UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Clairvoyant: Customizable Artificial Intelligence Algorithm Visualization Tool

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Fabado Gómez Lobo, Mario

Tutor: Onaindia de la Rivaherrera, Eva

Experimental director: Venkatesaramani, Rajagopal

ACADEMIC YEAR: 2023/2024

Abstract

Clairvoyant is a web application built to facilitate education on topics related to Artificial Intelligence and Algorithms in Computer Science. The main problem it aims to solve is that of providing an intuitive, open, and highly customizable learning environment for these topics. Currently, Clairvoyant supports two problem types: Graph Search and Adversarial Search, both foundational topics in Artificial Intelligence. The Graph Search visualizer provides tools to explore, at large (with built-in sample algorithms and cases) or specifically down to the code, how these types of algorithms work. It not only covers search methods like DFS, BFS, A*, etc; but also includes tools to learn about heuristic admissibility and consistency. Adversarial Search, on the other hand, provides a more direct use of Artificial Intelligence algorithms; playing zero-sum, perfect information games. One great strength of Clairvoyant is that the game and the algorithm are separate, and one can program a completely custom game very easily. The visualizer allows one to create a custom render function for any given game state to explore the game tree more effectively. Algorithms such as Minimax, Alpha-beta pruning, and Expectiminimax (for non-deterministic games) are prime examples of adversarial search algorithms.

In conclusion, Clairvoyant is both a learning and teaching tool that educators can tailor to provide an intuitive understanding of the foundational aspects of Artificial Intelligence Algorithms.

Key words: artificial intelligence, education, algorithms, visualization, intuition, web applications, React

Contents

List of Figures

List of Tables

Resum

Clairvoyant és una aplicació web dissenyada per facilitar l'educació en temes relacionats amb la intel·ligència artificial i els algoritmes dins del camp de la informàtica. El problema principal que busca solucionar és el de la creació d'un entorn d'aprenentatge altament intuïtiu, obert i personalitzable per a aquests temes. Actualment, Clairvoyant suporta dos tipus de problemes: Cerca en Grafs i Cerca amb Adversari, ambdós temes fonamentals al camp de la Intel·ligència Artificial. El visualitzador de Cerca a Grafs proporciona eines per explorar, tant de forma general (utilitzant els algorismes i casos per defecte) com específicament fixant-se en el codi, com funcionen aquests algorismes. No només cobreix algorismes de cerca, sinó que també inclou eines per fer anàlisis sobre l'admissibilitat i consistència de funcions heurístiques. El mòdul de cerca amb adversari proporciona un ús més directe dels algorismes d'intel·ligència artificial; l'anàlisi de jocs suma zero amb informació perfecta. Una qualitat important de Clairvoyant en aquest aspecte és que el joc i l'algorisme estan separats. I és senzill programar un joc completament nou des de l'aplicació. El visualitzador permet dissenyar una funció de renderització personalitzada per a qualsevol estat de joc per poder explorar l'arbre de posicions més efectivament. Algorismes com minimax, poda alfa-beta, o expectiminimax (per a jocs no deterministes), són grans exemples d'algorismes de cerca amb adversari.

En conclusió, Clairvoyant és una eina tant d'aprenentatge com d'ensenyament que els educadors poden adaptar per proporcionar una entesa fonamental molt útil sobre aspectes relacionats amb algoritmes d'intel·ligència artificial.

Paraules clau: intel·ligència artificial, educació, algoritmes, visualització, intuïció, aplicacions web, React

Resumen

Clairvoyant es una aplicación web diseñada para facilitar la educación en temas relacionados con la Inteligencia Artificial y los algoritmos dentro del campo de la informática. El problema principal que busca solventar es el de la creación de un entorno de aprendizaje altamente intuitivo, abierto, y personalizable para estos temas. Actualmente, Clairvoyant soporta dos tipos de problemas: Búsqueda en Grafos y Búsqueda con Adversario, ambos temas fundamentales en el campo de la Inteligencia Artificial. El visualizador de Búsqueda en Grafos proporciona herramientas para explorar, tanto de forma general (utilizando los algoritmos y casos por defecto) como específicamente fijándose en el código, como funcionan estos algoritmos. No solo cubre algoritmos de búsqueda, sino que también incluye herramientas para hacer análisis sobre la admisibilidad y consistencia de funciones heurísticas. El módulo de búsqueda con adversario proporciona un uso más directo de los algoritmos de Inteligencia Artificial; el análisis de juegos suma cero con información perfecta. Una cualidad importante de Clairvoyant en este aspecto es que el juego y el algoritmo están separados. Y es sencillo programar un juego completamente nuevo desde la aplicación. El visualizador permite diseñar una función de renderizado personalizada para cualquier estado de juego para poder explorar el árbol de posiciones más efectivamente. Algoritmos como minimax, poda alfa-beta, o expectiminimax (para juegos no deterministas), son grandes ejemplos de algoritmos de búsqueda con adversario.

En conclusión, Clairvoyant es una herramienta tanto de aprendizaje como de enseñanza que los educadores pueden adaptar para proporcionar un entendimiento fundamental muy útil sobre aspectos relacionados con algoritmos de Inteligencia Artificial.

Palabras clave: inteligencia artificial, educación, algoritmos, visualización, intuición, aplicaciones web, React

CHAPTER 1

Introduction

The field of Artificial Intelligence is ever-evolving. It has recently been brought to the forefront of modern life, and as such, advancements and research efforts have multiplied relentlessly.

With such a critically important field, it is paramount that education tools can keep up. This Capstone Project aims to provide such a tool, starting with the very foundations of Artificial Intelligence. While comparatively simple, fundamental AI algorithms like graph search or adversarial search are ubiquitous in everyday computer applications. However, educating students who may not have much experience with abstraction can complicate this fundamental task. Allowing educators to provide clear, flexible visualizations to facilitate intuition and allowing students to easily play around with different cases and algorithm variants is an effective way to make understanding the foundations of AI more accessible.

1.1 Motivation

Teaching the basics of Artificial Intelligence (not exclusively, as this is true for a lot of education on algorithms, such as sorting algorithms) usually involves a lot of work creating slides of different steps of an algorithm in a concrete, illustrative case. This is slow, busy work which could be better spent by educators. Moreover, this technique lacks flexibility; curious, proactive students who do not understand certain specifics may ask for different cases, or ask about edge cases or the functionality of the algorithm in practice. These are problems that prepared slides simply cannot solve.

With this being the case, we must find an effective alternative. Some already exist, but I believe they have some significant shortcomings. For example, in the case of graph search, most tools [5] only allow you to visualize the classic algorithms; and editing, importing, or exporting cases can be time-consuming. Some others [7] restrict you to classic algorithms on grids only. Those that allow you to dive into the code [6] have very lackluster user interaction.

With the intent of tackling this issue, I decided to develop Clairvoyant, an easily accessible web application where instructors can prepare their own cases to display. Moreover, I think it's critical that students can also choose to explore these cases on their own time. As such, this application is also aimed at students.

1.2 Objectives

The main goal of this project is to provide a platform where teachers and students can visually explore algorithms in the field of artificial intelligence.

As such, the itemized objectives for this project could be expressed as follows:

- For each problem type, an effective visualizer must be designed and developed.
- An efficient, safe way to evaluate user-sourced code must be developed. Furthermore, whenever there are errors in the source code, an informative and illustrative error should be displayed.
- For each problem type, default algorithms and cases must be provided for ease of use for students and to provide an illustrative example of how to develop these for users.
- The offered API, as well as the use of the app itself, must be properly documented in an accessible page within the application.
- The application must remain open source and properly documented in case anyone wishes to copy and extend the application for some specialized use, or to add wholly new problem types.
- The application must be deployed to be accessible via the internet without having to run the server locally.

