University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Web-based Graph Algorithm Game

George Yarr

February 7, 2024

Abstract

Graph algorithms are a foundational computing science concept. They are traditionally taught through lectures with some worked examples, online articles, or visualisation websites (2.1). This project adds another dimension to the visualisation of graph algorithms by implementing a 'Do It Yourself' (DIY) game where the user must control which steps in the algorithm are performed on a visualised graph. The web-based graph algorithm game (WGAG) was developed to implement visualisations of Breadth-First Search, Depth-First Search, Dijkstra's Shortest Path Algorithm, and then a DIY approach for each. To solve the algorithms, the user must decide for themselves which action to take at each step, thus consolidating and reinforcing their learning.

The website was well-received and helped users understand these graph algorithms. The DIY approach was shown to consolidate the information learned from the visualisations.

For the website, see `gyr1967.github.io/graph-algorithm-game/`.

For the GitHub repository, see `https://github.com/gyr1967/graph-algorithm-game`.

Acknowledgements

I would like to thank Dr. Sofiat Olaosebikan for her support and supervision of this project. I would often walk into a meeting feeling unsure and anxious about this project, but would walk out with a plan of action and the motivation to complete it.

I'd also like to thank my family, and my friends Sasha, Andrés, Ross and Josh for their support throughout the year.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes.

Signature: George Yarr Date: 7 February 2024

Contents

1 Introduction

Graph algorithms are a foundational concept in computing science and are applied in many different fields, such as consumer map applications and discrete state space optimisation in artificial intelligence. There are multiple approaches to learning the fundamentals of the graph data structure and simple graph algorithms. Two of these approaches are the traditional university-taught method of presenting lecture slides with some worked examples and online articles and tools as seen in 2.1.

Where university lectures provide little hands-on learning, products such as online articles and tools can integrate interactivity and visualisations much more effectively, allowing for more interaction and self-paced discovery. However, existing products (2.1) are generally centred on visualisation, with no facility for the user to execute the algorithm themselves using the website. Users have to resort to pen and paper, which, is time-consuming, not very engaging, and doesn't provide feedback.

The motivation behind this project is to add another dimension to learning graph algorithms by providing a deeper level of interaction from the user and including them in the decision-making process to reach a success state. This is achieved by the user engaging in a 'Do It Yourself' (DIY) paradigm, executing graph algorithms themselves using the web-based graph algorithm game (WGAG).

The WGAG is aimed at learners with some programming experience who are at the level to take their first algorithms and data structures course.

1.1 Goals

The initial goals of this project are as follows:

- **To develop a web-based graph algorithm game that is fun and engaging for students to play.**
  - The game should be web-based, fast and deployed on a server.
  - The game should be easy to use and understand.
  - The game should be fun and engaging.
- **To develop a game that can only be completed efficiently when the user understands the graph algorithms.**
  - The better the user understands the graph algorithms, the more efficiently the game can be completed.
- **To have a smooth development process, with continuous integration and testing.**
  - The game should be developed using test-driven development.
  - The game should be regularly deployed.
  - GitHub actions should be used to run tests on push to develop.
  - GitHub actions should be used to deploy the game on push to master.
- **To develop a game that uses pedagogical techniques to teach graph algorithms.**
  - The game should be effective in teaching graph algorithms.

2 Background

This project aims to integrate more of the ideas that a constructivist learning model for teaching graph algorithms might exhibit, such as greater interaction and autonomy in solving problems. Schell (2013) says of the constructivist model:

> The constructivist model of learning opposes the objectivist idea that the best way to transmit knowledge is dissemination from expert to learner. Instead, proponents of the constructivist learning model argue that the learner should have more control over the learning process and that individuals learn better when they discover things independently. ... constructivist proponents believe that the process of determining the correct answer for oneself, or at least formulating an idea and thinking about the question, is a very important aspect of the learning process.

In the context of this project, visualisations are the instructor telling the answer to the student, and the DIY sections implemented in this project (5.3.3) represent a more constructivist learning process. This is the core idea that was to be brought to life, where thinking about how to solve an algorithm and then solving it is a powerful way to become familiar with and gain intuition about graph algorithms.

Whilst existing products (mentioned in 2.1) encourage the learner to think and formulate ideas, the process of 'doing it yourself' is not implemented in these products. This was to be implemented in some form and evaluated to see if it was received well.

Furthermore, there is support for increased interactivity and autonomy in online learning from Pelz (2010). Bill Pelz has three principles of online pedagogy that, whilst applied to online courses, are still relevant to this scenario. Two of them in particular:

- Let the students do (most of) the work.
- Interactivity is the heart and soul of effective asynchronous learning.

have strong relevance for this project. With these principles in mind, existing products were reviewed and critiqued.

2.1 Existing Products

The WGAG, in part, visualises graph problems. However, many products already exist that do this. A handful of websites were inspected to gain inspiration and inform the project requirements and implementation, including considering how their approaches could be modified/extended to allow for greater user interactivity with graphs.

2.1.1 VISUALGO.net

VISUALGO.net (Visualgo) is an online algorithm visualisation and quiz platform developed by Steven Halim, Felix Halim and student researchers at the National University of Singapore (NUS). Halim (2011) developed the website to teach students at NUS different algorithms.

- **Pedagogical Approach**

  - **Textual Explanations –** Each topic section has a series of explanations of the concepts taught in the section. These are optional and can be closed.
  - **Configurability –** Visualgo provides configurable visualisations of many different algorithms, including graph traversal algorithms. Users can choose the speed of playback and step through algorithms manually.
  - **Variety of graphs –** There are many different graphs on which visualisations can take place.
  - **State of Algorithm Shown –** The states of algorithms are shown via highlighted pseudo-code, and refinements for the current action being performed are shown next to the pseudo-code.
- **Usability**
  - **Visualisation Speed –** Whilst the option to step through is available, the slowest playback speed is still fast.
- **General Notes**
  - Visualgo is a prime example of what was sought to be built upon. It is a good learning tool but does not offer a DIY-style approach. A full page layout dedicated to the graph and algorithm is the most feasible layout to add DIY-style interaction.

2.1.2 A Visual Guide to Graph Traversal Algorithms by WorkShape.io

This interactive article is motivated by Dent (2016)'s frustration with their experience with the traditional delivery mechanism of the theory being presented in lecture slides with a single example. It is a long-form article that explains basic concepts about graph traversal, supplemented heavily by visualisations of algorithms performed on user-editable graphs.

- **Pedagogical Approach**
  - **Introduction, Example, Explanation –** The structure of the article for each algorithm follows the pattern of a short introduction to the algorithm, then the visualisation, and finally, the pseudo-code with a more in-depth explanation
- **Usability**
  - **Simple –** The article is clearly separated into sections, and each visualisation has a box with the graph, a box with the algorithm steps, and a bar at the top to configure the visualisation.
- **General Notes**
  - WorkShape.io presents a friendly, interactive way to understand graph traversal algorithms. The explanations give the user a linear journey, starting simple and then discussing the concepts' applications.

2.1.3 Introduction to the A* Algorithm from Red Blob Games

This is another interactive article of a similar style to WorkShape.io. Patel (2014) gives an in-depth, interactivity-dense, beginner-friendly tutorial on the A* search algorithm. The basics are covered, such as how a computer can see a real-world problem as a graph problem.

- **Pedagogical Approach**
  - **Interactivity –** Nearly every section has some way to interact with the graphics or visualisations. Hovering over certain parts of the text can change how the visualisations look.
  - **Drawing Comparisons –** Users can step through algorithms simultaneously that are shown side-by-side. A greedy best-first-search is shown next to a uniform cost search to show the difference in behaviour.
- **Usability**

  - **Simple –** Similarly to WorkShape.io, all interactions are clearly labelled, simple, and internally consistent.
- **General Notes**
  - The controls of stepping through the algorithm allow for scrubbing and skipping, which can help the user understand the big picture in path-finding grids; however, they may not be as useful for smaller graphs and are hard to implement dynamically.

2.1.4 algmatch

'algmatch' is a matching algorithm animator created by Lau (2021) and Ormond (2023) under the supervision of Dr. Sofiat Olaosebikan for their honours projects. It features multiple matching algorithms using the graph data structure to visualise them.

- **Pedagogical Approach**
  - **Pseudo-code and Justifications –** The visualisation includes the graph, variables, and the algorithm's pseudo-code with the step highlighted. The pseudo-code is complemented by justifications for each step, which helps rationalise what is happening in the graph.
- **Usability**
  - **Three-column Layout –** The pseudo-code on the left of the page, the graph in the middle, and the auxiliary information on the right is a clear structure from which this project's implementation has taken inspiration from.
- **General Notes**
  - This is a good example of the visualisation of algorithms. It has a pleasing interface, and it is clear what happens at each step.

2.1.5 Visualization of Classical Graph Theory Problems

This website is an honours project by Al-Sayegh (2023). It provides fun tutorials on the basics of different graph theory problems, with interactive quizzes and games to test and improve understanding. It is straightforward to use and ensures the games are very simple to complete whilst still demonstrating the underlying concept.

- **Pedagogical Approach**
  - **Textual Explanations –** Each concept has its own page with a concise explanation of the concept, complemented by visualisations.
  - **Interactivity –** Each concept has a visualisation that can be interacted with, often in a game-like manner. One example is the graph colouring section. It displays a graph and asks the user to colour the graph such that no two colours are adjacent. It starts with an easy example and provides a more challenging one afterwards.
- **Usability**
  - **Simple Design –** The website uses space effectively to avoid a cluttered look. It has the graphs on the left-hand side of the page, with the text, navigation and controls on the right-hand side.
- **General Notes**
  - The approach of providing multiple levels of difficulty works well to build understanding.

2.2 The Algorithms

2.2.1 Breadth-First Search

Breadth-First Search (BFS) is a simple graph traversal algorithm. Each vertex adjacent to the source is added to a queue to be 'visited', and then each visited vertex's neighbours are added to the queue to be visited. This repeats until the queue is empty. Using a queue means that each level in the graph will be fully explored before going deeper.

Input: Graph G, starting node s

Initialize an empty queue Q;
`Enqueue(`Q, s`)`;
while `¬Empty(`Q`)` **do**
  $u \leftarrow$ `Dequeue(`Q`)`;
  `Visit(`u`)`;
  foreach *neighbor* v *of* u *in* G *not in* Q *where* `¬isVisited(`v`)` **do**
    `Enqueue(`Q, v`)`;
  end
end

Algorithm 1: Breadth-First Search (BFS) algorithm pseudo-code

2.2.2 Depth-First Search

Depth-First Search (DFS) is another simple graph traversal algorithm similar to BFS. The difference is that instead of a First-In-First-Out queue, DFS uses a First-In-Last-Out stack. Using a stack means that deep branches will be explored fully first.

Input: Graph G, starting node s

Initialize an empty stack T;
`Enqueue(`T, s`)`;
while `¬Empty(`T`)` **do**
  $u \leftarrow$ `Pop(`t`)`;
  `Visit(`u`)`;
  foreach *neighbor* v *of* u *in* G *not in* T *where* `¬isVisited(`v`)` **do**
    `Push(`T, v`)`;
  end
end

Algorithm 2: Depth-First Search (DFS) algorithm pseudo-code

2.2.3 Dijkstra's Shortest Path Algorithm

Dijkstra's Shortest Path Algorithm (DSPA) finds the shortest path from a source vertex to all other vertices in a weighted graph.

```
Input: Graph G, source node s
foreach vertex v in G.Vertices do
    dist[v] ← ∞;
    prev[v] ← null;
    Enqueue(Q, v);
end
dist[s] ← 0
while ¬Empty(Q) do
    u ← v ∈ Q | v = MinDist(dist);
    Dequeue(Q, u);
    foreach v ∈ Q | v = Neighbour(u) do
        alt ← dist[u] + Graph.Edges(u, v);
        if alt < dist[v] then
            dist[v] ← alt;
            prev[v] ← u
    end
end
```

Algorithm 3: Dijkstra's Shortest Path Algorithm (DSPA) pseudo-code from Wikipedia contributors (2024)

3 Requirements

3.1 MoSCoW Requirements

To solve the problem of developing a web app, requirements were drafted of the WGAG using the MoSCoW method, developed by Clegg and Barker (1994), to capture the main strokes of what would constitute a successful product.

- **Must Have**
  - A visualised tutorial for the BFS, DFS and Dijkstra's graph algorithms
  - Users can carry out algorithm-specific actions themselves - e.g. in BFS, managing the queue and visiting vertices
  - Users can attempt to solve the chosen algorithm with guidance from the computer
  - Graphs used must be well-formed graphs that would be seen in examples elsewhere - e.g. Wikipedia, online tutorials, or university lectures.
  - Pseudo-code displayed for the algorithms, with the current step highlighted
- **Should Have**
  - Background/General information on the algorithms to avoid the user having to look to other sources to understand the basics of the algorithms.
  - An intuitive user interface.
- **Could Have**
  - Additional algorithms, such as Tarjan's Strongly Connected Components Algorithm or Hopcroft-Karp.
- **Would Be Nice To Have**
  - Greater customisation from the user, such as randomised graphs or user-defined graphs.

3.2 Issue Tracking

From these requirements, the project management and issue tracking tool ClickUp was used to create actionable, refined issues. A Kanban-style workflow was created where a backlog of issues was defined. As the project progressed, issues were removed from the backlog, marked as 'In Progress' while being worked on, 'Review' once the changes were merged into develop, and finally 'Done' once the changes were deployed. See A.3 for the Kanban board.

