DEPARTMENT OF COMPUTER SCIENCE

Designing a Framework for Generating Graph Theory Questions for Education

Paolo Mura

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Thursday 4th May, 2023

Abstract

There are many existing websites that provide educational content for graph theory, including algorithm visualisations and quizzes. However, most of these resources are presented as a fixed set of topics, with no scope for extending them.

I developed a website called Graph Quest together with an accompanying Python package called *graphquest*. The *graphquest* package defines abstract base classes for generic question types. Teachers are able to extend these to write their own specific questions and upload them to the Graph Quest website. The website then takes care of the front-end presentation of the questions. This allows teachers to focus on the logic of the question rather than the visualisation.

The result was a success, meeting all of the initial requirements and surpassing some of the features of similar solutions. The highlight of Graph Quest is its extensibility; it is very straightforward to add new features and question types. There are some minor bugs that have been identified and several further improvements that could be made to enrich Graph Quest's feature set.

Dedication and Acknowledgements

I am extremely grateful to my supervisor, Dr John Lapinskas, who regularly provided helpful guidance and feedback on both my implementation and thesis. I am also thankful to Zac Woodford for giving constructive peer-review on some sections of my thesis. Lastly I would like to thank the participants who took time out of their studies to interview for my project, all of whom contributed valuable feedback.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Paolo Mura, Thursday 4th May, 2023

Contents

List of Figures

List of Tables

Ethics Statement

This project fits within the scope of ethics application 0026, as reviewed by my supervisor, Dr John Lapinskas.

Supporting Technologies

- I used React Bootstrap for many of the generic UI components.
- I used Material UI for some additional UI components, most notably icons.
- I used CytoscapeJS for the graph visualisation.
- I used NetworkX for converting the back-end graph representation to CytoscapeJS format.
- I used Flask for building the server API.
- I used Flask JWT Extended for authentication tokens.
- I used NumPy and Shapely for vector manipulation and line intersection functions.
- I used Docker to containerise the application.
- I used AWS LightSail to host the website.
- I used the "Rise of Kingdom" font by Vladimir Nikolic from FontSpace [64] for the website title.

Notation and Acronyms

API	:	Application Programming Interface
AQA	:	Assessment and Qualifications Alliance
AWS	:	Amazon Web Services
CORS	:	Cross-Origin Resource Sharing
CPU	:	Central Processing Unit
CRUD	:	Create, Read, Update, Delete
CSS	:	Cascading Style Sheets
DoS	:	Denial of Service
GDPR	:	General Data Protection Regulation
HTML	:	HyperText Markup Language
HTTP	:	HyperText Transfer Protocol
ICL	:	Imperial College London
iff	:	If and only if
I/O	:	Input/Output
JSON	:	JavaScript Object Notation
JSX	:	JavaScript XML
JWT	:	JSON Web Token
MIT	:	Massachusetts Institute of Technology
NP	:	Nondeterministic Polynomial
OCR	:	Oxford, Cambdridge and RSA Examinations
OS	:	Operating System
RAM	:	Random Access Memory
REST	:	Representational State Transfer
SSO	:	Single Sign-On
SVG	:	Support Vector Graphics
UCL	:	University College London
UI	:	User Interface
UML	:	Unified Modelling Language
URL	:	Uniform Resource Locator
UX	:	User Experience
VM	:	Virtual Machine

Chapter 1

Introduction

There are currently no existing solutions that I have come across that provide an interface for generating practice questions for graph theory. Similar solutions each provide a subset of the required functionality. For instance, there are many quiz generator websites, but none that specialise in graph theory. There are also existing sites that deliver practice question on graph theory, but only serve a fixed set of topics with no opportunity for extension. This is a particular problem for higher education courses, which often tend to specialise in niche topics that are not covered by these websites.

The project requirement is therefore a website that allows teachers to easily define their own questions for graph theory. Henceforth I make the distinction between the two users of the website as being "teachers" (who generate the content) and "students" (who consume the content). I also use the term "topic" interchangeably with the word "quiz". The specific aims are as follows:

1. Abstract away the front-end visualisation of graphs and the presentation of the question, so that the teacher only has to concern himself with the logic of the question.
2. The solution should equip the teacher with enough power for creative freedom and to design a wide variety of types of questions, which support the needs of their individual syllabus.
3. The solution must be highly extensible, such that new features may be added in a modular fashion.
4. The interface should be simple, intuitive and easy-to-use for both teachers and students.
5. The website design should be accessible to a wide range of students.

I met these criteria by developing a website called Graph Quest with an accompanying Python package called *graphquest*. The name "Graph Quest" was chosen as a double entendre. It is an abbreviation for "graph question generation" and also implies the student's use of the site is a journey or "quest" through graph theory.

The *graphquest* package may be installed by a teacher with *pip* [24]. It contains abstract base classes which they must extend with their own Python classes. The base classes represent fundamental methods of answer entry (such as text input or multiple choice), while the teacher's classes should define the logic for specific questions. For example, the teacher may extend the text input base class to implement a question that asks the student "what is the distance between vertex u and vertex v?".

Once the teacher has finished writing their question scripts, they upload them to the Graph Quest website. The website is made up of a Python Flask server, which serves a front-end React application as well as RESTful API endpoints for the back-end. The entire application is containerised with Docker [49] and hosted on AWS Lightsail [2].

The back-end of the website exposes authentication-protected endpoints for performing CRUD (create, read, update, delete) operations on the teacher's uploaded files. The teacher is also able to construct "topics" (i.e., quizzes) via the website's graphical front-end interface. After sequencing their uploaded questions into one or more topics, they then share the topic code(s) with their students.

When a student accesses the Graph Quest website with a topic code, the server looks up the topic and accesses each question script referenced by it. It dynamically imports these scripts at runtime, calling their methods to generate the data for each question before sending the serialised data for the entire topic back to the student.

The front-end student page is constructed from a nested hierarchy of React components. The top level component receives the topic data and passes it down to its children as "props" (a.k.a. properties).

A progress bar contains different tabs for each question of the topic. Each question is laid out as a graph visualisation on the left-hand side and a question panel on the right. The graph component is built on top of Cytoscape.js, a JavaScript graph visualisation library [27]. It supports various styling and layout options for the graphs. The component exposes its functionality to the question via an event interface. To be more specific, it broadcasts events when the user interacts with the graph (such as by tapping on a node), and listens to action requests (such as to highlight a specified edge).

The question panel houses the logic of the the question that corresponds directly to the Python abstract class that the teacher's question is based on. The JavaScript functions specific to this question (such as answer construction and verification) are loaded dynamically when the question panel first mounts.

I performed a user study with five computer science students who had previously studied graph theory. The study involved an individual interview per participant with some general questions followed by an interactive demonstration of the website. The feedback was mixed, with students highlighting the automatic question generation and feedback, graph interactivity and minimalist design aesthetic as being the website's strengths. However, they also suggested that the interactivity could be taken further, raised issues with accessibility and uncovered a small handful of bugs. I was able to address most of these issues in subsequent development.

Ultimately the website met all of the aims above to some degree, successfully providing a framework for generating practice questions for graph theory, with the most notable success being its extensibility. This is something that other similar solutions do not provide. Graph Quest surpasses these solutions in some areas (particularly visualisation of weighted graphs) but falls short in others (such as number of topic-level configuration settings).

Chapter 2

Background

Graph theory is the study of graph data structures and their algorithms. Graphs model nodes (a.k.a. vertices) and edges (connections between node pairs). The focus of this section will not be the fundamentals of graph theory since I assume the reader already has prior knowledge of the topic. For a better introduction, a textbook that covers the content well is [11]. It is also assumed throughout that the reader has a basic understanding of web technologies (HTML, CSS, JavaScript, HTTP, etc.).

The subject is often taught at secondary and higher education as part of mathematical and computer science courses. AQA and OCR are two UK A-Level exam boards that provide computer science courses. They both cover almost identical content with respect to graph theory, including basic terminology, trees, binary search trees and traversals (depth-first and breadth-first search) [4] [66].

In higher education, there is also overlap in the topics taught, however some courses cover topics that others do not. According to [32] and [33], The University of Cambridge, University of Oxford, University of St Andrews, Imperial College London, University College London and the University of Bristol are among the highest ranking UK universities for computer science in 2023. Table 2.1 shows the topics within graph theory that are taught in a selection of some of the mathematics or computer science courses at these universities. In this table, "storage" refers to graph representation (e.g. adjacency lists).

There is certainly some overlap between the topics taught. In particular, paths and cycles (including Eulerian circuits and Hamilton cycles) are common to all courses shown in the table. The next most common topics are trees and matchings in bipartite graphs. By contrast, planar graphs, extremal problems (such as Turán's theorem [88] or the Zarankiewicz Problem [103]) and different variations of tree data structures are less widely covered.

To be more specific, the Cambridge Part 1A CST Algorithms 1 course [67] teaches B-trees, the Oxford Algorithms and Data Structures course [15] teaches splay trees and the Bristol Algorithms II course [55] teaches 2-3-4 trees. Although they are all examples of balanced trees, they each have distinctions. Every time a node is accessed in a splay tree, it is moved to the root of the tree. 2-3-4 trees are a specific case

Table 2.1: Graph theory topics taught in various courses at top UK universities. Courses that are specifically graph theory-focused are in italics, the rest include graph theory as a subset of the course content (such as an algorithms module).

Topic	**Total**	**Cam [68]**	***Oxf* [57]**	**ICL [16]**	***UCL* [18]**	***SAn* [69]**	**Brs [55]**
Storage	3	Yes		Yes			Yes
Paths and cycles	6	Yes	Yes	Yes	Yes	Yes	Yes
Sort and search	3	Yes		Yes			Yes
Spanning trees	3	Yes		Yes			Yes
Matchings	4	Yes	Yes		Yes		Yes
Colourings	3		Yes		Yes	Yes	
Planar graphs	2		Yes			Yes	
Trees	4		Yes		Yes	Yes	Yes
Binary search trees	1						Yes
Union-find	1						Yes
Flow networks	3	Yes	Yes				Yes
Extremal problems	2		Yes		Yes		

of the generalised B-tree, in which nodes may be combined and split.

Course content for Algorithms II is delivered through online videos and lecture slides, while Blackboard (see 2.1) is used to provide quizzes that aid learning. The quizzes are intended to help students in "keeping up with the unit" and cover a spread of the whole content, with a quiz for every week of the course. The quizzes test students' understanding of definitions and theorems as well as the steps and complexities of algorithms. All questions use a multiple-choice format and some include accompanying images. Some examples of questions include the following, each of which displays an accompanying image of graph(s):

- Does G1 have an Euler walk from v1 to v4?
- Find the distance between vertices x and y in each graph.
- What is the third edge added to the output minimum spanning tree by Kruskal's algorithm?

All in all, it is evident that in higher education, there is a much wider variety of topics taught than at A-Level. Some courses teach content that is rarely covered by other universities. However, quiz formats need little variety to support them; as a minimum, multiple choice with accompanying images is enough.

2.1 Blackboard

Blackboard is an educational technology product with support for creating "Tests" (quizzes) [47]. The teacher configures general settings for the Test, such as visibility, due date, number of attempts and time limit. Blackboard also partners with Examity [44] and Respondus [46] to mitigate cheating during assessments via proctoring and browser lockdown.

It supports nine different question types including multiple choice, matching, and true/false questions. Blackboard's "Question Banks" [45] provide a method to store individual questions for re-use across different Tests.

In terms of randomisation, there are options to do this at the Test level as well as the question level. For tests, the order of questions can be randomised and "Question Pools" [48] are a way to give each student a different random subset of the Topic's questions. At the question level, the order of answers can be randomised in the case of questions like multiple choice. There is also a "Calculated Formula" [43] question type, which allows the teacher to include variables and simple mathematical formulas in the question generation as seen in Fig. 2.1.

The teacher is able to upload images to be included inline as part of each question. The "Hotspot" [42] question requires students to drop a "pin" on the image, with their answer being correct iff it is within region(s) specified by the teacher.

Finally, Blackboard supports automatic answer verification for most questions whereby the teacher specifies the correct solution or formula as part of the question. Written feedback can also be displayed on submission, but unless the teacher submits it manually after the student's submission, it will be generic.

When considering Blackboard as a tool for generating questions for graph theory, its strengths include the number of configurable settings and question types. Although it does support an element of randomisation and feedback, it is quite limited. There is no way for graph images to be randomised, feedback to be tailored to students' answers, or for more sophisticated question generation beyond variables and simple mathematical formulas.

2.2 Numbas

Numbas is an alternative question creation tool [92]. It offers many of the same features as Blackboard, including a variety of question types, image import and randomised ordering. However, it takes randomisation *much* further through its use of variables and scripting.

Unlike Blackboard, its variables are not limited to simple numerical ranges and mathematical formulas. Instead, Numbas uses a scripting language called JME to define variables [94], which allows for much more powerful randomisation. The teacher can also write "marking algorithms" through JME that are used to determine both the student's mark and customised feedback based on their answer [95]. On top of this, the teacher can use JavaScript to override default Numbas functionality in some cases [96], or write extensions more generally [97].

Numbas provide their own first-party extensions [93]. These include wrappers for the data visualisation libraries GeoGebra [29], JSXGraph [99] and Eukleides [65] as well as their own graph theory extension.

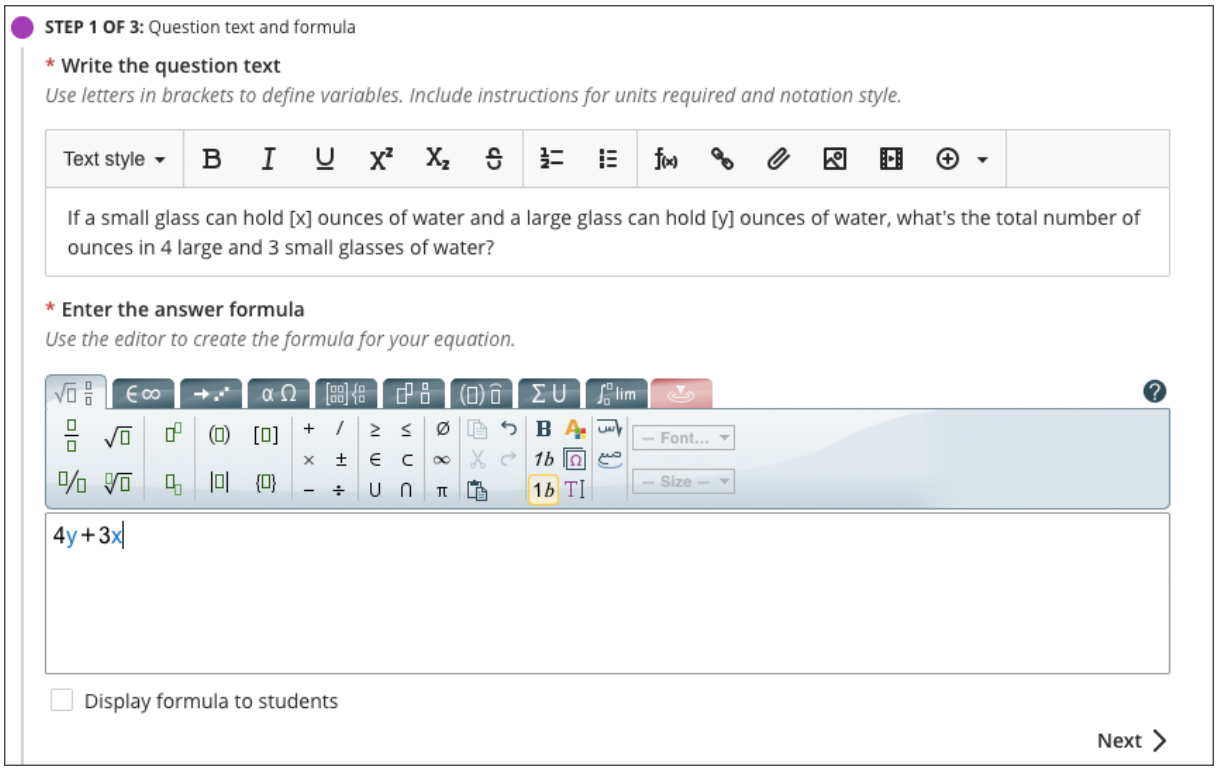


Figure 2.1: Creating a Calculate Formula question type in Blackboard [43].

None of the data visualisation libraries inherently support graph data structures (they are primarily designed for displaying charts or plots). The graph theory extension has a very basic set of features, with little more than a few common algorithm implementations and a graph drawing function for use in a question. An example use of this extension is shown in Fig. 2.2. It demonstrates the generation of a simple graph that is displayed in the question. Since the extension lacks graph interactivity features, the student is required to enter their answer in a table.

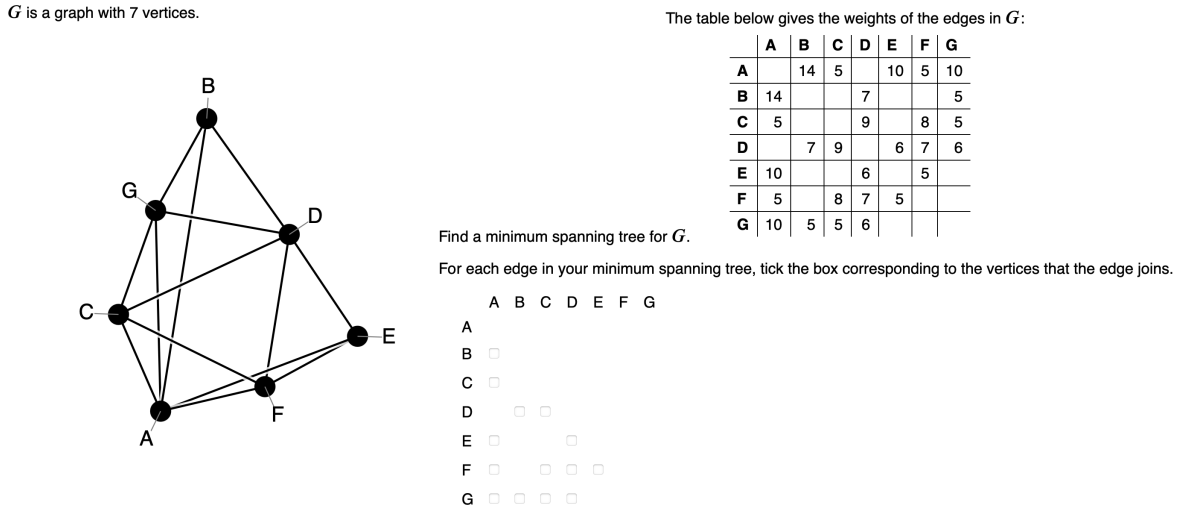
Overall, Numbas is an extremely powerful question generation tool. Its main advantages are its randomised question generation capabilities and extensibility through scripting. Although there are ways to generate graph images for use in questions, these are limited. It doesn't natively allow teachers' question generation or marking algorithms to interface with an extensive graph theory library.

2.3 VisuAlgo

VisuAlgo on the other hand is a website designed specifically for teaching algorithms, overlapping heavily with graph theory. It was first introduced in [38], where it was intended to provide a "unified" platform with an interactive and visually consistent approach to learning algorithms.

Although its original features centred around algorithm visualisation (see section 2.5), it has since added automatic question generation as well [35]. Currently it supports twenty-four topics, including the following from graph theory:

- Binary heap
- Binary search tree
- Graph structures
- Union-find data structure
- Fenwick tree
- Segment tree
- Graph traversal
- Minimum spanning tree
- Single-source shortest paths
- Cycle finding
- Network flow
- Matchings
- Minimum vertex cover
- Steiner tree
- Travelling Salesman Problem



	A	B	C	D	E	F	G
A		14	5		10	5	10
B	14			7			5
C	5			9		8	5
D		7	9		6	7	6
E	10			6		5	
F	5		8	7	5		
G	10	5	5	6			

Figure 2.2: Example question from Numbas using the Graph Theory extension [91].

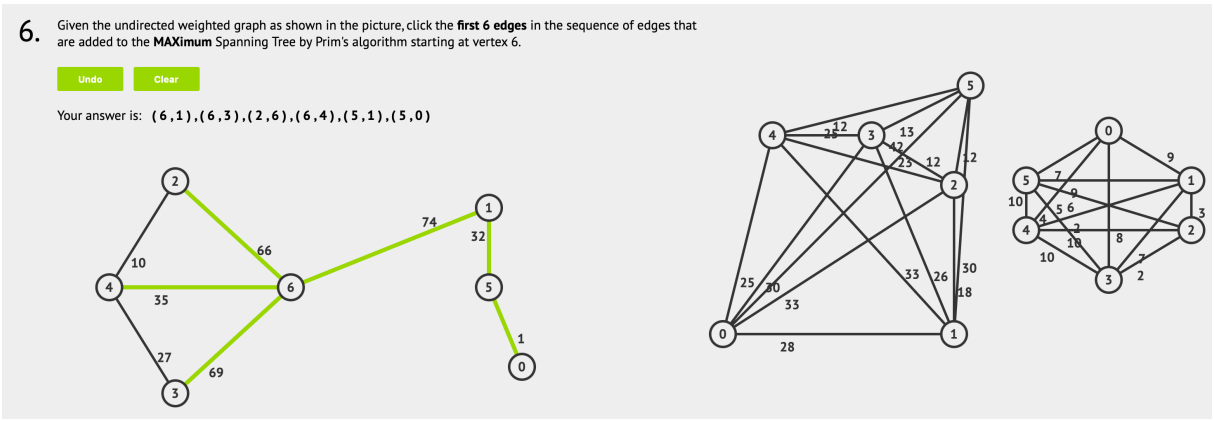


Figure 2.3: From left to right: an example question from VisuAlgo's Min Spanning Tree topic; two examples of poorly displayed weighted graphs [36].