1.3 Expected Impact

This project was created with the goal of it being used in classrooms for educational purposes. If instructors and educators are willing to make use of it; our expectation is that, when it comes to topics covered by the application, their teaching will be much more in-depth and clear. Playing on this very concept, the name of the application, "Clairvoyant", suggests a clear vision and understanding of AI topics.

Our hope is that with the aid of this application, students can explore the complex but fascinating field of Artificial Intelligence algorithms and get an intuitive understanding of them, allowing them to further dive into the implementation details if they so wish.

The expected impact does not end with students. We expect educators to be able to use this tool to significantly reduce the time it takes to prepare lessons on these topics, thus allowing them to focus on more important parts of their lessons.

Lastly, if the application gets sufficiently popular; hopefully, educators would be able to leverage its open-source nature to create wholly new cases, algorithms, and even entire modules.

1.4 Memory Structure

What follows is a short description of the remaining major blocks this memory will contain;

- **Chapter 2. State of the Art**. This chapter describes the current state of the art on this specific problem and related topics such as Computer Science education in general.
- **Chapter 3. Methods**. This chapter describes the tools and general methodology used to organize and develop this project.
- **Chapter 4. Analysis and Requirements**. This chapter describes the initial analysis stage and outlines, in detail, the requirements to be fulfilled by the application. It also describes other concerns with the product and general project planning.
- **Chapter 5. Design and Data Architecture**. This chapter describes the broad design principles of the application, as well as the specific design of certain pages and components, and the internal data architecture of the application.
- **Chapter 6. Development and Deployment**. This chapter describes the details of the development process of the application, as well as how the locally developed program was deployed into a live server, so as to make it externally accessible.
- **Chapter 7. Testing**. This chapter describes the testing and debugging methodology used for the application, as well as some use case scenarios that describe, step-by-step, how the application works in practice.
- **Chapter 8. Conclusions and Potential Improvements**. This chapter provides a summary of the conclusions of this project with relation to the field of Computer Science. It also briefly goes over the potential ways to improve this app into a more fleshed out product in the long term.

CHAPTER 2
State of the Art

When developing software projects of this nature, it is important to understand the currently available alternatives to what we are creating. In the previous section, some graph visualization web applications were already mentioned [5] [6] [7]. We will go into some detail on what those offer, as well as exploring the state of the art on the broader field of Computer Science education.

2.1 Pathfinding Visualization Tools

Pathfinding visualization is intrinsically connected to Graph Search. After all, every pathfinding algorithm is, at its core, a graph search algorithm over some graph representing traversable terrain. This makes tools that allow one to visualize these kinds of algorithms a really close example of what the state of the art looks like in education when it comes to Graph Search algorithms.

One such relevant tool worth highlighting is pathfindout [4]. This tool allows the user to modify a square grid by adding walls or terrain of varying weights, along with the ability to run one of four popular Graph Search algorithms (DFS, BFS, UCS, A*) on that grid. It excels in terms of performance and visually it is quite clean. It also has a fair amount of options which suffice for the vast majority of cases. Clairvoyant aims to provide a very similar product, with the added benefit of being able to program any kind of algorithm and also work with generic graphs.

2.2 CS Education

The purpose of Clairvoyant is to be a tool that both students and instructors can use to help elucidate certain processes and algorithms, it is not meant to be something a student can use to teach themselves a topic. However, that latter kind of application fits within the same niche of CS education, with a significantly different paradigm. Many alternatives exist for teaching oneself AI algorithms or CS topics in general. Furthermore, many of them use really interesting visualization methods that can be applied to Clairvoyant's development.

2.2.1. Brilliant

Brilliant, also known as brilliant.org [8], is a platform that makes heavy use of gamification elements and encourages a very hands-on approach to learning. Its courses are based on user interaction and incremental learning. This is one of the main inspirations of Clairvoyant as a product.

The differences, however, are also very significant. Brilliant is a paid service, and it covers a lot of topics, not just algorithms (Figure 2.1).

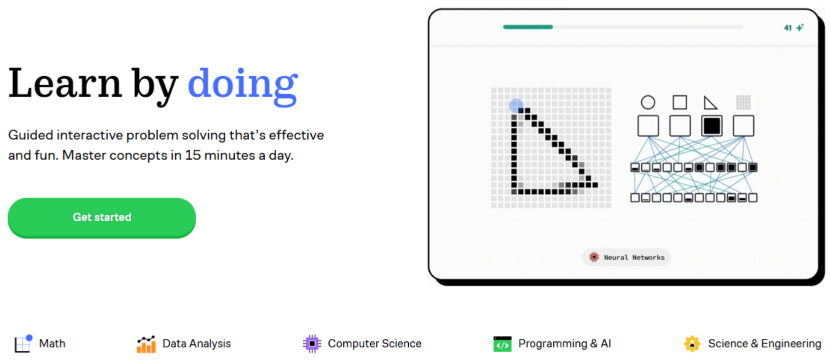
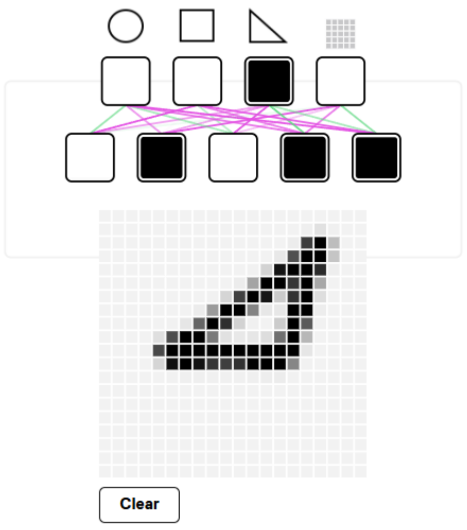


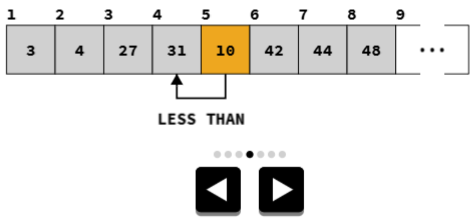
Figure 2.1: Brilliant's Course Selection, with a depiction of its Neural Network module on the side.

Brilliant does a really good job at teaching the fundamentals of Computer Science, but more specific topics can only be found in wiki articles, such as A* search [9]. It succeeds particularly well at teaching and visualizing the function of Neural Networks (Figure 2.2a) and sorting algorithms (Figure 2.2b).

Note the step-by-step visualization provided by the sorting algorithm visualizer, this is also the approach Clairvoyant uses for its Graph Search module.