4 Design

To create a design for the WGAG, simple ideas were drafted on paper to get a first idea of what would be suitable. This chapter often references the concept of a 'step' in an algorithm. A step means the action/change caused by evaluating a line in algorithm pseudo-code. For example, a step in Dijkstra's Shortest Path Algorithm is **"Set the neighbour's previous vertex to the current vertex"**.

The website was designed to have two pages dedicated to each of the three algorithms. One page is for a visualisation that the user can step through, and another for the DIY section, where the user can manually execute the algorithm by making choices on what operations to perform on a vertex.

On each algorithm's page, there are three columns. The centre column contains a graph (4.3) and, if necessary, the DIY controls (4.5). The left column contains the pseudo-code (4.6), the controls to start the algorithm and choose which vertex/graph to work on, the legend (4.6), and helpful hints (5.3.5) for the DIY sections. The right column contains auxiliary information on the algorithms not suitable to be shown on the graph, such as a queue, the current vertex or the shortest paths.

Many figures in this section were created before the implementation began and, therefore, contain features that were not implemented and omit features that are now present.

To keep the algorithms simple, they do not function correctly on disconnected graphs. This is because the implementations of these algorithms that work on disconnected graphs have additional steps that were not considered essential in understanding the algorithms and would present a more significant learning curve.

4.1 Website Layout

The website layout was chosen to be very minimal in complexity. It has a homepage and a game page. The homepage contains basic information on the content of the website, an interactive explanation of pseudo-code, explanations of the algorithms, and a link to the game page.

The game page contains a nav-bar at the top with options to click on the six sections: Breadth-First Search, Depth-First Search and Dijkstra's Algorithm - each with a visualisation and a 'Do It Yourself' (DIY) section.

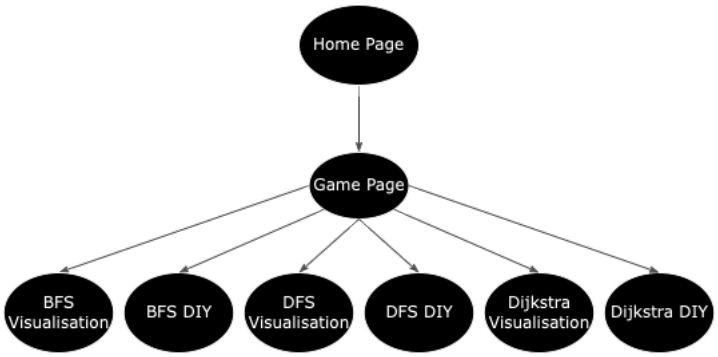


Figure 4.1: *The sitemap of the website. The game page is the central access point for each algorithm.*

4.1.1 Home Page

The home page is the first thing the user sees. This page, however, should not be a barrier to the user interacting with the website's main content. To achieve this, the content on the home page is hidden inside dropdown boxes or popup dialogs that provide information on the algorithms, what the website content is about, and how to interpret the pseudo-code. The largest, most prominent button on the screen is the link to the game page, intended to encourage the user to get into the action immediately, as the intended demographic includes computing scientists who may already have a basic understanding of the home page content.

4.1.2 Game Page

The game page contains only a nav-bar that allows navigation to each algorithm.

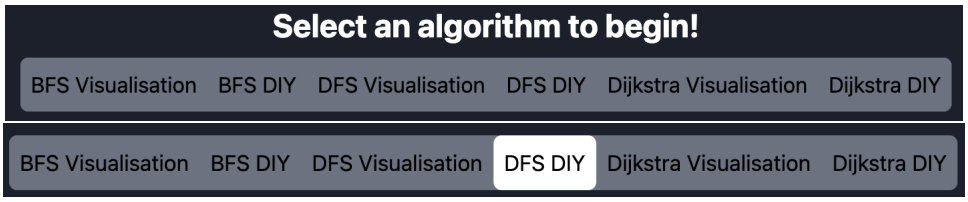


Figure 4.2: *The nav-bar on the game page, and also the subsequent algorithm pages. The top image shows the default state where no algorithm is chosen, and the bottom image shows the nav-bar once an algorithm is chosen.*

4.1.3 Algorithm Pages

This is where the main content of the website is centred. Each page has the same nav-bar as the game page, allowing for quick and easy navigation between the pages. On each page, the visualisation or DIY content is displayed.

4.2 User Interface

The user interface was made to be simple and to avoid clutter. Inspiration was taken from Ormond (2023) and Halim (2011) for the look of the game page. In the game pages, the graph takes up the centre of the screen, and pseudo-code and other relevant information occupy the

sides, forming a three-column structure. Namoun (2018) shows that a three-column structure can improve user experience and readability.

4.2.1 Constraints

Due to the time constraints on the project, a mobile-friendly version was not implemented, nor was the website tested on screen sizes smaller than the 13-inch laptop screen used for development. The evaluation (6.1.2) showed it does work well on large-screened tablets.

4.3 The Graph Data Structure

To visualise the algorithms, the graphs on which these algorithms must be performed were to be visualised. Circles were used for vertices, connected by thin lines to represent the edges between vertices.

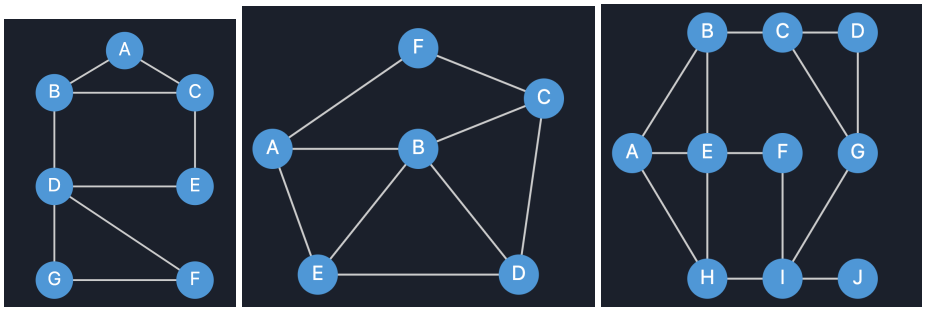


Figure 4.3: *The three unweighted graphs from the WGAG, represented as labelled circles and lines*

This style of graph is externally consistent with how graphs are represented in other products, such as Visualgo from Halim (2011), so should be instantly recognisable to users who have encountered the graph data structure before.

4.4 Visualisation

The WGAG visualises three different algorithms and allows the user to step through different graphs. In addition to the graph itself, information about the algorithm must be presented to the user.

The wireframe below is a minimal expression of the necessary features on the visualisation page for Breadth-First Search. It shows the three-column structure present in the current iteration of the website. The media controls are shown in a 'pause, play and rewind' format, however the website uses a 'next' button only, since there is no need to pause due to the user stepping through manually, and there is no back button (7.1.5). During implementation, it became clear that more information was needed to allow the user to follow the algorithms, so a legend (see 4.6), the current vertex, the source/start vertex, and the shortest paths in DSPA were all added.

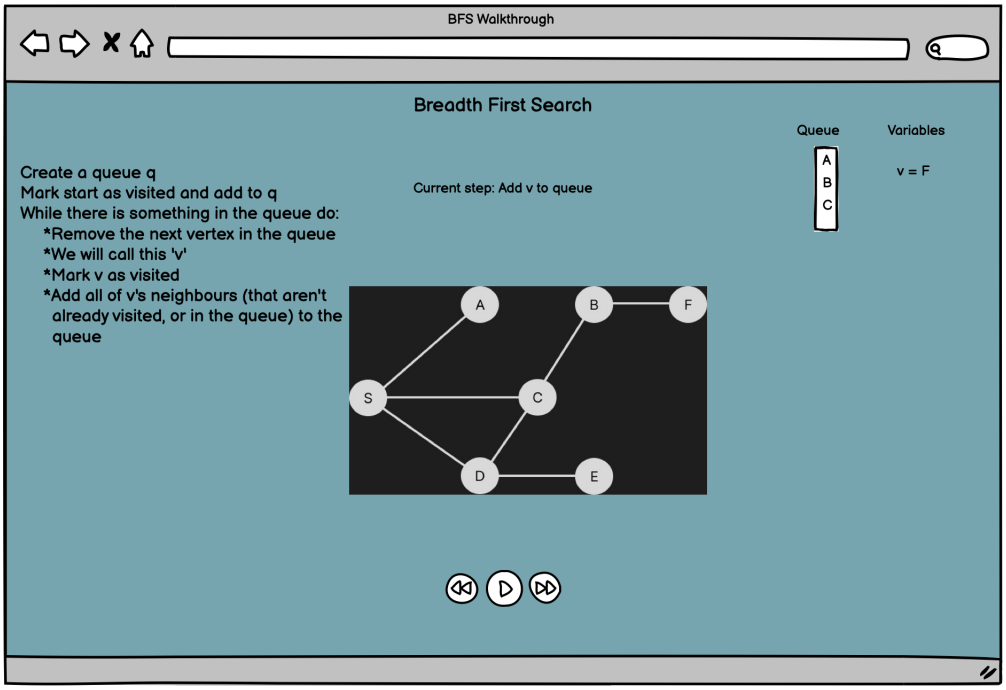


Figure 4.4: *Wireframe of BFS Visualisation (originally called walkthrough)*

To represent a weighted graph, as is used for DSPA, the edge weights and distances from the source vertex would be shown on the graph. The edge weights are displayed next to the edges, and the distances above the vertex circle. In the final iteration, the distances are shown inside the circles. See 5.3.1 on how the edge weights are drawn to avoid intersecting with geometry.

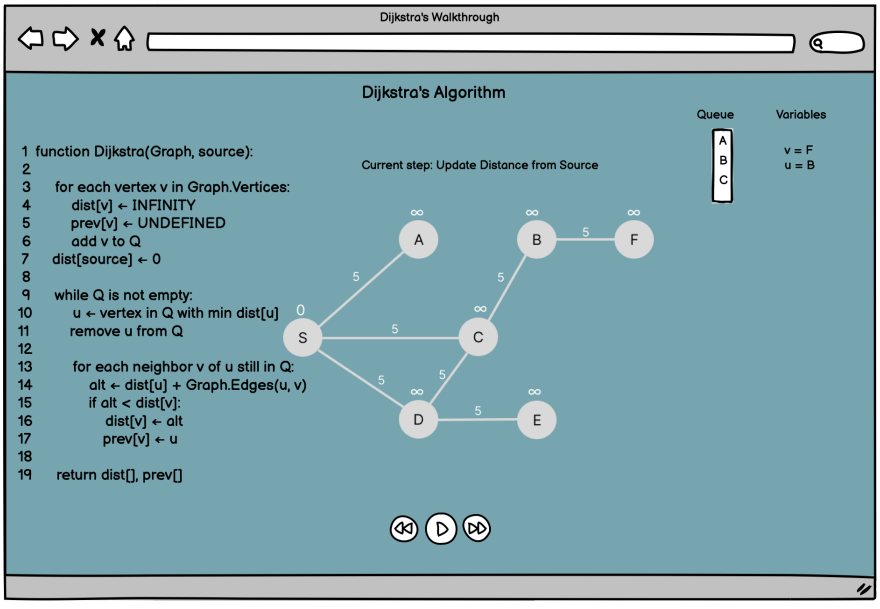


Figure 4.5: *Wireframe of Dijkstra Visualisation (originally called walkthrough)*

4.4.1 Updating the Graph

To provide information to the user, the graph itself must change to reflect updates in the algorithm. Changing colours and shapes can succinctly represent different states, such as if a vertex has been visited or is the current vertex. A legend shows what each colour and shape in the graph represents for each algorithm. Each item in the legend has a description where the meaning can be deduced from looking at the pseudo-code.

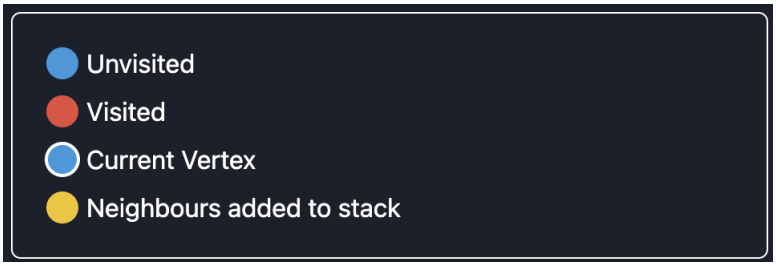


Figure 4.6: *The legend for Depth-First Search indicates the meaning of the circle colour/shape in the graph shown in the centre of the page.*

Breadth-First and Depth-First Search Updates

- A blue circle represents the default state of a vertex where it has not been visited.
- A red circle represents a vertex that has been visited.
- A circle with a white stroke border represents the current vertex
- A yellow circle represents a vertex recently added to the queue/stack (queue for BFS, stack for DFS) to be visited.

Dijkstra's Shortest Path Algorithm Updates

- A blue circle represents the default state of a vertex.
- Inside each vertex is either a number or the ∞ symbol. This represents its distance to the source vertex
- A circle with a white stroke border represents the current vertex
- A green circle represents a vertex undergoing the relaxation step.
- A pink circle represents a vertex part of the path of vertices from current to source.

4.4.2 Right-hand Side Panel

The right-hand side of the screen was dedicated to representing information that doesn't belong on the graph itself. The queues and the stack were internally represented as lists but shown as a vertical geometric representation of the data structure, where the next element in the queue to be removed is shown at the bottom, or the next element in the stack to be popped is shown at the top. Additionally, these elements are tagged with text to make it more explicit. The rationale behind making this as user-friendly as possible is that the task of deciphering the auxiliary information should not be more difficult than necessary so as to not distract from the user's learning and engagement. At a quick glance, it should be obvious which vertex is the next in line to be handled. In DSPA, the priority queue is sorted by shortest distance to always show the closest vertex ready to be dequeued.