This is a good spread, however it is not comprehensive enough to cover all the topics required by every university course. In particular, it lacks coverage of some of the most commonly taught topics such as Eulerian circuits and Hamilton cycles, as well as more niche ones such as splay trees as discussed at the start of this chapter.

Each topic includes a quiz component, which uses rules to generate the questions and verify students' answers so that the same question will have a different variant each time the student attempts it. As well as text input and multiple choice question types, it also supports several graph interaction questions. These require students to select edges or vertices in a given graph, as in Fig. 2.3 (left), or construct a graph themselves as their answer.

The main strengths of VisuAlgo are its automatic question generation and answer verification, and graph interactivity. Its graph visualisations are generally good, although they can sometimes render poorly, especially in the case of weighted graphs as can be seen in Fig. 2.3 (right).

Despite aiming for a "unified" approach, this is actually one of its biggest limitations. As a closed-source solution, support for additional topics is purely at the discretion of the website's developers, so university courses with specific needs are not catered for. It is unclear how extensible the website is for the developers, but there is currently no way for external teachers to add their own questions.

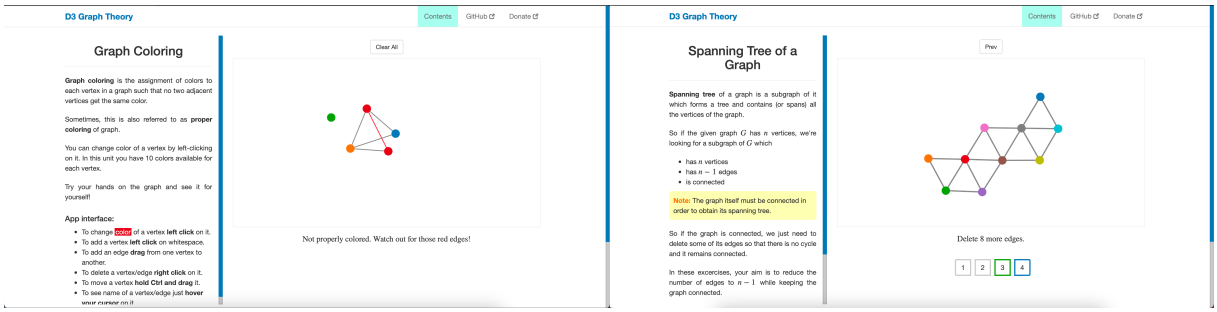


Figure 2.4: Two example topics from the D3 Graph Theory website [71].

2.4 D3 Graph Theory

D3 Graph Theory is another website specialising on teaching graph theory [71]. Unlike VisuAlgo, it does not include traditional quizzes, instead providing more of a set of open tasks. It does focus on interactive graphs however, utilising the powerful D3 [7] JavaScript library to visualise the graphs in an aesthetically pleasing manner as demonstrated in Fig. 2.4. This is achieved through the use of force-directed layout algorithms, while also providing a highly intuitive mechanism for manipulating the graph through clicking on nodes and edges. A more comprehensive look at D3 and force-directed layouts can be found in section 3.5.1.

While its graph visualisations are very impressive, D3 Graph Theory does have the same pitfall as VisuAlgo: it only provides a set number of topics. The full list is as follows:

- Vertices and Edges
- Order and Size of a Graph
- Degree of a Vertex
- Degree Sequence of a Graph
- Graphic Sequence
- Havel-Hakimi Algorithm
- Pigeonhole Principle
- Regular Graph
- Complete Graph
- Bipartite Graph
- Complete Bipartite Graph
- Walk
- Open vs Closed Walks
- Connectivity
- Eulerian Circuit
- Eulerian Trail
- Graph Coloring
- k-Colorable Graph
- Chromatic Number
- Trees
- Rooted Trees
- Spanning Tree of a Graph

As with VisuAlgo, they cover a good spread of content, but still leave out some of the common concepts including Hamilton cycles and binary trees, and do not support any of the more advanced topics such as 2-3-4 trees.

However, D3 Graph Theory is open source under an MIT license at [73], allowing it to be forked and modified. It is designed in an extensible way such that new topics may be added by writing the following:

- A JavaScript file for the graph visualisation;
- A CSS file for the styling of the graph; and
- A JSON entry that holds the text and data for the question.

This provides a fairly simple interface for adding new topics. However, adapting it for automatic question generation involves many steps for the teacher, including the creation of a question framework, the individual questions themselves and each accompanying graph visualisation, which uses D3. This is explored further in section 3.5.1, but in summary, the library is a low-level tool that requires a lot of work to visualise graphs.

Overall, D3 Graph Theory has a strong visualisation component, though it is limited in topics covered, and adapting it for automatic question generation is an involved process.

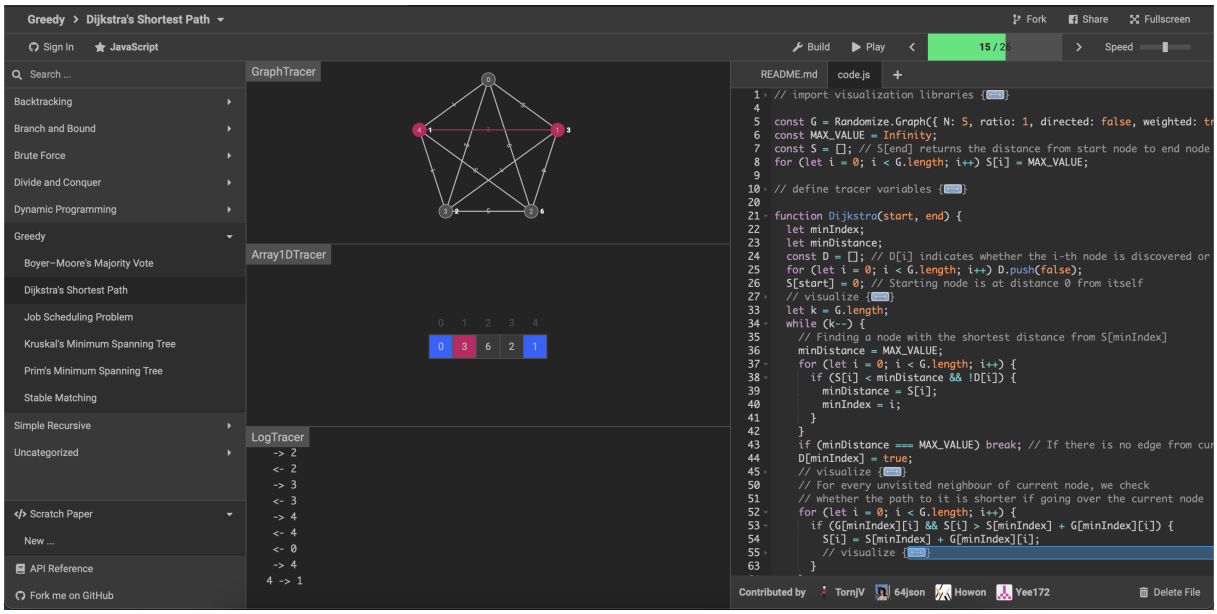


Figure 2.5: Algorithm Visualizer's Disjkstra's Shortest Path page [100].

2.5 Algorithm Visualizer

Algorithm Visualizer specialises in defining a framework for teachers to generate algorithm visualisations [74]. An algorithm visualisation refers to the process of using visual representations to explain how an algorithm works. In the case of graph theory, it often involves stepping through the algorithm, while an accompanying image of a graph is manipulated accordingly. This may be by changing values, modifying the graph structure or highlighting elements in the graph. Fig. 2.5 shows an example of this in Algorithm Visualizer; the algorithm is on the right-hand side and data structures are shown in the centre. As the student plays through the algorithm, the data structures are updated. Many websites have been designed to provide this feature for a set of pre-defined algorithms, including VisuAlgo and several others [17] [28].

What distinguishes Algorithm Visualizer is its open-source framework. This framework allows teachers to write their own algorithm in JavaScript, C++ or Java, inserting "tracers" throughout the code. Tracers are classes that encapsulate the visualisations that are shown in the centre of the screen at a very high level. Currently, the following five Tracers are supported:

- Array1DTracer
- Array2DTracer
- ChartTracer
- GraphTracer
- LogTracer

Chart represents a bar chart, Log represents a text stream (i.e., logging) and the rest are self-explanatory. Each Tracer exposes methods that may be used to interact with the visualised data. For example, GraphTracer includes a `select()` method to highlight an element in the graph. There is also a `delay()` method, which is used to pause the simulation at that line of code.

All in all, the teacher is able to write any algorithm of their choice with the full power of JavaScript, Java or C++. Using Tracers allows them to interact with the visualised data structures and focus their attention on writing the algorithm rather than the implementation of the front-end visualisation. Some of the downsides of Algorithm Visualizer are that its graph Tracer only supports circle layouts (where nodes are arranged in a circle) and it does not feature any framework for adding practice questions.

2.5.1 Summary of Existing Solutions

Table 2.2 shows a summary of the features that the aforementioned solutions provide. Each one specialises in a subset. For example, D3 Graph Theory specialises in graph layout and interactivity, but has no

Table 2.2: Summary of the provision of required features by existing solutions.

Feature	**Blackboard**	**Numbas**	**VisuAlgo**	**D3GT**	**Algorithm Visualizer**
Question Generation	Randomised	Randomised + scripted	Randomised + scripted	None	None
Feedback	Generic	Customised	Customised	N/A	N/A
Graph visualisation	Image import	Image import + simple layout	Advanced layout	Advanced layout	Simple layout
Graph interactivity	N/A	None	Good	Great	None
Easily interface with a graph theory library	No	No	No	Yes	Yes
Extensible	Yes	Yes	No	Possible	Yes

support for practice questions. Numbas is at the opposite end of the spectrum, with powerful question generation through scripting, but little support for graph visualisations. Algorithm Visualizer is a great example of a solution that balances a high degree of creative freedom with abstraction of the front-end visualisation. However, none of these examples are able to deliver all the required features set out in chapter 1.

Chapter 3

Project Execution

3.1 Overview

My solution is a website called Graph Quest, which involves two main components: a Python package called *graphquest* and a web server as illustrated in Fig. 3.1 (left). The *graphquest* package defines the interface that teachers should use via Python abstract base classes such as the example outlined in Fig. 3.1 (right). Each class encapsulates a general question type, or "shell", such as *QSelectPath* for questions that require the student to select a path of nodes in a given graph. Teachers import the *graphquest* package and extend its classes to create their own specific question types locally. For example, they could create an Euler walk question or a shortest path question, both of which would utilise the *QSelectPath* functionality, since they both require the student to submit a path as their answer.

Once the teacher has written their question subclasses, they upload them to the Graph Quest website as Python scripts to be stored in the server's file store. The teacher constructs "topics" (i.e., quizzes) via a web interface. These topics are stored as JSON files that include a unique "topic code", settings and references to the questions to be used in that topic.

When a student accesses a given topic via its topic code, the server constructs the questions by dynamically loading the teacher's scripts and calling their methods. The resulting JSON schema is sent to the student's browser, which handles the client-side rendering of the questions. This includes the question description, graph layout and answer collection, which depends on the *graphquest* question type used. For example, the *QTextInput* questions will involve a simple text box, whereas questions based on *QSelectPath* require more complex interaction with the graph itself.

On submitting their answer, it is either matched against a list of accepted solutions that were supplied with the question; or their answer is sent to the server to be processed by a verification script which returns customised feedback.

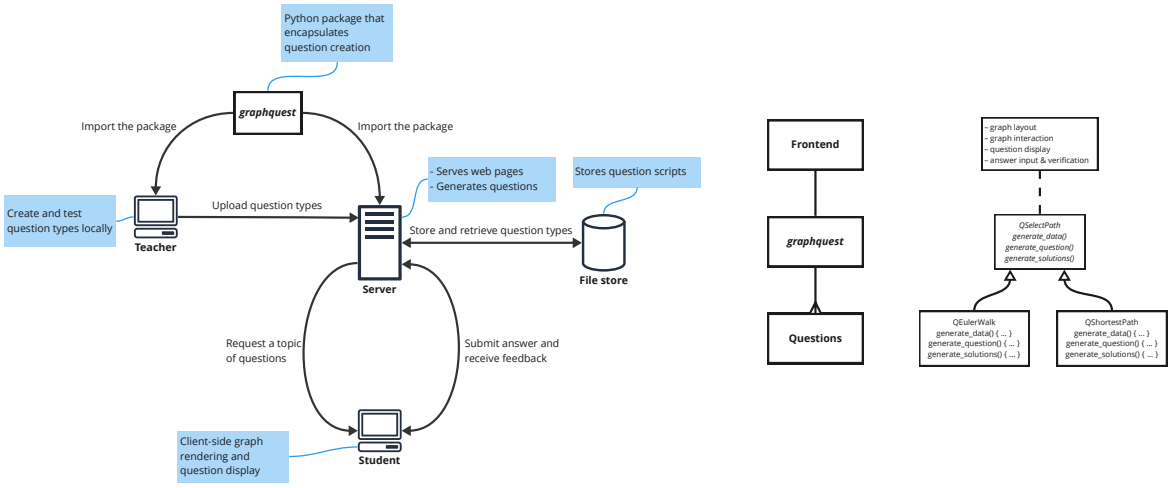


Figure 3.1: Left: high-level architecture diagram. Right: the levels of question definitions.

3.2 The *graphquest* Package

The primary aims of the project required a method for teachers to define questions in a simple, yet powerful manner. The solution should enable them to write code that determines the logic of the question, without having to detail the visualisation of the graph or the presentation of the question.

With this in mind, I decided to create the *graphquest* package. It is a Python package complete with documentation, which includes two modules: *question* and *graph*. The *question* module provides abstract classes that represent shells for different question types (multiple choice, text input, etc.). These classes define abstract methods that interact with the NetworkX graph theory library (discussed in section 3.2.1). To create a new type of question, the teacher extends the base class and implements these methods. The *graph* module provides useful functions that extend the features of the NetworkX library.

3.2.1 Choosing a Language and Graph Theory Library

I spent time considering several options before settling on Python and NetworkX. My requirements for a graph theory library were ordered as follows, from most to least important:

1. Support a wide variety of features
2. Popular, well-maintained library
3. Simple and intuitive to use
4. Interface well with Cytoscape (my chosen graph visualisation library; see section 3.5.1)
5. Performance

The reason performance is less of a concern is because in practice, all questions are likely to involve small graphs of much less than 100 nodes. Even algorithms with expensive time complexities or brute force solution verification functions would not adversely affect runtime.

Consider the example of a question which, with probability $p = 0.5$, generates a graph $G(V, E)$ with a Hamilton cycle. This requires it to check whether G contains such a cycle, which is an NP-complete problem. Using the brute force approach of checking every possible sequence of vertices takes $O(n!)$ time, where $n = |V|$. However, for a small graph with 5 vertices, this is only 120 iterations.

The decision of which programming language to use went in tandem with the choice of graph theory library to support. Recent statistics show Python as being among the most popular languages of 2022/23 when considering metrics such as number of Google searches and developer surveys [98] [87] [84]. It is an easy scripting language to use thanks to its simple syntax, strong community support and large number of libraries, with over 400,000 projects in the Python Package Index [24]. This, paired with the fact that many graph theory libraries are compatible with Python makes it the obvious choice, since the fundamental requirement is that teachers use a tool that is both simple and powerful. Of the Python-compatible libraries, the main contenders seemed to be IGraph [12], Graph-tool [75] and NetworkX [34].

The IGraph library is written in C and supports multiple languages (Python, R, C/C++ and Mathematica). It is well-documented, updated frequently and supports a wide range of features. It also includes functions to convert to and from both NetworkX and graph-tool [40].

The Graph-tool library is written in C++ and is based on the Boost library. It also takes advantage of OpenMP parallelism, making it significantly faster than NetworkX and was also shown by [56] to outperform IGraph on certain benchmarks. However, it takes a long time to compile, with [76] stating that it can take around 100 minutes using clang and around 3 GB of RAM. I also found its documentation to be slightly less welcoming than those of IGraph and NetworkX, since it has far fewer examples or explanations beyond the API reference.

NetworkX is written in Python, which means it is slower than the other two libraries but does not need to be compiled. Since the early years of its development, it has prioritised ease-of-use, being well-documented and integrating with other libraries [34]. Today it is still updated frequently and now has an extensive set of supported algorithms. On top of this, it also has several functions for exporting its graphs in a variety of formats for visualisation by other libraries, including Cytoscape [63].

Ultimately I chose to use NetworkX due to its popularity and ease of use. NetworkX interfaces well with Cytoscape, and IGraph has functions to convert between NetworkX and graph-tool. This means that by supporting NetworkX directly, teachers would still be free to use IGraph or graph-tool and then simply convert to NetworkX for use with *graphquest*.

3.2.2 The *question* Module

The *question* module includes an abstract base class called *Question* and several other abstract classes that inherit from the *Question* class. Each of these child classes represent a particular type of question. Fig. 3.2 shows a subset of the classes, with *QSelectPath* and *QTextInput* as example children.

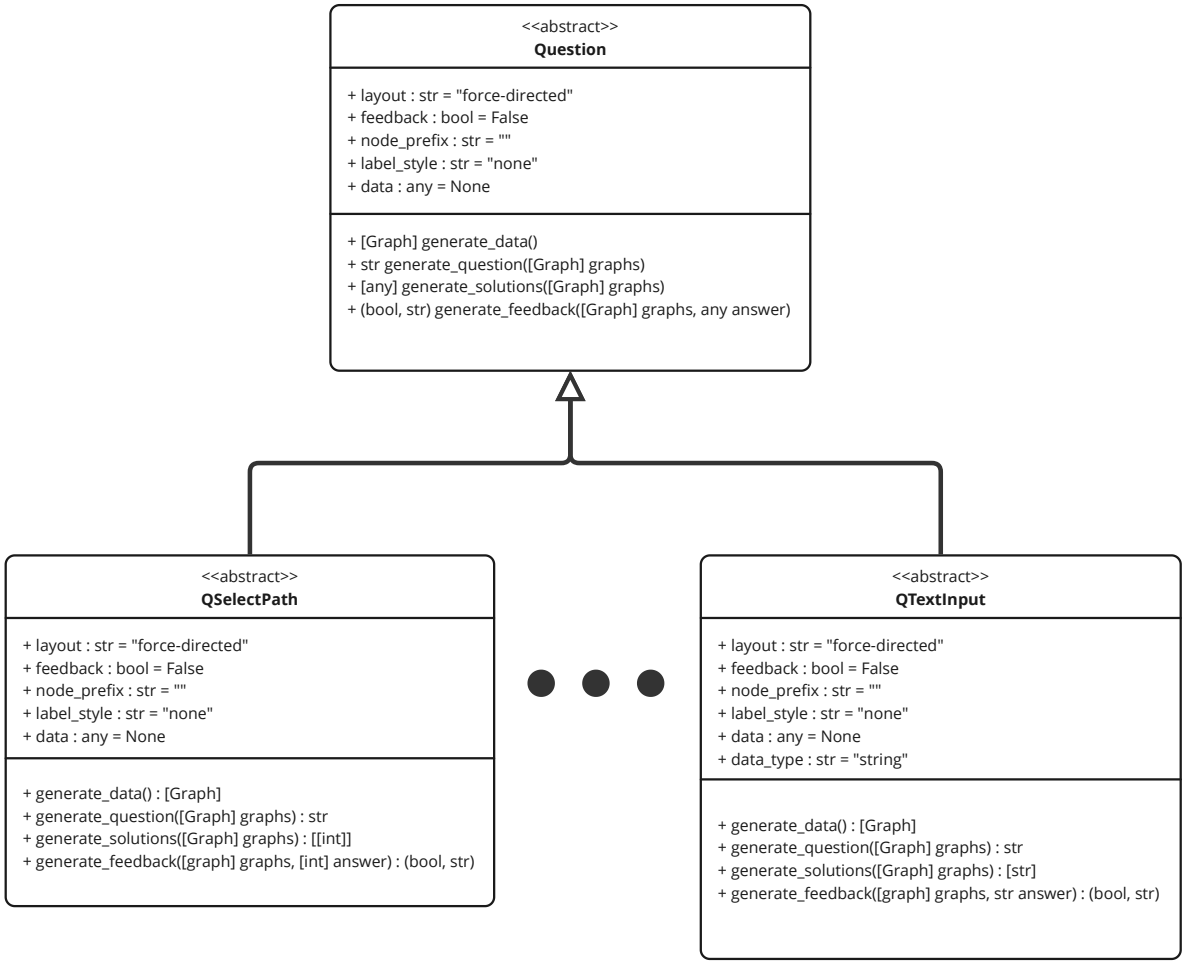


Figure 3.2: UML diagram for the abstract base class Question and two of its child classes, QSelectPath and QTextInput.

