-insert heuristic algorithm working on a 4-vertex Euclidean graph. The user has selected vertex a as the start vertex. From there, at each step, the algorithm examines vertices not in the partial tour but nearest to some vertex in the partial tour, and inserts into the tour the nearest among *them*. When we insert the last vertex into the tour, it is complete.

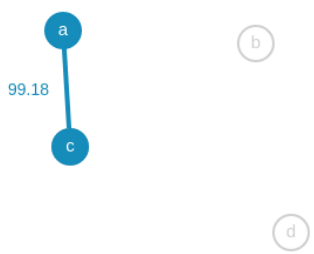
Cheapest Insert Heuristic

The cheapest insert heuristic algorithm is almost identical to the nearest insert heuristic algorithm, except for one difference: the inserted vertex is not the one that is nearest to the current tour, but the one that minimizes the length of the newly created tour. The insertion itself is done identically to the nearest neighbor heuristic algorithm. Like the nearest-insert heuristic, cheapest-insert also produces a tour that is at most only two times as long as the optimal one. [16]

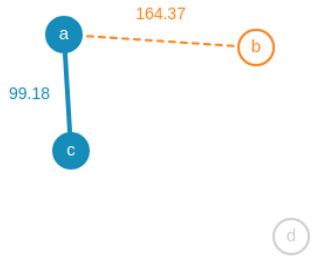
Figure 21 illustrates the cheapest-insert heuristic algorithm working on a 5-vertex Euclidean graph. The user has selected vertex 1 as the start vertex. At each step, the algorithm evaluates the cost (the length added to the tour) of inserting each vertex not already in the tour, and inserts the vertex with the minimum cost. In the first insertion, that vertex is vertex 4. The next one added is 2 because tours 1-4-3-1 and 1-4-5-1 are both longer than 1-4-2-1. Similarly 5 is inserted next and 3 is inserted last to complete the tour.



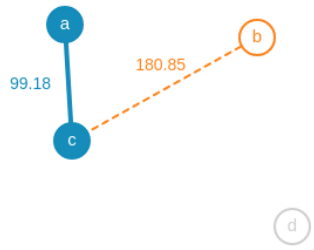
(i) Start with a, nearest neighbor c



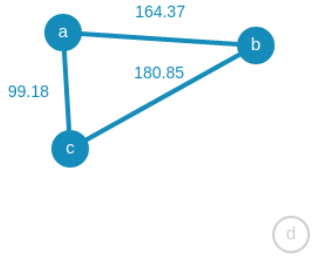
(ii) Insert c to tour



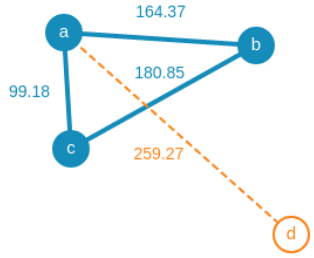
(iii) Nearest remaining neighbor of a is b



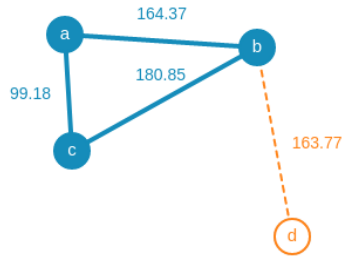
(iv) Nearest remaining neighbor of c is b



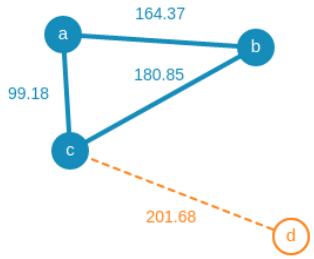
(v) Insert b to tour



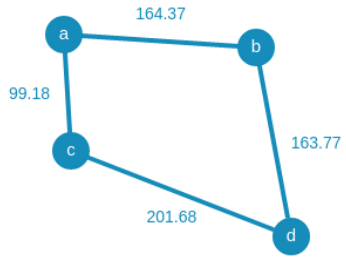
(vi) Nearest remaining neighbor of a is d



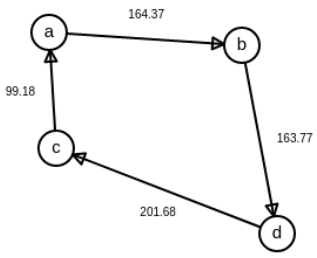
(vii) Nearest remaining neighbor of b is d



(viii) Nearest remaining neighbor of c is d



(ix) Insert d to tour, tour complete

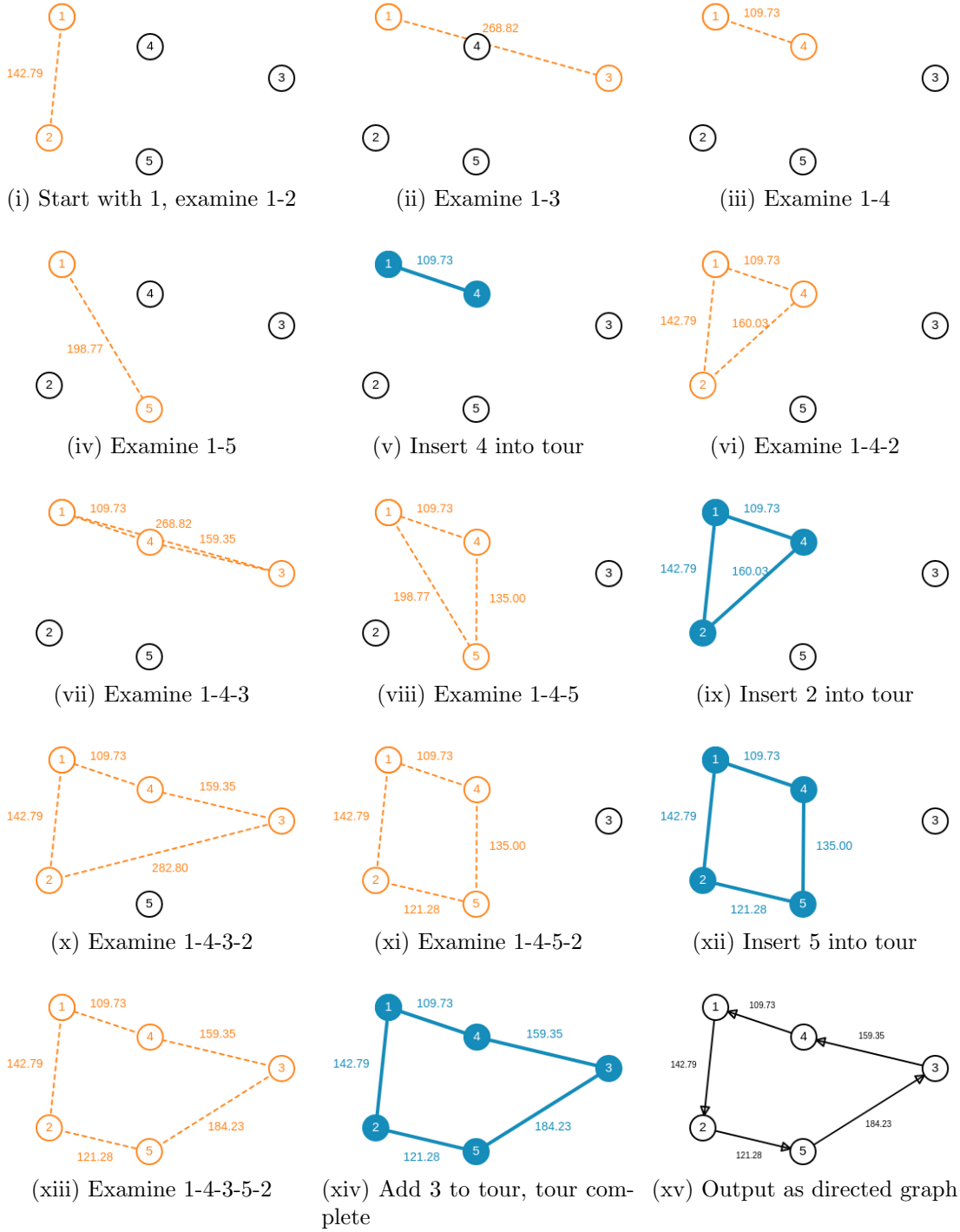


(x) Output as directed graph

Figure 20: TSP Approximation with Nearest-Insert Heuristic

MST-Based Approximation Algorithm

The Minimum Spanning Tree (MST) based approximation algorithm starts by first creating an MST of the given graph. In our implementation we use Kruskal's



(i) Start with 1, examine 1-2 (ii) Examine 1-3 (iii) Examine 1-4

(iv) Examine 1-5 (v) Insert 4 into tour (vi) Examine 1-4-2

(vii) Examine 1-4-3 (viii) Examine 1-4-5 (ix) Insert 2 into tour

(x) Examine 1-4-3-2 (xi) Examine 1-4-5-2 (xii) Insert 5 into tour

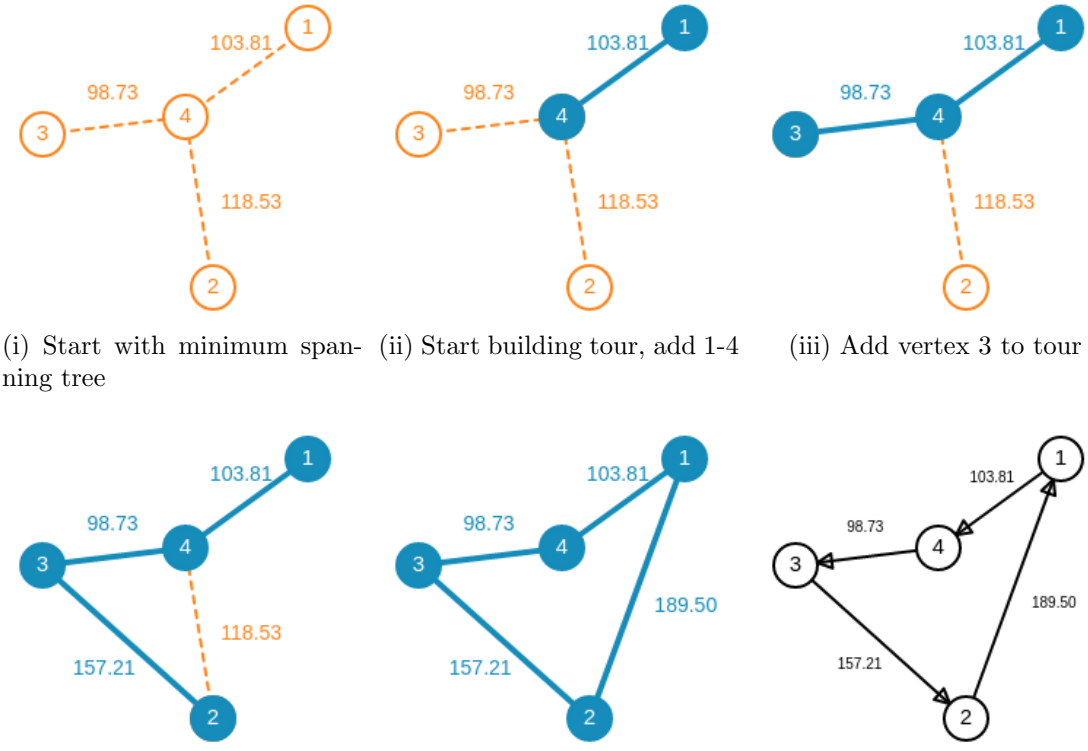
(xiii) Examine 1-4-3-5-2 (xiv) Add 3 to tour, tour complete (xv) Output as directed graph

Figure 21: TSP Approximation with Cheapest-Insert Heuristic

algorithm to create the tree. A trail is built from the MST by traversing the edges using the right-hand rule (always keep the edge to your right hand side). The trail

is then converted to a proper Hamilton Circuit by traversing the trail but skipping already-visited vertices. This approach produces a tour that is at most twice the length of the optimal TSP tour. [16]

Figure 22 illustrates the steps of the MST-based approximation algorithm. The figure omits the steps taken by Kruskal's algorithm to compute the MST. Once we have the MST, the tour is built by starting at an arbitrary vertex (1 in this case) and adding vertices to the tour by following the tree. In this way we add 4 and 3 to the tour. As we travel along the tree, vertices already in the tour are skipped, so we add 2 next, and then finally move to 1 to complete the tour.