4.5 Do It Yourself

The design for how the user controls the DIY section was refined from different approaches.

The first approach would have the user perform shortcut-style actions that make sophisticated use of the mouse, involving double-clicks and click-and-drag interactions. It heavily emphasizes interacting directly with the graph and a visualisation of the queue instead of relying on buttons. This was designed to reduce clutter and save screen space by removing the need for separate buttons.

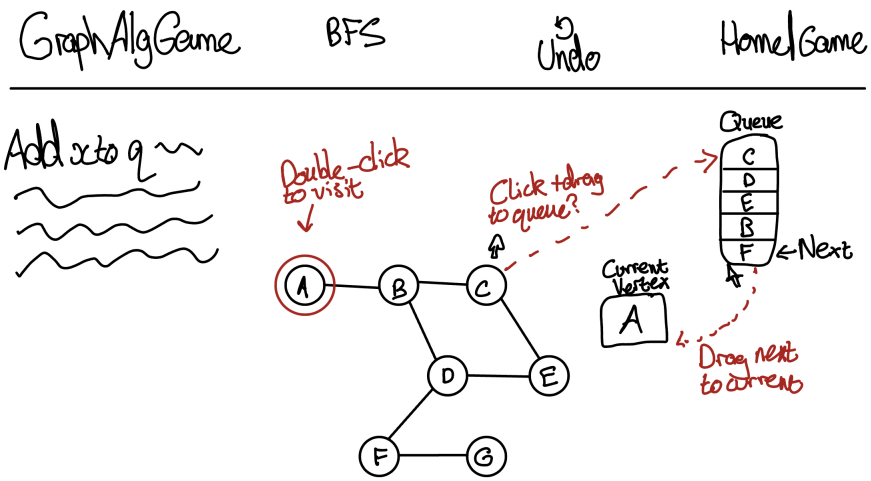


Figure 4.7: *Hand-drawn wireframe indicating potential buttonless user interactions with the BFS algorithm. Red ink text reads as "Double-click to visit", "Click + drag to queue?", "Drag next to current?".*

Once implementation began, the second approach (shown below 4.8) was chosen, as the first design (4.7) was deemed to have too great a learning curve. The user performs actions on a per-vertex basis. The user clicks a vertex to reveal the options available and then makes a choice. In the final product, instead of the options being hidden until a vertex is selected and popping up next to it, the options are shown in a box below the graph. This way, the graph is never obscured, and the user is always aware of their options.

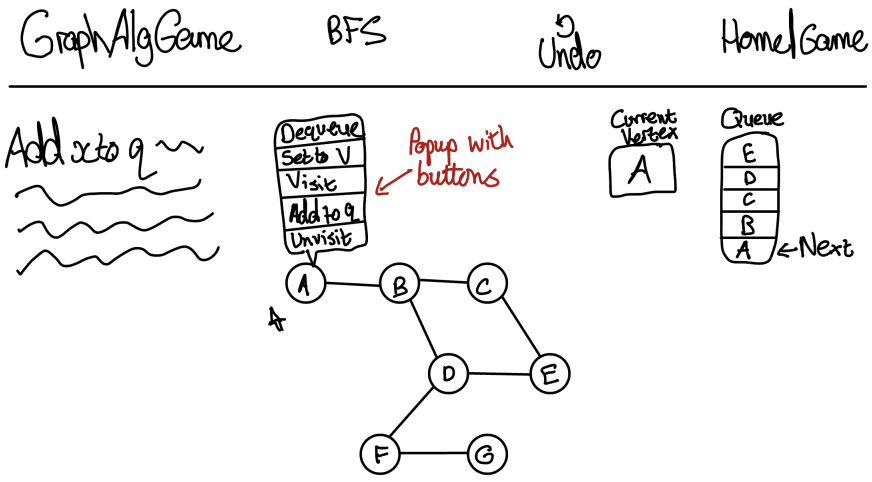


Figure 4.8: *Hand-drawn wireframe indicating user interactions based on buttons with the BFS algorithm. The options inside popup read as "Dequeue, Set to v, Visit, Add to q, Unvisit". The red ink text reads as "Popup with buttons"*

4.6 Pseudo-code

Pseudo-code for each algorithm is shown in the top left, and is vital to the user's understanding. It is expressed in natural language instead of a prescriptive mathematical style. This is to allow a more natural and intuitive thought process to form around the algorithm, where the user does not need to create a mental map of notation to what they see in the visualisation - e.g. the Wikipedia contributors (2024) pseudo-code for Dijkstra's Shortest Path Algorithm as can be seen in 4.5 mentions `prev[v]` whereas the WGAG pseudo-code refers to this as the previous vertex. This is also more accessible to those not as comfortable with mathematical/programming notation as was found in initial user studies at the beginning of development.

The pseudo-code is highlighted on the appropriate line corresponding to the current step with a clear contrast in colour. This ensures external consistency and takes inspiration from Ormond (2023) and Lau (2021)

4.7 DIY Hints

As can be seen in 4.4 and 4.5, there is an indicator of the current step above the graph. This idea was refined for use in the DIY sections to become an accurate hint of what the user should do to make progress.

4.8 DIY Difficulty Level

To give a wider range of difficulty, there were originally different stages with different amounts of help, known as 'Guided' and 'DIY', where the guided page showed hints and highlighted the pseudo-code and the DIY page had just the pseudo-code visible. These concepts were combined into one configurable page for simplicity. Users can optionally hide the hints, the highlighting of pseudo-code, and the pseudo-code entirely to give as much or as little challenge as desired.

4.9 Explanations

To introduce users who are not already familiar with many of the concepts shown on the website, the homepage contains explanations for some concepts, including one for each algorithm. The explanations use metaphor, comparing vertices and edges to cities and roads or the queue data structure to a human waiting line.

4.9.1 Pseudo-code Tutorial

The home page has two examples to introduce the user to the WGAG's representation of algorithms in pseudo-code: one showing an if-condition, and the other a conditional loop. The if-condition has a numeric input box with a submit button and shows confetti if the number entered is less than ten; otherwise, it shows an error. The conditional loop evaluates $x \leftarrow x + 1$ until $x = 10$.

4.10 Variety

To add variety to the website, rule-based randomly generated graphs were considered, meaning users could practice on an infinite number of graphs. This was found to be infeasible to implement, due to time constraints and feature prioritisation. Instead, users can choose from three different graphs and start from any vertex in a graph. The three graphs that were made were designed to show different behaviours in the algorithms. A wide graph with multiple depth levels was made to show the difference between BFS and DFS in their priorities. A graph with interesting edge-weights was made to show the relaxation step of DSPA, where the lowest-cost path is not always the path with the fewest vertices.

5 Implementation

5.1 Software Development

This project was created paying attention to software engineering best practices. A stack of VueJS, TypeScript, Tailwind CSS and Headless UI was used to write the WGAG code.

5.1.1 VueJS

It is common to use a JavaScript framework to develop a website. VueJS 3 was chosen for this project. It improves greatly upon vanilla JavaScript by introducing the idea of components. Components encapsulate user interface elements and logic into reusable building blocks. This allows for significant savings of development time, code duplication, and coupling and more maintainable and readable code.

The syntax of VueJS is declarative, with a `<template>` and `<script>` tag in each component, declaring the HTML to render to the screen and the logic of the component, respectively.

```
<script setup lang="ts">
   defineProps<{
      name: string;
   }>();
</script>
<template>
   <h1 class="font-bold my-2">
      Hello, {{ name }}!
   </h1>
</template>
```

Listing 5.1: *A simple 'Hello, person' VueJS component, which takes in, as a prop, the name to say hello to.*

VueJS uses double curly braces to insert TypeScript into the template. The VueJS function `defineProps` defines the parameters being passed into the component, and if used with TypeScript as it is here, defines the type of each prop (see 5.1.2). The class attribute contains Tailwind CSS (see 5.1.3).

5.1.2 TypeScript

VueJS uses JavaScript by default but easily integrates with TypeScript. TypeScript is a super-set of JavaScript which introduces syntax for types, and allows for the creation of custom types, which were helpful during development to understand the nature of data and variables. TypeScript is strongly typed, which maintains strict rules to reduce the introduction of bugs. Bogner (2022) shows that, on average, repositories written in TypeScript have higher code quality than JavaScript, especially where the `any` generic type is not used. To align with this, it was ensured there were zero uses of the generic `any` type in the source code, instead using proper typing.

```
export type DFSYieldData = {
   visited: Set<number>;
   stack: Vertex[];
   step: DFSStep;
   currentVertex: Vertex | null;
};
```

Listing 5.2: *The custom type of the yield data from the Depth-First Search generator function*

An example of a custom type is the yield data type for the DFS generator function (see 5.3.2). This type was assigned to all the yield statements in the Depth-First Search generator function, ensuring consistency in the data. Nested inside is a mixture of generic and custom types, demonstrating the flexibility of TypeScript.

5.1.3 Tailwind CSS

Tailwind CSS was used to style the website. Tailwind provides abstract and succinct classes for inline styling. This avoids bulky CSS files and speeds up the feedback loop for rapid development. Additionally, the need to name CSS classes disappears entirely, and the possibility of breaking the styling on one part of the site when trying to fix another is eliminated. The downside of Tailwind is CSS code duplication. However, this is mitigated mainly through the reusability of VueJS components.

The VueJS example 5.1 contains an `<h1 />` tag with a class attribute containing Tailwind CSS styling. It serves as an example of the brevity and readability of Tailwind. The equivalent of `font-bold` in plain CSS is `font-weight: 700;`, and the equivalent for `my-2` is `margin-top: 0px; margin-bottom: 0px;`. Tailwind is more readable, more concise, and has exhaustive documentation.

5.1.4 Headless UI

Headless UI is a free component library made by the creators of Tailwind CSS. It provides useful UI components such as dropdowns, dialogs, and tabs. The dialog popup component is used for the algorithm explanations on the homepage and the dropdown component for the graph configurator.

5.1.5 CI/CD

The idea of continuous integration was developed by Fowler and Foemmel (2006), and was extended by Humble and Farley (2010) into continuous development, which, as seen at PaddyPower from Chen (2015), is beneficial to productivity, development speed and product quality.

To minimise time spent manually testing new releases and deploying new changes, a set of GitHub actions were created that would run automatically when specific actions were triggered. A testing pipeline was run for each push/merge to develop. On each merge to the master branch (deployment), the same testing pipeline is run, and if that passes, the website is automatically deployed to GitHub Pages. To avoid unnecessary processing, pipelines were not run unless the `src` source folder was included in the pushed changes.

5.1.6 Branching Strategy

The Git-Flow branching strategy was used to manage the different versions that emerged during development. This allowed for a working version to be constantly deployed while features were

developed on separate branches. Once features were merged, they would pass the develop branch tests, ensuring that the develop branch was always a working branch ready for deployment. Changes were never required to be rolled back; however, the architecture would have easily allowed for this.

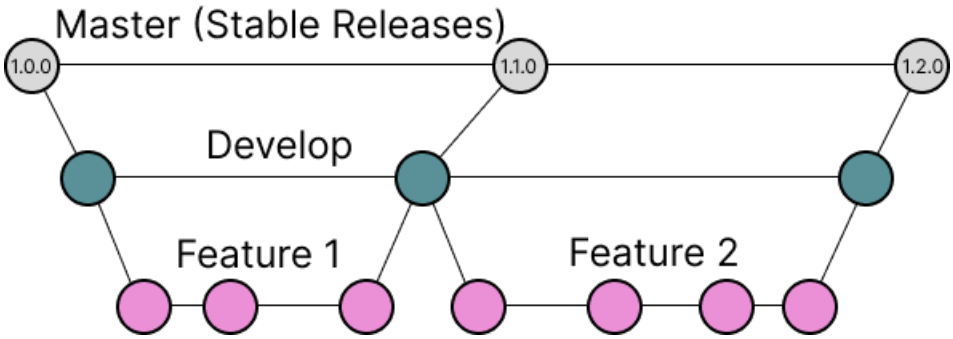


Figure 5.1: *Basic example of the Git-Flow branching strategy. Each circle represents a new state of the repository after a commit.*

5.1.7 Pre-commit Hook

To help ensure good code quality, a pre-commit hook was configured to run the ESLint TypeScript linter and Prettier code formatter to enforce good practices and a standard coding style. This would run on each commit locally, acting as a filter through which all code must pass before reaching the remote repository. It also saves a step in the GitHub Actions pipelines. The pre-commit hook, in combination with the CI/CD pipeline, ensures that code quality is scrutinised and working properly in the absence of other software developers who could perform code reviews.

5.1.8 Unit Testing

As mentioned in 5.1.5, each push to the development or master branch triggers a test pipeline. This testing pipeline was made up solely of unit tests using the Vitest and Vue Test Utils frameworks. In each test, a component is mounted, manipulated, then probed to assert the state is as expected. For the different algorithms, there are tests that assert the state is as expected at various stages in the algorithm to ensure correctness. This, however, is not sufficient to ensure a lack of bugs, so on top of this, manual testing was carried out with each pull request.

As stated in 1.1, the goal was to use a test-driven development workflow. However, this was not done, and all tests were written 'after the fact'. The tests cover the core functionality of algorithm correctness by asserting a certain configuration of variables is present at a certain point in an algorithms execution. The correct rendering of content in the hint boxes and vertex option menu is also asserted.