All question types follow the format of the *Question* class: their attributes correspond to settings and they have four methods for defining the question's behaviour. These methods are as follows:

- *generate_data()* returns the graph(s) to be displayed.
- *generate_question()* returns a string representation of the question itself.
- *generate_solutions()* returns a list of accepted answers.
- *generate_feedback()* takes in the student's answer as a parameter, then returns a pair saying whether the answer is correct along with feedback as a string.

The teacher only needs to implement one of *generate_solutions()* and *generate_feedback()*. *generate_solutions()* will be used to pre-compute accepted solutions server-side, then the student's answer is verified client-side by checking it against this list. If *generate_feedback()* is implemented, the website will wait for the student to submit their answer, then the *generate_feedback()* processes it server-side. Although this may delay response time in some cases, it does give the opportunity to provide the student with highly tailored feedback.

Each child is a base that represents a particular method of user input. Currently, *graphquest* provides the following five abstract question types:

```python
from graphquest.question import QVertexSet
import networkx as nx
import random

class EvenDegrees(QVertexSet):
    def __init__(self):
        super().__init__(layout="circle")

    def generate_data(self):
        n = random.randint(5, 10)
        G = nx.gnp_random_graph(n, p=0.4)
        return G

    def generate_question(self, graph):
        return "Select all vertices with even degree."

    def generate_solutions(self, graph):
        solution = [n for (n, d) in graph.degree if d % 2 == 0]
        return [solution]

    def generate_feedback(self, graph, answer):
        return ""
```

Figure 3.3: An example of a question class called *EvenDegrees* that extends the *QVertexSet* abstract base class.

- *QTextInput* (the student enters their answer via a text box)
- *QMultipleChoice* (the student selects from multiple options)
- *QVertexSet* (the student selects a set of vertices by clicking on them in the graph)
- *QEdgeSet* (the student selects a set of edges by clicking them in the graph)
- *QSelectPath* (the student selects a path by clicking on vertices in the graph; it will prevent them from making invalid selections)

The only differences between classes from the teacher's perspective are additional settings, whether they can generate multiple graphs or just one, and the data type used to represent students' answers. For the *QTextInput* type, the answer is simply a string. For the *QSelectPath* type, the data type is a list of integers. For a comprehensive guide, see [60].

An example use case is shown in Fig. 3.3. In this instance, the teacher wants to define their own question type, one where the student needs to identify the number of vertices with even degree. They start by importing the *QVertexSet* shell from the *graphquest* package as well as the *networkx* and *random* libraries.

Next, they define their own class called *EvenDegrees* which extends the *QVertexSet* class. Constructor arguments to the parent class are used to configure settings. In this case, they choose the "circle" layout, which will arrange nodes in a clockwise circle.

They then implement the three methods *generate_data()*, *generate_question()* and *generate_solutions()*, leaving *generate_feedback()* as a stub function. Their *generate_data()* method uses the NetworkX library to create a random graph with n nodes and with probability of including an edge as $p = 0.4$ [61]. The actual visualisation and interactivity of this graph are all handled by the website as detailed in section 3.5. The teacher's *generate_question()* method simply returns the wording of the question description. Finally, the *generate_solutions()* method returns a list of accepted solutions. It uses Python's list comprehension together with NetworkX's *Graph.degree* property to create a list of nodes that have even degree [62]. It returns this list as the one and only acceptable solution.

Overall, the *question* module involves a common format with just three functions requiring implementation per question. It should be clear to see that the same generic question shell can be used as the template for countless questions. The generic *QVertexSet* shell used in the "even degrees" example

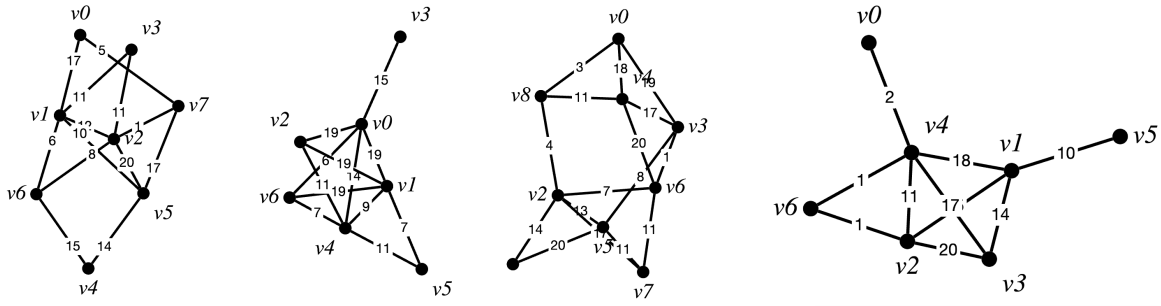


Figure 3.4: Visualisations of graphs generated using NetworkX's *gnp_random_graph()* function [61].

could just as easily be used for a question on selecting the kth vertex visited in a depth-first search, or selecting vertices that make up an independent set. The result strikes a balance between abstraction and power. On the one hand, data entry and visualisation are taken care of such that the teacher only needs to consider the raw data (the graph data structure, the question and the answer). On the other, by writing Python functions the teacher has the full range of features supported by NetworkX to implement the logic of the question, as well as any other Python library they like.

Not only does this solution give the teacher great versatility, but it is also highly extensible. Adding to the *graphquest* module simply involves writing the abstract base class in the *question* module together with its corresponding front-end functionality as explained in section 3.6.

3.2.3 The *graph* Module

At the time of writing, the *graph* module only includes one function, *random_planar_graph()*. I implemented this function because when generating random graphs, I found difficulty in generating visually pleasing weighted graphs (section 3.5). The issues arose when edges crossed, or the angle between incident edges was so small that their weights overlapped. This made the weights impossible to read as can be seen in Fig. 3.4. It is also the same problem that VisuAlgo has in Fig. 2.3 (right).

The ideal solution would be a random planar graph generator. A planar graph is one that may be drawn on a plane such that no edges intersect. Thus my criteria for a generator function are as follows:

- A planar graph;
- With no small angles between incident edges; and
- With vertices spread evenly.

Unfortunately, NetworkX does not include such a function. IGraph and Graph-tool seemed promising with their force-directed implementations, however none of them fully suited my needs.

As an example, IGraph provides an implementation of the Davidson-Harel algorithm introduced in [14]. Their criteria for a "nice" graph is stated as "(1) distributing nodes evenly, (2) making edge-lengths uniform, (3) minimizing edge-crossings, and (4) keeping nodes from coming too close to edges". These criteria align very closely with my requirements and generally the results are very good. However, the algorithm does not completely eliminate edge-crossings and sometimes generates graphs with incident edges that are close together.

Another contender is the Kamada-Kawai algorithm detailed in [54]. It is based on the spring model first introduced by [20] and has similar aims to the Davidson-Harel algorithm for generating visually-pleasing graphs. However, it has the same shortcomings and although [54] mentions an adaptation for weighted graphs, this is deceivingly for adjusting edge lengths to be proportional to their weights, rather than displaying the weights themselves in a pleasing manner.

Ultimately, I found no easily accessible algorithms for generating random planar graphs that suited my criteria, so I decided to implement my own algorithm.

My *random_planar_graph()* function takes in three parameters: n (the number of vertices), *connected* (whether connectedness must be enforced) and s (a sparseness parameter). The algorithm is shown in Algorithm 3.1.

Sorting the edges prioritises connections between nearby vertices. For my implementation, small angles are defined as being less than 15°. The sparseness requirement is met when $s \cdot E_c$ nodes have

Choose n vertex positions randomly in Cartesian space such that no two vertices are within some threshold distance of each other
Let E_c be the edges of the complete graph on these vertices
Sort the edges in ascending order of Euclidean distance
Let $G(V, E)$ be the graph we are constructing, where $V = [n]$, $E = \emptyset$
foreach $e \in E_c$ **do**
    If inserting e maintains planarity and it makes no small angles with incident edges in G, add e to G
    If every node belongs to at least one edge and the sparseness and connectedness requirements are met, **break**
end
return G

Algorithm 3.1: RandomPlanarGraph

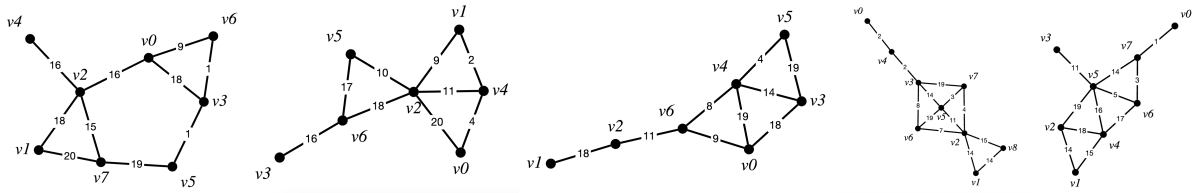


Figure 3.5: Visualisations of graphs that were generated using my *random_planar_graph()* function and then had weights given to their edges.

been processed. A value of s close to 0 gives sparse graphs, whereas a value of 1 will result in the most dense graph possible while satisfying my criteria; these often tend to be triangular lattices with additional pendent paths leaving them. Fig. 3.5 shows five such graphs which were generated sequentially, with n chosen randomly from $\{7, 8, 9\}$, $connected = True$ and $s = 0.3$.

The algorithm does not sample uniformly from the set of planar graphs. However, unlike the available library functions, it does satisfy all my requirements through the vertex positioning and loop invariants. I found it to be good enough at providing a variety of different configurations for quiz questions. In summary, the algorithm fulfils the need for generating weighted graphs that may be visualised clearly.

3.2.4 Hosting and Documentation

I host the *graphquest* package in a GitHub repository (`https://github.com/PaoloMura/graphquest/`) which allows the package to be easily installed with pip. I also include documentation, which can be found at `https://graphquest.readthedocs.io/`. Although I could have kept the documentation self-contained within GitHub via GitHub Pages [51], ReadTheDocs is an alternative that is also free and popular, hosting over 80,000 open source projects [52]. Unlike GitHub Pages, ReadTheDocs is designed specifically for documentation, and through Sphinx (a documentation generator for Python) [85] I was able to automatically generate the API documentation from the Python docstrings. This helped minimise redundant duplication of information, leading to faster development and less chance of inconsistencies. ReadTheDocs also integrates with GitHub via webhooks, so it is automatically updated when changes to the repository are pushed. All in all, this delivers the same benefits as GitHub Pages and more.

3.3 Server

The backend of the Graph Quest website is comprised of a Python server that uses a Flask RESTful API. Its requirements are as follows:

- Serve the static front-end web pages
- Handle teacher requests to upload, modify and delete question scripts and topics.
- Test and run the question scripts.
- Handle student requests for a topic or feedback by constructing and then transferring the data to the student.

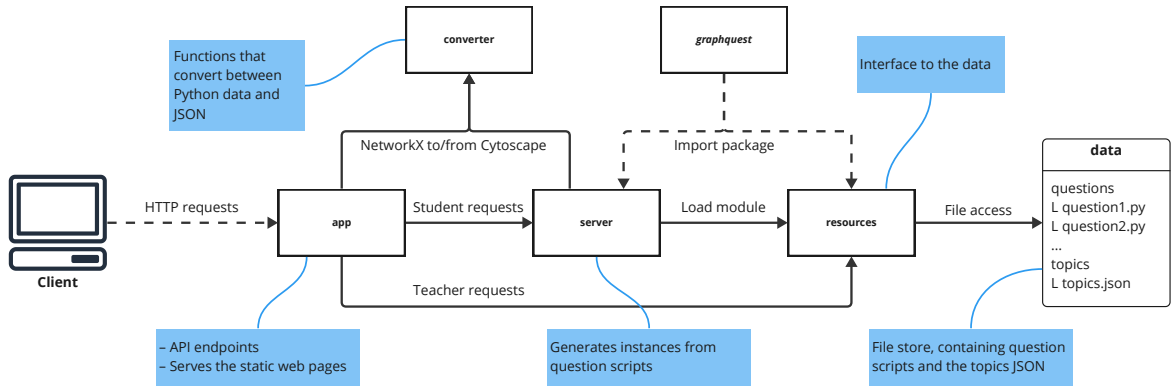


Figure 3.6: Architecture diagram of the server.

The overall structure of the solution is outlined in Fig. 3.6. The client (whether that be the teacher or a student) makes HTTP requests to the *app*. This contains all the API endpoints and serves the static front-end web pages. When a teacher sends requests, these make direct use of the *resources* script to perform CRUD (create, read, update and delete) operations. The *resources* script is the interface to the file store, exposing methods for interacting with the questions and topics. When a student makes a request to the server, the *app* uses the *server* script to generate instances from the question classes, which must in turn be loaded via the *resources* script. Using the question scripts requires the *graphquest* package to be imported (see section 3.2). Finally, the *converter* script is used to convert data from the NetworkX format used by the question scripts to the JSON Cytoscape format used by the frontend (section 3.5.1).

3.3.1 Choosing a Back-end Framework

Further dissecting the requirements of the server revealed the following properties. The teacher only needs to perform simple CRUD operations on the data and the student's requests could be made to be stateless. This is because beyond accessing the web pages, students only perform two requests to the server: download a topic and get feedback to a given answer. If the student has a local reference to which file generated their data along with the data itself, they can send this information to the server, eliminating the need for the server to track state. Another benefit of stateless is that cookies may be avoided, making GDPR compliance simpler. The server must also be able to test and run the question scripts, which were written in Python using the *graphquest* package (section 3.2). Data needs to be encoded and decoded for communication with the client.

I chose to use Python as the server-side scripting language because it fulfilled all of the criteria above. It was obviously the clear choice for testing and running the Python question scripts since a server written in Python would be the most convenient way of interacting with them. Specifically, it has a library called *importlib* [21] which makes it very straightforward to import Python modules as covered in section 3.3.3. On top of this, Python has many other easy-to-use built-in libraries for file I/O including *os* [23] and *json* [22].

Flask and Django are frequently ranked as the most popular Python web frameworks [53] [70]. Flask is a more flexible and lightweight option, whereas Django is a full-stack web framework. This makes Flask more suitable for small to medium sized web apps and RESTful APIs. Since the server is only required to serve static web pages and provide stateless API endpoints, defining a RESTful API using Flask more closely suited the specification.

3.3.2 API Endpoints

The *app* script contains the Flask endpoints. These include the following for teachers:

- *get_content()*
- *access_file()*
- *access_topic()*

The *access_file()* function handles GET, PUT and DELETE requests for the question file resource. In other words, the teacher can download, upload an delete their question files on the server by making requests to this endpoint. The *get_content()* function returns a list of the question files and topics stored on the server. These details are then displayed in tables on the client-side. When the user selects a specific topic from these tables, the *access_topic()* function is used to provide GET, PUT and DELETE functionality for this topic. A more detailed look at how the frontend interacts with these endpoints is covered in section 3.4.

These endpoints are protected, requiring authentication via a JWT access token included in the request header. There are also endpoints for setting, refreshing and removing this token. This is all covered in detail in section 3.7.2.

The following two endpoints are defined for students:

- *access_topic_data()*
- *get_feedback()*

The *access_topic_data()* function is called when the student makes a request for a topic. Once the server has processed this request, a JSON file with the topic metadata and generated question data is returned. The *get_feedback()* function is called when the student submits their answer for server-side verification. Similarly, this returns a small amount of data including the correctness of their solution and any generated feedback. The next sections explain these functions in depth.

To test these endpoints in isolation without having to spin up the frontend every time, I created requests in Postman [81]. Postman is a web service that facilitates the testing of API endpoints through an intuitive interface. Since I only had a handful of functions that needed testing, writing the requests and manually checking output was an adequate way to test during development without creating additional overhead through a complex test framework. I was able to save and reuse the requests in Postman so I could keep testing the endpoints at various stages of development when either the frontend or backend evolved.

3.3.3 Dynamic Class Import

Thanks to Python's *importlib* module [21], running the teacher's question scripts is made simple. The *importlib* module provides an implementation of Python's *import* command, allowing modules to be imported dynamically at runtime, rather than statically at the start of the program. This is exactly what the *resources* script does in order to access a specified question file. A major benefit of importing classes in this way is that the teacher has the freedom to name their classes anything they like and store multiple classes in the same file.

When generating question instances for a topic, the *server* script instantiates the requested question class from the imported script, and the class's methods are called to generate the data for the question. This data includes the wording of the question as a string and a list of NetworkX graphs to be displayed. If the class's *feedback* attribute is set to *False* (the default), a list of accepted solutions is also generated. The datatype of each accepted solution depends on the particular question type.

If the *feedback* attribute is set to *True*, then the student will make a request on submitting their answer, and a similar process takes place for generating feedback. This also involves dynamically importing the corresponding question file, instantiating the class and calling its *generate_feedback()* method, passing in the student's answer and the graphs used in their question instance as parameters.

3.3.4 NetworkX to Cytoscape Conversion

The graphs created by the teacher's question classes are of the NetworkX *Graph* type. However, the Cytoscape graph visualisation library used in the frontend has its own implementation for displaying graphs. Thankfully, Cytoscape accepts JSON data as a way of initialising its graphs, and NetworkX provides two functions *cytoscape_data()* and *cytoscape_graph()* for converting to/from this JSON format respectively [63].

Rather than use these functions directly, I wrapped them in my own *converter* script so as to fix their problems and layer any of my own additions on top. There were only a few minor problems that needed resolving: the Cytoscape graph required edges to have IDs but the *cytoscape_data()* function was not providing these. The node IDs associated with edges were being stored as strings in Cytoscape, but NetworkX required them to be integers. The use of a *converter* script allowed me to resolve these issues in a self-contained wrapper.

```
{
    "d83hf": {
        "name": "Paths and Cycles",
        "description": "This is a quiz for topic 1",
        "settings": {
            "linear": true,
            "feedback": "each",
            "random_order": false
        },
        "questions": [
            {
                "file": "cycles.py",
                "class": "EulerWalk"
            },
            {
                "file": "cycles.py",
                "class": "HamiltonCycle"
            }
        ]
    }
}
```

Figure 3.7: JSON structure of the topics file, which is stored on the server.

3.3.5 JSON Schema

I made careful consideration when deciding on the schema to use for backend-frontend data transfer. Ultimately, I settled on using a JSON structure for both storing the topic data and transmitting all generated data to the student when they request the topic. The reason for using JSON is because it is a simple, lightweight solution that is language independent, making it ideal for web applications. Not only do Python (section 3.3.1) and JavaScript (section 3.4.1) have built-in support for JSON serialisation, but the graphs are already in JSON format for transmission (section 3.3.4).

Fig. 3.7 shows an example section of the *topics.json* file used to store topic data. The topic code is a unique randomly generated five-character string. This is used as the key to index each individual topic. In the example, this code is "d83hf". The topic includes four properties: name, description, settings and questions. The name is simply the name of the topic that is displayed to teachers and students. The description is a text segment for the teacher to add their own notes to the topic. The settings are topic-wide and apply to the overall quiz. Currently there are three settings: linear (whether progression should be sequential or allow students to move between questions), feedback (whether to provide feedback after each question, at the end or not at all) and random order (whether questions should be shuffled). It is straightforward to add new settings and future versions may include examples such as time limit or number of allowed attempts. Finally, questions contains a list of objects, each of which has a reference to the file and the question class within that file to be used.

Not only does the topic code index the topics, but it is distributed by the teacher to their students and is used as part of the URL to access the topic via a web page. I felt that having a single index was the simplest way to reference the topics.

When the student requests a topic via this code, the corresponding data is retrieved from the *topics.json* file. The *server* then iterates over each question, loading and generating its data as described in section 3.3.3. At this stage, it packs the question data into a Python dictionary as shown in Fig. 3.8.

The file is the name of the file that the question belongs to. The class is the name of the question class. The type is the name of the base class (e.g. *QTextInput*) from which the question derives. The settings are the class's attributes and the remaining items are the data generated by the class's methods.

When transmitting the generated data to the student, rather than assemble a new structure, a copy of the actual topic entry (Fig. 3.7) is sent to the student, with each question object replaced by the generated data (Fig. 3.8).

Overall, JSON is a natural choice due to its simplicity and compatibility. My solution prioritises reuse, whether that be the topic code for indexing and distribution, or the stored topic JSON making up the base of the response data to the student.

```
{
    "file": q_file,
    "class": q_class,
    "type": q_type,
    "settings": q_sett,
    "description": q_descr,
    "graphs": q_graphs,
    "solutions": q_sols
}
```

Figure 3.8: Python dictionary for question data.

3.4 Website Design

3.4.1 Choice of Front-end Framework

My frontend consists of a React application written in JavaScript that primarily uses components from the Bootstrap package, along with some additional components from Material UI. The reason I chose to use JavaScript as the front-end language was primarily because of its popularity. According to [101], JavaScript "is used as client-side programming language by 98.6% of all the websites". This means there is ample support and examples for JavaScript when designing a front-end.