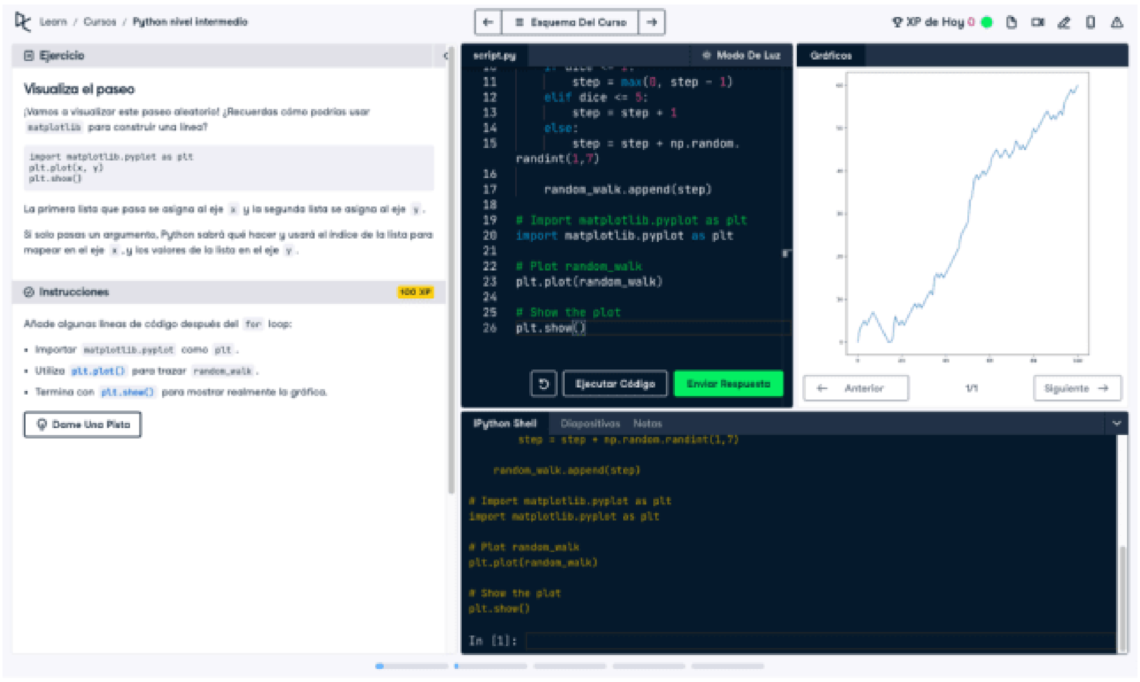
(a) Brilliant's Visualization of a Neural Network with hidden layers



(b) Brilliant's Visualization of the Insertion Sort algorithm

2.2.2. Datacamp

Datacamp [10] uses gamification elements much like Brilliant does. However, it's more focused on teaching Data Science with Python and R rather than pure Computer Science. In some ways, it is more hands-on and practical than Brilliant, with hands-on coding (Figure 2.3) and more complex scenarios. In a way, this kind of hands-on coding approach is what we went with when designing Clairvoyant. Allowing the user to see the code that creates what they are visualizing is an invaluable way to build practical experience and allow the user to experiment and see what works and what doesn't.



Visualiza el paseo

¡Vamos a visualizar este paseo aleatorio! ¿Recuerdas cómo podrías usar `matplotlib` para construir una línea?

```
import matplotlib.pyplot as plt
plt.plot(x, y)
plt.show()
```

La primera lista que pasa se asigna al eje `x` y la segunda lista se asigna al eje `y`.

Si solo pasas un argumento, Python sabrá qué hacer y usará el índice de la lista para mapear en el eje `x`, y los valores de la lista en el eje `y`.

Instrucciones 100 XP

Añade algunas líneas de código después del `for` loop:

- Importar `matplotlib.pyplot` como `plt`.
- Utiliza `plt.plot()` para trazar `random_walk`.
- Termina con `plt.show()` para mostrar realmente la gráfica.

```
            step = max(0, step - 1)
        elif dice <= 5:
            step = step + 1
        else:
            step = step + np.random.
randint(1,7)

        random_walk.append(step)

# Import matplotlib.pyplot as plt
import matplotlib.pyplot as plt

# Plot random_walk
plt.plot(random_walk)

# Show the plot
plt.show()
```

```
        step = step + np.random.randint(1,7)

    random_walk.append(step)

# Import matplotlib.pyplot as plt
import matplotlib.pyplot as plt

# Plot random_walk
plt.plot(random_walk)

# Show the plot
plt.show()

In [1]:
```

Figure 2.3: Datacamp exercise sample shown in Datacamp's home page

2.2.3. Other Course Platforms

Many other course platforms, such as Khan academy [11] or codecademy [12] present courses with combinations of video-based learning and articles, and in some cases for simpler topics, also have a set of exercises. While these are accessible platforms for learning computer science, they don't really go in-depth and have a hard time explaining complex topics. Khan academy, for example, only has an article on Breadth-First Search, and nothing else on Graph or Adversarial Search.

CHAPTER 3

Methods

The methodology used for the completion of this project has been a relatively loose Agile methodology. Many specific agile methodologies [1] focus on how to handle teams and users of a live version of the app, as well as stakeholders and the like. In the case of this project, I am the sole developer and must handle all tasks, so team cohesion policies are irrelevant. Regardless, the project is loosely based on Scrum (Figure 3.2) and Feature-Driven Development (Figure 3.1). In both methodologies, the design and development processes are divided into small chunks (Called Sprints in Scrum, and Iterations in FDD). We will use the FDD terminology from now on.

In practice, here's how the methodology goes.

1. An initial product model is created
2. The product model is used to create a list of new features
3. For each feature, a design plan, full design, and development build is created
4. Once all features of the model are complete, the model is refined with additional content. Then the flow of development returns to step 2.

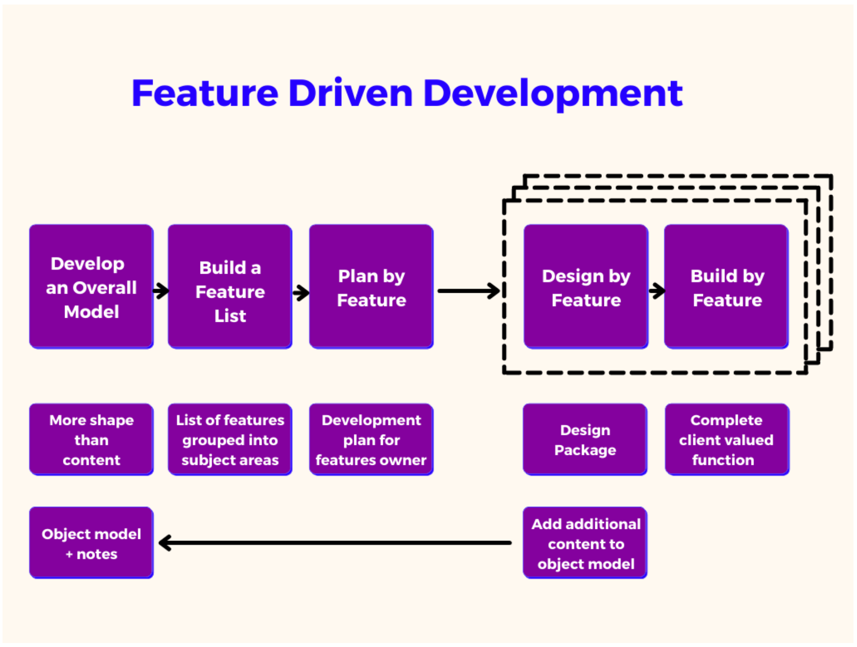

Feature Driven Development
Develop an Overall Model
Build a Feature List
Plan by Feature
Design by Feature
Build by Feature
More shape than content
List of features grouped into subject areas
Development plan for features owner
Design Package
Complete client valued function
Object model + notes
Add additional content to object model

Figure 3.1: Feature Driven Development Methodology Chart

The elements taken from Scrum are the backlogs. Oftentimes, when developing features, certain design decisions can impact changes on other parts of the model. In this case, a backlog task is created, which is logged in the Kanban Board (Figure 3.3). Backlog items are then prioritized as either essential or refinement. Essential fixes are covered immediately, while refinement items (those that aren't part of the formal requirements of the project but can improve user experience, see section 4.1.3) are taken care of after all formal requirements are finished or whenever the development of a feature would easily encompass the implementation of such item.