5.2 Component Hierarchy

The six content pages, as seen in 4.1, were created following the same architecture. These components can be reused by virtue of being parametric, where parameters or 'props' are passed in. For example, the legend component `<SearchLegend />` is used by both DFS and BFS and takes in, as a prop, either the string 'stack' or 'queue' to be displayed. Each algorithm page follows the component structure shown below. To communicate between components, events

are emitted upwards to the parent page component, which updates a reactive variable which is the prop for another child component. See listing 5.7 and figure 5.6 for more details.

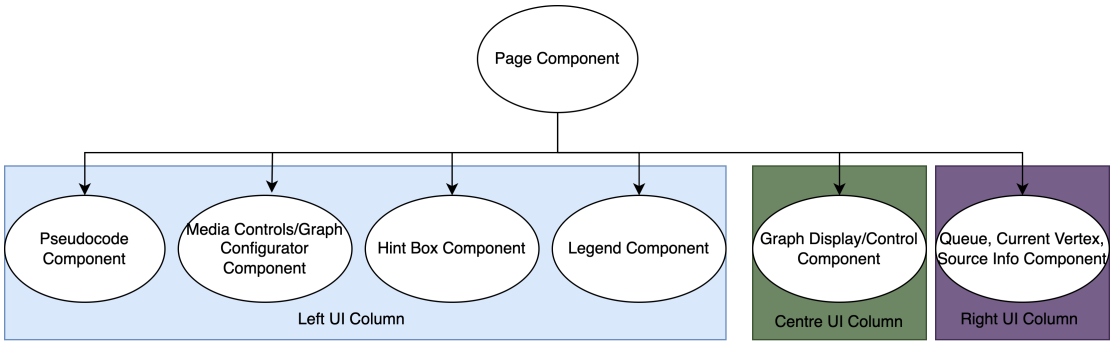


Figure 5.2: *The common hierarchy of VueJS components for the pages. The diagram is not exhaustive; each component may have many child components. The left blue box includes all the content on the UI's left-hand column. The centre green box contains the graph, and for DIY sections, the controls. The right purple box contains the 'side panel' information that shows additional information on the algorithm's state. The hint box component is only present in the DIY sections.*

5.3 Visualisation and DIY

5.3.1 SVG Graphs

HTML's built-in SVG capability was used alongside VueJS components to create a reusable and duplication-free implementation for drawing the graphs on the screen. A file with JavaScript objects containing the information for each node in a graph (x and y coordinates, text, edges) was fed into a VueJS component, which plugs the information into `<Circle>` and `<Line>` SVG elements, with the corresponding weights, colours, etc, all rendered. This also made the creation of graphs trivial, as the workflow only involved adding another entry to the JSON file.

```
{
   1: {
      A: {id: "A", x: 175, y: 30},
      B: {id: "B", x: 100, y: 75},
      C: {id: "C", x: 250, y: 75}
   }
}
```

Listing 5.3: *A simplified example of vertex data from "nodeDatas" that describes a graph with three vertices, with x and y coordinates for use in the SVG*

This `nodeDatas` example is defined in `graph-data.ts` alongside the description of the edges: `linkDatas`

```
{
   1: {
         A_B: {
         v1: "A",
         v2: "B",
```

```
            x1: nodeDatas[1]["A"].x,
            y1: nodeDatas[1]["A"].y,
            x2: nodeDatas[1]["B"].x,
            y2: nodeDatas[1]["B"].y,
            stroke: "#ccc",
            strokeWidth: "2",
            weight: 1,
        },
        A_C: {
            v1: "A",
            v2: "C",
            x1: nodeDatas[1]["A"].x,
            y1: nodeDatas[1]["A"].y,
            x2: nodeDatas[1]["C"].x,
            y2: nodeDatas[1]["C"].y,
            stroke: "#ccc",
            strokeWidth: "2",
            weight: 2,
        },
    }
}
```

Listing 5.4: *A simplified example of edge data from "linkDatas". There are two edges: 'A_B' and 'A_C'*

Here, to create an edge between vertices, an object must be defined with fields describing the vertices, the coordinates of the vertices, styling, and edge weight. The edge weight is defined for all graphs, since they are used both as unweighted graphs for BFS and DFS, and as weighted graphs for the DSPA sections. It uses a reference to `nodeDatas` to get the vertex coordinates, meaning that if the vertex is repositioned, this is also automatically reflected in the edges.

Weighted Graphs The graphs in the Dijkstra's Shortest Path sections include edge-weights and distances from the source vertex. The distances from the source vertex were drawn inside the circle, below the character label of the vertex. These positions could be hard-coded to the coordinates of the circle's centre, with a vertical offset of $\frac{radius}{1.5}$.

The edge-weights are shown next to the edges, at the midpoint between their vertices. An offset of y was either added or subtracted from the midpoint coordinates to avoid the drawn line intersecting with the number. To decide this, the angle, in degrees, of the line from the positive x axis was calculated using the formula

$$\theta = arctan(\Delta y, \Delta x)\frac{180}{\pi} \tag{5.1}$$

If this angle θ ranged from $-90°$ to $90°$, then the position of the edge-weight was lowered, else it was raised.

Node and Link Components The visualisation can be updated dynamically via props. The `Node` component inherits the following props from its parent:

```
const props = defineProps<{
    cx: number;
    cy: number;
    r: number;
    fill: string;
    text?: string;
    currentVertex?: boolean;
```

```
    uncheckedDistance?: number;
}>();
```

Listing 5.5: *Props in the Node component. We have the x and y coordinates, the radius r, the colour, the label text, a boolean if it's the current vertex, and the distance from the source.*

These props are used when drawing the SVG and are updated reactively by the parent components that handle the algorithm's progression. Notice also the `?` question marks next to some of the props, which indicate the prop is optional, allowing for the Node to contain edge-weight text or have nothing shown if it isn't necessary. This lets the same component be used in both weighted and unweighted graphs. The text label is also optional as the `<Circle />` component is reused, without any label, in the legend 4.6.

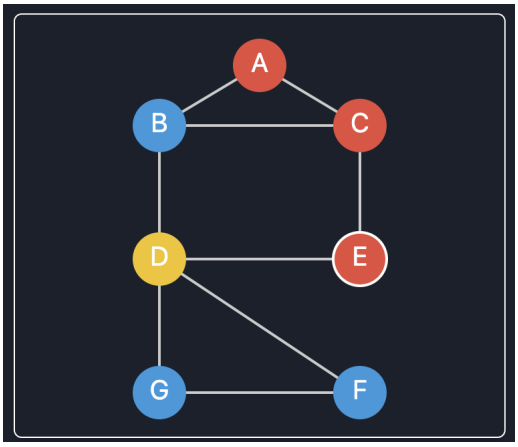


Figure 5.3: *The first of three graph options, midway through the DFS algorithm. Red indicates a visited vertex, blue is unvisited, and yellow is a vertex to be added to the stack. E has a white circle around it, indicating it is the current vertex.*

This graph shows the different states available in the DFS algorithm. The start and end coordinates of the link components are the centre points of the vertices they connect. The node components must be drawn after the link components to ensure they appear on top.

5.3.2 Visualisation with Generator Functions

To implement the stepping-through of algorithms in the visualisation sections, JavaScript's generator function capabilities were used. A generator function yields a value in a sequence until it reaches the last yield statement. Each click of the 'next' button performs the current step in the algorithm, progressing it by one step until it reaches the end.

```
*dfsGenerator(startVertex: Vertex): Generator<DFSYieldData, void, unknown> {
    // ...
    while (this.stack.length !== 0) {
        // ...
        yield {
            visited: this.visited,
            stack: this.stack,
            step: "removeFirstAndMakeItCurrent",
            currentVertex,
```

```
        };
        // mark v as visited
        currentVertex.setVisited(true);
        this.visited.add(currentVertex.getIndex());
        this.changeVertexColour(
           numToLetter[currentVertex.getIndex() + 1],
           "#e74c3c",
        );
        yield {
           visited: this.visited,
           stack: this.stack,
           step: "markVAsVisited",
           currentVertex,
        };
        // ...
     }
     // ...
}
```

Listing 5.6: *An abridged snippet from the DFS generator function in <VisDFSGraphDisplay.vue /> that handles the visualisation progression.*

Each yield statement returns out of the function until the `next()` function is called when the user presses the 'next' button. In between each yield statement is the logic for a step. This example (listing 5.6) shows how a vertex becomes visited, where an array is updated, and the colour of the vertex is changed to red. This component `<VisDFSGraphDisplay.vue />` handles the display of the graph, and the visiting step does not have side effects outside of this component. For other steps, such as adding a vertex to the queue, these changes must update the display of different components. To achieve this, VueJS emits events to a parent component, from where the displays of other components can be updated.

```
const performDFSStep = () => {
   if (dfsGenerator.value) {
      const result = dfsGenerator.value.next();
      if (result.done) {
         reset();
      } else {
         emit("update:pseudoStep", result.value.step);
         emit(
            "update:currentStack",
            result.value.stack.map((v) => {
               return numToLetter[v.getIndex() + 1];
            }),
         );
         graph.currentVertex.value = result.value.currentVertex;
      }
   }
};
```

Listing 5.7: *The function called on each click by the user to execute a step in the algorithm.*

In this function, `emit()` is called, which updates triggers an event in the parent component. VueJS provides support for reactive variables that, once changed, their dependent components will be re-rendered with the new information.

5.3.3 DIY Implementation

Generator functions were not suitable for the non-deterministic nature of the algorithms, where there are multiple equal options that the user can choose from. If the user has the option to add both vertices B and C to the queue in Breadth-First-Search, there is no guarantee of their behaviour, and there was no need to enforce an order of precedence. To account for this unpredictability meant that a generator function was not appropriate for the DIY section, and instead, standard functions, such as `graph.visit(nodeId: string)`, were defined for each operation the user could perform. These functions mirrored the behaviour of each chunk of the generator function in between yields but were in separate, callable functions. Each option the user could choose would be validated, and if correct, all necessary variables and the graph display would be updated. If the wrong choice was clicked, it flashed red to indicate an incorrect decision.

To control the algorithm, the user clicks a vertex and can then choose to perform operations on that vertex. For example, the user clicks a vertex, and is presented with three options to perform on a vertex in the Breadth-First Search DIY section: "Add to queue", "Mark as visited", "Remove from queue and set to current vertex". Of these three, only one will be correct at a time for a given vertex. When one of these options is clicked, the corresponding validation function checks the state of the algorithm - the current step, the contents of the queue, and if the currently selected vertex is the correct one to perform this step on - and if these all pass validation, the user has chosen the correct option and the algorithm progresses one step.

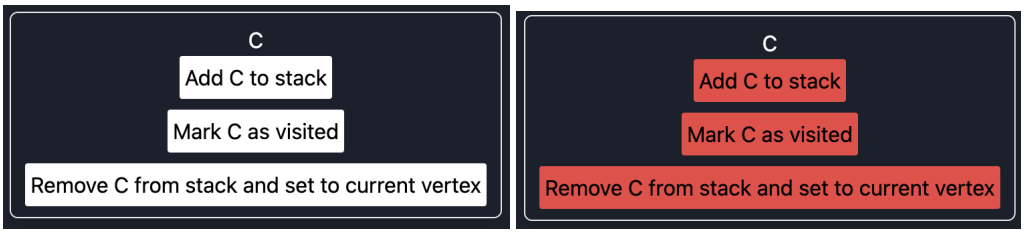


Figure 5.4: *Depth-First Search DIY controls where a user has clicked on vertex 'C'. The left shows the configuration at rest. Right shows the temporary red flash triggered by clicking an incorrect option.*

5.3.4 Pseudo-code

The pseudo-code is shown in the top left, with the current step highlighted. As the algorithm progresses, the current step is updated, and the changes are emitted up the component hierarchy until the pseudo-code component is updated.

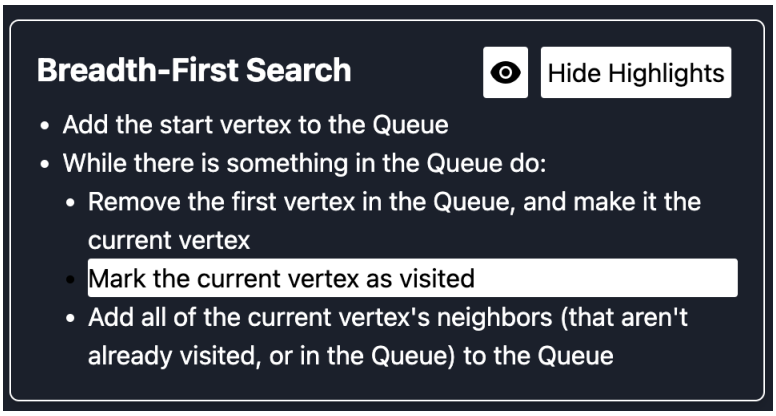


Figure 5.5: *BFS pseudo-code, with the current step highlighted.*

For the pseudo-code component to be aware of what the current step was, the components controlling the progression of the algorithm would emit the updates to the parent, where the new value would be assigned to a reactive variable, which is passed into the pseudo-code component, meaning the pseudo-code is dependent on the reactive variable, and therefore is re-rendered.