The results of a 2022 developer survey are revealed in [70], and place React.js as the most popular front-end framework with 42.62% of respondents stating they had used it extensively in the past year and would use it again. I personally had previous experience with React both in a hackathon and job interview. In each case, I only had a few hours to learn and use the framework, but it proved to be incredibly intuitive and I managed to quickly achieve successful outcomes in both cases. Finally, my chosen graph visualisation library (section 3.5.1) had support for React via a JavaScript package [80], making it the clear choice.

The popularity and modular architecture of React gives rise to one of its greatest advantages: its support for a wide range of packages. Bootstrap [1] is one such package that includes a selection of commonly used components such as buttons, grid layouts and modals. It is easy to use and has a simple, consistent style, allowing for a clean and minimalist design. For cases where Bootstrap lacked features such as icons, I turned to Material UI [86]. This is another package, which is far more extensive than Bootstrap. It is based on Google's Material Design guidelines [31], which introduces familiarity and consistency to users. Using these packages enabled me to focus on the application architecture and unique problems rather than designing standard website components such as grids.

3.4.2 UX Design

The frontend is made up of five web pages as shown in Fig. 3.9. The Home page is the main entry point to the website, with two branches: one for the teacher and one for the student. These initial pages are shown in Fig. 3.10.

The *TeacherLogin* page is used for authorisation as detailed in section 3.7.2. The *ChooseTopic* page allows students to enter a given topic code to access the corresponding topic (see section 3.3.5).

The *Teacher* page provides the teacher with a graphical interface to the resources on the server (Fig. 3.11). This consists of a list of question files and a list of topics, with buttons and drop-down menus to trigger HTTP GET, PUT, POST and DELETE requests to the server.

The *Student* page is the main feature of the website, and is where the student accesses a topic (Fig. 3.13). It includes a progress bar on the top, graph display on the left and question description on the right. On mobile, the graph is displayed above the question.

I kept to a minimalist design in line with the original aims. Only the key information is presented on each page and this is done in a consistent style through the use of Bootstrap components such as the buttons as can be seen in Fig. 3.10. I stuck to a simple colour scheme, primarily using blue and a light shade of grey throughout the website.

I also included several reusable components throughout. One of the most notable examples of this is the table row used throughout the teacher side of the website. This can be seen clearly in Fig. 3.11, which shows table rows being used for the list of question files and the list of topics on the Teacher page (left), and the list of questions within a specific topic, displayed within the *TopicModal* component (right).

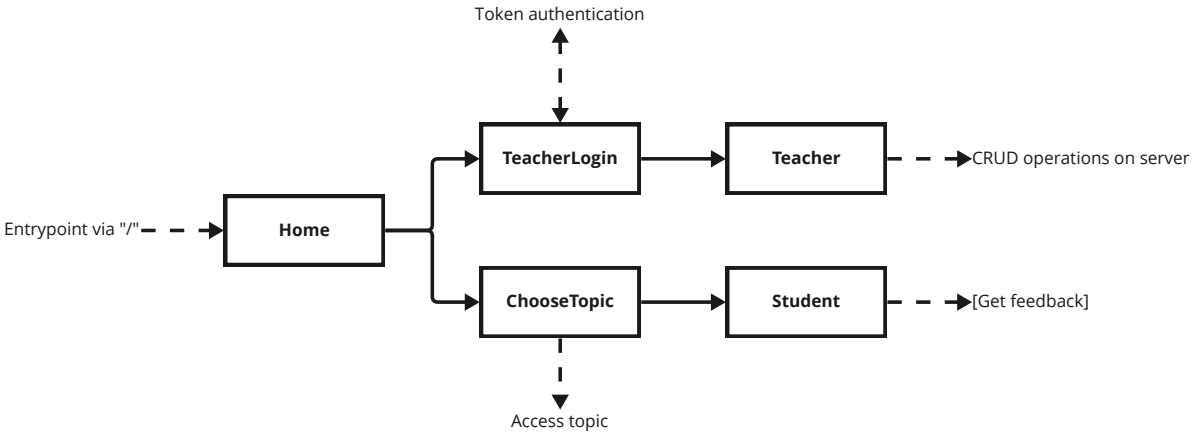


Figure 3.9: Frontend web pages.

Figure 3.10: From left to right: *TeacherLogin* page, Home page and *ChooseTopic* page.

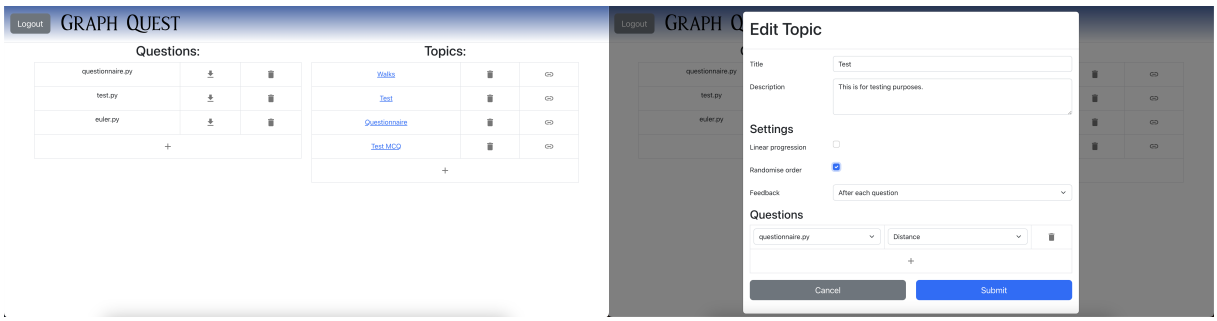


Figure 3.11: Left: *Teacher* page. Right: *TopicModal*.

```
<TblText
    myKey
    text
    onClick
/>
```

```
<TblButton
    myKey
    icon
    onClick
/>
```

```
<TblDropdown
    myKey
    selected
    choices
    onChange
/>
```

Figure 3.12: The *TableRow* components: *TblText*, *TblButton* and *TblDropdown*.

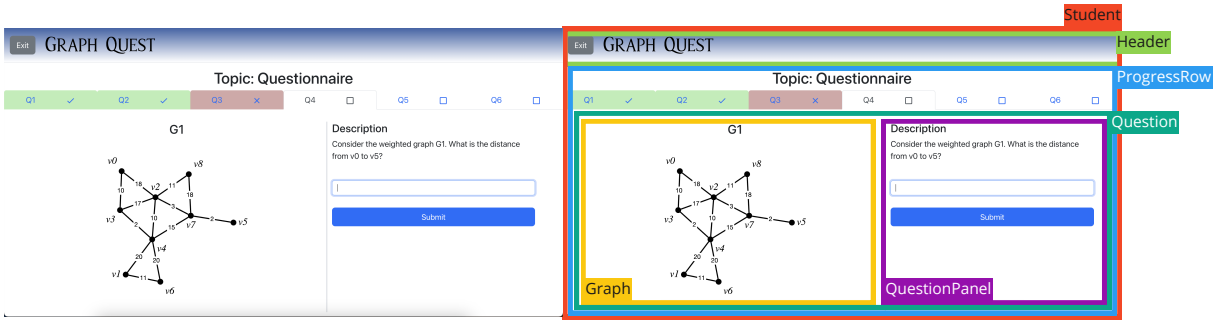


Figure 3.13: Left: the *Student* page. Right: the same page annotated with the subcomponents that make up that page.

Fig. 3.12 shows how the *TableRow* components are used in the JSX code. There are just three types of row items: *TblText*, *TblButton* and *TblDropdown*. *TblText* simply displays text, which is optionally clickable when the *onClick* property is supplied. The *TblButton* component is used for clickable icons, where the *icon* property determines which Material UI icon to display. Finally, the *TblDropdown* component is used for dropdown menus and is built on top of Bootstrap's *Form.Select* component. These three components are modular, allowing tables to be constructed using any number of them in any order. Overall, this helped achieve both a consistent theme from the user's perspective and a reusable structure from a development standpoint. It is very straightforward to add new column types, which is in line with the project requirement of extensibility.

3.4.3 Student Page

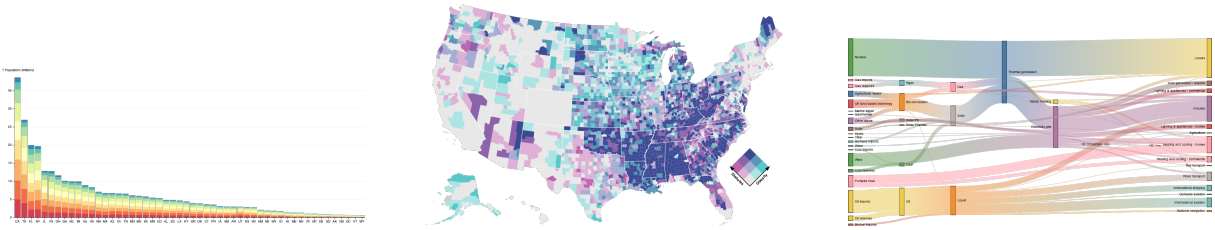
The student page is the core feature of the Graph Quest website. This is where the topic and its questions are rendered along with their associated graphs. Fig. 3.13 shows an example topic that consists of six questions. The *Student* component is itself split into nested sub-components, each of which corresponds to regions within the page. This is shown in the right-hand image in Fig. 3.13.

The *Header* component is common to all pages and includes a title designed with the font from [64]. The topic is first accessed via a HTTP request from the *Student* component, which then passes the necessary data down to its sub-components as properties.

The main body of the student page is made up of the *ProgressRow* component, which stores the state associated with the student's progress through the topic. This state includes their answer for each question paired with the question's status (i.e., unanswered, correct or incorrect). It displays the name of the topic along with a progress bar, which is a Bootstrap *Tabs* component, coloured according to each question's state.

Each Bootstrap *Tab* contains a *Question* component. For reasons explained in section 3.5.5, it is imperative that the *Question* component does not store any state. Instead, it simply contains the *Graph* and *QuestionPanel* components, and selects which answer type to use within the question panel.

The *Graph* component is a wrapper for Cytoscape's React component and is covered in depth in section 3.5. The *QuestionPanel* displays the question description and handles the logic of the question type. It is explained in further detail in section 3.6.



(a) Bar chart [6]. (b) Map [8]. (c) Sankey Diagram [5].

Figure 3.14: Three example graphics created using D3.

3.5 Graph Visualisation

3.5.1 Choosing a Graph Visualisation Library

The very first decision I made in this project was the library to use for visualising the graphs. It was essential to make a fixed commitment on this from the beginning, since the rest of the application would have to integrate with it. Ultimately, I chose to use Cytoscape because of its popularity, ease-of-use and out-of-the-box features. These fulfilled most of my requirements, which are detailed as follows.

Requirements

The library had to be popular and well-maintained so that it would be supported for the coming years. It should also be well-documented, have good community support and implement as much of the required functionality out-of-the-box, making development easier and allowing me to focus more on application integration, rather than lower-level rendering. Speed was not an important factor since practice questions tend to only deal with small graphs of far less than 100 nodes. However, the solution still had to be responsive.

I determined the specific required features of the library from the original project requirements, as well as the features supported by the best solutions discussed in chapter 2 in order to make my solution comparable at the very least. These requirements are as follows:

1. Display nodes and edges (the elements)
2. Labels for both elements (including within the node circle)
3. Style for both elements (including colour and size)
4. Arrows on the edges
5. Tree layout
6. Force-directed layout
7. Interactive (user should be able to drag elements, and click on them to change their style and properties)

Achieving requirements 1 to 5 would help meet the needs of most graph variants (simple graphs, trees, flow networks, etc.) that are covered by higher education courses as discussed in chapter 2.

The reason I included a force-directed layout (see section 3.5.4) as a requirement was because this is one of the standout features of D3 Graph theory (section 2.4). Similarly, interactive graphs are one of the main advantages of both D3 Graph Theory and VisuAlgo (section 2.3). Hence the inclusion of these as requirements would allow my solution to match if not surpass the benefits of those existing sites.

D3

D3 (Data Driven Documents) is a JavaScript library with a collection of modules for visualising data [7]. It is a generic, low-level framework that focuses on providing base functionality for any form of data visualisation, from bar charts to maps and other diagrams such as those shown in Fig. 3.14.

It is the most popular of the libraries covered in this section (see the summary below) with great documentation and community support. However, its low-level nature would require building and fine-tuning all the required features. Although there are plenty of existing open-source graph theory projects

Table 3.1: Summary of the features of graph visualisation libraries. Data correct as of 28th April 2023.

	D3	**Sigma.js**	**Graphology**	**G6**	**Cytoscape**
GitHub stars	105k	10.6k	919	9.7k	9.1k
GitHub forks	23.3k	1.6k	62	1.2k	1.6k
Last commit	1st Apr 23	4th Nov 22	3rd Mar 23	23rd Apr 23	27th Apr 23
Documentation	Very good	Poor	Ok	Very good	Good
Required features	7/7	6/7	6/7	7/7	7/7

built on top of D3 that could serve as a starting point [73] [102] [9], integrating them, resolving bugs or extending their functionality would necessitate dealing with this low-level framework.

Sigma.js

Sigma.js is a JavaScript library that specialises in visualisations for graph theory specifically [79]. Since 2021, its core data structure and algorithms have been split into Graphology [77], while Sigma.js retains the rendering and interactivity components. It renders its graphs using WebGL, which is faster than the Canvas or SVG-based rendering used by the other solutions discussed here. This makes it the ideal option for visualising large networks, though as mentioned above, this is not required for Graph Quest. It has support for many of the required features include styling, interactivity and force-directed layouts, but lacks built-in hierarchical tree layout algorithms. The Graphology API is well-documented but there is little further support beyond a few examples [78].

G6

G6 is a library by AntV, which is a data visualisation arm of Alibaba Group, with full support for graph theory [3]. The library has a rich set of features, from common graph theory algorithms to popular layout algorithms including trees and force-directed options. The documentation is thorough and involves many examples. G6 can also integrate with React via its sister library, Graphin. It is likely one of the most powerful and user-friendly visualisation options for graph theory, striking a good balance between supported features and abstraction of the low-level visualisation. The main disadvantage of G6 is that most of its documentation and community support is written in simplified Chinese, so for a non-Chinese reader, running into problems midway during development could potentially be very difficult to resolve.

Cytoscape

Cytoscape is another JavaScript library built specifically for graph theory visualisation as introduced in [27]. A 2023 update in [26] states that it is now "in the top 0.01% of software packages by popularity measured by number of user stars" on GitHub. One of its main claims is usability for both developers and end users, which it provides through comprehensive documentation and examples, integration with React (see section 3.4.1) and plenty of extensions for various layout algorithms. This aligned naturally with my own objective to develop an extensible solution.

Summary

Table 3.1 summarises some key information about the graph visualisation libraries. GitHub stars and forks are used as an indicator of popularity, with D3 taking the lead by a significant margin. All libraries are well-maintained with frequent and recent commits.

Although D3 has the capacity to support all the required features, it is very low-level and does not directly support graph theory. Sigma.js and Graphology are not as well documented as the others and although the G6 documentation is comprehensive, much of it is written in Chinese.

Cytoscape appeared to be the most developer-friendly thanks to good documentation, examples and community support, while striking a balance between providing all the required features and using a high enough level of abstraction. Ultimately, I committed to using this library through react-cytoscapejs, a React package with a component that renders a Cytoscape graph. See section 3.4.1 for a discussion on React.

```
<CytoscapeComponent
    id={id}
    elements={elements}
    layout={layout}
    stylesheet={stylesheet}
    cy={cy => {...}}
/>
```

Figure 3.15: Use of *CytoscapeComponent*.

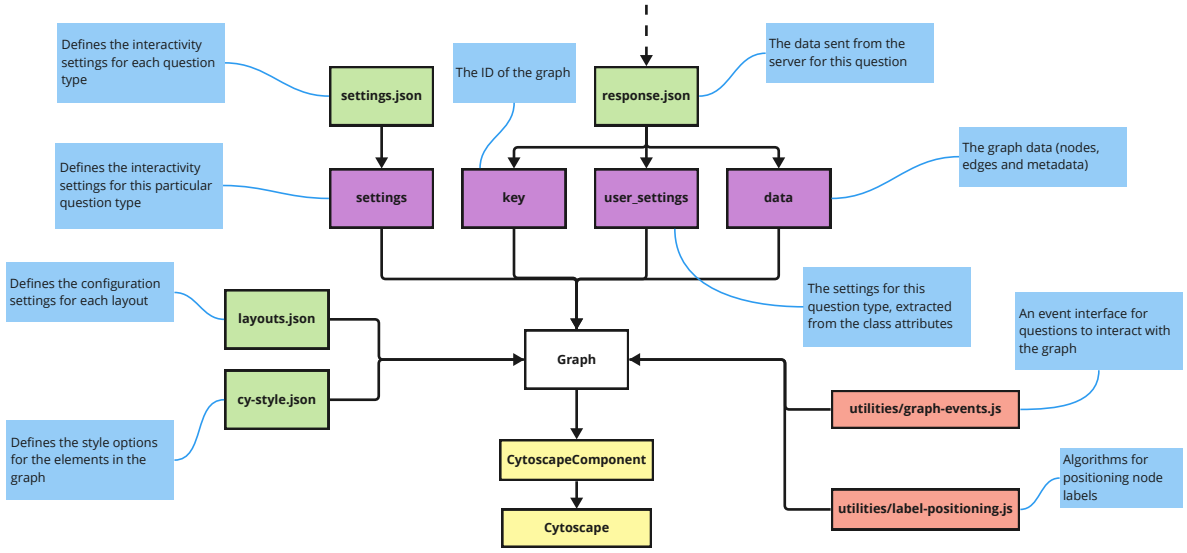


Figure 3.16: The files, data and interactions involved with the *Graph* component. Green for JSON files, purple for React properties, red for JavaScript functions, yellow for Cytoscape and blue for comments.

3.5.2 Overview of the *Graph* Component

The react-cytoscapejs package includes a component called *CytoscapeComponent*, which enables use of the Cytoscape library within a React front-end application [80]. This component is applied as shown in Fig. 3.15. The *id* attribute is that of the HTML *div* element used by this component. The *elements* attribute takes in the data to be used for the graph, which includes the nodes, edges and metadata (such as whether the graph is directed). The *layout* attribute defines the configuration of the layout algorithm to be used on the graph. Cytoscape layouts are extensions, which anyone can define themselves as a mapping from elements to positions. However, they do provide plenty of built-in ones already including a *random* layout, *circle* layout and *breadthfirst* layout (see section 3.5.4). The *stylesheet* attribute takes in JSON to style the elements of the graph. As with CSS, the stylesheet includes selectors for elements in the graph and defines their style (such as size or colour). Finally, the *cy* attribute is a function that gives access to a reference to the underlying Cytoscape API.

Rather than setting the elements and layout via the *CytoscapeComponent*, I wrapped this component in my own *Graph* component and used the *cy* reference for more control. An overview of the data involved with my *Graph* component is displayed in Fig. 3.16. The graph takes inputs as React properties from its parent component (section 3.4.3), which include the graph data and the settings generated by the teacher's question as covered in section 3.2.2. It also takes in configuration data defined in JSON files. These include the interactivity settings for the question type (e.g., whether a student is allowed to click on nodes); the layout settings (e.g. the specific configuration to be used for circle layouts); and the stylesheet (which defines the style for different types of elements in the graph). The *Graph* component also makes use of two utility files: *graph-events* (for interaction with question logic) and *label-positioning* (for layout of node labels). Finally, *Graph* returns a *CytoscapeComponent*, which itself is a wrapper for the underlying Cytoscape JavaScript library.

The sequence of logic within *Graph* is as follows:

```
{
    "selector": "node.ring",
    "style": {
        "background-color": "#fff",
        "border-style": "solid",
        "border-width": 2,
        "width": 30,
        "height": 30,
        "text-valign": "center",
        "text-halign": "center"
    }
}
```

Figure 3.17: An entry in the *cy-style.json* stylesheet, which sets the style for any nodes in the graph that have the "ring" class.

1. Set general style and layout options.
2. Import the graph data.
3. Set the initial node positions.
4. Trigger the layout algorithm.
5. Apply the interactivity settings for this question type.
6. Set the node label positions.
7. Apply styling to graph elements.
8. Register events.
9. Start listening for events.