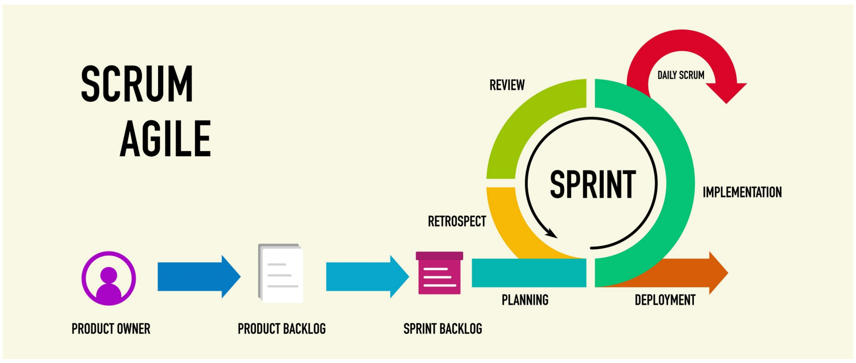
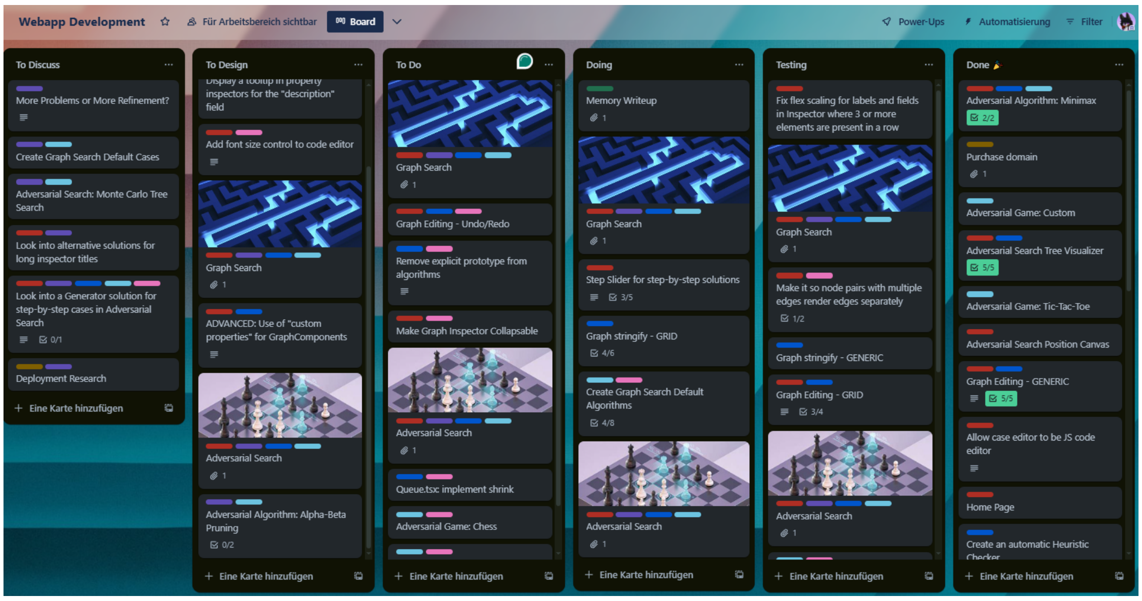


Figure 3.2: Scrum Methodology Chart

3.1 Project Management Tools

The main tool used to keep track of project items is a kanban board. This board's utility is twofold. Firstly, it allows us to keep track of items that need designing, building, testing, fixing, or discussing. Secondly, it allows tutors and stakeholders to keep track of what is being done. At this stage, the experimental director is the sole stakeholder.



To Discuss
- More Problems or More Refinement?
- Create Graph Search Default Cases
- Adversarial Search: Monte Carlo Tree Search
- Look into alternative solutions for long inspector titles
- Look into a Generator solution for step-by-step cases in Adversarial Search 0/1
- Deployment Research

To Design
- Display a tooltip in property inspectors for the "description" field
- Add font size control to code editor
- Graph Search 1
- ADVANCED: Use of "custom properties" for GraphComponents
- Adversarial Search 1
- Adversarial Algorithm: Alpha-Beta Pruning 0/2

To Do
- Graph Search 1
- Graph Editing - Undo/Redo
- Remove explicit prototype from algorithms
- Make Graph Inspector Collapsable
- Adversarial Search 1
- Queue.tsx: implement shrink
- Adversarial Game: Chess

Doing
- Memory Writeup 1
- Graph Search 1
- Step Slider for step-by-step solutions 3/5
- Graph stringify - GRID 4/6
- Create Graph Search Default Algorithms 4/8
- Adversarial Search 1

Testing
- Fix flex scaling for labels and fields in Inspector where 3 or more elements are present in a row
- Graph Search 1
- Make it so node pairs with multiple edges render edges separately 1/2
- Graph stringify - GENERIC
- Graph Editing - GRID 3/4
- Adversarial Search 1

Done
- Adversarial Algorithm: Minimax 2/2
- Purchase domain 1
- Adversarial Game: Custom
- Adversarial Search Tree Visualizer 5/5
- Adversarial Game: Tic-Tac-Toe
- Adversarial Search Position Canvas
- Graph Editing - GENERIC 5/5
- Allow case editor to be JS code editor
- Home Page
- Create an automatic Heuristic Checker

Figure 3.3: Trello Board used for this project

The kanban items are divided into six different categories:

- **To Discuss**. Items that are to be discussed with project tutors or researched or that do not constitute part of a feature.

- **To Design**. Items that need to be designed for some feature or refinement.
- **To Do**. Items that have not yet been started but are part of active development and have been designed.
- **Doing**. Items that are being actively worked on or are partially completed.
- **Testing**. Items that have been completed and are being tested or are pending testing once related features are completed.
- **Done**. Items that have been completed and tested.

Furthermore, each category has a divider which splits general items or items dedicated to a specific part of the model. In this case, the project model parts are modules (Graph Search and Adversarial Search).

3.2 Version Control

For version control, Git was used. During this stage of the project, and as agreed upon with the experimental director, the project is private. Approved collaborators and tutors who have obtained access to the repository are able to see the entire commit history. If other members were to join the project, each commit would also easily indicate who is responsible for its changes.

Version control is an extremely useful feature, as it can help roll-back changes that are fatal to the application. It will also make deployment simpler. Once the project is released, it will be properly open source, which also depends on this technology. This, along licensing details, is further detailed in chapter 4.5.

CHAPTER 4

Analysis

Before starting the design process of the app, we need to come up with some conceptual basis that will help us guide the design of the product. The following sections will outline and explain what these bases are.

4.1 Software Requirements Specifications

The following is a list of formal requirements that the application must meet. It will follow the structure laid out in IEEE's Guide for Software Requirement Specifications [2].

4.1.1. Introduction to the SRS

Purpose

The purpose of this Software Requirements Specification is to describe precisely, both to developers, stakeholders, and users, what functions the product is meant to have in an unambiguous, specific, and formal way.

Intended Audience

The intended audience of this SRS is mostly the development team, as a way to provide guidance on how to design the application. Beyond that, it is useful for project overseers (such as the tutor and experimental director), as it helps ground some of the decisions taken in the project design stages.

Intended Use

During the development process, this document is meant to be used effectively as a guide for what to add to the product model and what constraints any of the developed features must abide by.

Scope

Clairvoyant is a solution for AI algorithm visualization. It allows instructors to showcase step-by-step algorithms in a quick and customizable way to students, thereby significantly cutting down on the preparation time for lessons and allowing student users to experiment with the covered material in a fun and intuitive way.