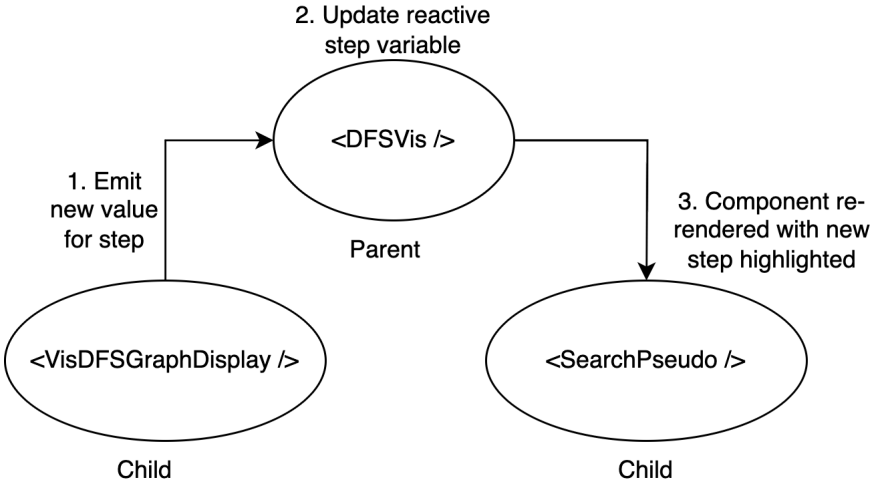


Figure 5.6: *Component diagram showing the flow of updating a variable and communicating this update to a sibling component. Step 1 shows an emission (for example, the next step button is clicked). Step 2 shows the parent receiving this emission and updates the reactive variable. Step 3 shows the sibling component receiving the update from the parent and re-rendering.*

The structure of having a child speak to its parent, which speaks to another child, is the common structure for each of the algorithms and is a powerful way to have communication between components.

5.3.5 Player Hints

Similarly to the pseudo-code, hints would be displayed based on the algorithm's state and the current step. If there were only one correct action, it would be shown precisely, e.g. 'Remove A from the stack'. A broader hint would be shown if there were multiple options, e.g. 'Update the distance of an adjacent vertex'.

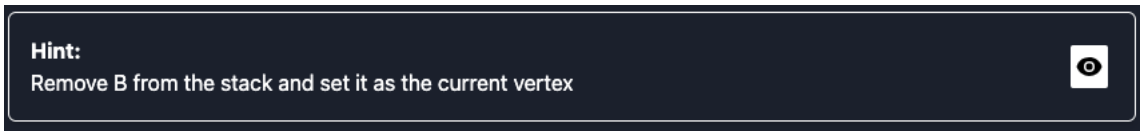


Figure 5.7: *The hint box explaining the current step is to remove the vertex B and set it as the current vertex in the DFS DIY algorithm.*

5.3.6 Hiding the Assistance

Users have access to the pseudo-code, the highlighted current step of the pseudo-code, and a hint showing precise information on the exact action to take. Users can hide any number of these functions to add a layer of challenge. Blurring the pseudo-code or hints is done by having button-activated boolean flags which decide if the Tailwind CSS blur class is applied to the element. To hide the highlighting of the step, a white-background CSS class is conditionally applied to each line of pseudo-code using a button-activated boolean flag as well.

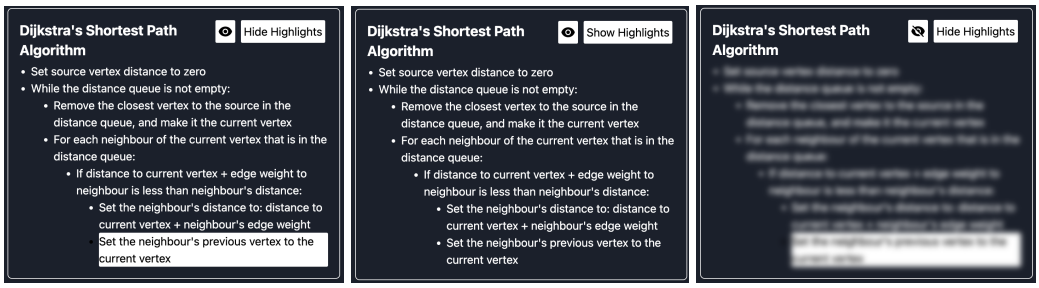


Figure 5.8: *Pseudo-code for DSPA DIY with some progression made on the algorithm. The Left shows the default configuration, the middle shows the pseudo-code with the 'Hide Highlights' option clicked, and the right shows the pseudo-code with the blur option (denoted by the eye) clicked.*

5.4 Extensibility

As development began, attention was given to code reuse, where components were generic. BFS and DFS share many of the same components, however when adding DSPA, it was found that some of the components were not suitable to refactor to allow for DSPA to be seamlessly integrated.

5.4.1 Dijkstra's Shortest Path Algorithm

As Dijkstra's Shortest Path Algorithm (DSPA) is very different to BFS and DFS, it was not possible to reuse as many of the components to implement the features necessary for the algorithm. To make the code more extensible, it would have been prudent to use a generic way to manipulate graphs that is parametric and able to be manipulated by logic uncoupled from it. As this approach was not taken, code duplication was more common, and UI changes took longer to implement.

This being said, code duplication was reduced in many places thanks to VueJS and the component architecture, such as the rendering of the SVG vertices and edges. See 7.1 for more details.

6 Evaluation

6.1 Unsupervised Survey

A Google Forms survey was created to gather anonymous qualitative data on user's perceptions of the WGAG. The survey was sent to family members, friends, and public forums and discord servers to get responses. The number of responses was zero initially, which prompted me to run supervised evaluations (6.2) as another evaluation method. After these supervised evaluations were complete, however, another push to get responses with a more **polished product** yielded a small number of responses.

To gain demographic information, respondents were asked about their level of experience in programming, their familiarity with pseudo-code, and their familiarity with graph algorithms.

The survey then asked the respondent to:

> Visit the game page, and try running through the visualisations for each algorithm. Pay attention to the state of the graph and the algorithm whilst doing so. Try starting on different vertices, and on different graphs to get a feel for how the algorithm behaves in different scenarios.

After this, they would answer some questions about their understanding and experience with the visualisations, including how capable they would be of explaining graph algorithms to someone from memory.

Respondents were then asked to perform the same tasks but for the DIY sections and asked similar questions, including again how capable they would be to explain graph algorithms from memory. The aim of this was to see how using the DIY section affects one's understanding of graph algorithms.

The questions covered the respondents' ability to explain graph algorithms from memory, how easy the visualisations/DIY sections were to understand, how easy the controls were to use, how fun and engaging the website was, if they would use the website, their general thoughts, and finally any technical issues encountered. The survey can be accessed from the appendices: A.1.

6.1.1 Limitations

There is inevitably bias in asking respondents about their confidence in explaining graph algorithms at both stages, as after the DIY section, they will have been exposed to the concepts for longer. This, however, is still a relevant question, as if the respondents' confidence increases, then the website has caused that effect. To get an accurate representation of which mode is more effective in isolation, it would be better to split respondents into trying only one of the modes. However, the WGAG contains both modes to complement each other, so this is unnecessary.

The question of how many years of computer programming experience a respondent has is flawed, as the computing experience gained in secondary school, university, and industry all yield completely different levels and areas of expertise. To ascertain the respondent's skill/knowledge level, it would have been more prudent to ask for their highest level of education in computing science as well.

Additionally, the sample size is too small to draw concrete conclusions and observe trends.

6.1.2 Results

Five respondents answered the survey. Of these five, three had one or fewer years of experience with computer programming. Three had encountered pseudo-code before, and only two had heard of Breadth-First Search. When asked how many were familiar with graph algorithms on a five-point scale of 'completely unfamiliar' to 'expert', nobody ranked themselves greater than the second increment. Considering this, all participants were unfamiliar with graph algorithms and can be regarded as beginners.

Each question in 6.1.2 Task 1 and 6.1.2 Task 2, and all ranking questions in the 6.1.2 General Questions section uses a four-point scale, forcing either a positive or negative response.

Task 1 When asked how easy the visualisations were to understand, four of five responded positively, with the outlier giving the lowest rank, indicating they were highly confused.

When asked if they easily understood the visualisation media controls, three responded positively, and the other two responded negatively.

When asked if they could explain the graph algorithms to another person from memory after working through the visualisations, three answered negatively, and two positively. Nobody chose the most negative option.

Task 2 Similar questions were presented to respondents to see how their perceptions compared with the visualisation and DIY sections, and how their perceptions had changed regarding knowledge and confidence.

When asked how well they understood the state of the graph and algorithm during the DIY sections, all answered positively, with two selecting the most positive option.

All five chose the second most positive option when asked how easily they interacted with the DIY interface.

When asked again if they could explain the graph algorithms to another person from memory after using the DIY section, four answered positively, and only one responded negatively. Of the four that responded positively, there was an even split between the second most positive and most positive options.

General Questions When asked if they found the WGAG fun and engaging, all answered positively, with three choosing the most positive option.

When asked if they would use the website to visualise and better understand graph algorithms, all answered positively, with only one choosing the second most positive option.

Respondents were then asked their general thoughts on the WGAG, if they perceived the website as an effective learning tool, and if they had learned anything. The general consensus was positive.

All five participants noted that they found it to be an effective learning tool.

Two participants noted that they felt the visualisation built a foundational understanding, and the DIY section helped consolidate that knowledge:

- *The visualisations help build initial understanding, and the DIY section helped me cement it.*
- *Yes the Visualisation helps lead into the DIY sections.*

Positive Feedback Some quotes that show respondent's satisfaction with the WGAG are as follows

- *I think it's excellent. The explanations are all very clear and user-friendly and there is just the right amount of text. I definitely learned how the algorithms work.*
- *Yes, worked well on my iPad, if I was studying computer science again this would help me a lot due to my need to visualise things to help understand them or commit them to memory*
- *Definitely an effective tool. ... If there was anything I didn't understand I could practice with the guide still visible and once I felt I was secure I turned it off to test my knowledge.*
- *It was good to see 2 similar algorithms side-by-side, and by actually going through the steps in the DIY sections I could see the difference, and I got a real feel for the process. When I went wrong in the DIY sections the hints were really helpful and clear. By the time I got on to the third graph of each I knew exactly what I was doing and could remember it all. If I had needed more practice, I could have used different starting vertices, so I had lots of learning opportunities.*

Room For Improvement Some suggestions and negative feedback were given as well.

- *The DFS and BFS are understandable for me but not the Djykstra one*
- *I found the Dijkstra's algorithm much harder and the reference to Google Maps confused me a bit. During the visualisation I was unclear about what Dijkstra's algorithm was actually doing. A clearer 'shortest paths' box would help me better understand the algorithm.*
- *I think an extra section with more info about the application of the algorithms would be a good learning opportunity for me.*

6.2 Supervised Evaluations

In-person supervised evaluations were conducted where the participant was given a laptop with the website open and asked to simply explore and try it out. No instruction or help was given to ensure the website was interacted with naturally. During the evaluations, notes were taken of what each participant did, what they did well, and what they struggled with. At the end, they were asked two questions. The first was their general thoughts, which were intended to open a discussion about their experience. The second question was about their preference for either the visualisation or the DIY method.

Seven participants were primarily selected to come from a computing science background, as the target demographic is undergraduate computing science students. However, some participants were taken from other degree disciplines with little to no experience with algorithms and data structures. Faulkner (2003) ran user studies with different numbers of participants, and found that in usability testing, their evaluations with five participants found anywhere from 55% to 99% of issues, whereas their evaluations with ten participants, at the lowest, found 80% of issues. The combination of the seven supervised evaluation participants and the five survey respondents gives a large enough sample size that a significant portion of the issues with the project have likely been identified. See A.1 for the link to the GitHub Wiki where the notes are kept.

6.2.1 Limitations

All participants were undergraduate students at the University of Glasgow, and all the computing science students were in the same year of study. This means all computing science student participants have had the same university education in algorithms and data structures, which may limit some biases, but give a less broad range of opinions.

6.2.2 UI/UX Feedback/Results

As more supervised evaluations were conducted, more bug fixes and UI changes were implemented to address issues as they were identified. Many of the issues raised were fixed, so the later evaluations should be considered the most accurate for deciding on future work.

Difficulty Starting DIY Users found it difficult to start the DIY sections. The appropriate workflow is to click start, then click the start/source vertex and start choosing operations to perform. The main point of confusion was discovering that the vertices were intended to be clicked to select them. It often took more than ten seconds of trial and error before the participants realised they could click on vertices. Users would also click start, then try to use the drop-down menus that adjust the starting vertex and which graph to use. This was fixed by disabling those drop-down menus once the user clicked start.

Undo/Back Button Three of the seven participants expressed their desire for an undo or back button. They stated that they accidentally went too far and lost their train of thought and that an undo or back button would help them better understand the algorithms.

DSPA Pseudo-code Highlighting The pseudo-code for DSPA contains a while loop, a for loop, and an if-condition that are all evaluated without a dedicated step for each. So when users step through the visualisation or work through the DIY section, they can make progress, shown in the graph and the right-hand side panel, but the highlighted step does not move, even though the vertex in question will be different. This confused many of the participants and caused them to take time to try to understand what had happened.

Thoughts on the UI Participants generally responded well to the UI, and as the WGAG became more polished, there were fewer instances of confusion and more positive feedback. The introduction of the legend was well-received.

6.2.3 Effectiveness as a Tool Results

Learning Effectiveness Every participant in the supervised evaluation completed at least the BFS/DFS DIY section without relying on brute force but instead by using the knowledge they gained of the algorithm from studying the visualisation and working through the DIY section itself. Every participant, bar one, could complete the DSPA DIY section without help. This shows that the WGAG was effective in helping people learn the BFS and DFS algorithms. It was effective in helping those with some computing science experience learn DSPA but not as effective for those with no computing science experience.