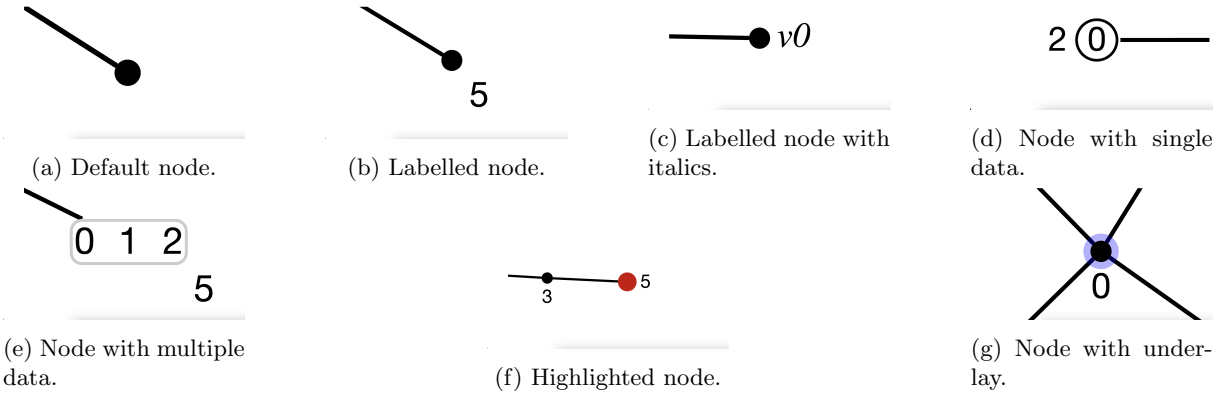
3.5.3 Styling

The stylesheet that *react-cytoscapejs* uses is JSON but follows CSS conventions. An example entry in this stylesheet is shown in Fig. 3.17. In this example, the selector indicates that the style should apply to any node in the graph that has the "ring" class. The style includes CSS-like properties, which in this case set any matching node to be a 30x30 px circle with a white background, black border and centred label. It is very easy to modify which classes graph elements have at runtime using Cytoscape's functions such as `eles.addClass()` and `eles.removeClass()` [13].

Fig. 3.18 and Fig. 3.19 visually summarise all the styles that I have written support for in the *cy-style.json* stylesheet. The *Graph* component applies the corresponding classes to elements in the graph based on criteria from different sources. Some styles, such as node labels and underlay, are determined by teacher-specified settings when they set the class attributes in their question scripts (section 3.2.2). Element highlighting is caused by the student interacting with the graph in certain questions (section 3.5.5). Other attributes are inferred from the graph data. For example, edges are given arrows if the graph metadata has the "directed" attribute, and each edge with a "weight" attribute is given a label with its value.

Node data is slightly more involved because Cytoscape does not allow elements to have multiple labels. However, it does have a feature called "parenting" whereby nodes may "contain" child nodes. This is the approach I took when representing nodes with data such that the node itself becomes a "parent" and contains "child" nodes, each of which stores an item of data but with normal node styles hidden. As with edge weights, the data is inferred from a "data" attribute given to nodes. If the node has no data attribute, the default styling is used. Otherwise the ring (for a single data item) or rounded rectangle (for multiple data items) is used along with node parenting.

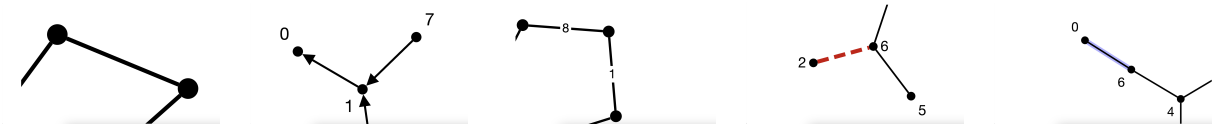
In order to meet my original aims, I made sure to follow a user-centric approach so that the results would be accessible to a variety of students. In the case of element highlighting, I not only changed the colour, but also the size of nodes and line style (dashed vs. solid) for edges. This was to cater to



(a) Default node. (b) Labelled node. (c) Labelled node with italics. (d) Node with single data.

(e) Node with multiple data. (f) Highlighted node. (g) Node with underlay.

Figure 3.18: Node styles defined in the *cy-style.json* stylesheet (screenshots from Graph Quest).



(a) Default edge. (b) Directed edge. (c) Weighted edge. (d) Highlighted. (e) Underlay.

Figure 3.19: Edge styles defined in the *cy-style.json* stylesheet (screenshots from Graph Quest).

colour-blind students who would need an alternative way of distinguishing highlighted elements other than colour. I also adapted my stylesheet in response to user testing (chapter 4.1), making such changes as increasing the default size of nodes for visually impaired students and adding support for element underlay as a way for the teacher to provide visual feedback within the graph.

Another one of my aims was to be able to support a good spread of the syllabus content for higher education courses. With the selection of style options I provide, Graph Quest should be able to support almost all major graph variants taught by the courses introduced in chapter 2. To be more explicit, it can support rendering of:

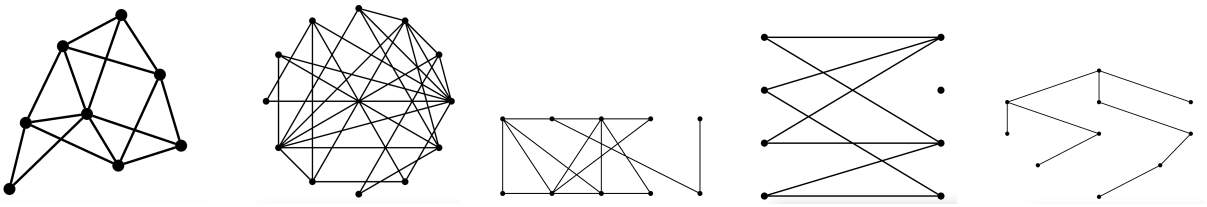
- Simple graphs (default style),
- Weighted graphs,
- Directed graphs,
- Spanning trees (by highlighting/underlaying the tree on the graph)
- Trees (optionally with node data),
- Flow networks (by using node data together with directed edges),
- B-trees (by using node data),

The only topic covered in those courses that is not fully supported by Graph Quest is graph colouring. However, thanks to the extensible nature of the project, it would be trivial to implement in a future version. This would be done by adding a new style in the stylesheet, which uses the node's "colour" attribute to set the colour of the node. The teacher could write their question class *generate_data()* function to assign this attribute to its nodes, with no changes required in the backend or *graphquest* module.

Although B-trees are supported, Graph Quest cannot yet render them in a hierarchical fashion (see section 3.5.4).

3.5.4 Layout

Graph Quest supports five layout options, which the teacher specifies via a constructor argument in their question class (section 3.2.2). Each one uses a Cytoscape layout extension with specific configuration settings defined in the *layouts.json* file.



(a) Force-directed. (b) Circle. (c) Grid. (d) Bipartite. (e) Tree.

Figure 3.20: The five supported graph layouts (screenshots from Graph Quest).

```
"tree": {
    "name": "breadthfirst",
    "fit": true,
    "padding": 30,
    "nodeDimensionsIncludeLabels": true
  }
```

Figure 3.21: The "tree" entry in the *layouts.json* file.

The "force-directed" layout is the default. A force-directed layout is one that uses a physics simulation-based algorithm to position nodes and edges in an aesthetically pleasing manner. One of the first force-directed algorithms was proposed in [89] and then [20] later introduced a spring-based layout whereby nodes are modelled as rings and edges are modelled as springs between these rings. The nodes have attractive forces applied, while edges have tension forces determined by Hooke's Law. Around 100 iterations is generally sufficient for the simulation to reach an equilibrium such that edges have approximately equal lengths and edge crossings are minimised.

The force-directed layout used in Graph Quest is from the Cola extension for Cytoscape. Its JavaScript library is introduced in [19], which claims that it achieves higher quality and more stable layouts than d3-force, the force-directed algorithm used by D3 [7]. The downside is that Cola does not scale well to large graphs, though it is suitable for graphs with under 100 nodes, which is acceptable for the Graph Quest use case. I found the force-directed layout to be a good general solution, but it can lead to "hairball" visualisations for larger graphs. For these situations, I added the "circle" layout, which is a built-in Cytoscape layout.

Another built-in layout is their "grid" algorithm as shown in Fig. 3.20c. I initially used this to support bipartite graph displays, where nodes are assigned to one of two columns depending on their "bipartite" attribute (whose value is either 0 or 1). This is set automatically when generating a bipartite graph with NetworkX, but the teacher is also free to assign it via their code. Although the current implementation requires this attribute in order to assign a node to a column, a future version could run an algorithm to infer the column assignments just from their edge connections.

Finally the tree layout uses the built-in "breadthfirst" layout algorithm, which assigns node positions based on a breadth-first traversal of the graph. More specifically, the root node is positioned at the top, then rank 1 nodes are positioned in the space below, with rank 2 nodes below them, and so on.

Supporting multiple layout options helped achieve the project aim of providing tools for a range of higher education graph theory topics. The force-directed layout is a good general-purpose solution (and the only one used by D3 Graph Theory [71]). However, since trees and matchings in bipartite graphs are amongst the most commonly taught topics in graph theory courses (2), it made sense to include specific layouts for them. Another objective is achieved through the mere choice of Cytoscape itself, since its layouts are extensions by nature, fundamentally aligning with my aim for extensibility. Adding support for another layout to Graph Quest is as simple as importing the layout or designing its algorithm yourself, and then including a configuration entry for it in the *layouts.json* file. The tree layout entry is shown for reference in Fig. 3.21. It specifies the name of the layout to be used (in this case the built-in "breadthfirst" algorithm) and the layout configuration settings (fit the graph to the screen, add padding around the edge of the graph and include node labels when calculating spacing). After including this entry, no changes were required on either the server or *graphquest* package; the teacher can now simply set their *layout* constructor argument to "tree" and the "breadthfirst" layout will be used.

On the whole, Cytoscape made layout options very straightforward. However, problems arose that I had to resolve myself in order for the layouts to work as intended.

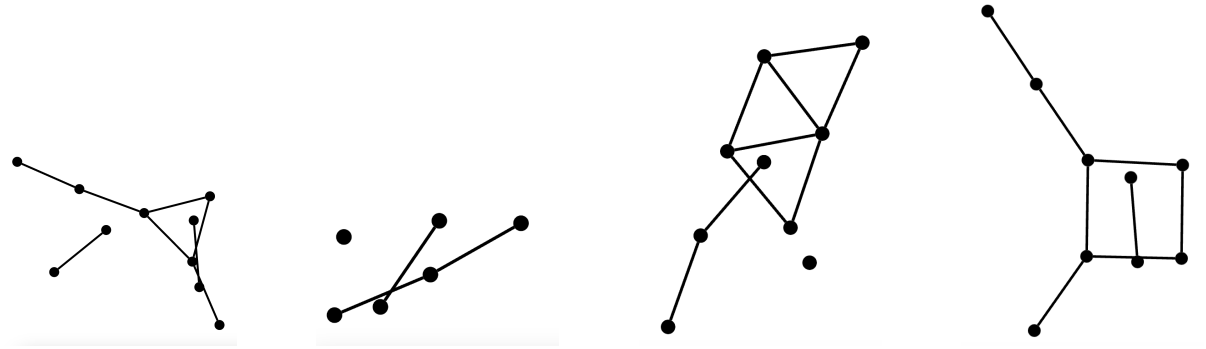


Figure 3.22: Examples of force-directed layouts where multiple components are rendered on top of each other (screenshots from Graph Quest).

(a) 2 components. (b) 3 components. (c) 4 components. (d) 5 components.

Figure 3.23: Initial component positions for graphs with different numbers of components (screenshots from Graph Quest).

Problem 1 - Imperfect Force-directed Algorithm

There were two main issues with Cola's force-directed algorithm. The first was that if the graph was made up of multiple components, they would often get tangled on top of each other as shown in Fig. 3.22.

To resolve this, I made sure to set the Cola layout configuration settings such as "handledisconnected", "avoidoverlap" and a higher value for "nodespacing" in order to try to spread nodes apart more and prevent component overlap. When these failed to solve the issue, I resorted to setting initial node positions such that all nodes belonging to the same component are initialised to the same position and different components are allocated different starting positions around the screen. The starting positions are chosen using the formulas

$$x = k \cdot \cos\left(\frac{i}{n_c} \cdot 2\pi\right)$$

for the x-coordinate and

$$y = k \cdot \sin\left(\frac{i}{n_c} \cdot 2\pi\right)$$

for the y-coordinate, where k is a constant, i is the component index and n_c is the number of components. This arranges starting positions evenly-distanced around a circle as can be seen in Fig. 3.23. The result was far more effective, with components successfully repelling each other without getting trapped in each other's space.

Another issue with the force-directed layout was that weighted graphs often rendered poorly due to edge crossings obscuring the weight labels. Since this is not a bug with the layout itself, but rather an inevitability caused by the fundamental data of the graph, I opted to solve this by implementing my own generator function. This is covered in detail in section 3.2.3.

Problem 2 - Issues with the Breadthfirst Algorithm

The breadthfirst algorithm often produced poor results since it allows multiple roots, and its default method of determining these roots is by choosing the highest degree nodes. The results are visualised in Fig. 3.24. I resolved this issue with the following methods:

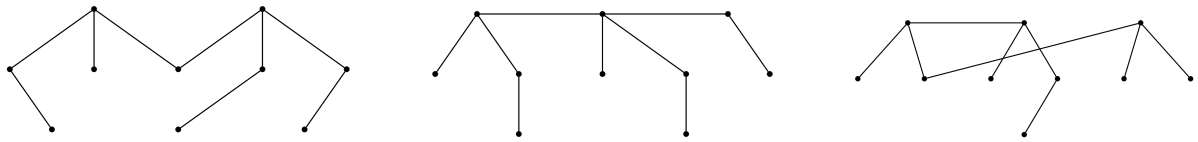


Figure 3.24: Examples of the breadthfirst layout choosing multiple nodes as roots (screenshots from Graph Quest).

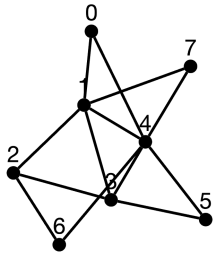


Figure 3.25: A force-directed layout with default positioning for node labels.

1. Allow the teacher to specify the root of the tree through a *roots* constructor argument in their question class.

2. If no roots are specified by the teacher, set an optimal root.

I defined "optimal" as being the midpoint of the longest path such that the tree is displayed with minimum layers. According to [90], finding the longest path in a graph is an NP-hard problem and cannot be solved in polynomial time. However, they present a linear time solution for trees. My own algorithm is based on this (and adapted to use NetworkX functions) as outlined in Algorithm 3.2.

Input: a graph $G(V, E)$.
1 Choose a random r_1 from V.
2 Let G_1 be the directed graph created from a breadth-first traversal on G starting at r_1.
3 Let p_1 be the longest path in G_1 from r_1, and r_2 be the final node in this path.
4 Let G_2 be the directed graph created from a breadth-first traversal on G starting at r_2.
5 Let p_2 be the longest path in G_2 from r_2.
Result: the midpoint of p_2.

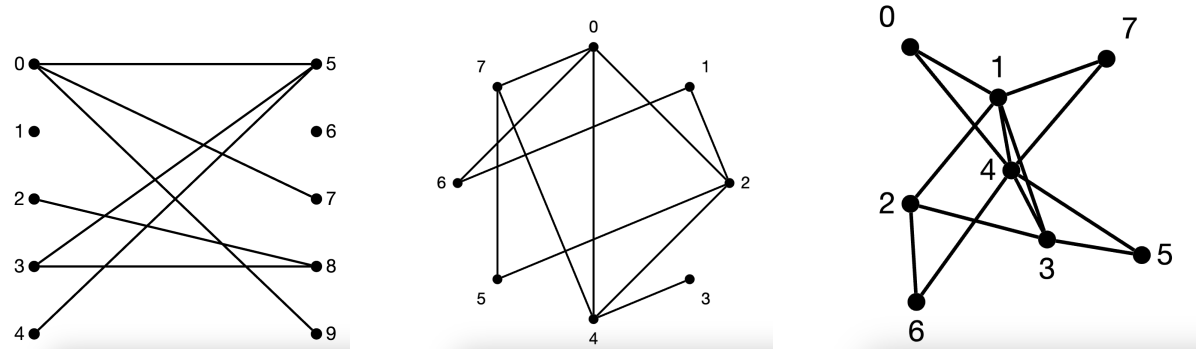
Algorithm 3.2: OptimalRoot

Ultimately, this is a linear time solution and contributed much better results than Cytoscape's default breadthfirst layout, such as the tree shown in Fig. 3.20e. Another issue with the breadthfirst algorithm was that it was not sophisticated enough to be able to handle "parent" nodes (as in the Cytoscape parent nodes discussed in section 3.5.3). For now, I have opted to use the force-directed layout to display trees whose nodes have multiple data items, regardless of whether the teacher specified that a tree layout should be used. A future version of Graph Quest could explore extending the breadthfirst layout to handle these kinds of structures.

Problem 3 - Absence of Node Label Positioning Algorithms

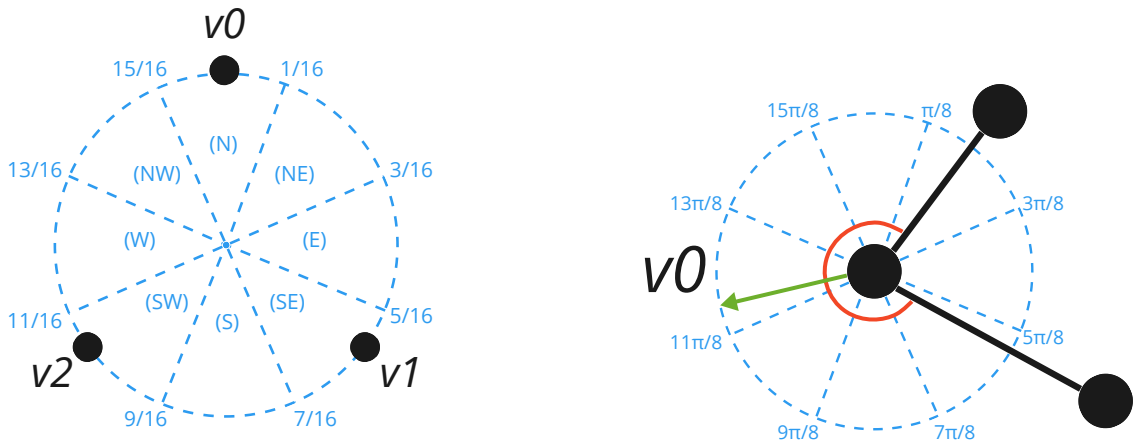
Cytoscape does not include built-in functionality for positioning node labels optimally. The default is for node labels to appear above the node, which leads to labels being obscured by edges as shown in Fig. 3.25. The only option for customising label positioning is through setting the node's style such that its label appears in one of eight compass positions around the node.

To resolve this issue, I implemented three node label positioning algorithms for the different layouts: bipartite, circle and a generalised solution, the results for which are displayed in Fig. 3.26. All these algorithms are implemented in the *src/components/utilities/label-positioning.js* script. The bipartite algorithm is by far the simplest: for nodes in the left-hand column, their labels are allocated to the "west" position while nodes in the right-hand column have labels allocated to the "east" position.



(a) Bipartite. (b) Circle. (c) Force-directed.

Figure 3.26: Examples of layouts with different label positioning algorithms applied (screenshots from Graph Quest).



(a) In the circle layout, a label's position is determined by the node's proportional location around the circle.

(b) In the general case, a label's position is determined by the midpoint of the largest angular gap between neighbouring edges.

Figure 3.27: Diagrams to help visualise the circle (left) and generalised (right) node label positioning algorithms.

The circle layout algorithm is best described through Fig. 3.27a. The Cytoscape circle layout positions nodes evenly spaced around a circle, with the first node always appearing at the top. To determine the optimal positions of each node's label, I first calculate i/n where i is the index of the node and n is the total number of nodes. This proportion is then used to determine which compass "zone" the node fits. For example, v_1 in Fig. 3.27a should have its label displayed in the south-east position, since $\frac{5}{16} \leq \frac{i}{n} < \frac{7}{16}$ where $i = 1$ and $n = 3$.

The final algorithm I wrote was a generalised solution, used in force-directed, grid and tree layouts. For nodes with no neighbours, the label position defaults to "North" of the node. For nodes with a single neighbouring edge, the label position is the compass zone opposite this edge. When the node has two or more neighbouring edges (such as the example in Fig. 3.27b), we iterate over these edges to find the largest angular gap between any two of them. This is represented as the red line in the diagram. Then, we find the midpoint of this angle (as illustrated by the green arrow). The compass zone that contains this midpoint is the label's new position.

The results were highly satisfying, with labels successfully being allocated to the optimum positions (i.e., the largest available space around a node) as shown in Fig. 3.26c.

```
"QSelectPath": {
    "selectNodes": true,
    "selectEdges": false,
    "boxSelection": false
}
```

Figure 3.28: The QSelectPath entry in the *settings.json* file.