Definitions, Acronyms, and Abbreviations

This specification will use the following terms with the associated definitions:

Term	Definition
Algorithm	*Always italicized when used with this meaning*. In the context of this application, refers to a specific program that converts a *case* input of a given problem type into a solution for that *case*.
Case	*Always italicized when used with this meaning*. In the context of this application, refers to a specific input for a given problem type.
Instructor User	An expert user making use of the app with the purpose of educating others, who may or may not be users of the app.
Problem Type	Refers to a broad class of problems in the realm of artificial intelligence. Currently, problem types include Graph Search and Adversarial Search.
Student User	Any user of the app who is not using it with the capacity of instructor.
User	Someone who interacts with the application. The user may be a *student* or an *instructor*.
API	Application Programming Interface; in this case, referring to the functions exposed to the user-provided source code.
DoN	Degree of Necessity; the degree to which a specific requirement is necessary, it can be either *mandatory*, *desirable*, or *optional*.
CORS	Cross-Origin Resource Sharing
VPS	Virtual Private Server

Table 4.1: SRS Definitions and Abbreviations

4.1.2. General Description

Product Perspective

This is a standalone product and isn't part of a larger product.

Product Functions

In general, the users of this product will have the following needs for this product:

- **Obtain information on how the product works**. For the user to be able to use the product, it is critical to provide a general guide on how it is meant to be used, both in terms of the application as a whole and in terms of the exposed API.

- **Visualize graphs in a generic way**. Graphs are a core part of visualizing search algorithms. For this purpose, it must also be possible to alter the style of individual graph nodes and edges.

- **Read and edit code in an ergonomic way**. *Algorithms* will always be expressed in the form of code. So the app must provide a usable, comfortable, source code editor. Furthermore, it should be possible to safely evaluate the code the user provides.

User Characteristics

There are two major user archetypes for this product, let's call them the *Student Profile* and the *Instructor Profile*.

A user matching the student profile will usually have some basic understanding of algorithms, and should understand the structure of a graph. Furthermore, they'll have a varying understanding of the JavaScript programming language. They should also have access to a computer with the capacity to run a browser application.

A user matching the instructor profile will usually have an advanced understanding of specific AI algorithms and graphs, and should be able to understand the JavaScript code and API without much difficulty. For the sake of instruction, they should have access to a desktop or laptop computer with access to a larger display for students to see, which may often be a projector.

Furthermore, and with regards to accessibility, since this is a visualizer, both user profiles have a functional amount of vision.

General Constraints

There are largely four groups of constraints that will affect the development and function of the application.

- **Hardware Limitations and Performance (Front End)**. The application is most likely going to be handling small *cases* most of the time. However, the size of *cases* that the application will have to handle is unconstrained, as the user will have direct control over it. Given this, the standard *cases* and *algorithms* should model a reasonable amount of complexity for most systems. *Algorithms* should be as efficient as possible given their actual use in the visualization, even if this makes them less accurate with respect to their implementation in industry.

- **Hardware Limitations and Performance (Back End)**. The server should be able to run as a VPS, so, if possible, functions that require interaction with the back end server should be limited. The user-provided code should never be run on the server, which affects both the performance and safety aspects.

- **Safety**. The application should be safe with respect to the back end server. Since arbitrary code can be run, the application must make sure that the user cannot make use of HTTP requests or similar modules that could bypass CORS policy restrictions on the back end. The proposed solution for this is making user-run code have no access to imports or context variables. Additionally, all user-provided code must be run in the user's computer as opposed to the back end server.

- **Accessibility**. The application should be usable in a wide variety of screen sizes. For example, it should be possible to make code visible on a projector screen from the back of a classroom, but it should also be comfortably usable on a tablet device or smaller laptop. The product is not designed for use in smartphones, though it should be possible to use the visualization elements in landscape mode, the programming elements are not a priority for mobile devices. As such, text should be resizable, and it should be possible to resize the sections that various elements take up to make it easier to focus on one part or another of the application.

Assumptions and Dependencies

The following assumptions condition the requirements outlined in this specification. Meaning that the requirements, and overall the design aspects of the application, are written with these assumptions in mind.

- The back end server will be open to requests at all times.
- The user will be utilizing a modern browser with modern functionalities, as opposed to legacy browsers.
- Furthermore, a relatively fast internet speed and an amount of memory concordant with the use expectations of the user is assumed.

4.1.3. Specific Requirements

This section describes, in a formal way, each of the specific requirements that must be fulfilled by the application.

For ease of understanding and for the sake of readability, each type of requirement (functional, non-functional), will be divided into certain subcategories.

Prerequisites for each requirement only represent *direct* requirements. Note that this means that prerequisites are more expressive, since they represent what a function directly relies on; but not exhaustive. Prerequisites are inherently transitive, so exhaustive prerequisite lists can be obtained, if required, by following the chain of prerequisites.

Functional Requirements

Functional requirements are those that indicate an input-output constraint. Meaning, anything that indicates *what* the product should do given a specific input.

Back End Functional Requirements

ID	**FR1**
Title	Relay Default *Algorithms* and *Cases*
Description	Default *algorithms* and *cases* must be fetchable from the back end server by problem type. The server must be able to provide (1) all available *cases* and *algorithms* for a given problem type and (2) the plain-text data necessary to present, load, and execute a given *case* or *algorithm* on the front end.
Prerequisites	–
DoN	Mandatory

Table 4.2: FR1. Relay Default *Algorithms* and *Cases*

ID	**FR2**
Title	Minimum Default Inputs
Description	For every problem type, the server must provide at least 1 default *algorithm* and at least 3 default *cases* which are compatible with the algorithm.
Prerequisites	FR1
DoN	Mandatory

Table 4.3: FR2. Minimum Default Inputs

ID	**FR3**
Title	Alter Default *Algorithms* and *Cases*
Description	A designer with access to the back end server should be able to create, edit, or remove default *algorithms* and *cases*.
Prerequisites	–
DoN	Mandatory

Table 4.4: FR3. Alter Default *Algorithms* and *Cases*

General Functional Requirements

ID	**FR4**
Title	Code Editing
Description	Regardless of the problem type, the user must be able to view and edit the code that composes its active *algorithm* and (if applicable), its active *case*. This editor should be ergonomic and provide at least Syntax Highlighting.
Prerequisites	–
DoN	Mandatory

Table 4.5: FR4. Code Editing

ID	**FR5**
Title	Error Feedback
Description	Whenever an exception stemming from user code takes place, the server should employ a best-effort approach to point to the user the exact source of the error through means of a stack trace of the error. The error should also be displayed in an area relevant to the error, ideally with the code editor where the erroring code is contained.
Prerequisites	FR4
DoN	Desirable

Table 4.6: FR5. Error Feedback

ID	**FR6**
Title	Documentation
Description	The application must provide suitable, in-depth documentation on the exposed API so that a user can properly make use of visualizer methods and write their own *algorithms* and *cases*.
Prerequisites	–
DoN	Desirable

Table 4.7: FR6. Documentation

ID	**FR7**
Title	Usage Tutorial
Description	The product must provide a way for new users to understand how it functions in general. This includes selecting default *cases* or *algorithms*, running the selected *algorithm*, and interact with the visualizer.
Prerequisites	–
DoN	Desirable

Table 4.8: FR7. Usage Tutorial

ID	**FR8**
Title	Property Inspection
Description	Many of the elements used within the product will use *properties* as a way to expose internal values to the user. The application must allow the user to view and edit the values of these properties, concordant with property-specific constraints.
Prerequisites	–
DoN	Desirable

Table 4.9: FR8. Property Inspection

Graph Search Functional Requirements

ID	**FR9**
Title	Serialize or Deserialize Arbitrary Graph
Description	Graphs, or the specific *cases* for the Graph Search module, must be convertible back and forth from a human-readable plain text form and an internal data representation form. The plain text representation will be named *graph notation*.
Prerequisites	–
DoN	Mandatory

Table 4.10: FR9. Serialize or Deserialize Arbitrary Graph

ID	**FR10**
Title	Visualize Arbitrary Graph
Description	Given the data for an arbitrary graph, the application must be able to graphically represent it in a way that is intuitive to the user.
Prerequisites	FR9
DoN	Mandatory