(i) Start with minimum spanning tree

(ii) Start building tour, add 1-4

(iii) Add vertex 3 to tour

(iv) Skip visited vertex 4, add vertex 2 to tour

(v) Complete tour by adding 2-1

(vi) Output as directed graph

Figure 22: TSP Approximation with MST-Based Algorithm

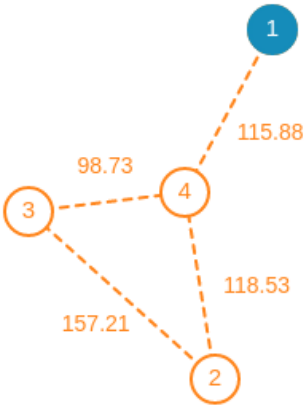
Christofides Algorithm

Christofides algorithm is an improvement upon the MST-based approximation algorithm. After finding the minimum spanning tree, it takes the set of vertices that have odd degree in the tree, and creates a subgraph of the underlying Euclidean graph containing just those vertices and the edges between them. Then, a minimum-weight prefect matching on that subgraph is found. The Handshaking Lemma guarantees that there are an even number of odd-degree vertices in any graph, so a perfect matching always exists. After the minimum-weight perfect matching has been found, a multi-graph is formed by adding the edges from the matching to the minimum spanning tree. The resulting multi-graph will only contain even-degree vertices. An Euler cycle is found on the multi-graph and the cycle is converted into a Hamilton circuit by traversing it and skipping already-visited vertices. Christofides algorithm produces a tour that is at most 1.5 times as long as the optimal TSP tour. [17]

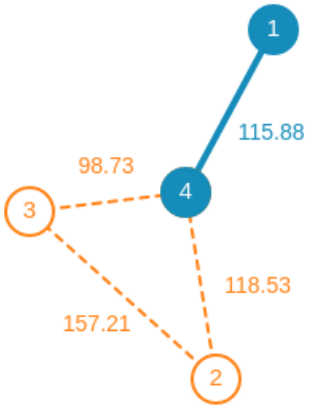
The execution of Christofides Algorithm on a 4-vertex Euclidean graph is shown in Figure 23. Once again we start with the minimum spanning tree of the vertices. Then in the next step, a minimum-weight matching between the odd-degree vertices (of which there are an even number by the Handshaking Lemma) is found. This is performed internally using the Blossom algorithm. [18] The minimum-matching edges are then added to the minimum spanning tree, forming a multigraph. In Figure 23 (ii), we can see the added edge $(3, 2)$. There are actually two edges between vertices 1 and 4, but since the system lacks support for displaying multigraphs, only one is shown. An Euler trail is found on the multigraph using Fluery's Algorithm. This multigraph is then traversed starting from an arbitrary vertex (1 in this case) and vertices already in the path are skipped, creating a complete tour in the end.

(i) Start with minimum spanning tree

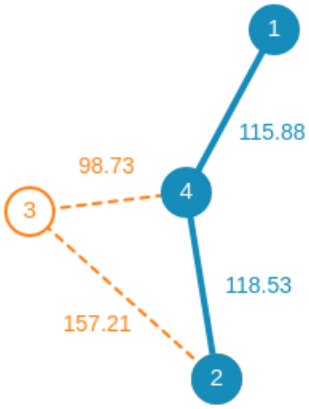
(ii) Add minimum-matching edges (3-2 and 1-4)



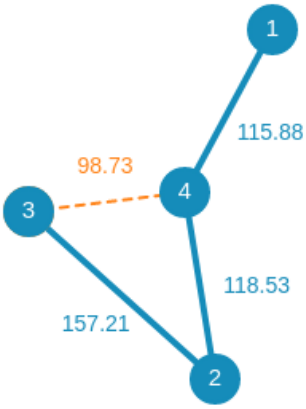
(iii) Start building tour with 1, following Euler Cycle



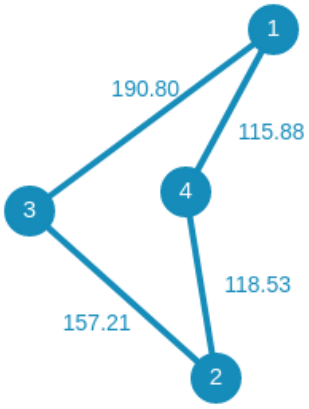
(iv) Add 4 to tour



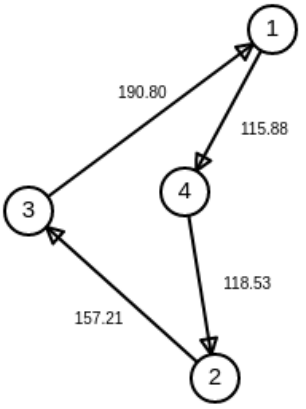
(v) Add 2 to tour



(vi) Add 3 to tour



(vii) Skip visited vertex 4, complete tour



(viii) Output as directed graph

Figure 23: TSP Approximation with Christofides Algorithm

2.6.10 Network Flow Algorithms

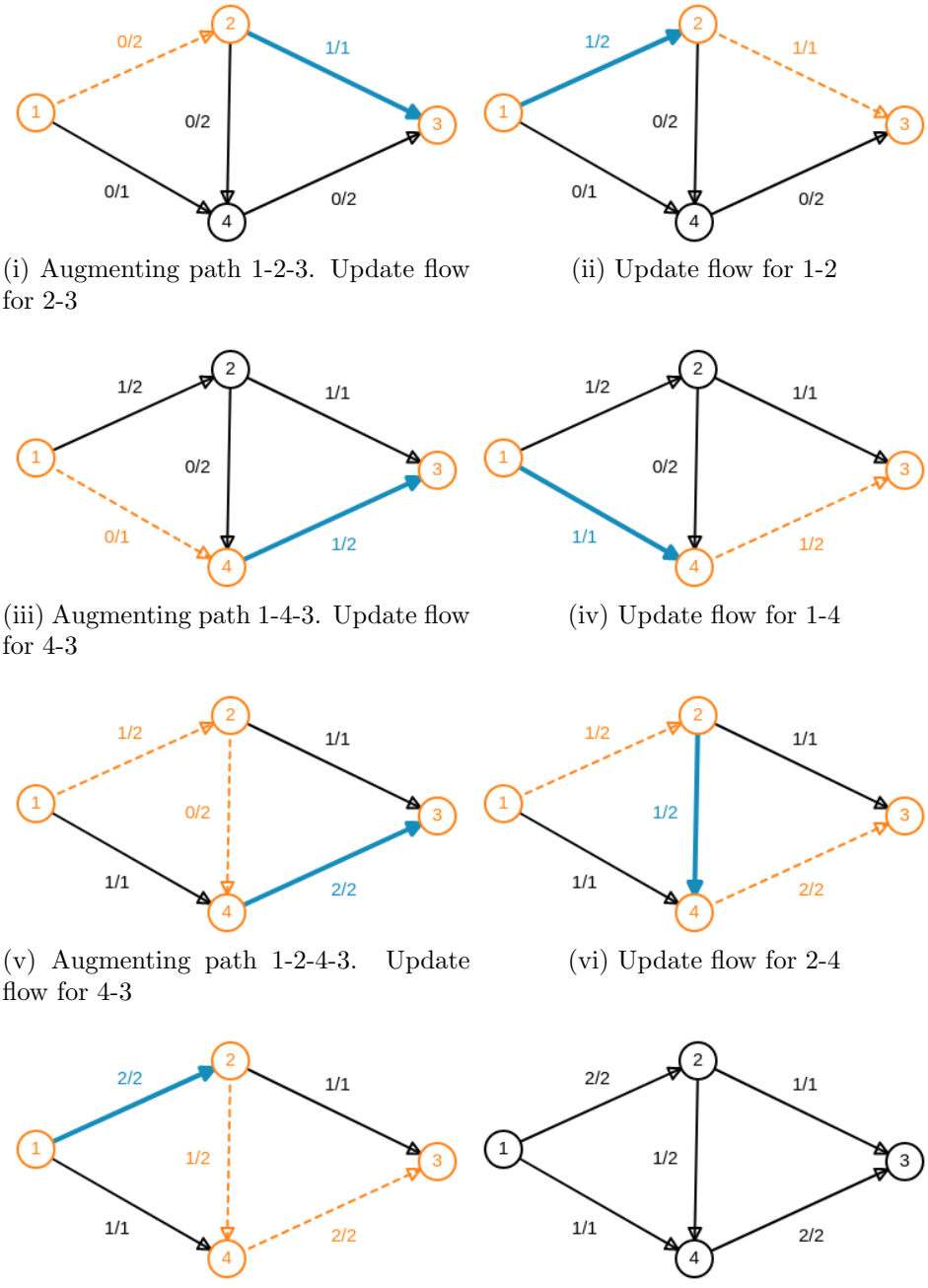
Edmonds-Karp Algorithm

The system implements the Edmonds-Karp Network Flow algorithm, which is equivalent to the Ford-Fulkerson method implemented using a Breadth First Search to find augmenting paths. The algorithm requires two inputs: a source vertex and a sink vertex, and it attempts to find the maximum flow from the source to the sink.

The algorithm works by finding the shortest-distance augmenting path at each step. An augmenting path is a path from the source to the sink where the maximum flow along the path is lower than the minimum capacity along the path. This means that more flow can be sent along the path. The algorithm reassigns flow values to the maximum possible flow and goes on to find the next augmenting path until no augmenting paths can be found.

The output of the graph is a directed weighted graph with flow values of edges as their respective weights.