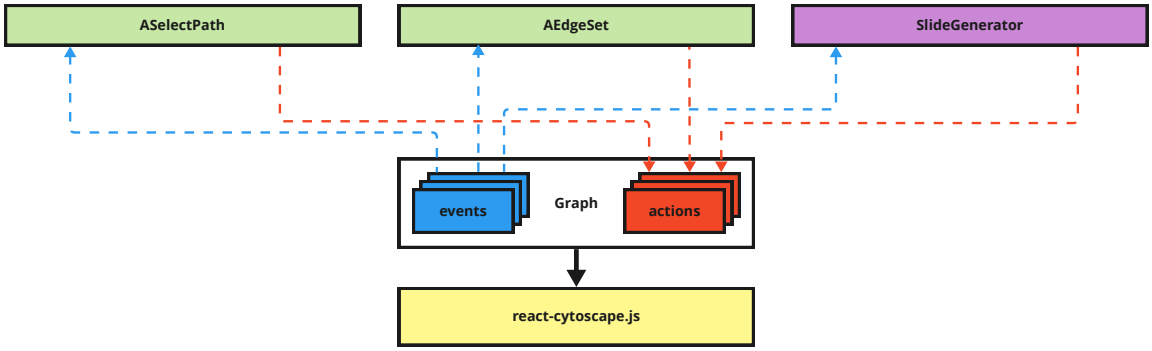


Figure 3.29: High-level diagram of the *Graph* component's modular event system interface. The *SlideGenerator* component is a potential future addition, not currently supported by Graph Quest.

3.5.5 Interactivity

Cytoscape includes various interactivity features, allowing the user to click on elements in the graph, select or move them, pan and zoom the viewport and detect a range of user input events. The *Graph* component exposes a subset of this functionality through two approaches.

The first is a *settings.json* file, which defines the general interactivity settings for each question type. An example entry is shown in Fig. 3.28, whose settings are self-explanatory. This JSON file is loaded in during initialisation and the interactivity settings are toggled according to the question type.

The second approach to handling graph interactivity is an event system, which allows other components to listen to events that the graph triggers, as well as to order actions in the graph. This is represented in Fig. 3.29. It is an elegant solution, since it not only solves a major complication to do with React rendering, but opens up possibilities that perfectly align with my project aim of extensibility.

The problem with React rendering in Graph Quest was as follows. A React application is made up of JavaScript functions which represent the components of a website. Fig. 3.30 illustrates how the *Question* component is a parent with two child components, *Graph* (to display the graph) and *QuestionPanel* (the text instructions and answer submit area, covered in detail in section 3.6). Each component can store "state" that is associated with it, and may pass this state down to to its children as "properties" (a.k.a. "props"). When state is updated in a component, this triggers a re-render of the component and any children who receive this state via props. This works well for React components, but unfortunately, the way we interact with Cytoscape from the *Graph* component is directly via its (non-React) *cy* reference (see section 3.5.2). Re-rendering *Graph* returns a new instance of the Cytoscape component, which is a particular issue for force-directed layouts since the layout algorithm would have to be fired again every render, often converging to a different final state. It is very jarring for the graph layout to re-organise itself with every tap of a node. The intuitive, React-like approach would be to store the answer state (in this case an integer list of selected nodes in the path) in the parent *Question* component and pass the state down as props to both *Graph* (to highlight the path in the graph) and *QuestionPanel* (to update the answer box). But this would mean that every time the student clicks on a node in the graph, the state would be updated in *Question*, triggering a re-render that resets the Cytoscape graph in the *Graph* child.

After careful consideration, the solution I settled on was to move the answer state down to be stored exclusively in the *QuestionPanel* component, and setup an event system interface on the *Graph*, which uses the observer design pattern. The *Graph* component now publishes a set of input events (or simply "events") and subscribes to a set of action events ("actions"). Now, when the answer state changes, it

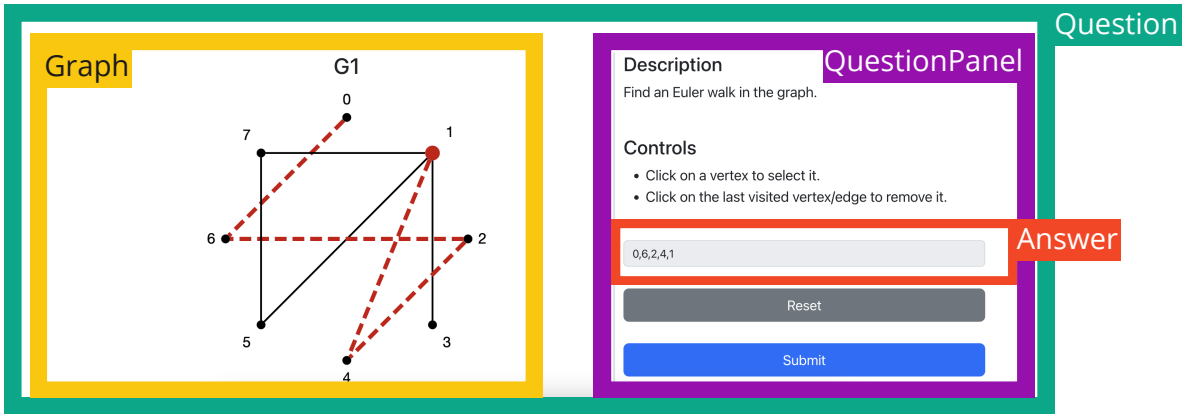


Figure 3.30: Annotated screenshot from a QSelectPath Graph Quest question showing the React components that make up the question.

```
const params = {
      vertex: myVertex,
      type: 'colour',
      highlight: True
    }
triggerGraphAction('highlightVertex', params, graphKey)
```

Figure 3.31: An example of triggering the highlightVertex action, which will cause the graph to colour the *myVertex* node.

only triggers a re-render in the *QuestionPanel* component, yet the *Graph* and *QuestionPanel* are still able to "communicate" via this event system. The events the *Graph* currently supports are the following:

- *tap_bg* (student tapped on the background)
- *tap_node* (student tapped on a node)
- *tap_edge* (student tapped on an edge)
- *cxttap_bg* (student right-clicked on the background)
- *cxttap_node* (student right-clicked on a node)
- *cxttap_edge* (student right-clicked on an edge)
- *box_end* (student box-selected elements)

When it triggers an event, it also supplies additional information such as the node ID when the student taps a node for example. The action requests it currently responds to are:

- *highlightVertex* (highlight a given vertex in the graph)
- *highlightEdge* (highlight a given edge in the graph)

When a component triggers an action, it should also include parameters. In the case of *highlightVertex*, these are the node ID, the type (colour or underlay), whether to highlight or un-highlight and the graph key. The graph key is an identifier which is needed to distinguish between graphs when multiple graphs are displayed side-by-side in the question.

The specification for triggering actions and events is set out succinctly in the *utilities/graph-events.js* file. An example for triggering the highlight vertex event is shown in Fig. 3.31.

These actions and events are what underpin the interactivity provided by the *QVertexSet*, *QEdgeSet* and *QSelectPath* questions, contributing towards the project aims by providing students with a more

intuitive, easy-to-use input method than the alternative forms of answer entry, such as those provided by Numbas (section 2.2).

All-in-all this solution not only resolves the React rendering problem, but actually contributes perfectly towards my project aim of extensibility. It is very straightforward to add support for new actions and events, with the only change needed being the definition of the handler function (in the case of actions) or the trigger call (in the case of events). This enables the *Graph* to be completely decoupled from the *Answer* components that interact with it; neither needs to know anything about the other, aside from the actions and events supported by the interface.

A further benefit of decoupling the *Graph* component from the logic of individual questions is that the entire project could easily be extended further to support more than just quiz questions. A suggested example shown in Fig. 3.29 is a fictitious *SlideGenerator* component for generating lecture slides. The component could step through an algorithm, triggering *Graph* actions and saving a screenshot of the graph every step of the algorithm before combining the results into a PDF presentation. This would require no further modification to *Graph*; the new *SlideGenerator* component would simply interact with its highlighting capabilities through the existing event interface.

3.6 Question Modelling

As previously illustrated in Fig. 3.30, the *Question* component contains both the *Graph* visualisation on the left as well as the *QuestionPanel* on the right, which handles the logic of the question. When the *Question* first mounts, it checks the *graphquest* question type being used, and then dynamically imports the functions it needs from the appropriate answer script. There are five answer scripts, each corresponding to one of the question types defined in the *graphquest* package:

- *ATextInput.js*
- *AMultipleChoice.js*
- *AVertexSet.js*
- *AEdgeSet.js*
- *ASelectPath.js*

Every file must export the functions shown in Fig. 3.32. *initialAnswer()* returns the initial answer, which is used at the start and when the student presses the "Reset" button. *controls()* may return a list of controls for the question, which *QuestionPanel* displays as a bulleted list. The *onReset()* function may be used to reset any augmentation done to the graph (e.g. by removing highlighting) when the "Reset" button is pressed.

When the student presses the "Submit" button, *validate()* is called, which should return an error message as a string (then displayed as shown in Fig. 3.33 [right]), or the empty string if the answer is valid. If the answer is valid, then it is verified for correctness. If the teacher specified the "feedback" setting, then an HTTP request is made for server-side processing of the answer (section 3.3.2). Otherwise the *verify()* method is called, which returns a boolean indicating whether the answer is correct by checking it against the list of acceptable solutions.

The final two functions that the answer script must implement are *Answer* and *DisabledAnswer*, which are a pair of React components displayed when the question is in progress or after submission respectively. For most questions, they display a (often disabled) text box, or radio/check boxes in the case of the *AMultipleChoice.js* file.

These components also handle the main logic of the answer, which involves taking in user input and modifying the answer state accordingly. In the case of *ATextInput.js*, this is as simple as setting the answer state to be whatever the user entered in the text box. The *ASelectPath.js* logic is far more involved, since it registers to listen to the graph's *tap_node* and *tap_edge* events, determines if the event it is a valid selection based on the graph data, and if so, triggers the graph's *highlightVertex* and *highlightEdge* actions appropriately. The graph-answer event interface is covered in detail in section 3.5.5.

It should be clear to see that this client-side question framework is highly modular, such that the logic associated with each question type may be swapped with one another, thus helping to fulfil the project extensibility requirement. The complete workflow for creating a new base question type is simple:

1. Define the Python base class called "QNewQuestion" in *graphquest.question*. In particular, consider:

```
export function initialAnswer (question) { return '' }

export function controls (question) { return [] }

export function onReset (question, answer) {}

export function validate (question, answer) { return '' }

export function verify (question, answer) { return false }

export function Answer ({ question, answer, progress, setAnswer, setError }) {
    return (
        <div>Hello World!</div>
    )
}

export function DisabledAnswer ({ question, answer, progress }) {
    return (
        <div>Hello World!</div>
    )
}
```

Figure 3.32: The functions to be implemented by an answer script.

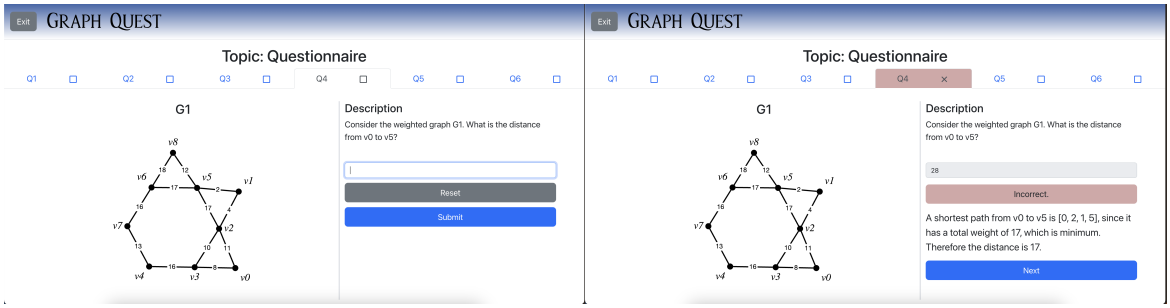


Figure 3.33: Left: an example QTextInput question when first loaded. Right: the same question after submission.

(a) What is the data type of the answer?

(b) Are there any additional settings unique to this question type?

(c) Can the question support multiple graphs or just one?

2. Create the corresponding JavaScript answer script called "ANewQuestion.js" in the *src/components/answers/* directory.

   (a) Implement each of the seven functions outlined in Fig. 3.32.

3.7 Deployment and Security

3.7.1 Need for Security

The Graph Quest application involves hosting a Python Flask web server, which allows teachers to log in and upload Python scripts. When students then access quiz topics from the website, these scripts are run on the server. Allowing users to upload and run untrusted Python code on a server raises a large security concern.

One of the risks is privilege escalation, leading to such threats as arbitrary code execution on the host machine, or access and corruption to (sensitive) data. Similarly, the Python scripts may include malware such as viruses that autonomously corrupt the server's data or affect its functionality. Several other potential issues also revolve around disrupting the server's activity. For example, the scripts could be used in a denial of service (DoS) attack on the server, or abuse resources such as CPU or RAM, making the server unresponsive. This may not even necessarily be due to malicious intent; there is a risk of teachers uploading code which unintentionally enter an infinite loop for example.

I have only implemented minimal layers of security for Graph Quest, including token-based authentication, sandboxing via containerisation and limiting functionality in production environments when compared to a localhosted development environment. I would not recommend running Graph Quest unrestricted in a production environment without adding further layers of security on top of these. I detail both my own efforts as well as suggested improvements in the subsequent sections.

3.7.2 Authentication and Authorisation

Authentication is used to verify the identity of a user, while authorisation determines their rights to access a resource or perform an action. In the case of Graph Quest, the teacher is currently authenticated against hard-coded login details and JSON web token (JWT) access tokens, and then authorised to access the teacher API endpoints within the Flask application. The hard-coded login details are only used in local development testing (see section 3.7.4) as a simple stand-in for future improvements. I opted to use token-based authentication over basic HTTP or session-based methods because basic HTTP requests send the unencrypted login credentials in every request header, which lacks confidentially and requires HTTPS in order to be secure. Session-based authentication on the other hand uses a session ID after initial login, so credentials do not need to be sent on every request. However, it stores cookies on the client's web browser and is stateful, since the session needs to be stored on the server, which does not pair well with Graph Quest's existing RESTful Flask API. Token-based authentication is stateless, since the server does not need to store any data. Instead, it issues the client with a token after initial login, which they save in local storage and include in the headers of subsequent requests. The server can then verify the token's validity by checking its signature against a secret key, rather than storing data for each client session. The particular library I chose to use is called flask-jwt-extended [30]. It balances security with ease-of-use, since it provides many built-in features such as automatic token refresh and token expiration.

There are several ways that future versions of Graph Quest could improve on this authentication further. Firstly, the login details are hard-coded, which is not secure and only permits a single teacher entity to login. A secure identification database of some sort could be included to allow multiple teacher accounts to exist with hashed passwords. Alternatively, a more sophisticated form of authentication and authorisation could be used such as single sign-on (SSO) authentication. In the case of SSO, a service like Azure Active Directory (Azure AD) [59] would enable the teacher to sign in through Microsoft's own authentication service rather than some Graph Quest database. If they already have login credentials associated with their educational institution on Azure AD, this could help authorise only users with valid emails (such as those with *.ac.uk* top level domains) via Conditional Access [58]. Once authenticated, a JWT token could be assigned as before.

3.7.3 Sandboxing

Sandboxing creates a safe, isolated environment in which to run untrusted code, decoupled from the surrounding infrastructure. Several solutions exist specifically for sandboxing Python code. PyPy has a sandbox implementation for Python that uses Linux's seccomp technique to provide OS-level security and restrict the inputs and outputs for a given process [82]. However, it may not work so well for Python 3 (which Graph Quest uses) and has not been as well-supported since 2020. Restricted Python is another alternative for CPython, which adds restrictions on which functionality a given Python script is allowed access to [25]. With more time, I would be tempted to include this as an additional layer of sandboxing for when running code from individual teacher-provided scripts.

Instead, I opted to containerise the server using Docker [49]. A Docker container creates a completely isolated environment from the host operating system and can have configuration settings applied to limit the container's resources such as CPU and RAM, which may help protect against some of the risks outlined in section 3.7.1. Containerising the entire server makes efforts to protect the host system, but does not protect the server itself from the untrusted question scripts, leaving it vulnerable to DoS attacks or resource abuse.

Although containerisation improves the security, it is not foolproof and may still be susceptible to an attack if misconfigured or vulnerabilities in the server are exploited. Additional layers of sandboxing could be applied in future versions of Graph Quest, such as virtual machines as provided by the Qubes architecture [83]. As well as sandboxing the server, the question scripts themselves could be sandboxed individually. This could be done by having the scripts run in a dedicated second Docker container whose sole purpose it to run the scripts and communicate results with the main server. With the Qubes OS, this could be made more secure by running the two parts of the server in two separate virtualised "qubes"; with the main server given a higher level of access such as for network resources and the question script processing server limited to minimal required resources only. Another alternative for separation is for the main server to spin up a new container or VM every time a question script is to be run. This is likely more secure (since it limits the time window an attacker has to control the system), but at the cost of additional overhead. A similar idea would be to use a serverless cloud service such as AWS Lambda [41], which can run Python code in an isolated environment with restrictions on resources, and only spins up resources for the script as and when the Lambda function is triggered, saving on cost. Although Lambda has limitations on execution time and resources, this should not have an impact on Graph Quest, since the question scripts are lightweight.

In summary, I containerise the entire server with Docker to help protect the underlying system. However, it would be sensible for future versions of Graph Quest to protect the server itself by splitting the question script processing into its own module. This module would either use virtualisation, containerisation, or serverless architecture, with each option having a different trade-off between security and speed/overhead. Personally, I would lean towards AWS Lambda due to its convenience and low-overhead, but a more security-conscious provider (particularly one running it on their own dedicated server rather than via cloud services) may opt for virtualisation.

3.7.4 Hosting

Full deployment instructions are set out in the server's README. The server code has two "modes": development and production. Which one is run depends on the value of an environment variable called *GRAPH_VIS_ENV*. In development mode, the app runs as normal, with the React frontend being served on localhost:3000 and Flask server on localhost:5000. To get around the CORS (cross-origin resource sharing) security feature [39], which restricts sites from making requests to a site served from a different origin, I setup a proxy to forward requests from React to Flask.

When the environment variable is set to run in production mode, the teacher side of the app is made inaccessible by simply not including their API endpoints. This is because I felt my security provisions were inadequate during development. The front-end React application is built locally into static files, which are then served from the same Flask server as the back-end API endpoints.

As mentioned in section 3.7.3, I containerised the server with Docker. Not only did this improve security, but it also made deployment far more streamlined. To take this one step further, I setup GitHub Actions [50] to automatically build and launch the container on AWS LightSail [2]. Since I was already using a GitHub repository for version control, using GitHub Actions was the obvious method for adding a continuous deployment pipeline. On top of this, it provided a simpler way for building the Docker image on a Linux Ubuntu machine, rather than my M1 MacBook (which would require additional steps to ensure compatibility), so that the container could run on AWS LightSail or other Linux servers.

AWS Lightsail was very easy to setup and free with AWS's Free Tier. It has a limited number of features and control, but is highly suited to small-scale web applications, making it ideal for Graph Quest.

Overall, I was able to automate deployment, making it easy to spin the application up and down for user testing (chapter 4.1) in a matter of minutes. Although I chose to use LightSail because it was cheap and convenient for user testing, the Docker image should allow the website to be deployed quickly on other cloud services or a dedicated server.

Chapter 4

Critical Evaluation

4.1 Evaluation via User Study

4.1.1 Overview

I designed a user study in order to test and improve the student experience with the Graph Quest website. The study involved interviewing a small set of volunteer participants and taking them through an interactive demonstration of the website. I recorded their anonymised responses and analysed them, forming aggregated feedback and a list of actionable requirements. The study took place at the start of the 2023 Easter break, which gave me enough time afterwards to make development changes that addressed most of the issues raised.

4.1.2 Participants

I recruited participants by posting a message on a group chat open to all third and fourth year computer science students undertaking individual projects at the University of Bristol. The message included a participant information sheet describing the nature of the interviews.

Five fourth-year students volunteered to take part. All of them had taken the second-year Algorithms II module, which has strong overlaps with graph theory (see chapter 2). Since it had been approximately two years since taking the module, most of the participants had reduced recollection of the syllabus content. Four of the five students took part in-person in a quiet room on University campus and one of them joined remotely via Microsoft Teams. Each student participated individually.

4.1.3 Design

Prior to the interviews, I designed a strict format and schedule in order for the interviews to run smoothly and consistently. This was comprised of two parts: questions that covered a general overview of the student's experience with graph theory; and a practical, interactive demonstration of the website, where the students would attempt topic questions under realistic conditions while being prompted to give feedback. Both parts were tailored towards the Algorithms II module since I only sourced participants that had previously taken it. The first part was comprised of the following five questions:

1. How would you rate your own ability with respect to graph theory?
2. Have you had any exposure to graph theory outside of the Algorithms II module?
3. Which definitions or graph algorithms did you find the most challenging?
4. What online tools and resources did you use to learn graph theory?
    - What are their benefits?
    - What are their shortcomings?
5. Which type of questions did you find hardest in the Blackboard quizzes?

The aim of questions 1 and 2 was to determine the skillset and background of the participants. Questions 3 to 5 were designed to better understand the needs of students when learning graph theory. I deliberately worded the questions to be open-ended and non-leading, so as to encourage a wide variety of responses that I myself may not have considered.