Table 4.11: FR10. Visualize Arbitrary Graph

ID	**FR11**
Title	Visually Edit Graph
Description	Given a graphically represented graph, the user should be able to make small, cumulative alterations without directly modifying the graph notation of the case. They should be able to add or remove edges, alter properties like heuristics for nodes or weights for edges, and – in the case of generic graphs – add or remove nodes. Furthermore, this should update the loaded *case* graph notation accordingly to represent the new structure.
Prerequisites	FR8, FR9, FR10
DoN	Desirable

Table 4.12: FR11. Visually Edit Graph

ID	**FR12**
Title	Run Graph Search *Algorithm*
Description	Given valid code for a graph search algorithm, the application must be able to run this code to obtain a series of *solution steps*, which must persist locally for further exploration.
Prerequisites	FR1 or FR4, FR9
DoN	Mandatory

Table 4.13: FR12. Run Graph Search *Algorithm*

ID	**FR13**
Title	Step Through Graph Search Solution
Description	Given valid solution steps and a valid graph concordant with those solution steps, the application must be able to let the user visualize every step in that solution. To do so, it should alter the look of affected nodes or edges, and relay debug or log information from the algorithm.
Prerequisites	FR9, FR10, FR12
DoN	Mandatory

Table 4.14: FR13. Step Through Graph Search Solution

Adversarial Search Functional Requirements

ID	**FR14**
Title	Initialize Adversarial *Case*
Description	Given a valid *case* and *algorithm* for adversarial search, the user must be able to initialize the *case*, creating a tree with the initial position and setting up the *algorithm* for a step-by-step solution.
Prerequisites	FR4 or FR1
DoN	Mandatory

Table 4.15: FR14. Initialize Adversarial *Case*

ID	**FR15**
Title	Visualize Game Tree
Description	Given a valid game tree, the application must be able to create a graphical representation of a sub-tree. The sub-tree must be determined by which nodes the user wishes to collapse or expand.
Prerequisites	FR14
DoN	Mandatory

Table 4.16: FR15. Visualize Game Tree

ID	**FR16**
Title	Visualize Position
Description	Given a selected position in the game tree and a valid *render* function in the active *case*, the visualizer must render the given position along with the game tree.
Prerequisites	FR14, FR15
DoN	Mandatory

Table 4.17: FR16. Visualize Position

ID	**FR17**
Title	Run Adversarial Step
Description	Given a valid game and *algorithm*, the application must be able to run the *algorithm* step by step until it reaches an exhaustive solution or until some constraint (such as memory) prevents it from continuing.
Prerequisites	FR14
DoN	Mandatory

Table 4.18: FR17. Run Adversarial Step

Non-Functional Requirements

Non-functional requirements are those that indicate a constraint on the performance of the product. In other words, they indicate *how* the product should handle certain specific inputs.

Back End Non-Functional Requirements

ID	NFR1
Title	Bounded Time Request Responses
Description	Requests made to the server must always be answered either in constant time (where the amount of work the code does doesn't depend on the size of the request), or in bounded time (where the amount of time the server dedicates to the request has some upper bound, at which point the server drops or denies the request). A request should take no longer than $100ms$ of active processing time from the server. This is critical because problem sizes in our application are unbounded, which means that any request made to the back end where the problem is an input may lead to arbitrarily large workloads. This ensures that this does not lead to the server hanging or dropping requests from other users.
Prerequisites	–
DoN	Mandatory

Table 4.19: NFR1. Bounded Time Request Responses

ID	NFR2
Title	Efficient Algorithms and Control Yielding
Description	The default *cases* and *algorithms* stored in the back-end server must be designed to (1) avoid infinite loops without control flow yielding and (2) yield control to the application after significantly costly steps. This ensures that the application will not freeze while trying to run default code, and provides programmers with an example of good practices for our application.
Prerequisites	–
DoN	Desirable

Table 4.20: NFR2. Efficient Algorithms and Control Yielding

ID	NFR3
Title	Restricted Access
Description	Our application will be deployed on some server. For security reasons, we must ensure that the server (physical or virtual) running our application's back end is only accessible by authorized users.
Prerequisites	–
DoN	Mandatory

Table 4.21: NFR3. Restricted Access

ID	**NFR4**
Title	Server Reliability
Description	The back end system should have an uptime of at least 99.9%, excluding maintenance periods.
Prerequisites	–
DoN	Mandatory

Table 4.22: NFR4. Server Reliability

ID	**NFR5**
Title	Service Scalability
Description	The back end system should be able to handle requests from 1000 users using the application normally without impacting performance.
Prerequisites	NFR1
DoN	Mandatory

Table 4.23: NFR5. Server Reliability

User Experience and Accessibility Non-Functional Requirements

ID	**NFR6**
Title	Resizable Text
Description	All components of the application must have text that can be resized either as part of the whole page being resized or as part of an individual component with a zoom or font size feature.
Prerequisites	–
DoN	Desirable

Table 4.24: NFR6. Resizable Text

ID	**NFR7**
Title	Resizable Components
Description	Wherever one page contains various elements which are not meant to be in focus simultaneously, at least one of the conflicting elements shall be made resizable or collapsable as to allow the user to shift focus in the application physically.
Prerequisites	–
DoN	Desirable

Table 4.25: NFR7. Resizable Components

ID	**NFR8**
Title	Ergonomic Code Editing
Description	The code editor must provide at least syntax highlighting and line numbers, and ideally other common code editor features such as keyboard-based selection, search and replace by literal or regex, block collapse/expand, and error underlining.
Prerequisites	FR4
DoN	Mandatory

Table 4.26: NFR8. Ergonomic Code Editing

ID	**NFR9**
Title	Themes
Description	Users should be able to switch between dark and light themes in both the code editor and the application as a whole.
Prerequisites	FR4
DoN	Optional

Table 4.27: NFR9. Themes

ID	**NFR10**
Title	Usability and Learning Curve
Description	The application should be designed so that the average target user can learn the basic usage of the application within 5 minutes of starting to use it. A programmer trying to learn how to use the API for a specific problem type should be able to achieve it in under 1 hour.
Prerequisites	FR6, FR7
DoN	Desirable

Table 4.28: NFR10. Usability and Learning Curve

ID	**NFR11**
Title	Platform Target
Description	The application should be designed for desktop computer use, which will be the most common platform during instruction. However, it should still be usable in landscape mode on mobile devices such as tablets or cellphones, excepting programming functionality.
Prerequisites	–
DoN	Desirable

Table 4.29: NFR11. Platform Target

4.2 Requirement Analysis by Categories

While the document above is very exhaustive, it is worth considering and expanding upon many of the specific concerns that our analysis ought focus on. This will make heavy use of references to the SRS in the previous section.

4.2.1. Security Concerns

As for any application that allows the user to run any arbitrary code, it is critical that we take an in-depth look at security and how to mitigate any potential risks. This is reflected by the Bounded Time Request Responses analysis (Table 4.19, NFR1) as well as the very light responsibilities of the back-end server (Table 4.2, FR1).

Additionally, note that we won't allow any of the user-provided code to run on the server (since that is happening on the client side) or to make requests to the server (since we won't allow arbitrary package imports).

From the perspective of the safety of the user, we could interpret that our default *cases* and *algorithms* are arbitrary code being run on their machine. So, while the code that can be run from these algorithms is safe with respect to gathering and spreading information (since it does not have access to JS libraries allowing it to request certain browser interaction permissions), we should still make sure that these default inputs are alterable only by trusted personnel on the back end server (NFR3, Table 4.21). In all likelihood, the worst thing a poorly designed case or algorithm could do is freeze the user's tab or crash the application. This is reflected in NFR2 (Table 4.20).

4.2.2. Legal Analysis

This product does not store any personal data, so the legal concerns are rather minor on that front. However, given that this is an open source program, it is worth analyzing the kind of licensing that will be made available.