We can see the steps of an execution of Edmonds-Karp on a 4-vertex flow network in Figure 24. The user has selected vertex 1 as the source and vertex 3 as the sink for this example. First, the algorithm finds the augmenting path 1-2-3 and sends a flow of 1 along that path. Then the augmenting path 1-4-3 is found, and a flow of 1 is sent along that path as well. The final augmenting path found is 1-2-4-3, along which an additional flow of 1 is sent. After this no augmenting path with any available capacity is found, and the algorithm terminates. The output is a flow graph with edge weights representing the value of the flow in that edge.



(i) Augmenting path 1-2-3. Update flow for 2-3

(ii) Update flow for 1-2

(iii) Augmenting path 1-4-3. Update flow for 4-3

(iv) Update flow for 1-4

(v) Augmenting path 1-2-4-3. Update flow for 4-3

(vi) Update flow for 2-4

(vii) Update flow for 1-2

(viii) All flows computed

Figure 24: Edmonds-Karp algorithm for computing network flow

List of References

[1] npm, Inc. “npm - npm.” Accessed: 2021-07-05. [Online]. Available: https://www.npmjs.com/package/npm

[2] The Webpack Team. “webpack.” Accessed: 2021-07-18. [Online]. Available:

https://webpack.js.org/

[3] A. Lavrenov. "Konvajs." Accessed: 2021-07-05. [Online]. Available: https://konvajs.org

[4] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2016, pp. 263–273.

[5] The Web Hypertext Application Technology Working Group. "HTML Standard web storage." Accessed: 2021-07-02. July 2021. [Online]. Available: https://html.spec.whatwg.org/multipage/webstorage.html

[6] P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

[7] S. G. Kobourov, "Force-directed drawing algorithms," in *Handbook of Graph Drawing And Visualization*, R. Tamassia, Ed. CRC Press, 2013, ch. 12, pp. 383–408.

[8] The Web Hypertext Application Technology Working Group. "HTML Standard range state." July 2021. [Online]. Available: https://html.spec.whatwg.org/multipage/input.html#range-state-(type=range)

[9] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956. [Online]. Available: http://www.jstor.org/stable/2033241

[10] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, p. 269–271, Dec. 1959. [Online]. Available: https://doi.org/10.1007/BF01386390

[12] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," *J. ACM*, vol. 9, no. 1, p. 61–63, Jan. 1962. [Online]. Available: https://doi.org/10.1145/321105.321111

[13] M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196–210, 1962. [Online]. Available: https://doi.org/10.1137/0110015

[14] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, p. 372–378, June 1973. [Online]. Available: https://doi.org/10.1145/362248.362272

[15] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, ser. Computer software engineering series. Pitman, 1978. [Online]. Available: https://books.google.com/books?id=n8c8PgAACAAJ

[16] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II, "An analysis of several heuristics for the traveling salesman problem," *SIAM journal on computing*, vol. 6, no. 3, pp. 563–581, 1977.

[17] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1976.

[18] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of mathematics*, vol. 17, pp. 449–467, 1965.

CHAPTER 3

Design and Implementation of Graph Playground

In this chapter we discuss the design and implementation of Graph Playground and present a step-by-step example of extending the system by implementing a new algorithm visualization. Section 3.1 provides an overview of the design of the software, describing the main organizational parts of the system. Section 3.2 goes into more detail about the implementation of the system, describing the major interfaces, the classes that implement them, and user interface handlers. Section 3.3 discusses the tools used to create the system and the technologies, frameworks and design patterns used. Section 3.4 goes into the problems encountered while designing and implementing the system and how they were solved. Section 3.5 describes the programming interface that algorithms use to communicate with the rest of the system and create visualizations. Section 3.6 is a description of the steps needed to extend the system by implementing a new algorithm, demonstrated with a real example.

3.1 Overview of Design

This overview is organized by *modules*, which are sub-systems that have a particular objective. Modules consist of a collection of type definitions, interfaces and classes that implement them. Each of the sub-headings below briefly discusses a module, and a more detailed description is given in Section 3.2. It should be noted that since the system is implemented in the TypeScript programming language, the words *interface*, *class* and *type*, when used in a context to describe the implementation, refer to their meanings within the TypeScript language.

Graph Core

At the core of the system are the interfaces and classes representing graphs themselves, without regard to how they are drawn or shown to the user. The details of the layout and display of graphs, including positions of the vertices and edges, the placement of labels, and the decorations made by algorithms are stored in and manipulated through the drawing classes discussed in Section 3.2.2, the primary one being `GraphDrawing`. Using the terminology of the implementation, we will use the word 'graph' to mean the graph data structure without regard to drawing and layout concerns. We will use 'graph drawing' to mean a graph together with its drawing. We will sometimes use just 'graph' to refer to both when the distinction is irrelevant.

The graph core is centered around the `Graph` interface, which provided methods for adding and removing vertices and edges, getting the neighbors of a vertex and setting labels for vertices. A `Weighted` interface supplements the `Graph` interface, providing methods specific to weighted graphs. The classes that implement these interfaces are discussed in Section 3.2.1.

Drawing

The drawing module is concerned with producing graph drawings from graphs, giving users the ability to interact with graph drawings and change the underlying graph, and allowing algorithms (through decorators) to change how the graph drawing is displayed.

The drawing module consists of three classes: `VertexDrawing` to represent a drawing of a vertex, `EdgeDrawing` to represent a drawing of an edge, and `GraphDrawing` to tie the vertex drawings and edge drawings together, representing a graph drawing. All three classes also perform event handling to respond to user interactions and requests by algorithms.

Algorithm

The algorithm module is made of an algorithm interface that all algorithms implement, an algorithm runner class that deals with how algorithms are run, and the algorithms themselves. The algorithms included in the system are introduced in Section 2.6 and implementation details are discussed in Section 3.2.5. As mentioned before, it should be noted that the word 'algorithm' is used here to mean not just an algorithm in the usual sense, but an implementation of the algorithm in Graph Playground in a way that the user can apply it to graphs.

Decorator

The decorator module exists to effectively decouple the implementation of algorithms from the specifics of graph drawings and the rest of the UI. This is done both for maintainability of the system and for easy extensibility. The most important part of the module is the interface called `Decorator` that algorithms use, among other things, to set visual states of vertices and edges, set labels for edges or vertices, and display a textual status. The default implementation of the

`Decorator` interface forwards all its requests to the `GraphDrawing` class.

User Interface

The user interface module is composed of the classes that implement functionality for the user interface components that are visible to the user. These include the tab bar, the drop down menus, the components within the side bars, the algorithm control panel and, of course, the graph canvas.

Layouts

The layouts module is made up of the `Layout` interface which the graph drawing class uses to get positions for laying out the vertices in the graph. There are several implementations of the layout interface in the system, and four of these correspond directly to the four auto-layout options visible in the right sidebar.

3.2 Implementation

In this section we discuss the system's main interfaces and their implementors in more detail. The interfaces, types and classes are described at a high level without going into the details of the code written.

3.2.1 Core Graph Interface and Classes

The `Graph` Interface

Before discussing the `Graph` interface itself, it is appropriate to describe how vertices are represented in the system. The system uses numeric vertex ids to identify vertices, and edges are specified by the pair of vertex ids to which they are incident. The numeric vertex ids can be optionally specified by the caller when adding a vertex to the graph, with the restriction that the vertex ids should be unique; specifying an already present vertex id is not permitted. If a vertex id is not specified while adding a vertex, the graph implementation generates a unique vertex id and returns it to the caller.

Textual items called labels can also be optionally associated with vertices in the graph, but they play no role in the behavior of the graph itself. Labels are used by the graph drawing so that users can differentiate between vertices.

The Graph interface specifies a set of methods for working with graphs. The methods are listed in the UML class diagram shown in Figure 25. It includes methods like `getVertexIds()` which returns a set of the numeric ids of all the vertices in the graph, `addVertex()` which optionally takes a vertex id and a vertex label, adds a vertex to the graph and returns the newly added vertex's id. The `clone()` method creates and returns a copy of the graph and the `toJSON()` method returns the graph as a JavaScript Object, suitable for writing to a text file. The 'Bookmark Graph' and 'Save Graph' functionality makes use of this method.

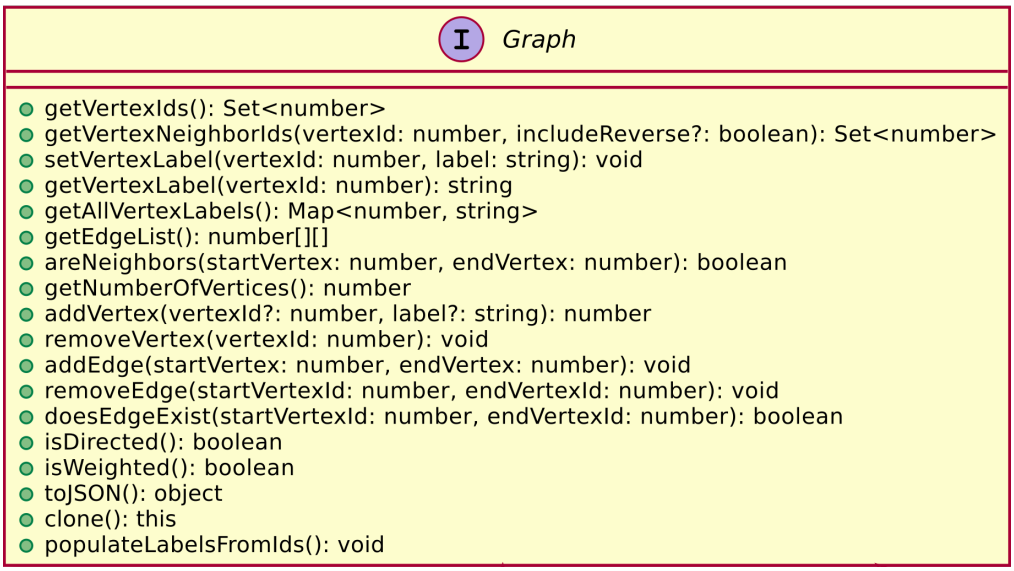


Figure 25: UML Class Diagram of the Graph Interface

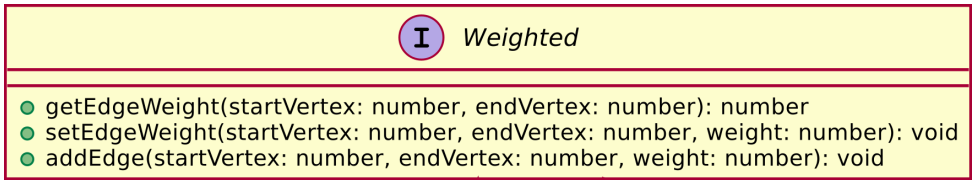


Figure 26: UML Class Diagram of the Weighted Interface

The `Weighted` Interface

An additional interface called `Weighted` includes methods specific to weighted graphs, including those for getting and setting edge weights and adding a new weighted edge. The UML class diagram for the interface is shown in Figure 26. A weighted graph must implement both the `Graph` interface and the `Weighted` interface.