Experience Level and Ease of Use Participants who had more experience with computing science and algorithms and data structures were able to understand and begin to solve the DIY sections much faster than those who had less experience.

Preference of Visualisation or DIY Of the seven participants, four found the visualisation the most useful, with three saying they found both equally useful - more specifically that the visualisation was a good jumping-off point, and the DIY sections helped to consolidate the understanding built with the visualisation. Two participants noted how the DIY after learning the patterns, became more of an exercise in muscle memory than consciously making decisions - especially with BFS and DFS. This muscle memory could potentially imply an intuition of the algorithms, especially when tried on different starting vertices and graphs. On the other hand, it could mean that the controls have a level of abstraction that is too high, where users can learn button patterns easily instead of learning how the algorithms work.

Not Beginner Friendly It was noted that there were a lot of jargon terms that go unexplained. One of the inexperienced participants struggled greatly with understanding what terms meant, which hurt the benefit they received from the WGAG - for example, they were unfamiliar with the term vertex, and there is nowhere on the website that defines the term.

Use of Hints Some participants obscured the hints and, after looking at the visualisation, tried to solve the DIY without computer assistance. They only looked at the hints when they lost track or became stuck. This is what the hints were intended for: to give a more significant challenge to those who choose it. The fact that some participants used the WGAG this way indicates that this feature was a success.

General Sentiment Participants generally liked the website, found it interesting and could see its use, especially as a supplement to self-directed learning. The visualisations were found most helpful, with the DIY section being a method to consolidate and test understanding. For non-computing scientists, it was found to be too concise and mathematical in its use of jargon but it was still able to teach intuition of the algorithms. Furthermore, most of the struggles participants encountered were due to a poorly implemented UI, which was refined as the evaluation progressed. As the UI was refined, participants encountered fewer obstacles in interacting with the website as intended and were able to learn more effectively.

6.2.4 Issues Raised That Were Remedied

Many of the issues raised during the supervised evaluations were fixed before the next evaluation as a means to polish the product iteratively. Examples of these issues are:

- Clumsy wording, such as using the word 'adjacent' as a noun, where 'neighbour' is more suitable, or the use of priority queue and distance queue in different places referring to the same thing.
- Poor placement of buttons, such as bunching the DIY controls in with the buttons that control which vertex and graph to use. This caused confusion for the second participant, who did not see these as separate and tried using the 'start vertex' dropdown during execution.
- Participants were unsure of what the colours and outlines meant, which led to implementing a legend explaining these meanings.

6.3 Testing

6.3.1 Unit Tests

As this website is frontend-only, the testing could be limited to unit tests that check functionality and rendering correctness. The project was built using Vite, which gives access to the testing framework Vitest, which is used in conjunction with Vue Test Utils to mount components, manipulate their state, and make assertions.

The unit tests do not cover the entire codebase, however they test the basic functionality of each of the modes for BFS and DSPA. They cover the correctness of the rendering of many components to ensure the right text is displayed at the right time.

6.3.2 Manual Testing

Each feature was thoroughly tested manually before being merged into the develop branch. Each algorithm was tested by stepping through the visualisation and DIY sections, checking that the algorithms still behaved as intended and that the UI appropriately reflected changes.

6.4 Verification of Requirements

6.4.1 Goals - 1.1

The initial goals, whilst not concrete requirements, are still a good baseline of what was set out to achieve.

Develop a game that is fun and engaging.

- All supervised evaluation participants showed they were engaged, and many stated they had fun and enjoyed trying out the WGAG. All survey respondents found it fun and engaging.
- As shown in 5.1.5, the website is hosted on GitHub Pages, which is fast, and doesn't suffer from cold-starts. The WGAG itself is a single-page web app, where navigating between 'pages' loads different components nearly instantly.
- The game does suffer from UI/UX issues; however, in general, evaluation participants and survey respondents were able to understand the visualisations and DIY controls well.

To develop a game that can only be completed efficiently when the user understands the graph algorithms.

- The participants with greater experience with graph algorithms were much more efficient in completing the DIY sections, and those unfamiliar became much more competent and fast as they learned.

A smooth development process, with CI/CD and testing.

- The website was not developed using test-driven development, however unit tests that build confidence for core functionality were created and manual tests were carried out frequently.
- The website was frequently deployed with an average of just under one weekly deployment.
- GitHub Actions was used to run tests and deploy the website, as is seen in 5.1.5

Use pedagogical techniques.

- Background research was done on the constructivist model in 2, which informed the development of the WGAG to encourage users to think for themselves on how to approach the algorithms.

6.4.2 MoSCoW - 3.1

- Must Have

Visualised Tutorial for BFS, DFS, DSPA The application has visualised tutorials for each of the three specified algorithms. The algorithms are correctly implemented, with the caveat that they do not work on disconnected graphs. The algorithms are correct, as verified partially by unit tests and fully by manual testing. This can be seen on the website.

Users can carry out actions themselves The implementation of DIY sections provides functionality for the user to complete the algorithms themselves, where they must be aware of the algorithm and select the correct options to progress, as is shown in 5.4.

Users can solve with guidance from the computer Users in the DIY section are presented with hints that accurately describe the next step that is to be taken to progress in the DIY sections, as seen in 5.7.

Well-formed, externally consistent graphs The graphs shown are basic, connected, and similar to those found on other tutorial websites. The second graph (visible at 4.3) is taken from Wikipedia contributors (2024)

Pseudo-code with highlighting As seen in 5.5, the pseudo-code for each algorithm is shown, and the step is highlighted clearly.

- Should Have

Background information on algorithms 4.1.1 and 4.9 show the design intention for background information, and the homepage of the website shows the implementation of these features.

Intuitive User Interface Based on the supervised evaluations, users who interacted with the website early in the evaluation found significant barriers in starting the DIY sections, however these issues were mostly resolved, with the only persisting issue being that some users struggled to realise that clicking on the vertices was the intended workflow. This being the only glaring issue, and the positive feedback from the survey, the user interface can be considered to be mostly intuitive. Yet there is room for improvement (see 7.1). The user interface can be seen on the website, available at A.1.

- Could Have

Additional Algorithms As a stretch goal, Tarjan's Algorithm and the Hopcroft-Karp algorithm were not implemented due to time constraints.

- Would Be Nice To Have

Greater Customisation Originally, D3.js was used to generate random graphs, however this was abandoned for adaptability reasons, replaced by SVGs. There is no feature to upload user-defined graphs, nor is there a feature to randomise graphs. There was the feature to randomise edge-weights in DSPA; however, this was removed.

7 Conclusion

Graph algorithms are a foundational concept in computing science. There are multiple approaches to teaching graph algorithms, including university lectures and online articles with visualisations.

This project aimed to create a web-based graph algorithm game (WGAG) that would add a further dimension of interactivity and autonomous learning to the visualisation approach through a 'Do It Yourself' (DIY) mode. In this mode, the user must choose options and control the graph to execute Breadth-First Search, Depth-First Search, and Dijkstra's Shortest Path Algorithm. The project was aimed at learners with some experience in computing science who are at the level required to take their first algorithms and data structures course.

Background research was conducted into pedagogical approaches in online learning, and the principles of greater learner autonomy and allowing learners to formulate their own ideas and try things first-hand guided the design and implementation of the WGAG.

Using the MoSCoW method, requirements were drafted, and from there, issues were created using the project management tool ClickUp in a Kanban style.

The website was implemented using VueJS with TypeScript and Tailwind CSS. As development began, the groundwork for strong software development practices was laid, with a CI/CD pipeline, unit tests, and pre-commit linting & formatting. Commits were small and frequent, following the Git-Flow branching strategy.

To draw graphs to the screen, SVG graphics were used in a dynamic, decoupled manner to allow for easy graph creation. Generator functions were used in the visualisations to allow users to step through BFS, DFS and DSPA using a simple 'next' button. Each click of the 'next' button evaluates the generator function, updating the visualisation until a yield statement is reached.

A DIY section was created where users could interact with the graph and a list of operations to perform on each vertex. This way, users were responsible for executing the algorithm themselves. Each operation chosen would be validated. If incorrect, a red flash would indicate they had chosen poorly. If they chose the correct option, their specific action would be executed behind the scenes using individual functions to update variables and the user interface, and they would progress.

Users could choose from three different graphs for each algorithm and decide which vertex to begin on since the three algorithms have some notion of a 'start' or 'source' vertex. Additionally, where the visualisation shows one set way of executing an algorithm, the DIY section allows the user to decide, for example, which vertex to visit next or which vertex distance to relax. This provided variety and many areas to learn through the many permutations.

The WGAG was evaluated through an unsupervised survey and supervised evaluations. The unsupervised survey was responded to by people unfamiliar with graph algorithms, and the supervised survey was conducted among mostly computing science undergraduates. The survey reported an increased understanding of graph algorithms and high satisfaction with the WGAG, where every respondent found it fun and engaging. The supervised evaluation exposed many UI flaws, which were polished as much as possible as the evaluations continued. Participants all learned how to execute the algorithms and had positive feedback. Overall, the WGAG succeeded in its goals, with the most significant drawback being its sub-optimal user interface.

7.1 Future Work

7.1.1 Improved UI/UX

Issues with the user interface were the main obstacles to users benefiting from the WGAG. Most of the time spent developing the website was focused on functionality, and not enough time was spent designing and testing the user interface. Reworking and improving this user interface would provide better learning outcomes.

The current DIY control scheme was chosen partially due to ease of implementation. However, other control schemes were considered, which would potentially be more intuitive. Another control scheme should be implemented and evaluated to see which has a shallower learning curve.

Multiple participants said they would like definitions and explanations to be presented on the algorithm pages. One participant gave the suggestion of a definition showing when a word is hovered over by the mouse. At the very least, the algorithm explanations should be available on their respective pages.

7.1.2 Implementing More Algorithms

Due to the time constraints of this project, it was only viable to develop BFS, DFS and DSPA. There are, of course, many other graph algorithms, including some lesser-known algorithms that have fewer online teaching materials. Stretch goals for this project included implementing Hopcroft-Karp and Tarjan's algorithms. To implement Tarjan's, directed graphs would need to be created as a feature.

7.1.3 Refactoring for Extensibility

Adding more algorithms would be much easier if the code-base was refactored to have generic components for each of the common elements of a visualisation and DIY page. For example, a generic pseudo-code component that takes in a list of lines and their indent level, and an index for which line is to be highlighted would mean that one component can be used for all algorithms.

7.1.4 Colourblindness

The visualisation relies heavily on the colour of the vertices to represent different states and behaviours, which excludes those with colour blindness. To remedy this, a set of colours that are friendly to the different types of colour blindness should be used, as well as an option that does not rely on colours to represent states and behaviours but on shapes, such as stripes and polka dots.

7.1.5 Back Button

Multiple evaluation participants requested a back button to reverse progress in visualisations or an undo button for the DIY sections. This was initially planned, as seen in the wire-frames 4.4; however, it was not implemented due to feature prioritisation.

7.2 Reflection

7.2.1 Project Development

This project was an overall success. The groundwork of solid software development practices and processes, including CI/CD automation, testing, a pre-commit hook, as well as using a

programming language and web framework in which I had professional experience, meant that many obstacles that could have been faced were avoided.

Much of the time spent on this project was the initial requirements gathering, design and early development, meaning that the scope of the WGAG is not as large as intended, nor is it as polished as I would like. Two weeks were spent experimenting with the visualisation library D3.js, which was found to be 'too much gun', and a simple SVG implementation was all that was necessary.

7.2.2 Educational Use Case

Whilst this website did help the evaluation participants learn, it did not serve as a gentle introduction to the concepts covered. This would be best used as an optional supplement to a university algorithms course, where students are given links to different visualisation platforms, including this one, to better understand the content being taught.

7.2.3 Wrap Up

This project tested and built upon a wide range of my skills, including but not limited to time management, long-form writing, user studies, and self-directed work on a project this large. The requirements for this project were met, but if work were to continue, an updated set of requirements should be proposed, for a new version, as if given more time and attention, this product would be an effective learning tool with much potential to be extended.

A Appendices

A.1 Links

- **WGAG:** `https://gyr1967.github.io/graph-algorithm-game/`
- **GitHub:** `https://github.com/gyr1967/graph-algorithm-game/`
- **GitHub Wiki:** `https://github.com/gyr1967/graph-algorithm-game/wiki`
- **Survey:** `https://docs.google.com/forms/d/e/1FAIpQLSf6ntcGnIfXrxfJWcwYT1seUXHfo_1FIGUKsCgdl54-_PelFQ/viewform?usp=sf_link`

A.2 Directory Structure

The structure of the `/frontend` directory, visually represented with itemize.