Most of this analysis is carried out in section 4.5.1, but from a legal perspective, we can discuss broadly the Intellectual Property repercussions and what we want of them. This analysis has two prongs, constraining allowable freedom from two directions: firstly, the continued deployment of this application on stable servers, which will incur a cost (further detailed in section 4.4); and secondly, the continued improvement of the application via open source contributions. The first will likely require some kind of long-term donation-based approach for the upkeep of servers and domains, which means our license must be chosen in such a way as to prevent others from claiming the work invested in developing this product. The latter will require a license lax enough that contributors can work freely and without fear of copyright infringement or legal retribution.

This means that we want to maintain some type of copyright, and not give this away to the public domain. But we don't want to forbid things like alteration. Overall, a license with attribution and restrictions on direct monetization seems ideal.

4.2.3. Risk Analysis

The use of this application is meant to be largely initiated by proactive, interested educators. For the success of this product, it is critical that a reasonable amount of attention is received. As further outlined in section 4.5, maintaining an open source application requires constant enrichment, proper support channels, an accessible code base, and a lot of care.

With that in mind, we aim to create a product that is truly useful, appealing, and easy to use (NFR10, Table 4.28). It must create a user base where there exist users who are dedicated and invested enough to take a look at vast amounts of code and take the time to properly understand how it works and how to improve it, to set up the initial environment, to develop, test, and debug changes, and to create a good pull request.

This is no small amount of completely voluntary, uncompensated investment. And it is a big risk that none of the users will take the time to make such contributions. Another risk is that outlined briefly in the previous point about budget to support back end servers and domains. While the initial setup cost is not particularly high, it incurs a constant cost that builds up slowly but surely over time. Donations or some other kind of monetization may be required at some point. Not being able to secure the funds required for upkeep are also a very real risk the product could face.

Given this, it would be good to set up some metrics of how well the application is dealing with these risks after it is deployed. The following is a list of metrics of success that we could use to gauge how well the application is doing and whether outreach or updates are critically necessary:

- The number of pull requests and issues created on the Open Source repository.
- The amount of funds received through direct donations or otherwise generated by the product.
- The amount of traffic generated by the application, correlating with active users.

4.3 Use Cases

With the intent to further clarify and express, in more practical terms, what the product is supposed to allow users to do; a set of use cases which depict what is most commonly done within the context of the application have been drawn up. Figure 4.1 showcases a UML Use Case Diagram with three roles. The roles here do not determine permission levels, but rather what the user is doing. Aside from the generic user, the "Programmer" role defines a user whose intent is to define their own *case* or *algorithm* for any purpose. The "Designer" role defines a user with access to the back end server who wishes to alter the design of the default algorithms.

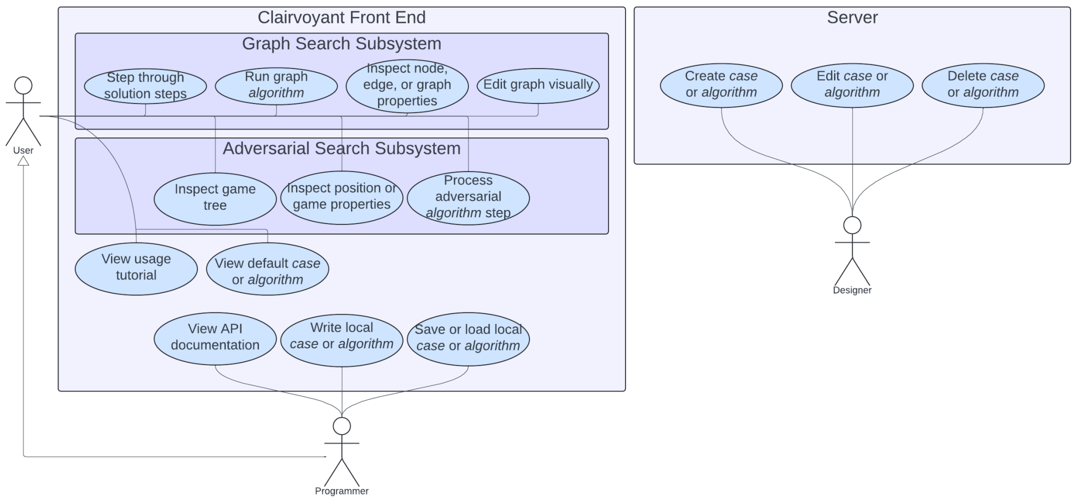

Clairvoyant Front End
Graph Search Subsystem
Step through solution steps
Run graph *algorithm*
Inspect node, edge, or graph properties
Edit graph visually
Adversarial Search Subsystem
Inspect game tree
Inspect position or game properties
Process adversarial *algorithm* step
View usage tutorial
View default *case* or *algorithm*
View API documentation
Write local *case* or *algorithm*
Save or load local *case* or *algorithm*
Server
Create *case* or *algorithm*
Edit *case* or *algorithm*
Delete *case* or *algorithm*
User
Designer
Programmer

Figure 4.1: UML Use Case Diagram

4.4 Budget

The budget for this project is largely divided in two parts, the monetary costs of developing this project and the human costs to be incurred.

4.4.1. Monetary Costs

Monetary costs of this server involve largely server and domain upkeep. All of these have no initial cost, but a compounding cost over time.

Service	Estimated Yearly Cost (EUR)
Domain (clairvoyantapp.com)	8.94
Back End Server	119.84
Total	**128.78**

Table 4.30: Budget: Monetary Costs

Note: The hosting plan we are using offers a USD 6.99/mo cost for the first two years. However, given the long-term nature of this budget, the USD 10.99/mo cost was used as reference.

The conversion rate from USD to EUR as of August 16, 2024 was used. The conversion rate was of 0.9088 EUR/USD.

4.4.2. Human Costs

Human costs refer to the time cost of development of the application. This is a very large project, and to estimate time costs, we will need to make a few decisions about the model to use.

We will use a method of estimation based around story points [3], which is suitable for agile development. Each story will receive a story size (S) and a complexity value (C) from 1 to 5. The story points that story will incur equals the product of the two values, this is a measure of Effort (Table 4.31). Given empirical data on some of these, a velocity value (in story points per human hour) will be calculated. Finally, an estimate for time cost in human hours will be produced.

Mathematically, one can express this estimation as follows:

$$T \approx \frac{E}{v} = \frac{\sum_{i=1}^{N} S_i \cdot C_i}{v}$$

Story	**S**	**C**	**E = SC**
Creation of the back-end Flask application	1	3	3
Design and Development of the application framework	3	3	9
Design of the Graph Search visualizer	1	2	2
Development of the Code Editor	2	3	6
Design of the Graph Search API	1	4	4
Development of the Graph Search Solution class	4	3	12
Design of Graph Search cases	2	2	4
Development of the Graph Search case language and system	3	2	6
Integration of Graph Search cases, solver, and visualizer	3	4	12
Implementation of default Graph Search algorithms BFS and DFS	2	1	2
Design of default Graph Search cases	2	2	4
Implementation of more complex default Graph Search algorithms like A* and UCS	2	3	6
Creation of the Documentation for the Graph Search API	2	2	4
Establishment of a standard visualizer layout and home tutorial page	3	2	6
Creation of the Adversarial Search problem page	2	2	4
Design of the Adversarial Search API	3	3	9
Design and development of the Adversarial Search Visualizer	3	2	6
Development of the Adversarial Search Game class	4	2	8
Development of the Adversarial Search Position class	2	2	4
Development of the Adversarial Search Solver class	4	3	12
Design and implementation of the default Adversarial Search case Tic-Tac-Toe	2	2	4
Development of the default Adversarial Search algorithm Minimax	3	3	9
Design and implementation of the default Adversarial Search case Custom	1	2	2
Design and implementation of various default Adversarial Search cases	3	2	6
Development of the default Adversarial Search algorithm Alpha-Beta pruning	3	4	12
(Optional) Development of the default Adversarial Search algorithm Monte Carlo Tree Search	4	4	16
Creation of the Documentation for the Adversarial Search API	2	3	6
Creation of the Documentation for the Properties API	1	1	1
Acquisition and setup of the domain	2	1	2
Deployment of the application	2	4	8
Total	–	–	**189**