`DefaultGraph`, `UnweightedGraph` and `WeightedGraph` Classes

The three classes that implement the previously discussed interfaces are `DefaultGraph`, `UnweightedGraph` and `WeightedGraph`. `DefaultGraph` is an abstract class and consolidates the functionality common to both weighted and unweighted graphs. `UnweightedGraph` and `WeightedGraph` both extend

`DefaultGraph` and weightedness-specific behavior. Moreover, `DefaultGraph` is a generic class parameterized by a type variable called `EdgeData`, which signifies the type of the object storing edge-related information. The `DefaultGraph` class itself does not care what information is associated with edges as that is for the implementing classes to handle. For example, `UnweightedGraph` does not store anything extra with the edges, so it extends `DefaultGraph<EmptyEdgeData>` where `EmptyEdgeData` is an empty object type. `WeightedGraph` implements `DefaultGraph<WeightedEdgeData>` where `WeightedEdgeData` is an object type with one numeric attribute: the edge weight.

Directedness

The system provides the ability to distinguish between directed and undirected graphs. The constructors of the `WeightedGraph` and the `UnweightedGraph` classes take a single boolean argument called `directed` that indicates whether the graph should be directed. For operations where the directedness of graphs matters, the implementations of all the methods perform the correct operation according to whether the graph is directed. For instance, if we have two vertices a and b, calling `addEdge()` with the arguments (a, b) creates a directed edge from a to b for a directed graph. For an undirected graph, the same call creates an undirected edge and is equivalent to calling the method with the arguments (b, a).

Internal Representation of Adjacency Data

The `Graph` interface does not specify how the vertices and edges should be stored, as those are implementation details and not relevant to the Graph interface. The `DefaultGraph` handles those details, and it uses an adjacency list representation to store the graph. More accurately, the representation could be called an adjacency *map* because lists are not involved and the JavaScript `Map` ob-

ject is used, which is internally implemented in the browser as a hash map "other mechanisms that, on average, provide access times that are sublinear in the number of elements in the collection." [1]

Since the data along with edges depends on more specific implementations, the `GraphAdjacencies` type is parameterized with the type parameter `EdgeData`, just like `DefaultGraph`. Precisely, the `GraphAdjacencies` type is defined as

```
type GraphAdjacencies<EdgeData extends EmptyEdgeData> =
    Map<number, Map<number, EdgeData>>;
```

The key of the outer `Map` is the starting vertex id, the key of the inner `Map` is the ending vertex id, and the value is the data associated with that edge.

The `EuclideanGraph` Class

The `EuclideanGraph` class is an implementation of the `Graph` interface to support a special graph type in the system: Euclidean graphs. Euclidean graphs, previously discussed in Section 2.3.1, are graphs for which vertex position is important and the edges are implicit, in the sense that every vertex is understood to have an edge to every other vertex, without the need to store them explicitly. The edges are undirected and weighted, and the weight of an edge is simply the Euclidean distance between the two vertices to which it is incident. For Euclidean graphs, a single set of vertex positions is shared between the graph and the graph drawing, whereas vertex positions are not relevant for other graph types and are only stored by the graph drawing visually.

3.2.2 Drawing Classes

The task of drawing graphs and handling the user's interactions with the graph drawing is handled by the drawing classes discussed here: `GraphDrawing`, `VertexDrawing`, and `EdgeDrawing`.

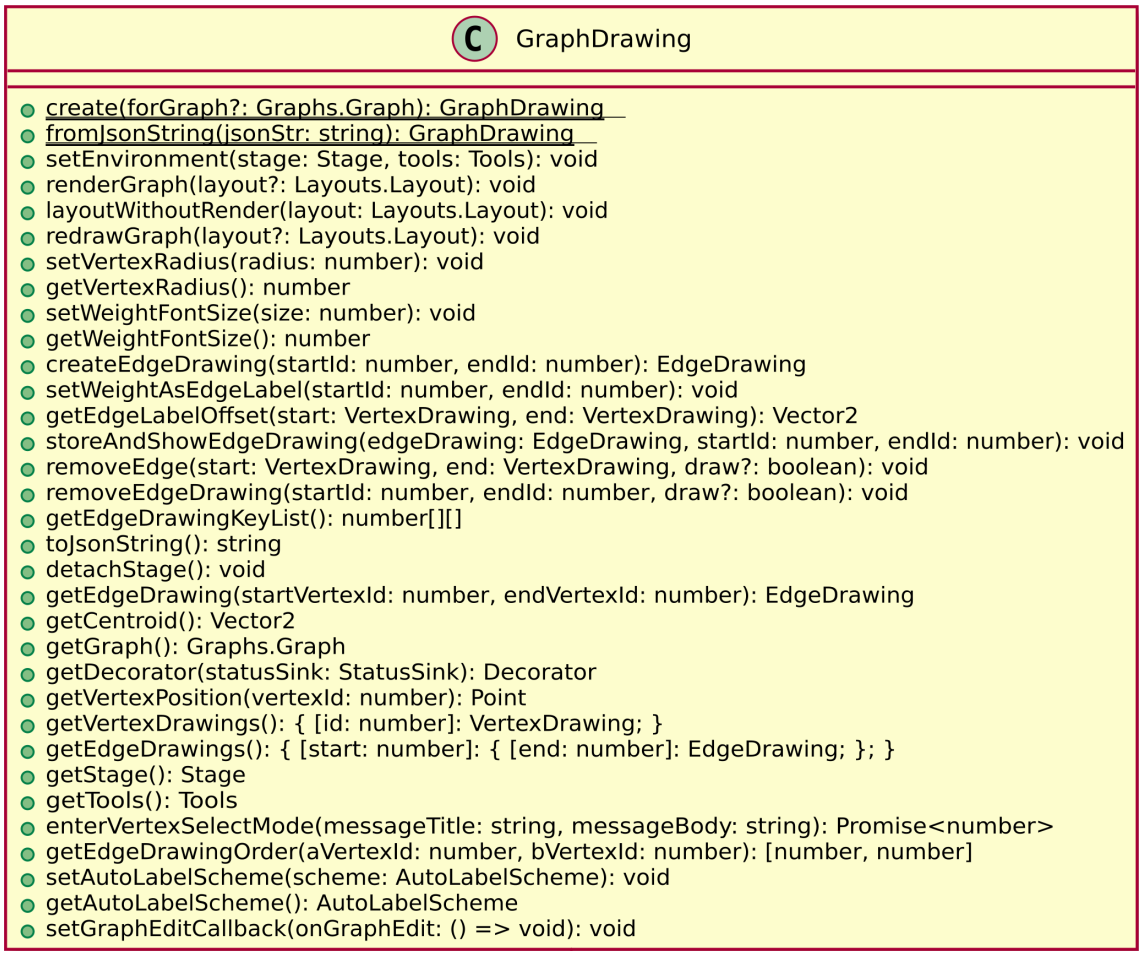


Figure 27: UML Class Diagram of the GraphDrawing Class, showing only public methods

GraphDrawing

The `GraphDrawing` class represents a drawing of a graph, including all its constituent objects (vertices, edges, labels, etc.) and their properties. This class contains a reference to a `Graph` of which it is a drawing. Figure 27 shows the UML class diagram of the class, with only public methods shown. In the figure, we can see the methods which other parts of the system use to interact with the graph drawing.

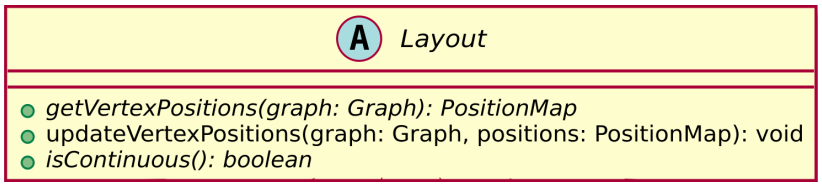


Figure 28: UML Class Diagram of the Layout abstract class

VertexDrawing

The `VertexDrawing` class represents the drawing of a vertex, including the circle that is the main visual component of the vertex and the label inside the circle. Since the system uses the KonvaJS library for drawing primitive shapes, the `VertexDrawing` class extends KonvaJS's `Group` class, and contains the corresponding KonvaJS objects for the label and the circle.

EdgeDrawing

The `EdgeDrawing` class represents the drawing of an edge, of which the main component is a line (for unweighted graphs) or an arrow (for weighted graphs). It also contains an optional label that is used, for instance, by weighted graphs to display the weight. Like `VertexDrawing`, the `EdgeDrawing` class also extends KonvaJS's `Group` class, and it contains the corresponding KonvaJS objects for the label and the line or arrow.

3.2.3 Layout Interface and Classes

To implement the auto-layout options provided to the user as discussed earlier in Section 2.3.9, an interface called `Layout` is used. The methods of the interface are shown in Figure 28. The important method of the interface is `getVertexPositions()`, which, given a graph as the argument, returns a `PositionMap`, which is essentially an object that maps vertex ids to vertex coordinates. The method `updateVertexPositions()` is the in-place version of the

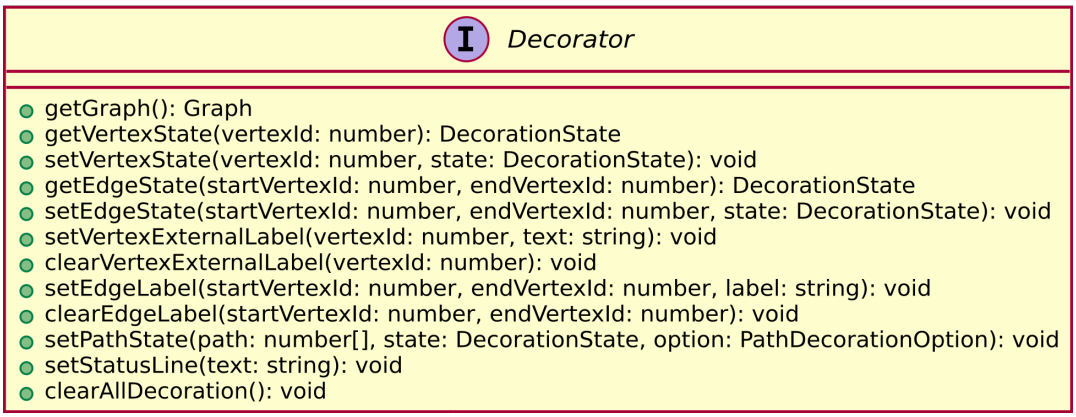


Figure 29: UML Class Diagram of the Decorator interface

method just discussed, i.e. instead of creating and returning an object with the positions map, it takes a reference to an existing position map as an input and updates that position map. There is another method called `isContinuous()` which returns a boolean value indicating whether the layout is a *continuous* layout, i.e. whether it continuously updates the vertex positions. Graph Playground only has one continuous layout: the force based layout discussed in Section 2.3.9. The classes implementing the layout interface are:

1. `CircularLayout`: for laying out the vertices in a circle.