- DIR: src
  - DIR: assets
    - GitHub and LinkedIn logos
  - DIR: components
    - DIR: dijkstra
      - DSPA-specific VueJS components
    - VueJS components
  - DIR: graph
    - TypeScript classes for the graph data structure internal representation
  - DIR: pages
    - DIR: BFS
      - BFS visualisation and DIY VueJS page components
    - DIR: DFS
      - DFS visualisation and DIY VueJS page components
    - DIR: Dijkstra
      - DSPA visualisation and DIY VueJS page components
  - DIR: types
    - TypeScript types and interfaces
  - DIR: utils
    - Helper function TypeScript files
  - App.vue
  - main.ts
- DIR: tests
  - unit tests
- config files e.g. package.json

A.3 Kanban Board (abridged)

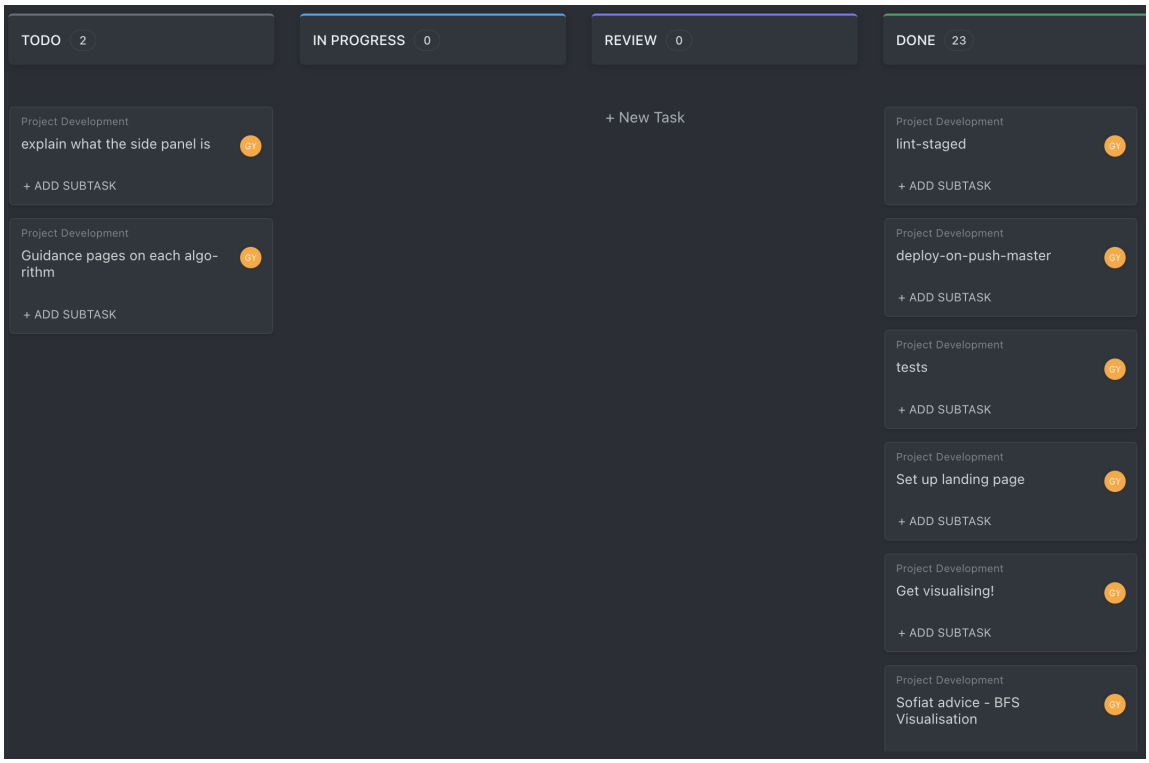


Figure A.1: *Kanban Board view of ClickUp project*

A.4 Unit Test Output

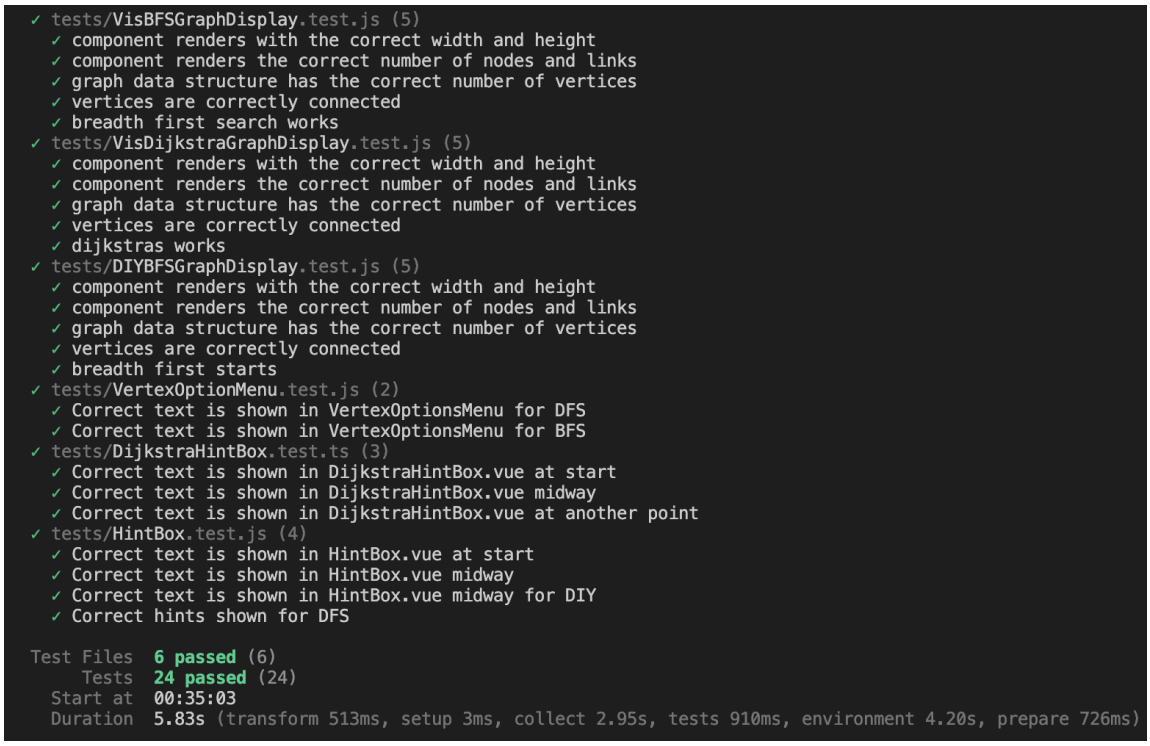


Figure A.2: *The output of* `npm run test|` *which is the npm script for* `vitest –reporter verbose|`

A.5 Survey Questions

These figures contains the survey questions seen by the respondents.

Graph Algorithm Visualisation DIY Game

As my level 4 honours project, I have developed a website that allows users to visualise some basic graph algorithms, and then carry the steps out themselves. The rationale behind this is that by going through the motions manually, the user will better understand the algorithm.

Please ensure you are using a desktop or laptop computer to interact with the website.
(https://gyr1967.github.io/graph-algorithm-game/)

To evaluate this project I need some user feedback.
You will be asked to try out some of these algorithms, and feedback your thoughts.
This is not a test of ability, I care most about honest answers.
You may withdraw at any point, and your answers will be kept completely anonymous.

Student Name: George Yarr
Email: 2553638y@student.gla.ac.uk
Supervisor: Dr Sofiat Olaosebikan

Do you consent to participating in this evaluation? *

- ◯ Yes
- ◯ No

How many years of computer programming experience do you have? *

	0	1	2	3	4	5	
Zero	◯	◯	◯	◯	◯	◯	5 or more

Please select the algorithms below that you have encountered/used before

- ☐ None of them
- ☐ Breadth-First Search
- ☐ Depth-First Search
- ☐ Dijkstra's Shortest Path Algorithm

Have you ever encountered/used pseudocode?

- ○ Yes
- ○ No

How familiar/knowledgeable are you with graph algorithms?

	1	2	3	4	5	
Completely unfamiliar	○	○	○	○	○	Expert

Please visit https://gyr1967.github.io/graph-algorithm-game/

Please have a look around, see what's on the website and familiarise yourself with the layout. There are explanations of each algorithm on the homepage; please refer to these if you are not sure what each algorithm is.

Task 1 (3-4 minutes)

Visit the game page, and try running through the visualisations for each algorithm. Pay attention to the state of the graph and the algorithm whilst doing so.
Try starting on different vertices, and on different graphs to get a feel for how the algorithm behaves in different scenarios.

These visualisations were easy to understand.

Please select how much you agree with the above statement

	1	2	3	4	
Strongly Disagree	○	○	○	○	Strongly Agree

I was able to easily understand the visualisation controls

Please select how much you agree with the above statement

	1	2	3	4	
Strongly Disagree	○	○	○	○	Strongly Agree

I can explain how the graph algorithms work to another person from memory.

Please select how much you agree with the above statement

	1	2	3	4	
Strongly Disagree	○	○	○	○	Strongly Agree

Task 2 (4-5 minutes)

Visit the game page, and try running through the DIY sections for each algorithm. In these sections you have to complete the algorithm yourself.
Try starting on different vertices, and on different graphs to get a feel for how the algorithm behaves in different scenarios.

I was able to understand the state of the graph and algorithm whilst completing the algorithms

Please select how much you agree with the above statement

	1	2	3	4	
Strongly Disagree	○	○	○	○	Strongly Agree

I was able to easily understand how to interact with the DIY interface

Please select how much you agree with the above statement

	1	2	3	4	
Strongly Disagree	○	○	○	○	Strongly Agree

I can explain how the graph algorithms work to another person from memory.

Please select how much you agree with the above statement

	1	2	3	4	
Strongly Disagree	○	○	○	○	Strongly Agree

General Questions (1 minute)

Description (optional)

I found the website fun and engaging.

Please select how much you agree with the above statement (be honest!)

	1	2	3	4	
Strongly Disagree	○	○	○	○	Strongly Agree

I would use this website to visualise and better understand graph algorithms.

Please select how much you agree with the above statement (be honest!)

	1	2	3	4	
Strongly Disagree	○	○	○	○	Strongly Agree

What are your thoughts in general on the website? Do you feel it is an effective tool? Did you learn anything?

Long answer text

Did you experience any technical issues? If so, please describe the issue(s).

Long answer text

Thank You!

Description (optional)

A.6 Survey Results

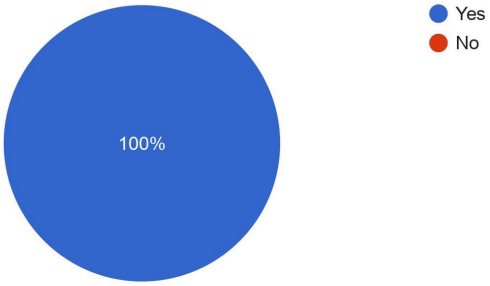
Graph Algorithm Visualisation DIY Game

5 responses

Publish analytics

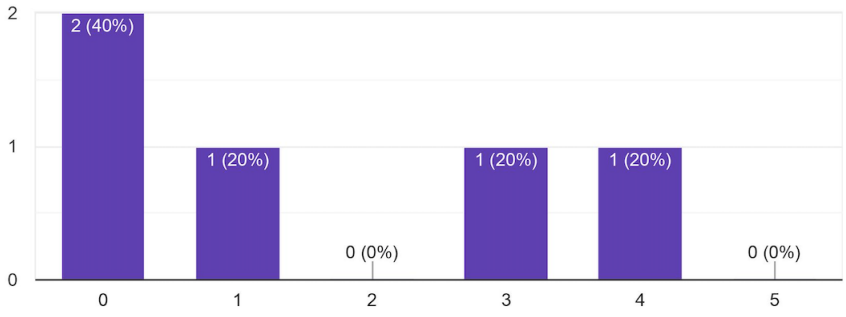
Do you consent to participating in this evaluation? Copy

5 responses



How many years of computer programming experience do you have? Copy

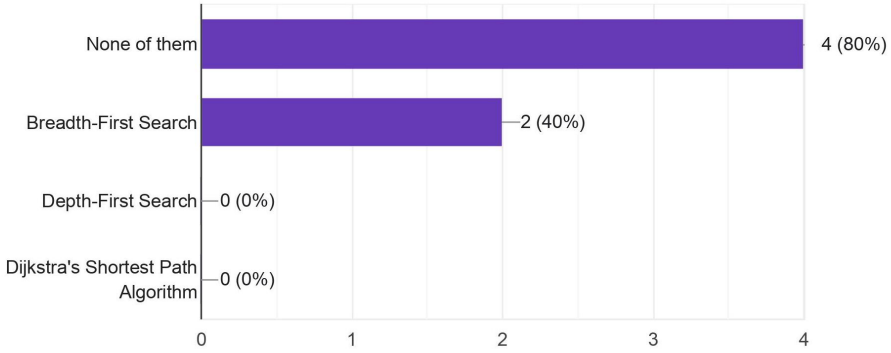
5 responses



Please select the algorithms below that you have encountered/used before

Copy

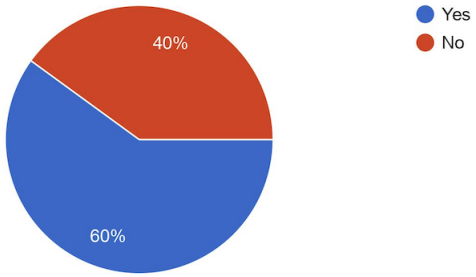
5 responses



Have you ever encountered/used pseudocode?

Copy

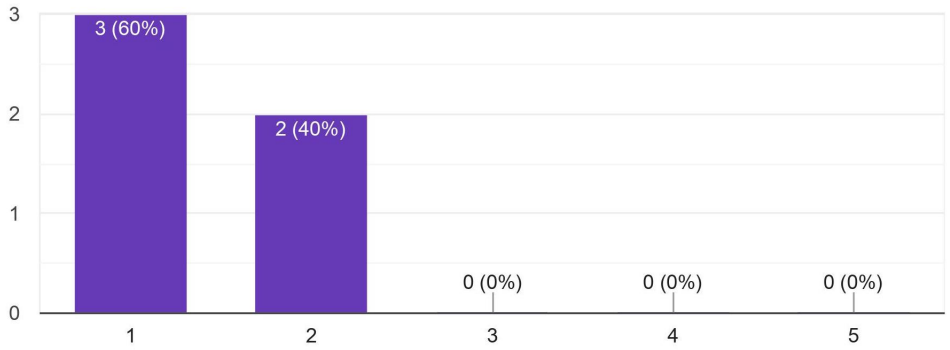
5 responses



How familiar/knowledgeable are you with graph algorithms?