The second part of the study required the student to use the Graph Quest website under "realistic" conditions. To simulate a realistic experience, I setup hosting via AWS Lightsail as detailed in section 3.7.4, which also helped contribute to testing the hosting on top of user experience. I also designed a mock topic, tailored specifically to emulate the kinds of questions a student would expect to face on the Algorithms II course. I wrote question classes (as a teacher would) for six question types, covering a broad range of the Algorithms II syllabus and using a variety of the base question types that Graph Quest supports. These questions are summarised as follows, and the original scripts are included within the *server/data/questions/* directory of Graph Quest. The format is *QuestionName(BaseClass) - description.*

- MinSpanTree(QMultipleChoice) - What is the weight of a minimum spanning tree in the graph?
- EulerWalk(QSelectPath) - Find an Euler walk in the graph.
- DFS(QVertexSet) - Which vertex will be explored sixth in a depth-first search of the graph?
- Distance(QTextInput) - What is the distance from v0 to v5?
- VertexCover(QMultipleChoice) - Which of the following statements are true? A is a vertex cover; B is a vertex cover; C is an independent set; D is an independent set
- MaximumMatching(QTextInput) - What is the size of the maximum matching in the graph?

The accompanying interview questions for this topic were as follows:

1. Can you describe the first things that catch your attention/your first impression of the website?
2. The student will then be asked to work through the questions one at a time. For each question, the following will be asked.
   (a) Complete the question, talking through your process as you go.
   (b) Is there anything that could be done to improve the question?
3. Discuss your thoughts on each of the following:
   (a) The clarity of the instructions.
   (b) The design of the interface.
   (c) The usefulness of the questions.
4. Do you have any final thoughts or comments on the application?

Question 1 was scheduled to be asked immediately on the student navigating to the provided URL to get their honest first impressions of the site. Question 2 was deliberately kept short to enable the students to work through the topic questions without interrupting their flow, so as to best simulate a real quiz. Questions 3 and 4 were scheduled to be asked on completion of the quiz as a way of prompting any general thoughts on the website. As with the first part of the interview, I tried to design the questions to be open-ended and non-leading to encourage unbiased, unrestricted responses.

4.1.4 Procedure

Participants had been given access to the participant information sheet prior to the interviews. I began each interview by summarising the contents of this sheet again, including an overview of what the interview would entail; data processing (I would record the interview but would not be collecting any personally identifying information); and the student's freedom to withdraw from the study at any point without having to give consent. The participant then signed a consent form and after giving verbal consent that they were happy to begin, I started recording the interview. I used Microsoft Teams to record a transcription and screen recording for all participants, whether in-person or online. Video cameras were turned off beforehand.

Each interview lasted between approximately 20 and 30 minutes. Even before beginning the questions listed in section 4.1.3, I reassured the students that their responses were anonymised, they were not being tested on graph theory and I was only interested in their feedback on the design and usability of the website. I went through each question from the first part in order with the interviewee. This was recorded, but otherwise involved little use of technology, with it simply being a face to face (or online) discussion. The recording meant I did not have to take notes, which allowed the conversation to flow naturally.

In the second part, I sent the URL to the online student via the Microsoft Teams chat box and asked them to open it in the browser window that they were sharing. For the in-person interviews, the student used the web browser on my laptop. Since most of the students had become slightly rusty on the Algorithms II content, I freely offered to remind them of definitions and technical terms throughout. I did not feel that this created any bias in their feedback of their user experience. Generally speaking, I tried not to interrupt the student's thought process, but occasionally prompted them to develop their explanations further to better understand their opinions. Some students tended to think through the questions in their head more than others, so I was more active in probing them with questions to get a sense for how they were finding the website.

When finishing the interview, I gave them the opportunity to ask any questions and provide general feedback. I also asked once again for verbal consent and to sign the user consent form for me to store and anonymise their recorded responses.

I conducted a post-processing stage after the interviews, which involved downloading the transcriptions and removing the unnecessary lines of timestamp metadata. I also downloaded the screen recordings and used video editing software to crop out the users' avatars to maintain anonymity. After downloading and anonymising these data, I deleted them from Microsoft Streams so that only the anonymised copies were stored on my local MacBook as promised in the participant information sheet.

Finally, I analysed the results by extracting the key information from each transcription, using the video recordings to help remind myself of what they were referring to.

4.1.5 Results

I have included summaries of the responses for the individual questions of the first part of the interviews below. For the second part, I have grouped together responses into four categories: the question design, the graph visualisation, the general design of the website and the specific questions used in the quiz. I was able to address most of the issues raised in the constructive criticism (see section 4.1.6) and adapt the code accordingly. Those points that I was unable to resolve in time are marked in italics.

Q1. How would you rate your own ability with respect to graph theory?

The most common response was 5/10.

Q2. Have you had any exposure to graph theory outside of the Algorithms II module?

Some respondents had been introduced to graph theory in decision maths at school, or revisited the topic as part of other university modules.

Q3. Which definitions or graph algorithms did you find the most challenging?

Most respondents could not remember, but a few said bipartite graphs and maximum matchings.

Q4. What online tools and resources do you currently use to learn graph theory?

Most respondents mentioned use of the Blackboard quizzes that were included as part of the Algorithms II module. Other answers were lecture slides, lecture videos and a few said they had watched YouTube videos of algorithm visualisations or explanations. The criticisms of these resources were:

- A lack of material to practice from.
- Questions only assess the basics.
- Videos were not an interactive way of learning.
- The resources had a lack of good feedback.

Q5. Which type of questions did you find hardest in the Blackboard quizzes?

The responses mentioned questions that involved long answers, multi-step calculations or proofs.

The Question Design

Positive feedback:

- It is nice to have instant feedback.
- It is amazing that it generates new question instances each time.

Constructive criticisms:

- The feedback should be more prominent.
- There could be more visual indications of progress (such as red-green colouring in the question panel as well as the progress bar).
- *The wording of multiple choice questions could be clickable as well as the check boxes.*
- *Technical words within the question descriptions could have tooltip definitions when you hover over them.*
- The reset button was not obvious enough.
- *The final submit button did not work for two of the students.*

The Graph Visualisation

Positive feedback:

- The interactivity features were fluid, intuitive and responsive.
- They particularly liked the interactivity of the *QSelectPath* question type.
- They preferred the visual layout of graphs generated using my *random_planar_graph()* function than those generated by NetworkX's *gnp_random_graph()*.
- They also thought the bipartite layout was nice.

Constructive criticism:

- There were sometimes overlapping nodes and edges in the force-directed layout.
- When the user tapped on the background, a grey circle would appear to indicate the tap, which respondents said confused them, as they thought this meant the background was interactive in some way.
- The graph was rendered too small for some respondents to see the elements clearly; it should fill the space more.
- For the depth-first search question, some students who got the question wrong suggested it would be helpful if the feedback included a highlighted path in the graph to better visualise the correct solution.
- *This could be taken further to have an animated traversal of the correct solution.*
- *The user should be able to interact with the graph for the purposes of working out the solution (as one would sketch or trace things out on paper).*
- For QVertexSet and QEdgeSet questions, if the selection limit is 1, clicking on another vertex/edge should transfer selection to it, rather than having to first deselect the currently selected element.
- *For the minimum spanning tree question, tapping on edges could automatically add their weights to a tally to avoid the student having to total the sum themselves.*
- *Submitting the answer would cause force-directed graphs to re-adjust slightly.*

The General Website Design

Positive feedback:

- The tab navigation in the progress bar was popular.
- The colour coding to indicate progress was good in this bar.
- The simple, intuitive and minimalist design was appreciated.
- The website's components were nicely spaced and clear.
- "Graph Quest" is a good title.
- It is good that you can see the question and graph together, side-by-side.

Constructive criticism:

- The only colour was grey, which is a bit of a depressing theme.
- Generally speaking, components should fill the space more.
- Buttons should be made thicker.
- Pressing the enter key on the topic code page should cause it to submit the topic code.
- Pressing the enter key in a text box should not cause a page reload.
- The website title should be less prominent.
- *In one instance, the screen flashed "topic not found" for a split second before loading the topic; the respondent found this a little odd.*
- Generally speaking, the text was too small.
- Red and green as progress indicators may be an issue for colour-blind students, so ticks and crosses could be used as well.
- The "check feedback" button on the final submission modal was counter-intuitive; respondents expected it to provide additional feedback, not to take them back to their answers.
- *The result bullet points in this final submission modal could link back to the corresponding answered question.*

The Specific Quiz Questions

Positive feedback:

- The instructions were clear, simple and concise.
- The questions were very useful and appropriately challenging.
- Many of the respondents said that they enjoyed the quiz.

Constructive criticisms:

- The solution feedback was sometimes not detailed enough.
- In the depth-first search question, it was not clear in the instructions that you had to start counting from $v0$ as the "first" vertex.
- In one case, the bipartite graph was too cluttered with edges.

4.1.6 Discussion

All of my interviewees were fourth-year students who had not studied graph theory in some time. This meant that most of them had forgotten the details of the things they had found challenging while studying Algorithms II or the resources they used. Many of them doubted their own abilities (as shown by their response to question 1), despite handling the quiz questions well. Their criticisms of online tools in question 4 reaffirmed the usefulness of Graph Quest in solving such problems as a lack of material or good feedback, since it is able to automatically generate instances of questions and provide customised feedback. The responses to question 5 centred around questions that are not as easily tested by visual graph algorithm quizzes (i.e., long answer, multi-step calculations and proofs). This suggests that although Graph Quest has the potential to improve on existing online learning resources, it may not be well suited to helping students with the types of questions they find most challenging.

Generally speaking, all interviewees gave similar feedback, both positive and negative, despite the interviews being taken independently. This was encouraging, since the consistency implied that their feedback was more objective and prevalent than subjective to individual opinions.

On the whole, students highlighted the automatic question generation and automated feedback as being one of main strengths of the website. Their praise on automation echoed their responses from the first part of the interviews. A couple of students suggested that a (potentially animated) highlighting of elements in the graph would benefit the explanations of feedback. This was particularly the case in the depth-first search question, where the student got the answer wrong and thought it would be useful to see the correct traversal of the graph highlighted along with the feedback. I was able to add this as a feature after the interviews, but only statically; animations would have to wait for a future version.

They were impressed with the questions that involved graph interactivity, particularly the *QSelectPath* question, which required them to select a path in the graph by tapping on nodes in order. It is not surprising when most of them mentioned only ever using the Blackboard quizzes when learning graph theory, which lacks graph interactivity features, and not having used tools like VisuAlgo or D3 Graph Theory (chapter 2) which do. I found that as the students got used to the interactive features, they saw the potential in it and were keen to see more. They put forward several good ideas on making the question panel more interactive, such as clickable text in the multiple choice questions and tooltip definitions when hovering over definitions. I did not have time to implement these kinds of additional features but they would be useful extensions in a future version of Graph Quest. I thought the idea of being able to "sketch" rough working on the graph while answering questions very interesting and see the potential for it in a future version (section 4.5).

The minimalist design was praised very often throughout the interviews, suggesting that the presentation of information was sufficient enough to understand without being overwhelming. However, accessibility was often criticised, with the size of text and buttons being too small for some participants to comfortably see. I was able to quickly resolve these issues after the interviews, as well as implement the suggestions for more coloured feedback, a less depressing colour scheme and icons to indicate progression alongside the existing red-green colours.

Several bugs were discovered during the course of the interviews. I was able to patch some of them (including the issues with the enter key causing a page reload in text boxes for example). In particular, I cover my response to the force-directed layout problems in detail in section 3.5.4. There were a small handful of bugs I did not have time to resolve, the most notable of which is the final submit button not working for two of the five students. I was not able to replicate the bug when hosting locally.

Finally, the design of the quiz itself was highly praised, with most students saying that they enjoyed it and would have found it useful when taking the Algorithms II unit. There were only a small handful of criticisms, the most common of which was that the description in the depth-first search question was not clear as to whether they were supposed to start counting from *v0* as the "first" or "zero-th" vertex. This feedback does little to comment on the success of the Graph Quest website; in reality it would be down to the teacher to design the individual questions. However, knowing that the quiz itself was mostly well-designed means that it was not a confounding variable for the rest of the genuine feedback.

4.2 Current Project Status

The central framework of Graph Quest is complete, deployable and extensible. It also supports numerous additional features. There are some bugs remaining as well as plenty of avenues to explore in terms of extensions. The central architecture of Graph Quest is essentially finished and includes a *graphquest* Python package hosted on GitHub, complete with documentation. The Graph Quest website itself is

made up of a Flask server that serves a React frontend, is containerised using Docker and deployable to AWS LightSail via a GitHub Actions script.

On top of the fundamental framework, Graph Quest currently supports five base question types:

- *QTextInput* (answer via a text box)
- *QMultipleChoice* (choose from a list of options)
- *QVertexSet* (select a set of vertices in the graph)
- *QEdgeSet* (select a set of edges in the graph)
- *QSelectPath* (select a valid path in the graph)

Each question allows the teacher to write functions that:

- Generate NetworkX graph(s) to be displayed.
- Generate the wording of the question.
- Generate a list of accepted solutions.
- Verify a given answer, providing tailored feedback.

The graph visualisation supports the following features:

- Force-directed layout
- Circle layout
- Grid layout
- Bipartite layout
- Rooted tree layout
- Weighted edges
- Directed edges
- Element highlighting
- Node labels
- 0, 1 or multiple data items per node

Question topics support the following settings:

- Linear progression or attempt in any-order.
- Randomised question ordering.
- Feedback after each question, at the end of the topic or not at all.

Other notable features include:

- Multiple graphs may be displayed side-by-side.
- The teacher can specify lists of elements to be highlighted, both at the start of the question and on giving feedback.
- The website may be used on desktop and mobile (but desktop provides a better user experience).
- *graphquest* includes a testing script that allows teachers to run basic validation checks on the question scripts they write.
- *graphquest* provides its own *random_planar_graph()* function, which generates aesthetically pleasing graphs.

A list of known bugs are as follows:

- Sometimes the final submit button has no effect.
- Teacher-specified graph highlighting works when giving feedback. However, if the student revisits that question later, the highlighting is not preserved.
- When there are multiple graphs, specifying the layout sets it for all of them, rather than on an individual level.

- Submitting an answer updates progress, causing the graph to re-render. This is jarring on force-directed layouts which don't always converge to the same layout.
- While a topic is loading, the page defaults to "404 not found".
- When the authentication token times out, it does not automatically log the teacher out. Instead it keeps them on the same page (but with limited functionality since they are no longer authorised to connect with the server endpoints).
- If the teacher uploads a bad script which fails the validation checks, they correctly receive an error and the file is not uploaded. However, even after fixing the errors in the file, the server continues to reject it (even after refreshing the page or logging out). Renaming the file is a current workaround.

4.3 Evaluation with Project Aims

I met all of the initial project aims (set out in Chapter 1) to some extent, though the level of success varied per objective. I believe the aims that were most successfully met were the abstraction of the frontend and the extensibility of the solution. It is also quite powerful, easy-to-use and accessible, though there is more scope to improve these areas further.

In terms of abstraction, the *graphquest* base classes are designed such that the teacher only needs to think about the graph data structures when designing their question (section 3.2.2). All the methods that they implement only deal with graphs in the form of NetworkX data and not the front-end representation. However, the teacher is given the freedom to configure constructor argument settings which dictate aspects of the front-end visualisation more directly, such as the "layout" or "label_style" options. This is only optional however, and the default settings (such as force-directed layout) should be sufficient for most questions.

I firmly believe that the extensibility of the solution is its greatest strength. Ultimately, Graph Quest may be easily extended on at least three different levels: the graph visualisation layer, the base question layer and the specific question layer as shown in Fig. 4.1. Extending the features supported by the graph visualisation layer simply involves defining new event handlers or triggers within the *Graph* component since it is decoupled from the question components. Thanks to Cytoscape's own extensible layout architecture, adding support for new graph layouts is also very feasible (section 3.5.4). Section 3.6 explains how new base questions can be defined by adding a Python class in *graphquest* and a corresponding JavaScript file which both conform to a specification. The final layer of extensibility is the specific questions that teachers write themselves. Since base classes represent very generic question types, teachers are able to write countless specific variations based on these classes.

This relates to the next project aim: power. Since the teacher is able to write Python code and use libraries such as NetworkX, they are given a very large amount of power when designing questions (section 3.2.2). However, with only five base classes and a lot of underutilised Cytoscape features, there is still a lot of potential to expand the base layer of the application further to support a wider range of interactivity. The current graph features should be capable of catering to most of the topics taught in higher education graph theory courses (table 2.1), including the most commonly taught ones (paths and cycles, matchings and trees). However, Graph Quest lacks tabular data displays which would be useful for questions on adjacency matrices. It also currently does not include support for graph colourings, another frequently taught topic, beyond a single colour.

The user study gave good insights into the usability and accessibility of the website, which suggested there was more of a balance between strengths and weaknesses (section 4.1.5). On the one hand, the interactive features and minimalist design were highly praised. On the other, text and buttons were often too small and the sole use of red-green colouring to indicate progress was flagged as being a potential issue for colour-blind students.

4.4 Evaluation with Existing Solutions

An updated version of table 2.2 is shown in table 4.1, which compares Graph Quest's provision of features against the existing solutions. It provides question generation capabilities on a similar level to Numbas with scripting through Python allowing for complex logic and randomised instances. This also allows it to respond to students' answers with customised feedback via implementation of the *generate_feedback()* method (section 3.2.2). The graph visualisation is at a similar standard to those of VisuAlgo and D3

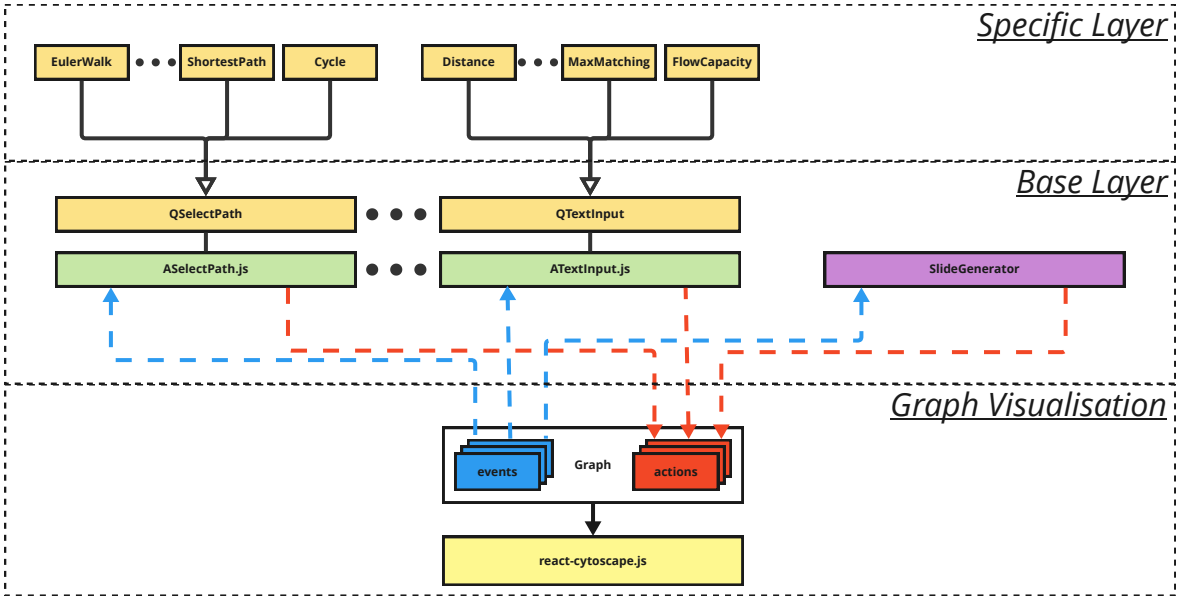


Figure 4.1: Layers of application extensibility.

Table 4.1: Summary of the provision of required features by existing solutions, including Graph Quest.

Feature	**Blackboard**	**Numbas**	**VisuAlgo**	**D3GT**	**Algorithm Visualizer**	**Graph Quest**
Question Generation	Randomised	Randomised + scripted	Randomised + scripted	None	None	Randomised + scripted
Feedback	Generic	Customised	Customised	N/A	N/A	Customised
Graph visualisation	Image import	Image import + simple layout	Advanced layout	Advanced layout	Simple layout	Advanced layout
Graph interactivity	N/A	None	Good	Great	None	Good
Easily interface with a graph theory library	No	No	No	Yes	Yes	Yes

Graph Theory. Where D3 Graph Theory only uses force-directed layouts, Graph Quest also includes others such as bipartite and trees. The force-directed layouts give similar results in both cases as illustrated in Fig. 4.2. Graph Quest is also capable of generating random planar graphs which help visualise weighted graphs in a much more aesthetically pleasing manner than the circle layout used by VisuAlgo as demonstrated by Fig. 4.3. The interactivity features of Graph Quest are comparable to those of VisuAlgo since it is able to support every base question type that VisuAlgo provides except one: questions that require the student to create or edit a graph themselves. The *graphquest* package was designed specifically to interface with the NetworkX graph theory library and as explained in section 3.3.1, it is also compatible with Graph-tool and iGraph.