2. `BipartiteLayout`: for laying out bipartite graphs with one independent set in each column.

3. `GridLayout`: for laying out vertices as points on a rectangular grid.

4. `ForceBasedLayout`: for laying vertices using attractive forces between adjacent vertices and repulsive forces between non-adjacent vertices.

5. `FixedLayout`: for representing a fixed, unchanging layout of the vertices.

3.2.4 Decorator Interface and Classes

The `Decorator` interface acts as a bridge between the algorithm implementations in the system and the drawing and UI modules, enforcing separation of concerns and loose coupling between distinct modules. The methods of the `Decorator` interface are shown in the UML class diagram shown in Figure 29.

Decoration States

As can be seen, most of the methods of the interface are concerned with setting 'Decoration States' for vertices or edges. 'Decoration States', encoded by the class `DecorationState`, are a concept used to let algorithm implementations make visual changes to the graph drawings they are operating on without having to worry about the details of how such visual changes are done. The algorithm can set vertices or edges to specific decoration states and the implementation of the decorator will change how the vertex or edge is drawn. For instance, in the default decorator in the system, vertices or edges with the `DISABLED` decoration state are shown grayed out. The full list of decoration states is

1. `DEFAULT`, which is used to convey that the vertex or edge is displayed as it is usually displayed in the absence of decorations. All vertices and edges are in this state initially.

2. `CONSIDERING`, meant to convey that the algorithm is considering or examining this vertex or edge, for example for inclusion in the final output. The default decorator colors vertices and edges orange to show this state. Edges are dashed.

3. `DISABLED`, meant to convey that the algorithm is yet to look at this vertex or edge, or has excluded it from consideration. The default decorator grays out vertices and edges to show this state.

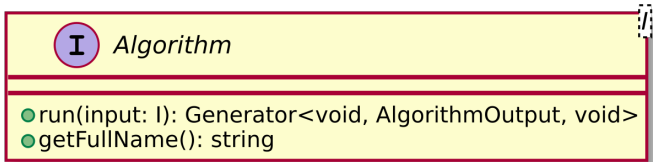
4. `SELECTED` is meant to convey that the algorithm has selected this vertex or edge in some sense, for example for inclusion in the final output or that it is considering the vertex or edge in a stronger way than vertices or edges assigned the `CONSIDERING` state. The default decorator colors vertices and edges teal-blue to show this state. Edges are drawn with a thicker stroke and vertex circles are filled.

5. Additionally, 'auxiliary states' are states that can be generated on demand and are distinct from all other states. They are useful if an algorithm needs more levels of visual distinction. Each auxiliary state has a numeric id and will be shown using a color distinct from the colors used for other auxiliary states and normal states.

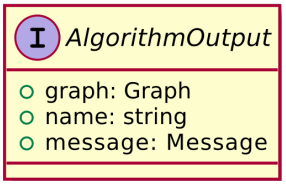
Edge Labels and Vertex Labels

The decorator interface also provides methods that allow an algorithm to set and clear labels for edges and vertices. Edge labels are arbitrary pieces of text that can be associated with edges, and they are shown in the graph drawing beside the edges. If the graph is a weighted graph, the edge label will prevent the weight from being displayed. The weight is displayed once again when the edge label is cleared.

Vertices have two kinds of labels: the ones that are stored in the Graph implementation itself, (See Section 3.2.1 for more detail. We will call these *internal labels* as they are displayed within the vertex circle.) and *external labels* that can be set from the decorator. While internal labels are stored with the graph data itself, external labels are comparatively ephemeral in the sense that they are treated like decoration states and edge labels: they are not stored when graphs are saved or bookmarked and are lost when the decorator's `clearAllDecoration()` method is



(i) The Algorithm Interface



(ii) The AlgorithmOutput Interface

Figure 30: UML Class Diagrams of the Algorithm Interface and the AlgorithmOutput Interface

called.

External vertex labels are useful when an algorithm needs to display extra information associated with an edge. The implementation of Dijkstra's Algorithm in the system uses external labels to show the distance from the source to a vertex.

3.2.5 Algorithm Interface and Classes

Another important interface in the system is the algorithm interface, and its methods are displayed in the UML class diagram shown in Figure 30 (i). The `getFullName()` method is implemented by algorithms to get the name of the algorithm to be displayed in the algorithm control panel. The other method in the interface is the `run()` method, which is called to execute the algorithm. `run()` also takes an input whose type depends on the type parameter of `Algorithm`. The type parameter can be void for algorithms that take no input. It should be noted here that the fact that algorithms take a graph as an input is implicit, and in implementations of algorithms, passing the graph is achieved through passing a `Decorator` object to the constructor of the algorithm class. Algorithms can retrieve the graph by calling the `Decorator.getGraph()` method.

The `run()` method is a generator method and its implementations should relinquish control from time to time using the `yield` statement to allow decoration changes to the graph drawing to be actually visible. The motivation for this

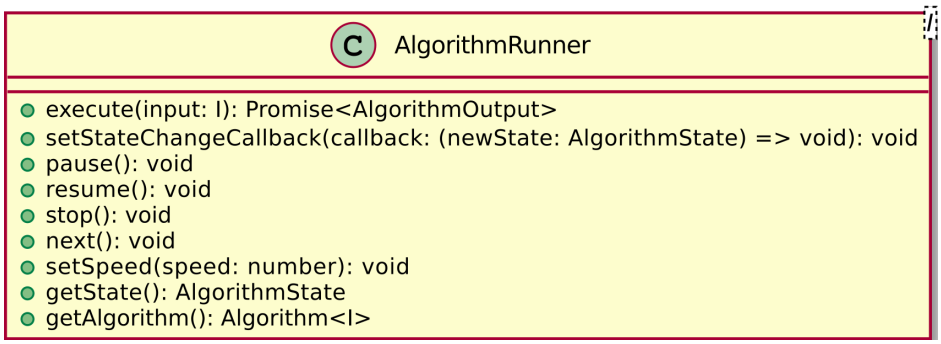


Figure 31: UML Class Diagram of the AlgorithmRunner class, showing only public methods

design choice is discussed in more detail in Section 3.4.2 and the interface between algorithm implementations and the rest of the system is discussed in Section 3.5.

The type of the value that the `run()` method returns is called `AlgorithmOutput`, and its contents are shown in the UML class diagram in Figure 30 (ii). The elements of the interface are discussed in Section 3.5.

A related class is the `AlgorithmRunner` class, whose UML class diagram is shown in Figure 31. It provides methods for implementing a stateful approach to algorithm execution, enabling essentially the core of the functionality of the algorithm control panel buttons.

3.2.6 UI Handlers and Components

The final module we discuss here is the user interface module, the main parts of which are discussed under the sub-headings below. We call the visible parts of the user interface the *components*, and the code that handles interaction with them the *handlers*.

Tools

Tools are implemented using the `Tools` class which handles user interaction with the toolbar on the left sidebar and lets the drawing module request the cur-

rently selected tool, letting it decide how to handle events for each tool. The `Tools` class also sets the appropriate cursor for each tool. Currently there are three tools included: (i) Add/Move Tool (ii) Delete Tool and (iii) Text Tool. The use of these tools to create and manipulate graphs is discussed in Section 2.3.

Algorithm Control Panel

The algorithm control panel is a UI component that sits below the graph canvas when an algorithm has been activated for a graph and contains buttons and inputs to allow the user to control the execution of the algorithm. The algorithm control panel is implemented by the `AlgorithmControls` class. It extends the browser's `HTMLElement` class, and uses a template to instantiate its user interface. Like the `Algorithm` interface and the `AlgorithmRunner` class, `AlgorithmControls` is parameterized by the type of input the algorithm takes. There are three implementations for each type of input currently possible in the system: `InputlessControls` for algorithms that take no input, `VertexInputControls` for algorithms that take one vertex as an input, and `SourceSinkInputControls` for vertices that take a source and a sink vertex as input. Algorithms requiring a different kind of input will require the implementing the corresponding kind of subclass to run.

Bookmarked Graphs

The bookmarked graphs feature is implemented by the `BookmarkedGraphs` class, which handles UI interactions with the list of bookmarked graphs in the left sidebar and the 'Bookmark' button in the top bar dropdown menu. `BookmarkedGraphs` uses `LocalGraphDrawingStore` to store graph drawings in the browser's local storage. The `LocalGraphDrawingStore` class provides methods for storing and retrieving JSON representations of graph drawings from the local

storage.

TabBar and GraphTabs

The tab bar is implemented by a `TabBar` class, which, like `AlgorithmControls`, extends `HTMLElement`. It provides methods for creating and listening to status updates on a set of tabs, without any regard to what is being displayed using the tabs. In fact, the tab bar is functionally independent of the graph canvas. The class `GraphTabs` ties the graph canvas and the tab bar together by listening to changes in the tab bar and updating the graph canvas accordingly. Since the `GraphTabs` class is the part of the system that places a graph drawing in the canvas and switches graphs when tabs are switched, it is the only part of the system that knows which graph drawing is currently being displayed to the user. As a result, an instance of `GraphTabs` is passed to other components that need to get the currently active graph drawing, and the components call the `getActiveGraphDrawing()` method of `GraphTabs` for that purpose.

Other UI Components

Other notable user interface components are listed below:

1. The auto-layout buttons panel, handled by the `AutoLayout` class.

2. The graph drawing display customizer which lets users set the size of vertices and font size of edge weights. It is handled by the `DisplayCustomizer` class.

3. The auto-label mode selector, which lets users select how they want the vertices to be labeled. It is handled by the `AutoLabelOptions` class.

4. The import-export manager, which provides functionality for saving and opening saved graphs. It is handled by the `ImportExport` class.

5. The algorithm menu, which activates the correct algorithm when the user clicks on the name of an algorithm from the top bar dropdown menu. It is handled by the `AlgorithmMenu` class.

3.3 Choice of Tools and Technologies

The choices of which tools, technologies and platforms to use to develop the system were made relatively early in the timeline of the project, based on informed guesses about the direction the development process would take and the needs that would appear later in the process. In the sections below we discuss the motivations for each of the major choices, and describe the strengths and limitations imposed by the choice.

3.3.1 Platform

Deciding that the system would be developed as a web-based application was a very straightforward choice, given the design goal that the system should be easy and fast to access. Creating a desktop application would mean that the user would have to go through the process of downloading and installing the application, with steps different for each operating system. On the other hand, only a URL and a browser is necessary to use a web application, and accessing a URL in a browser takes only a few moments. The application does not involve any login process for the same reason, and presents the full user interface to any user who accesses the system. In fact, there is no need for the user to log in at all, because the system does not have a server-side component other than . storage of the application files—there is no data stored in the cloud. The user has the option of storing graphs in the browser or in their computer.