Copy

5 responses



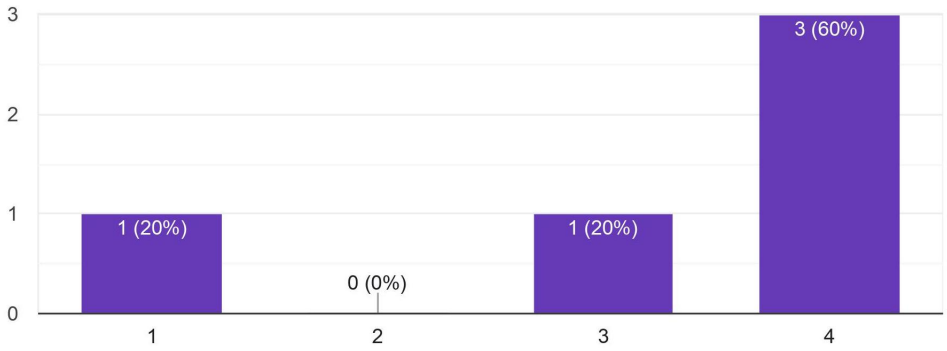
Please visit https://gyr1967.github.io/graph-algorithm-game/

Task 1 (3-4 minutes)

These visualisations were easy to understand.

Copy

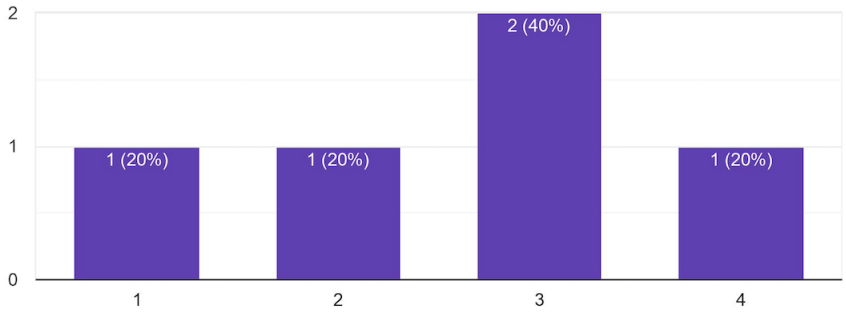
5 responses



I was able to easily understand the visualisation controls

Copy

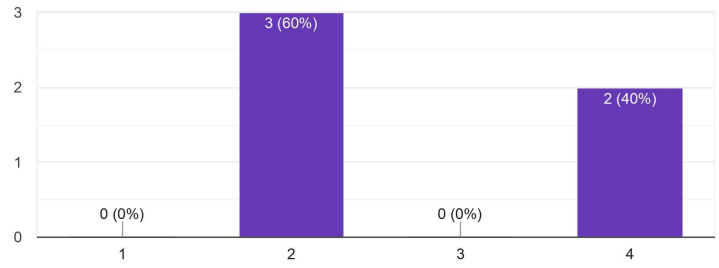
5 responses



I can explain how the graph algorithms work to another person from memory.

Copy

5 responses

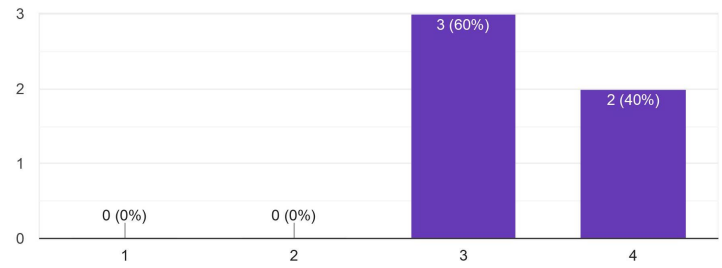


Task 2 (4-5 minutes)

I was able to understand the state of the graph and algorithm whilst completing the algorithms

Copy

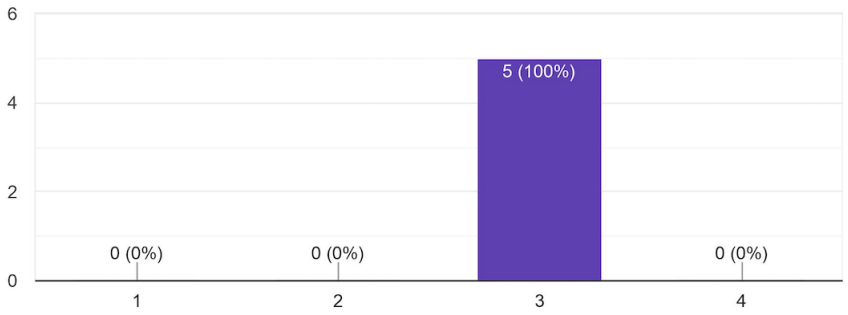
5 responses



I was able to easily understand how to interact with the DIY interface

Copy

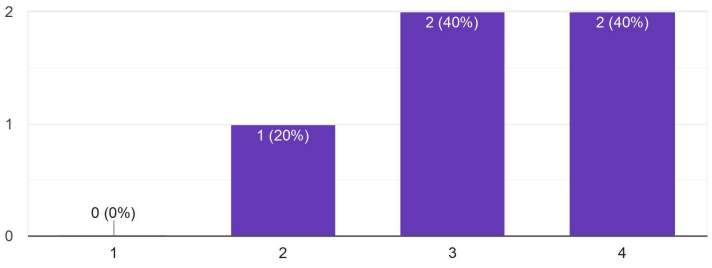
5 responses



I can explain how the graph algorithms work to another person from memory.

Copy

5 responses

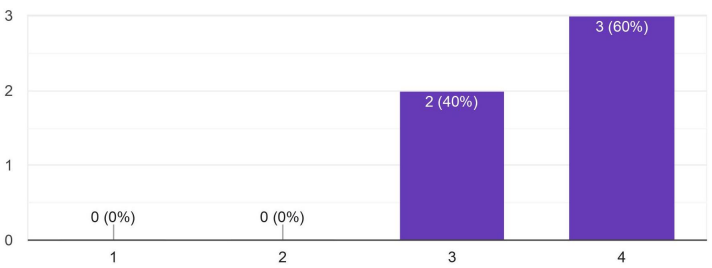


General Questions (1 minute)

I found the website fun and engaging.

Copy

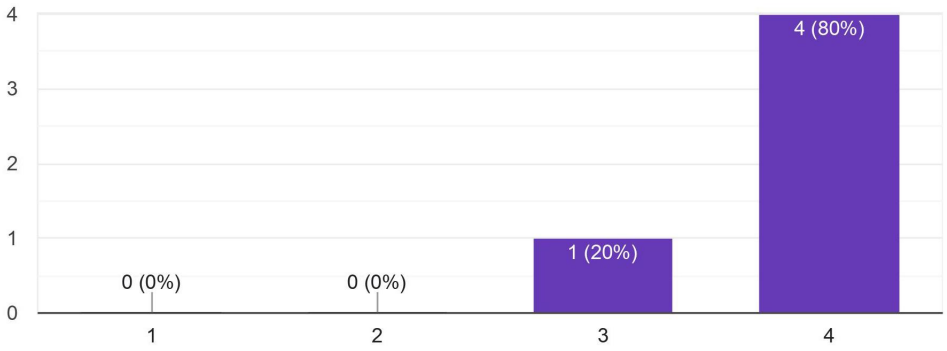
5 responses



I would use this website to visualise and better understand graph algorithms.

Copy

5 responses



What are your thoughts in general on the website? Do you feel it is an effective tool? Did you learn anything?

5 responses

Yes, worked well on my iPad, if I was studying computer science again this would help me a lot due to my need to visualise things to help understand them or commit them to memory

I think it's excellent. The explanations are all very clear and user-friendly and there is just the right amount of text. I definitely learned how the algorithms work. Playing the DIY game was more effective than when someone tried to explain it to me. It was good to see 2 similar algorithms side-by-side, and by actually going through the steps in the DIY sections I could see the difference, and I got a real feel for the process. When I went wrong in the DIY sections the hints were really helpful and clear. By the time I got on to the third graph of each I knew exactly what I was doing and could remember it all. If I had needed more practice then I could have used different starting vertices, so lots of learning opportunities. I found the Dijkstra's algorithm much harder and the reference to Google Maps confused me a bit. During the visualisation I was unclear about what Dijkstra's algorithm was actually doing. A clearer 'shortest paths' box would help me to understand the algorithm better. The website has left me wanting to learn more. I think an extra section with more info about the application of the algorithms would be a good learning opportunity for me.

Yes the Visualisation helps lead into the DIY sections. The DFS and BFS are understandable for me but not the Djykstra one

Definitely an effective tool. The visualisations help build initial understanding and the DIY section helped me cement it. If there was anything I didn't understand I could practice with the guide still visible and once I felt I was secure I turned it off to test my knowledge.

Fun, got the hang of things a bit

Did you experience any technical issues? If so, please describe the issue(s).

3 responses

There was no distance between A and C on the graph. I had to hit reset when I didn't think I should have to in the depth-first search.

No

No technical issues. The only visual improvement I think could be made would be a highlight on each point in the graph when you select it. Currently the only confirmation a selection has been made is the options underneath the graph changing. E.g. When clicking on point B having its border change colour.

Thank You!

A.7 Ethics Checklist

See `https://github.com/gyr1967/graph-algorithm-game/blob/develop/data/ethics/ethicschecklistsigned.pdf` and below for the signed ethics checklist:

School of Computing Science
University of Glasgow

Ethics checklist form for 3rd/4th/5th year, and taught MSc projects

This form is only applicable for projects that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, getting information about how a system could be used, or evaluating a working system.

If no other people have been involved in the collection of information, then you do not need to complete this form.

If your evaluation does not comply with any one or more of the points below, please contact the Chair of the School of Computing Science Ethics Committee (matthew.chalmers@glasgow.ac.uk) for advice.

If your evaluation does comply with all the points below, please sign this form and submit it with your project.

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.

   Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback

2. The experimental materials were paper-based, or comprised software running on standard hardware.

   Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, laptops, iPads, mobile phones and common hand-held devices is considered non-standard.

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.

   If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

   Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

4. No incentives were offered to the participants.

   The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

6. No participant was under the age of 16.
   Parental consent is required for participants under the age of 16.

7. No participant has an impairment that may limit their understanding or communication.
   Additional consent is required for participants with impairments.

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. All participants were informed that they could withdraw at any time.
   All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

10. All participants have been informed of my contact details.
    All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. In cases where remote participants may withdraw from the experiment early and it is not possible to debrief them, the fact that doing so will result in their not being debriefed should be mentioned in the introductory text.

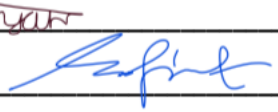
12. All the data collected from the participants is stored in an anonymous form.
    All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title Web-based Graph Algorithm Game

Student's Name George Yarr

Student Number 2553638y

Student's Signature 

Supervisor's Signature 

Date 21/03/2024

7 Bibliography

F. Al-Sayegh. Visualization of classical graph theory problems, 2023. URL https://visualise-graph-problems-with-me.netlify.app/.

M. Bogner, Justus; Merkel. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. In *Proceedings of the 19th International Conference on Mining Software Repositories*. Association for Computing Machinery, 2022. doi: 10.1145/3524842.3528454. URL https://doi.org/10.1145/3524842.3528454.

L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015. doi: 10.1109/MS.2015.27.

D. Clegg and R. Barker. *Case method fast-track: a RAD approach*. Addison-Wesley Longman Publishing Co., Inc., 1994.

G. Dent. Workshape.io, 2016. URL https://workshape.github.io/visual-graph-algorithms/.

L. Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, Computers 35, 379–383*, 2003. doi: https://doi.org/10.3758/BF03195514.

M. Fowler and M. Foemmel. Continuous integration, 2006.

F. Halim, S.; Halim. Visualgo.net, 2011. URL https://visualgo.net/en.

J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

L. Lau. algmatch, 2021. URL https://liamlau.github.io/individual-project/.

A. Namoun. *Three Column Website Layout vs. Grid Website Layout: An Eye Tracking Study*, pages 271–284. 06 2018. ISBN 978-3-319-91802-0. doi: 10.1007/978-3-319-91803-7_20.

C. Ormond. algmatch, 2023. URL https://callumormond.github.io/individual-project/.

A. Patel. Introduction to the a* algorithm, 2014. URL https://www.redblobgames.com/pathfinding/a-star/introduction.html.

B. Pelz. (my) three principles of effective online pedagogy. *Journal of Asynchronous Learning Networks, v14 n1 p103-116*, 2010. doi: https://eric.ed.gov/?id=EJ909855.

T. J. Schell, George; Janicki. Online course pedagogy and the constructivist learning model. *Journal of the Southern Association for Information Systems, volume 1, issue 1*, 2013. doi: http://dx.doi.org/10.3998/jsais.11880084.0001.104.

Wikipedia contributors. Dijkstra's algorithm — Wikipedia, the free encyclopedia, 2024. URL https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=1213317183. [Online; accessed 13-March-2024].