In summary, Graph Quest has better overall coverage of the required features than any of the existing solutions studied. However, the other solutions perform better in some specialised cases. In particular, Blackboard and Numbas supply a much more extensive set of topic settings (such as time constraints on quizzes) and D3 Graph Theory's visualisations are much more interactive, with capabilities for the student to reposition nodes in the graph and change the colour of nodes as two examples.

4.5 Future Extensions

Here I provide a list of potential future extensions, in order of what I feel is most achievable or useful.

1. Graph colouring: Allow teachers to include a "colour" attribute in their graph nodes which gets

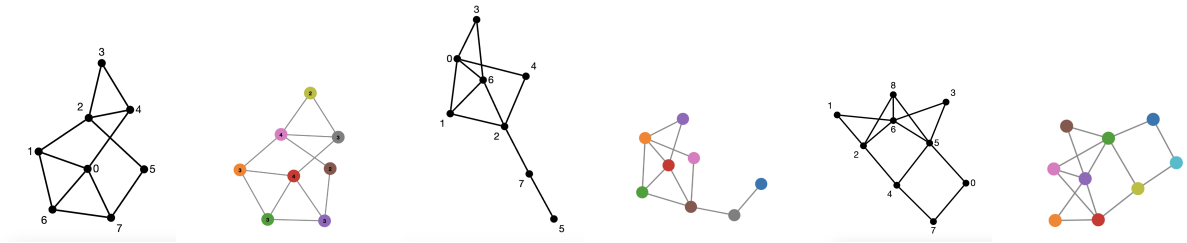


Figure 4.2: Pairwise comparison of graphs generated by Graph Quest (black) with equivalent graphs generated by D3 Graph Theory (coloured) [72].

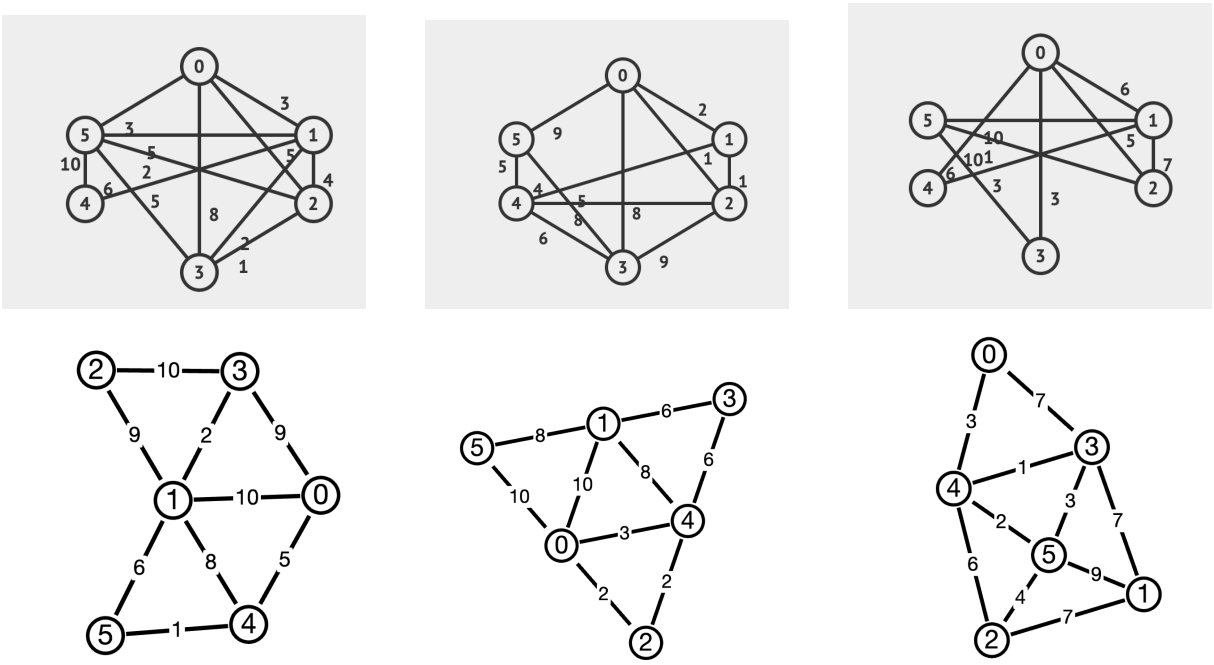


Figure 4.3: Comparison between 6-vertex weighted graphs displayed by VisuAlgo's circle layout (above) [37] and those generated with Graph Quest's random planar generator and displayed with its force-directed layout (below).

used to style the nodes.

2. A *QEditGraph* question base: This would enable students to create a graph or edit an existing one. This would include adding/removing nodes/edges, editing edge weights, changing direction of directed edges and colouring nodes. The question constructor argument settings should include toggles for the functionality, so the teacher can limit the students' controls depending on the needs of the specific question.

3. Improved security (section 3.7):

   - Separate the Python question script processing from the main server, use Restricted Python on it and run it in AWS Lambda.
   - Authenticate teachers via SSO with Azure AD.
   - Add timeouts so that jobs are cancelled if they run for too long.

4. Support for LaTeX: This could be implemented via MathJax [10] and be used for both question descriptions and feedback.

5. Caching: The student's progress could be cached in local storage so that they don't lose their progress when refreshing the page. This may also be used to solve the bug with highlighted elements (section 4.2).

6. Support tooltips with definitions: Based on user feedback (section 4.1.5), teachers could optionally supply definitions along with their question descriptions so that when a student hovers their cursor over the word, a tooltip is revealed with the definition.

7. Add B-trees to the *graphquest.graph* module: NetworkX does not have a dedicated data structure for B-trees. Providing an implementation as part of Graph Quest could also help towards the problem of rendering them hierarchically (section 3.5.4).

8. Support additional topic settings: These may include quiz timeouts, number of attempts, difficulty settings, marking systems, etc.

9. Graph sketching for working out: Run a user study designed to learn how students use pen and paper to help them while working out solutions to graph theory questions. Then design a digital solution that emulates this within Graph Quest.

10. Support other data structure visualisations: Examples include arrays, stacks and hash tables. This would involve adding another visualisation layer along with the *Graph* component with its own event system interface.

11. Lecture slide generation: Extend Graph Quest to include a module for generating graph visualisations for lecture slides (purple component in Fig. 4.1).

12. Integrate with Algorithm Visualiser: Algorithm Visualiser is a powerful framework for creating algorithm visualisations. Combining it with Graph Quest would lead towards an all-in-one solution for generic algorithm teaching resource creation.

4.6 Conclusion

As previously stated, I met all of the project's aims and presented a solution that fulfils the needs of higher education graph theory question generation in a way that other similar websites do not. Graph Quest strikes a balance between power and abstraction, while being highly extensible. There is still plenty of room for further enhancements, with security being one of the biggest shortcomings.

Graph Quest is built on top of many existing technologies, primarily NetworkX, Cyotsocape, Flask and React. My personal contributions were mostly in designing the framework that allows these technologies to interact in a modular way and designing algorithms to plug gaps in their shortcomings. The modularity included the event system interface for graphs (section 3.5.5), and the Python-JavaScript specification for dynamic import (sections 3.2.2 and 3.6). The algorithms include the random planar graph generator (section 3.2.3), optimal root finding and node label positioning (section 3.5.4).

Bibliography

[1] A. Abenoja et al. Bootstrap, 2023. URL: https://react-bootstrap.github.io.

[2] Inc. Amazon Web Services. Amazon lightsail: Build applications and websites fast with low-cost, pre-configured cloud resources, 2023. URL: https://aws.amazon.com/lightsail/.

[3] AntV. G6 diagram visualization engine, 2023. URL: https://g6.antv.antgroup.com.

[4] AQA. As and a-level computer science - 4.2 fundamentals of data structures, 2023. URL: https://www.aqa.org.uk/subjects/computer-science-and-it/as-and-a-level/computer-science-7516-7517/subject-content-a-level/fundamentals-of-data-structures.

[5] M. Bostock. D3 - sankey diagram, 2021. URL: https://observablehq.com/@d3/sankey.

[6] M. Bostock. D3 - stacked bar chart, 2021. URL: https://observablehq.com/@d3/stacked-bar-chart.

[7] M. Bostock. Data-driven documents, 2021. URL: https://d3js.org.

[8] M. Bostock. D3 - bivariate choropleth, 2022. URL: https://observablehq.com/@d3/bivariate-choropleth.

[9] D. Caldas, S. Hernández, T. Harrison, and A. Klopp-Tosser. react-d3-graph, 2021. URL: https://github.com/danielcaldas/react-d3-graph#readme.

[10] D. Cervone and V. Sorge. Mathjax, 2022. URL: https://www.mathjax.org.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*, volume Fourth edition, chapter VI. The MIT Press, 2022.

[12] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 11 2005.

[13] Cytoscape. Cytoscape.js documentation - collection style, 2023. URL: https://js.cytoscape.org/#collection/style.

[14] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, oct 1996. doi:10.1145/234535.234538.

[15] University of Oxford Department of Computer Science. Algorithms and data structures course page, 2022. URL: https://www.cs.ox.ac.uk/teaching/courses/2022-2023/algorithms/.

[16] Imperial College London Department of Computing. 40008 graphs and algorithms course page, 2022. URL: https://www.imperial.ac.uk/computing/current-students/courses/40008/.

[17] Technical University of Munich Department of Mathematics. Visualizations of graph algorithms, 2023. URL: https://algorithms.discrete.ma.tum.de.

[18] University College London Department of Mathematics. M0029 graph theory and combinatorics course description, 2023. URL: https://www.ucl.ac.uk/maths/sites/maths/files/math0029.pdf.

[19] T. Dwyer. cola.js - constraint-based layout in the browser, 2023. URL: https://ialab.it.monash.edu/webcola/.

[20] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[21] Python Software Foundation. Python documentation - importlib — the implementation of import, 2023. URL: https://docs.python.org/3/library/importlib.html.

[22] Python Software Foundation. Python documentation - json — json encoder and decoder, 2023. URL: https://docs.python.org/3/library/json.html.

[23] Python Software Foundation. Python documentation - os — miscellaneous operating system interfaces, 2023. URL: https://docs.python.org/3/library/os.html.

[24] Python Software Foundation. Python package index, 2023. URL: https://pypi.org.

[25] Zope Foundation and Contributors. Restrictedpython, 2023. URL: https://github.com/zopefoundation/RestrictedPython.

[26] M. Franz, C. Lopes, D. Fong, M. Kucera, , M. Cheung, M. Siper, G. Huck, Y. Dong, O. Sumer, and G. Bader. Cytoscape.js 2023 update: a graph theory library for visualization and analysis. *Bioinformatics*, 39(1), 2023.

[27] M. Franz, C. Lopes, G. Huck, Y. Dong, O. Sumer, and G. Bader. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311, 2015.

[28] D. Galles. Data structure visualizations, 2011. URL: https://www.cs.usfca.edu/~galles/visualization/.

[29] GeoGebra. Geogebra, 2023. URL: https://www.geogebra.org.

[30] L. Gilbert. Flask-jwt-extended's documentation, 2022. URL: https://pypi.org/project/Flask-JWT-Extended/.

[31] Google. Material design guidelines, 2022. URL: https://m2.material.io/design/guidelines-overview.

[32] The Guardian. Uk universities ranked by subject area: computer science and information systems, 2023. URL: https://www.theguardian.com/education/ng-interactive/2022/sep/24/best-uk-universities-for-computer-science-information-systems-league-table.

[33] Complete University Guide. Computer science subject league table 2023, 2023. URL: https://www.thecompleteuniversityguide.co.uk/league-tables/rankings/computer-science.

[34] A. Hagberg, D. Schult, and P. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[35] S. Halim. Visualgo – visualising data structures and algorithms through animation. *Olympiads in Informatics*, 9:243–245, 2015.

[36] S. Halim. Visualgo, 2022. URL: https://visualgo.net/en.

[37] S. Halim. Visualgo - steiner tree - training, 2022. URL: https://visualgo.net/training?diff=Medium&n=7&tl=0&module=steinertree.

[38] S. Halim, Z. Koh, V. Loh, and F. Halim. Learning algorithms with unified and interactive web-based visualization. *Olympiads in Informatics*, 6:53–68, 2012.

[39] S. Hobbs. Cors tutorial: A guide to cross-origin resource sharing, 2019. URL: https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/.

[40] IGraph. Visualisation of graphs - exporting to other graph formats, 2023. URL: https://python.igraph.org/en/stable/visualisation.html#exporting-to-other-graph-formats.

[41] Amazon Web Services Inc. Aws lambda: Run code without thinking about servers or clusters, 2023. URL: https://aws.amazon.com/lambda/.

[42] Anthology Inc. Blacboard help center - hotspot questions, 2022. URL: https://help.blackboard.com/Learn/Instructor/Ultra/Tests_Pools_Surveys/Question_Types/Hotspot_Questions.

[43] Anthology Inc. Blackboard help center - calculated formula questions, 2022. URL: https://help.blackboard.com/Learn/Instructor/Ultra/Tests_Pools_Surveys/Question_Types/Calculated_Formula_Questions.

[44] Anthology Inc. Blackboard help center - examity® online proctoring, 2022. URL: https://help.blackboard.com/Learn/Instructor/Ultra/Tests_Pools_Surveys/Examity.

[45] Anthology Inc. Blackboard help center - question banks, 2022. URL: https://help.blackboard.com/Learn/Instructor/Ultra/Tests_Pools_Surveys/ULTRA_Reuse_Questions/ULTRA_Question_Banks.

[46] Anthology Inc. Blackboard help center - respondus, 2022. URL: https://help.blackboard.com/Learn/Instructor/Ultra/Tests_Pools_Surveys/Respondus.

[47] Anthology Inc. Blackboard help center - tests, pools, and surveys, 2022. URL: https://help.blackboard.com/Learn/Instructor/Ultra/Tests_Pools_Surveys.

[48] Anthology Inc. Blackboard's help center - question pools, 2022. URL: https://help.blackboard.com/Learn/Instructor/Ultra/Tests_Pools_Surveys/ULTRA_Reuse_Questions/Question_Pools.

[49] Docker Inc. Docker, 2023. URL: https://www.docker.com.

[50] GitHub Inc. Github actions documentation, 2023. URL: https://docs.github.com/en/actions.

[51] GitHub Inc. Github pages, 2023. URL: https://pages.github.com.

[52] ReadTheDocs Inc. Read the docs, 2023. URL: https://readthedocs.org.

[53] Jetbrains. Dev ecosystem 2021 - python, 2021. URL: https://www.jetbrains.com/lp/devecosystem-2021/python/.

[54] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, apr 1989. doi:10.1016/0020-0190(89)90102-6.

[55] J. Lapinskas. Coms algorithms ii unit page, 2022. URL: https://uob-cs-algorithms-ii.github.io/.

[56] T. Lin. Benchmark of popular graph/network packages v2, 2020. URL: https://www.timlrx.com/blog/benchmark-of-popular-graph-network-packages-v2.

[57] University of Oxford Mathematical Institute. B8.5 graph theory course page, 2022. URL: https://courses.maths.ox.ac.uk/course/view.php?id=659.

[58] Microsoft. Azure – what is conditional access?, 2023. URL: https://learn.microsoft.com/en-GB/azure/active-directory/conditional-access/overview.

[59] Microsoft. Azure active directory (azure ad), 2023. URL: https://azure.microsoft.com/en-gb/products/active-directory.

[60] P. Mura. graphquest documentation, 2023. URL: https://graphquest.readthedocs.io/en/latest/index.html.

[61] NetworkX. Networkx reference - graph generators - gnp_random_graph, 2023. URL: https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.gnp_random_graph.html#gnp-random-graph.

[62] NetworkX. Networkx reference - graph.degree, 2023. URL: https://networkx.org/documentation/stable/reference/classes/generated/networkx.Graph.degree.html#networkx.Graph.degree.

[63] NetworkX. Networkx reference - reading and writing graphs - json, 2023. URL: https://networkx.org/documentation/stable/reference/readwrite/json_graph.html.

[64] Vladimir Nikolic. Rise of kingdom font, 2017. URL: https://www.fontspace.com/rise-of-kingdom-font-f27732.

[65] C. Obrecht. Eukleides, 2004. URL: http://eukleides.org.

[66] OCR. A level specification computer science, 2021. URL: https://www.ocr.org.uk/Images/170844-specification-accredited-a-level-gce-computer-science-h446.pdf.

[67] Department of Computer Science and University of Cambridge Technology. Part 1a cst algorithms 1 course page, 2022. URL: https://www.cl.cam.ac.uk/teaching/2223/Algorithm1/.

[68] Department of Computer Science and University of Cambridge Technology. Part 1a cst algorithms 2 course page, 2022. URL: https://www.cl.cam.ac.uk/teaching/2223/Algorithm2/.

[69] University of St Andrews. Mt4514 graph theory module page, 2022. URL: https://www.st-andrews.ac.uk/subjects/modules/catalogue/?code=MT4514&academic_year=2022%2F3.

[70] Stack Overflow. Stack overflow developer survey 2022 - most popular technologies, 2022. URL: https://survey.stackoverflow.co/2022/#technology-most-popular-technologies.

[71] A. Pandey. D3 graph theory, 2020. URL: https://d3gt.com.

[72] A. Pandey. D3 graph theory - vertices and edges, 2020. URL: https://d3gt.com/unit.html?vertices-and-edges.

[73] T. Panvey. D3 graph theory github, 2023. URL: https://github.com/mrpandey/d3graphTheory.

[74] J. Park. Algorithm visualizer, 2022. URL: https://algorithm-visualizer.org.

[75] T. Peixoto. The graph-tool python library., 2014. URL: https://doi.org/10.6084/m9.figshare.1164194.

[76] T. Peixoto. Graph-tool installation instructions - memory requirements for compilation, 2023. URL: https://git.skewed.de/count0/graph-tool/-/wikis/installation-instructions#manual-compilation.

[77] G. Plique. Graphology, a robust and multipurpose graph object for javascript, 2021. doi:10.5281/zenodo.5681257.

[78] G. Plique. Graphology documentation, 2021. URL: https://graphology.github.io.

[79] G. Plique and A. Jacomy. Sigma.js, 2014. URL: https://www.sigmajs.org.

[80] B. Postlethwaite et al. react-cytoscapejs repository, 2022. URL: https://github.com/plotly/react-cytoscapejs.

[81] Postman. Postman api platform, 2023. URL: https://www.getpostman.com.

[82] The PyPy Project. Pypy's sandboxing features, 2023. URL: https://doc.pypy.org/en/latest/sandbox.html.

[83] The Qubes OS Project et al. Qubes - architecture, 2023. URL: https://www.qubes-os.org/doc/architecture/.

[84] PYPL. Pypl popularity of programming language, 2023. URL: https://pypl.github.io/PYPL.html.

[85] Sphinx. Sphinx python documentation generator, 2023. URL: https://www.sphinx-doc.org/en/master/index.html.

[86] O. Tassinari and M. Brookes. Material ui, 2023. URL: https://mui.com.

[87] Tiobe. Tiobe index for april 2023, 2023. URL: https://www.tiobe.com/tiobe-index/.

[88] P. Turán. On an extremal problem in graph theory. *Matematikai és Fizikai Lapok*, 48:436–452, 1941.

[89] W. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 13:743–768, 1963.

[90] R. Uehara and Y. Uno. Efficient algorithms for the longest path problem. In R. Fleischer and G. Trippen, editors, *Algorithms and Computation*, pages 871–883, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[91] Newcastle University. Graph theory questions, 2021. URL: https://numbas.mathcentre.ac.uk/exam/27245/graph-theory-questions/.

[92] Newcastle University. Numbas, 2021. URL: https://www.numbas.org.uk/.

[93] Newcastle University. Numbas documentation - first-party extensions, 2021. URL: https://docs.numbas.org.uk/en/latest/extensions/first-party.html.

[94] Newcastle University. Numbas documentation - jme, 2021. URL: https://docs.numbas.org.uk/en/latest/jme-reference.html#.

[95] Newcastle University. Numbas documentation - marking algorithms, 2021. URL: https://docs.numbas.org.uk/en/latest/marking-algorithm.html.

[96] Newcastle University. Numbas documentation - parts - scripts, 2021. URL: https://docs.numbas.org.uk/en/latest/question/parts/reference.html#scripts.

[97] Newcastle University. Numbas documentation - writing an extension, 2021. URL: https://docs.numbas.org.uk/en/latest/extensions/writing-extensions.html.

[98] L. Vailshery. Most used programming languages among developers worldwide as of 2022, 2022. URL: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.

[99] B. Valentin and M. Gerhäuser. Interactive svg with jsxgraph. *Proceedings of the SVGOpen 2009*, 2009.

[100] Algorithm Visualizer. Algorithm visualizer - dijkstra's shortest path, 2023. URL: https://algorithm-visualizer.org/greedy/dijkstras-shortest-path.

[101] W3Techs. Usage statistics of javascript as client-side programming language on websites, 2023. URL: https://w3techs.com/technologies/details/cp-javascript/.

[102] Y. Yang. Graphrel, 2017. URL: https://github.com/yiboyang/graphrel.

[103] K. Zarankkiewicz. Problem p 101. *Colloq. Math*, 2:301, 1951.