3.3.2 Programming Language

Using the web as the development platform severely limits the number of programming languages available for use. Even though many other languages can be used for front-end web development using extra steps like compatibility layers, JavaScript is the only high-level language that all modern browsers support for

scripting in the front-end. However, instead of using plain JavaScript, the decision was made to use TypeScript as the language for development. TypeScript is a superset of JavaScript that adds static typing. The TypeScript compiler transpiles TypeScript to plain JavaScript code before it is run on a browser. The static typing of TypeScript is an attractive feature because type-checking catches many errors in the compilation stage which would otherwise appear only at runtime with dynamically typed languages. As a result, making the decision to use TypeScript for the development of the system was not very difficult.

3.3.3 Frameworks and Libraries

Since there are a great number of JavaScript/TypeScript frameworks and libraries for almost every conceivable use case on the web, choosing which ones to use for canvas drawing and the user interface was a considerable challenge.

There were several options considered for the canvas drawing library. PaperJS [2], KonvaJS [3] and FabricJS [4] were strong contenders. Among them, KonvaJS was selected as the library of choice because it was found to be simpler, more lightweight and easier to use than the alternatives.

Initially ReactJS [5] was considered as a possible choice for the user interface framework. However, initial attempts to build a prototype user interface with ReactJS that incorporated a graph canvas, a tab bar, and sidebar controls that could change the behavior of the objects in the graph canvas suggested that ReactJS would not be a good choice for the project. Specifically, ReactJS enforces a design philosophy that involves creating cohesive modules called *components*. Components contain immutable objects called *properties* or *props* that describe the state of the component. Components can be composed in the sense that larger components can contain smaller ones, and properties are passed top-down from the containing components to the contained components. This design philosophy was

found to be at odds with the natural design of the system, where we have a large amount of complex state data contained within KonvaJS objects. The KonvaJS objects that store drawing state also represent the drawings themselves, and there are a lot of complex interactions between between the drawing objects and the rest of the system. As a result it was decided not to use any user interface framework in the system.

Even though no user interface framework is employed, a CSS library called Bootstrap [6] is used to enforce a consistent look-and-feel across the system, especially for user interface elements like buttons, dialog boxes, etc. The JQuery [7] JavaScript library is used to make the access and manipulation of HTML elements easier.

3.4 Design and Implementation Problems Encountered

During the course of design and development of the system, several problems and challenges occurred. The sections below discuss the details of each design and implementation challenge and how they were solved in the system.

3.4.1 Separation of Rendering and Representation of Graphs

An observation made early in the development of the system was that the choice of the canvas drawing library could be changed later. Excessively close coupling between the state of the graph drawing and the objects of the drawing library would make the switch difficult. Allowing for change was especially important, as it was observed that although KonvaJS was easy to use and lightweight, it lacked some features required by our system. Additionally, the API documentation for some interfaces were sometimes insufficient or even missing.

A second motivation for separating the state of the graph drawing from the drawing objects of the graph canvas is the need to serialize and de-serialize the drawing objects for saving and opening saved graph drawings. A saved graph graph drawing file should contain only the information required to recreate later the graph drawing exactly as it looks like when saved, and no more. If we effectively decouple the representation of the visual state and the drawing of the state, we can just serialize the 'drawing state object' without worrying if we are serializing more information than is necessary.

Yet another concern that seems to support the separation of these two things is that algorithms manipulate the appearance of graphs, and in a well-designed system the algorithms should not know nor care about what library is used to draw the graph. So with the correct decoupling, the algorithm should just be able to update the 'drawing state object', and the drawing library should automatically respond by updating the drawing.

In the end, even though the aforementioned reasons supported a separation between the representation and rendering of graphs, it was decided that the separation would be kept for later. There were a couple of reasons. First, the `GraphDrawing` class already has a `toJSON()` method that lets us extract a serializable representation of the graph drawing, containing just the necessary elements. That function is used for creating file exports of graph drawings. A suitable alternative to KonvaJS that provided similar simplicity and ease of use wasn't found either. Second, with the `Decorator` interface being created for interaction between algorithms and the graph drawing, it was realized that even allowing algorithms to change the drawing state of the graph drawing was too much coupling, and algorithm implementations should not care about drawing attributes. For this reason, `Decorator` provides algorithms a high-level view of what can be done to a graph's vertices and edges, as discussed in Section 3.2.4.

3.4.2 Separation of Algorithm Implementation from Rest of the System

Since the system has easy extensibility as a design goal—especially when it comes to adding new algorithms—loose coupling between algorithm implementations and the rest of the system is a must. Ideally, someone implementing a new algorithm should not have to know how the rest of the system is implemented, including how graphs are drawn. This kind of 'spatial' separation is more or less straightforward to implement, as a clean bridge interface like the `Decorator` abstracts away the details of the rest of the system and provides a set of high-level operations that an algorithm can perform.

The other kind of separation necessary, namely 'temporal' separation between the algorithm and the rest of the system, turned out to be much more difficult to solve. In this case, temporal separation stands for the requirement that an

algorithm implementation should be able to perform its own operations and decorator calls without worrying about the speed at which it is executing, and without concern for the kind of environment where it executes. The latter is especially important: algorithm implementations should not have to worry about who calls them. On the other hand, the part of the system running the algorithm should be able to choose whether to execute the algorithm all at once to get the final output, or run it step-by-step, observing the results. This problem is uniquely challenging in the browser's JavaScript environment because everything in a browser runs in a single thread. If multiple threads were possible, we could simply execute the algorithm in a separate thread by itself, perhaps with `sleep()` calls of a variable length interspersed throughout, to execute the algorithm at an appropriate pace. Then the algorithm could be implemented any way the implementer likes.

But since everything executes in a single thread in the browser, the main event loop of the browser executes in the same thread as any code in our system. If an algorithm takes more than a trivial amount of time to execute, the user will notice this as the whole user interface freezes, not responding to the user at all. Since we want the user to be able to pause, resume or stop the algorithm while it is executing, allowing the user interface freeze is not an option. To solve this problem, the approach adopted initially was to require algorithms to be implemented with a `step()` function, which would do a small amount (for instance, an iteration of an outermost loop) of work and then return. The `step()` function would be called at regular intervals by the rest of the system while the algorithm is executing, but crucially the intervals between the `step()` calls would allow the event loop to run, responding to user interaction. The algorithm could signal that it has no more steps left to execute with a special return value.

While the aforementioned approach is fine for most algorithms, it enforces an

artificial restriction on algorithm implementations that is a product of the peculiarities of the platform chosen for the system. Furthermore, some algorithms have recursive implementations that are cleaner and easier to write and understand than their iterative counterparts, and using the aforementioned approach effectively prevents such recursive implementations without a fair amount of trickery. Therefore, alternatives to the approach were sought, and initially it appeared as if there was no way for an algorithm to be written in isolation as was required. Eventuallky two potential solutions were found. One was to use Web Workers, which is a relatively new feature of modern web browsers that allows execution of code in a separate thread. However, using Web Workers means that code executing in a web worker can only communicate with the rest of the system using a messaging interface, so direct function calls to `Decorator` would have to be bridged by implementing a special purpose RPC (Remote Procedure Call) layer. Fortunately the another solution was found that solved the problem without having to write a lot of extra code. The solution was to use *generator functions*, which is a feature of many modern high-level languages including JavaScript. A generator function is simply a function that has an additional way of relinquishing control back to the caller besides the usual return statement: the `yield` statement. Callers of the function access the result of the generator function as if they were accessing elements from an iterator, reading values consecutively until there are no more values. When a generator function `yield`s, control is transferred back to the calling part, but crucially the state of the function (its execution context) is maintained as-is. When the caller requests the next value from the generator function, execution continues from just after the `yield` statement. In our case, we don't need algorithms to pass back values when `yield`ing control back, we only need a result when the function returns.

The use of generators is reflected in the return type of the `run()` function in the `Algorithm` interface. Specifically, the return type is `Generator<void, AlgorithmOutput, void>`, whose type parameters indicate, respectively, that the `yield` statement doesn't pass any value back to the caller, the `return` statement passes back a value of type `AlgorithmOutput`, and that the caller does not need to pass a value to be received by the `yield` statement. This is discussed in more detail in Section 3.5 and an example implementation is provided in Section 3.6.

3.4.3 Placement of Edge and External Vertex Labels

Automatic label placement is a problem that has been studied in depth, especially in the context of labels for maps but also in the context of graphs [8]. The goal of most labeling algorithms is to place labels in such a way that a set of criteria are met as much as possible, like labels being close to the items they are for, and overlap between labels being minimized. Despite the existence of a large amount of literature and potentially better algorithms, due to time limitations and the need to focus on other problems, a simple approach was adopted to decide the automatic placement of edge labels (weights) and external vertex labels.

Edge labels are placed according to an *anti-centroid* approach, meaning that they are placed on the side of the edge that is away from the centroid of the vertex positions. The motivation for this approach is that other edges are likely to be where other vertices are, and this approach eliminates overlap between the label and other edges for at least the edges that are on the convex hull of the vertex set. The centroid of the vertex positions is obtained from `GraphDrawing`, and the `EdgeDrawing` class uses the positions of its two endpoints (the two vertices it is incident to) to compute a vector that is orthogonal to the line running between the two vertices, but pointing in a direction away from the side of the centroid. Then this direction vector is added to either the midpoint of the edge or (if the

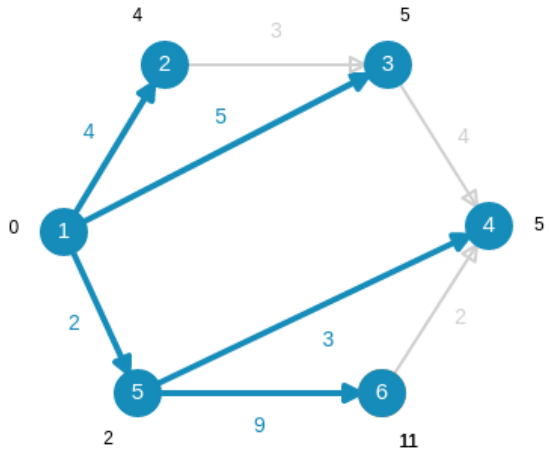


Figure 32: Edge labels (weights) and external vertex labels on a graph

edge has one) the curve point to obtain the position of the edge label.

For external vertex labels, since there can be a handful of edges incident to a vertex, the goal is to minimize the chance of overlap between vertex labels and edges. Intuitively, the approach that seems to do this satisfactorily is to place the vertex label between that pair of edges for which the angle between them is largest. The label is placed such that the angle it makes with the vertex center bisects the aforementioned largest angle. In case there is only one edge incident, the label is placed opposite to that edge. If there are no edges incident, the label is placed in the direction opposite to the centroid of the vertices in the graph.

As can be seen in Figure 32, the approaches used to place external vertex labels and edge labels produce satisfactory and somewhat visually pleasing label positions.

3.5 Algorithm API

In this section, the Algorithm Application Programming Interface (API) is described. The Algorithm API is the set of interfaces and operations used by algorithm implementations, the primary ones being for the purpose of making visual changes to the graph drawing.

Most parts of the algorithm API have already been discussed, and this section will restate them in a cohesive manner. As mentioned earlier, algorithm implementations need to implement the `Algorithm` interface. The `getFullName()` method needs to return a name for the algorithm, and the `run()` method is the entry point of the algorithm. Since most algorithms will need to make changes to the graph drawing, they will need an instance of a `Decorator`. The constructor of the class implementing the algorithm should take a single argument of type `Decorator`, and no more arguments. The graph itself can be retrieved using the `getGraph()` method in the decorator. The `Algorithm` interface has a type parameter that is the input type of the `run()` method, and the type parameter should be appropriately set according to the type of extra input (other than the graph) the algorithm requires.

Inside the body of the implementation of `run()`, calls to decorator methods and `yield` statements should be placed appropriately according to the behavior desired. Changes made to the appearance of the graph drawing (decoration states and labels) will not be visible to the user until the next `yield` statement, so `yield` statements need to be placed as often as visual updates are desired. It should be understood that a fixed interval of time will elapse between the execution `yield` statement and the execution of the next statement, so more `yield` statements will make the algorithm appear to run slower. The exact amount of time that will elapse between suspension of execution on the `yield` statement and resumption of

execution depends on the speed set on the algorithm control panel by the user.

The return value of `run()` should be of type `AlgorithmOutput`, which is an object type with three properties, as can be seen in Figure 30 (ii). The `graph` property is the output graph that the algorithm generates, for instance the search tree for depth first search. Algorithms that don't have an output can set it to `null` or `undefined`. The `name` property specifies the name to be used for the output graph, used for things like tab titles and file names. This property is required only if `graph` contains a valid graph. The `message` property specifies the message to be displayed to the user, and consists of a message level (indicating whether it is a failure, success, warning or just some information), the message text and the message title. Messages in the output are optional as well and can be set simply to `null` or `undefined`.

After the implementation of the `Algorithm` interface is complete, the algorithm needs to be registered with the system to be accessible by users. The registration is done in the `AlgorithmMenu` class, which contains a static member containing a list of all algorithms in the system. A portion of the list of algorithms is shown below.

```
1
2  type AlgorithmType<I> = new (d: Decorator) => Algorithm<I>;
3  type ControlsType<I> = new (algClass: AlgorithmType<I>,
4      graphTabs: GraphTabs, graphDrawing: GraphDrawing) => AlgorithmControls;
5
6  interface MenuEntry<I> {
7      controlsClass: ControlsType<I>;
8      algorithmClass: AlgorithmType<I>;
9      menuText: string;
10 }
11
12 export default class AlgorithmMenu {
13
14     static readonly algorithms: MenuEntry<any>[][] = [
```

```
15          [
16              {
17                  controlsClass: InputlessControls,
18                  algorithmClass: KruskalMST,
19                  menuText: "Kruskal's Minimum Spanning Tree",
20              },
21              {
22                  controlsClass: VertexInputControls,
23                  algorithmClass: PrimMST,
24                  menuText: "Prim's Minimum Spanning Tree",
25              },
26          ],
27          [
28              {
29                  controlsClass: VertexInputControls,
30                  algorithmClass: BreadthFirstSearch,
31                  menuText: "Breadth First Search",
32              },
33              {
34                  controlsClass: VertexInputControls,
35                  algorithmClass: DepthFirstSearch,
36                  menuText: "Depth First Search",
37              },
38          ],
39          ...
40      ];
41      ...
42  }
```

As seen above, an algorithm menu entry (whose type contract is defined by the `MenuEntry` interface) consists of (a) `controlsClass`, whose value should be the class name of a subclass of AlgorithmControls capable of asking the user for an input of the type the algorithm requires, (b) `algorithmClass`, whose value should be the class name of the algorithm, and (c) `menuText`, the text to be displayed on the menu item for this algorithm. `MenuItem`s grouped together in an inner array will be displayed together, divided from other groups with a horizontal divider in the menu.

After the algorithm menu entry is placed in the correct location, the system should automatically read the entry and populate the menu, thus allowing the user to execute the algorithm.

3.6 Extending the System with a New Algorithm

In this section a step-by-step example of implementing a new algorithm for the system is provided. The demonstration will be of a simple depth-first search based algorithm to count the number of connected components in an undirected graph.

3.6.1 Creating an Algorithm Class

Creating a class implementing the `Algorithm` interface is the first thing that needs to be done. The following code listing shows a syntactically correct class implementing the interface, but one that has no functionality of the algorithm itself implemented. Imports of the symbol names are omitted because import statements are dependent on the relative locations of the class files, which can change when the project's code structure is reorganized.

```
1
2  export class CountComponents implements Algorithm<void> {
3
4      constructor(private decorator: Decorator) {
5      }
6
7      *run() {
8          return {
9              graph: null,
10             name: null,
11             message: null
12         }
13     }
14
15     getFullName() {
16         return "Component Count Algorithm";
17     }
18 }
```

The class shown above takes a decorator as an argument and assigns it to a private member of the same name. It also has a `run()` function that just returns

an `AlgorithmOutput` object with null values. Finally there is a `getFullName()` function that returns the name.

3.6.2 Programming the Algorithm

After the previous step, we have a place where we can write the code for the algorithm. We will first write the code for the algorithm and insert decorator calls in the next step. The input graph will be obtained through the decorator and the number of components will be returned in the `AlgorithmOutput`'s message section to be displayed to the user.

The following code listing shows a working implementation of the DFS-based approach to counting the number of connected components. Only the `run()` function is shown here, because the rest of the class remains the same.

```
1  *run(): Generator<void, AlgorithmOutput, void> {
2      const graph = this.decorator.getGraph();
3      // Create a set of vertices we haven't yet visited. Initialize with all
4      // vertices.
5      const unvisited = graph.getVertexIds();
6      var components = 0;     // Variable to count the number of components
7      while (unvisited.size > 0) {
8          // Get one element from the unvisited set
9          const vertex = unvisited.values().next().value;
10         // Create an array to serve as a stack for DFS
11         const stack = [vertex];
12         while (stack.length > 0) {
13             // Pop a vertex from the stack, remove it from unvisited
14             const v = stack.pop();
15             unvisited.delete(v);
16             // Push all of its unvisited neighbors to the stack,
17             // disregarding edge direction for directed graphs
18             for (const n of graph.getVertexNeighborIds(v, true)) {
19                 if (unvisited.has(n)) {
20                     stack.push(n);
21                 }
22             }
```

```
23          }
24          // We're done exploring all vertices in this component
25          components++;
26      }
27      // Return output with message
28      return {
29          graph: null,
30          name: null,
31          message: {
32              level: "success",
33              title: "Execution Complete",
34              text: components.toString() + " components found in the graph."
35          }
36      };
37  }
```

Comments are placed at appropriate lines in the above listing to describe what the lines of code are doing. Essentially, all vertices are initially placed in a set called 'unvisited' and, as we do a depth first traversal of a component, we remove vertices from the set. If there are vertices remaining in the set after completing a depth first traversal, we increment the count of components and do another traversal. In the end we return an `AlgorithmOutput` object with a message stating the number of components.

3.6.3 Using the Decorator and Yield

At this point, the algorithm we just implemented runs on graphs and produces the correct result, but our implementation does not show any intermediate steps to the graph and does not make any decorations to the graph drawing. In this section, we will use the decorator passed to the class constructor and include some `yield` statements so the user can see the decoration.

The code for the `run()` method with decorator calls and `yield` statements added is shown in the listing below.

```
*run(): Generator<void, AlgorithmOutput, void> {
    const graph = this.decorator.getGraph();

    // Set all vertices and edges to disabled state
    for (const vertex of graph.getVertexIds()) {
        this.decorator.setVertexState(vertex, DecorationState.DISABLED);
    }
    for (const edge of graph.getEdgeList()) {
        this.decorator.setEdgeState(edge[0], edge[1], DecorationState.DISABLED);
    }
    yield;  // Let the decorations be displayed

    // Create a set of vertices we haven't yet visited, initially all of them.
    const unvisited = graph.getVertexIds();
    var components = 0;     // Variable to count the number of components
    while (unvisited.size > 0) {
        // Create a graph from this component for decoration later
        const thisComponent = new UnweightedGraph(false);
        // Get one element from the unvisited set
        const vertex = unvisited.values().next().value;
        // Create an array to serve as a stack for DFS
        const stack = [vertex];
        unvisited.delete(vertex);
        thisComponent.addVertex(vertex);
        this.decorator.setVertexState(vertex, DecorationState.SELECTED);
        this.decorator.setStatusLine("Exploring component " + (components + 1));
        yield;
        while (stack.length > 0) {
            // Pop a vertex from the stack, remove it from unvisited
            const v = stack.pop();

            var newPushed = false;  // For the yield later
            // Push all of its unvisited neighbors to the stack,
            // disregarding edge direction for directed graphs
            for (const n of graph.getVertexNeighborIds(v, true)) {
                if (unvisited.has(n)) {
                    unvisited.delete(n);
                    stack.push(n);
                    thisComponent.addVertex(n);
                    this.decorator.setEdgeState(v, n, DecorationState.SELECTED);
```

```
41                      this.decorator.setVertexState(n, DecorationState.SELECTED);
42                      newPushed = true;
43                  }
44                  thisComponent.addEdge(v, n);
45              }
46              if (newPushed) {
47                  yield;
48              }
49          }
50          // We're done exploring all vertices in this component
51          components++;
52
53          // Set vertices and edges of this component to an auxiliary state
54          const auxState = DecorationState.getAuxiliaryState(components - 1);
55          for (const v of thisComponent.getVertexIds()) {
56              this.decorator.setVertexState(v, auxState);
57          }
58          for (const e of thisComponent.getEdgeList()) {
59              this.decorator.setEdgeState(e[0], e[1], auxState);
60          }
61          this.decorator.setStatusLine(thisComponent.getVertexIds().size +
62              " vertices found in component " + components);
63          yield;
64      }
65      this.decorator.setStatusLine("Graph has " + components + " components");
66      // Return output with message
67      return {
68          graph: null,
69          name: null,
70          message: {
71              level: "success",
72              title: "Execution Complete",
73              text: "Graph has " + components + " components"
74          }
75      };
76  }
```

Lines 4–11, 17–18, 24–27, 32, 39–42, 44, 46–48, 53–63 and 65 contain the code inserted for decoration and `yield`ing. Initially we set all vertices and edges to the

DISABLED state. Then, as we visit vertices in the depth first search, we set the visited vertices and discovered edges to the SELECTED state. When we have visited all the vertices in a component, we set all the vertices and edges of that component to an auxiliary state, so that different components are shown in distinct colors and are distinguishable. We also send status messages to the decorator for displaying to the user as we explore vertices and as components are found.

3.6.4 Registering the Algorithm with the System

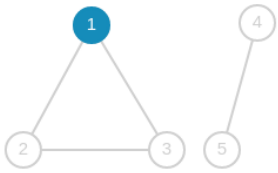
The final step necessary is the registration of the algorithm with the system, which can be done by inserting a MenuEntry object in the AlgorithmMenu class. The MenuEntry object interface and the list of menu entries is described in Section 3.5. Here we will just show the MenuEntry object that needs to be inserted.

```
1 {
2     controlsClass: InputlessControls,
3     algorithmClass: CountComponents,
4     menuText: "Component Count",
5 }
```

Placing this object in an inner array within the static array algorithms in the AlgorithmMenu class enables the system to read the entry and places an item named 'Component Count' in the algorithms dropdown menu, from which the procedure can be executed.

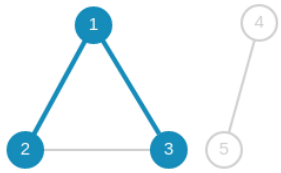
3.6.5 The Result

In this section, we show the result of the algorithm implemented in the previous sections. Step-by-step pictures of the decorations made by the algorithm to the graph are displayed in Figure 33. As shown, the requested decorations are performed on the graph as vertices and edges are set to the 'SELECTED' state as we traverse a component, and an auxiliary state is set for the component after



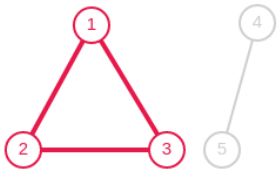
Exploring component 1

(i) DFS traversal starting at 1



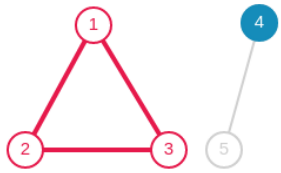
Exploring component 1

(ii) DFS continuing to 1 and 2



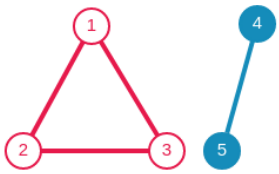
3 vertices found in component 1

(iii) Component 1 fully traversed



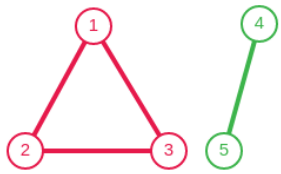
Exploring component 2

(iv) DFS traversal starting at 4



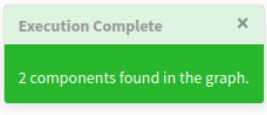
Exploring component 2

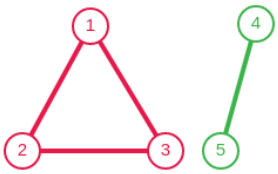
(v) DFS continuing to 5



2 vertices found in component 2

(vi) Component 2 fully traversed





Graph has 2 components

(vii) Output message with component count

Figure 33: The DFS-based component counting algorithm in action

completion of the traversal. The figure also shows the status line with the text set by our decorator calls, and the output message displayed in the message area for the final step. As we can see, the algorithm correctly finds that there are 2 components in the graph.

List of References

[1] *ECMAScript 2021 Language Specification*, Ecma International, 6 2021. [Online]. Available: https://tc39.es/ecma262/multipage/keyed-collections.html#sec-map-objects

[2] J. Lehni and J. Puckey, "Paperjs," accessed: 2021-07-15. [Online]. Available: https://paperjs.org/

[3] A. Lavrenov. "Konvajs." Accessed: 2021-07-05. [Online]. Available: https://konvajs.org

[4] "Fabric.js," The FabricJS Team, accessed: 2021-07-15. [Online]. Available: https://fabricjs.com/

[5] "React - a javascript library for building user interfaces," Facebook Inc., accessed: 2021-07-15. [Online]. Available: https://reactjs.org

[6] "Bootstrap," The Bootstrap Team, accessed: 2021-07-15. [Online]. Available: https://getbootstrap.com/

[7] "Jquery," The OpenJS Foundation, accessed: 2021-07-15. [Online]. Available: https://jquery.com/

[8] K. G. Kakoulis and I. G. Tollis, "Force-directed drawing algorithms," in *Handbook of Graph Drawing And Visualization*, R. Tamassia, Ed. CRC Press, 2013, ch. 15, pp. 489–515.

CHAPTER 4

Conclusions

4.1 Review of Goals Achieved

The primary design goals of the project were to build a system that (a) allows users to visually create and manipulate graphs, and (b) provides a set of algorithm implementations along with visualizations of their steps. The scope of the features and algorithms was planned to be more or less restricted to the ones necessary for teaching introductory graph theory and algorithms courses like CSC 340 and CSC 440, respectively, at URI.

The current implementation of the project contains most of the planned features and algorithms, and it is capable of supplementing conventional teaching methods for the following topics:

1. Graph isomorphism
2. Graph planarity
3. Kruskal's and Prim's minimal spanning tree algorithms
4. Breadth First Search and Depth First Search
5. Dijkstra's Shortest Paths Algorithm
6. Fleury's Euler Circuit Algorithm
7. Bellman-Held-Karp Algorithm for Hamilton Circuits
8. Hopcroft-Tarjan Algorithm for Biconnected Components and Articulation Points
9. Bellman-Held-Karp Algorithm for the Traveling Salesman Problem

10. Approximation Algorithms for the Traveling Salesman Problem

    (a) Nearest-neighbor heuristic

    (b) Nearest-insert heuristic

    (c) Cheapest-insert heuristic

    (d) Minimum Spanning Tree based algorithm

    (e) Christofides Algorithm

11. Edmonds-Karp Network Flow Algorithm

There are some items conspicuously missing from the list above, and a selection of them are discussed in the next section. However, the system as it exists now covers or touches on most of the graph-based topics taught in courses like CSC 340 and CSC 440, and that the undertaking was more or less successful in reaching its goals.

4.2 Future Work

There is a lot of potential for future work that can be done on the system, as the list of algorithms and features of the current implementation lacks some items that would be desirable in a system for graph theory pedagogy. The most notable among these are:

(i) Features to support manual graph coloring by users

(ii) Features to support multigraph creation

(iii) User-editable external vertex labels

(iv) User-editable edge labels

(v) Algorithms to find minimal graph colorings

(vi) Algorithms to test isomorphism

(vii) Algorithms to test planarity

Other features and algorithms would enhance the coverage, usability and visual appeal of the system include:

(i) Layout algorithms for minimum-crossing layout

(ii) Support for altering graph layout through the decorator interface

(iii) Graph operations like complement, dual and line graph

(iv) Topological sorting algorithms

The fact that the source code for the system is open-source and published under the GNU General Public License Version 3 (GPLv3) enhances its extensibility and the prospect for future work. The algorithm API as described in Section 3.5 has been designed to support extensibility and easy addition of new algorithms. With future work that builds on the work already done, the system can be made into a full-featured system for introductory graph theory and graph algorithm related pedagogy.

BIBLIOGRAPHY

"D3 graph theory - learn graph theory interactively." Accessed: 2021-07-02. [Online]. Available: https://d3gt.com/index.html

"Graph magics." Accessed: 2021-07-02. [Online]. Available: http://www.graph-magics.com/index.php

"Graph online." Accessed: 2021-07-02. [Online]. Available: http://graphonline.ru/en

"Graphanalyzer." Accessed: 2021-07-02. [Online]. Available: http://grafoanalizator.unick-soft.ru/en/

Bellman, R., "Dynamic programming treatment of the travelling salesman problem," *J. ACM*, vol. 9, no. 1, p. 61–63, Jan. 1962. [Online]. Available: https://doi.org/10.1145/321105.321111

"Bootstrap," The Bootstrap Team, accessed: 2021-07-15. [Online]. Available: https://getbootstrap.com/

Christofides, N., "Worst-case analysis of a new heuristic for the travelling salesman problem," Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1976.

Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, p. 269–271, Dec. 1959. [Online]. Available: https://doi.org/10.1007/BF01386390

Eades, P., "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

ECMAScript 2021 Language Specification, Ecma International, 6 2021. [Online]. Available: https://tc39.es/ecma262/multipage/keyed-collections.html#sec-map-objects

Edmonds, J., "Paths, trees, and flowers," *Canadian Journal of mathematics*, vol. 17, pp. 449–467, 1965.

"Fabric.js," The FabricJS Team, accessed: 2021-07-15. [Online]. Available: https://fabricjs.com/

"React - a javascript library for building user interfaces," Facebook Inc., accessed: 2021-07-15. [Online]. Available: https://reactjs.org

Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O., and Bader, G. D., "Cytoscape.js: a graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 09 2015. [Online]. Available: https://doi.org/10.1093/bioinformatics/btv557

Free Software Foundation. "Gnu general public license." [Online]. Available: http://www.gnu.org/licenses/gpl.html

Held, M. and Karp, R. M., "A dynamic programming approach to sequencing problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196–210, 1962. [Online]. Available: https://doi.org/10.1137/0110015

Hopcroft, J. and Tarjan, R., "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, p. 372–378, June 1973. [Online]. Available: https://doi.org/10.1145/362248.362272

Horowitz, E. and Sahni, S., *Fundamentals of Computer Algorithms*, ser. Computer software engineering series. Pitman, 1978. [Online]. Available: https://books.google.com/books?id=n8c8PgAACAAJ

Kakoulis, K. G. and Tollis, I. G., "Force-directed drawing algorithms," in *Handbook of Graph Drawing And Visualization*, Tamassia, R., Ed. CRC Press, 2013, ch. 15, pp. 489–515.

Kobourov, S. G., "Force-directed drawing algorithms," in *Handbook of Graph Drawing And Visualization*, Tamassia, R., Ed. CRC Press, 2013, ch. 12, pp. 383–408.

Kruskal, J. B., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956. [Online]. Available: http://www.jstor.org/stable/2033241

Lavrenov, A. "Konvajs." Accessed: 2021-07-05. [Online]. Available: https://konvajs.org

Lehni, J. and Puckey, J., "Paperjs," accessed: 2021-07-15. [Online]. Available: https://paperjs.org/

npm, Inc. "npm - npm." Accessed: 2021-07-05. [Online]. Available: https://www.npmjs.com/package/npm

"Jquery," The OpenJS Foundation, accessed: 2021-07-15. [Online]. Available: https://jquery.com/

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoč, D., "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2016, pp. 263–273.

Prim, R. C., "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

Rosenkrantz, D. J., Stearns, R. E., and Lewis, II, P. M., "An analysis of several heuristics for the traveling salesman problem," *SIAM journal on computing*, vol. 6, no. 3, pp. 563–581, 1977.

Rostami, M. A., Azadi, A., and Seydi, M., "Graphtea: Interactive graph self-teaching tool," *Communications, Circuits and Educational Technologies*, 01 2014.

The Web Hypertext Application Technology Working Group. "HTML Standard range state." July 2021. [Online]. Available: https://html.spec.whatwg.org/multipage/input.html#range-state-(type=range)

The Web Hypertext Application Technology Working Group. "HTML Standard web storage." Accessed: 2021-07-02. July 2021. [Online]. Available: https://html.spec.whatwg.org/multipage/webstorage.html

The Webpack Team. "webpack." Accessed: 2021-07-18. [Online]. Available: https://webpack.js.org/