Table 4.31: Effort Estimation by Stories

The velocity will be estimated as follows: given a set of sample stories S with an empirical time cost T_i and a number of story points (effort) E_i, we will calculate the average velocity in their development and the variance to get a sense of the error.

$$\hat{v} = \frac{\sum_{i=1}^{\|S\|} E_i}{\sum_{i=1}^{\|S\|} T_i} \pm \sigma_v^2 \tag{4.1}$$

$$\sigma_v^2 = \frac{\sum_{i=1}^{\|s\|} \left(\frac{E_i}{T_i} - \hat{v}\right)^2}{\|S\| - 1} \tag{4.2}$$

Using (4.1) we can readjust our time approximation to calculate $\hat{T}$. We will take into account the variance from (4.2). Let N be the number of actual stories. And E_i be their effort value (in story points).

$$\hat{T} = \frac{\sum_{i=1}^{N} E_i}{\hat{v} \pm \sigma_v^2} \tag{4.3}$$

For convenience. We can express this with the error in the numerator leveraging the following approximation, assuming $\varepsilon \ll x$.

$$\frac{1}{x \pm \varepsilon} \approx \frac{1}{x}\left(1 \mp \frac{\varepsilon}{x}\right) \tag{4.4}$$

Combining (4.3) and (4.4), we get:

$$\begin{aligned} \hat{T} &= \frac{\sum_{i=1}^{N} E_i}{\hat{v} \pm \sigma_v^2} \approx \frac{\sum_{i=1}^{N} E_i}{\hat{v}} \left(1 \mp \frac{\sigma_v^2}{\hat{v}}\right) \\ &= \frac{\sum_{i=1}^{N} E_i}{\hat{v}} \mp \frac{\sum_{i=1}^{N} E_i}{\hat{v}^2} \sigma_v^2 \end{aligned} \tag{4.5}$$

We have the following sample data:

Story	**E**	**T** (h)
Creation of the back-end Flask application	3	5.5
Development of the Code Editor	6	11
Development of the Graph Search *case* language and system	6	9
Design of Graph Search *cases*	4	6.5
Deployment of the application	8	9.5
Total	27	41.5

Table 4.32: Sample Stories for Time Estimation

Using the aforementioned formulae (4.1) and (4.2) for $\hat{v}$ and (4.5) for $\hat{T}$, we get the values:

$\hat{v}$	0.6506 ± 0.01507
$\hat{T}$	290.5 ± 6.729

Table 4.33: Values of Estimated Velocity and Time Cost

Given that value, we can estimate that it will take almost 300 hours to complete the project. This gives us a suitable margin to handle inaccuracies in our predictions or problems during development as well as dealing with tasks largely unaccounted for (such as collecting feedback, minor tweaks in the application, and writing this report). In total, this whole project (including writing the report) should easily take up to 360 hours.

4.5 Open Source

The decision to make this product open source was core to the entire endeavor. This was always planned to be an easily accessible tool which can be customized by a freely invested community.

The main goal of this project is not for personal enrichment or as a long-term monetary investment, but rather to create something that serves the community. For this purpose, a paid model makes no sense.

We must, however, not disregard the issues that stem from having a free-to-use model. After all, this product will have non-negligible maintenance costs (See section 4.4 for details).

The goal is to keep maintenance costs low and pay them through personal funding or donations.

4.5.1. Licensing

The license we choose for our product will have many ramifications, and it should not be underestimated. The purpose of a license is to clarify the rights that users are given to use our code.

It is critical that this decision be made early on in the project and before release, as changing it later can be quite problematic [14] and confusing to users.

There are, largely, four licenses commonly used at present for open source software projects [15]. Though we can also consider licenses used for all public works such as Creative Commons (CC) licenses.

One of the most restrictive, well-known open source licenses is GPL (GNU Public License). This is a copyleft license, meaning that any derivative works must also be licensed under the same license to be distributed. This is the licensed used by large non-profit open source projects such as the Linux kernel.

The MIT License, by contrast, is very permissive. It allows anyone to use, redistribute, or sell the work or derivatives thereof. It competes with the Apache 2.0 license for most used in open source in new, modern projects. The MIT license is used, for example, in the React project, which we are using (though not redistributing) for this project.

The Apache 2.0 license is another permissive license with a lot of similarities to MIT, though going into more detail on certain legal aspects like patent interactions. It also gives clearer instructions on what to do with derivative works making use of the license, wherein the specific licensed content must keep the Apache license and properly document any changes made to the original. An example user of the Apache license is the Android project.

The BSD license is very straightforward and very permissive, it simply requires that derivative work keeps the license contents and disclaimers (not necessarily the license itself) and forbids certain actions like claiming endorsements from the copyright holder for derivative works without written permission.

Lastly, Creative Commons is very often used for non-software projects. It is generally not recommended for software projects, but works well for things like multimedia content. While this is a viable option, it is not specialized for Software and better options are available.

At a first glance, the GPL license may seem like the best option. After all, we want users to share what they make using our work. However, GPL can oftentimes be very restrictive. For example, if someone wanted to use our graph implementation as a library, their whole project would also have to be licensed under the GPL. Using the lesser (LGPL) license could be a patch; however, this also introduces slight complications with monetization of our application. The initial idea is that the application stays free to use for everyone, but using GPL enforces us to never change the level of access of the product or any derivative works, so making a paid version would be impossible.

Thinking long-term, something like the MIT or Apache licenses seem like a better option, as we will be able to respond to changes more freely. It also grants users the ability to experiment with our software in their own projects without sacrificing their own choice in licensing.

The choice between Apache and MIT is complicated, but for the sake of contributors, it is probably best to pick the simplest one to use in practice. Apache goes to great lengths about patent rights, which are not necessary for the scope of this project; it also has very specific requirements. The MIT license, by contrast, is shorter and simpler. Thus, the project will go forward under the MIT license; though given its permissiveness, we could change it on later versions if we so decided, while keeping the original work licensed under MIT.

As an addendum, during the development process, a dependency to the BSD-licensed ACE Editor [24] was added. Since this is a permissive license, it does not restrict what licensing we can use, so long as the license itself is included in the documentation and source.

CHAPTER 5

Design

This chapter describes the decisions taken when going from the initial idea and requirements to a solid, fully fleshed out product design. The first thing to discuss is the overall architecture, what large-scale components our product will make use of and how they interact. The second section goes over the specific development-oriented design of certain responsibility-bearing components. Lastly, the third section documents the used frameworks and provides the reasoning for the decision to use these.

5.1 System Architecture

The full architecture for this application is relatively simple. We are using a traditional layered architecture with components on the front end. Since the product doesn't require, at least with current requirements, a dynamic database, no such layer is being used.

However, it would be possible, and quite simple, to connect a database to the application by using Python's sqlite3 package[13]. Note that this would be a local, file-based database. The reason we are using python is outlined in Section 5.3.1.

Before we get into how all components fit together, let's discuss each of the layers and what their functions are.

5.1.1. Product Layers

The product can largely be divided into two large, interacting processes. One of those is the front end and the other is the back end.

The front end is written with React (See section 5.3.2 for details) with Next.js. Next.js provides its own simple back-end for things like serving images. It will capture requests that are not directed at the API. The React front end will be working in port 3000.

The back end is written in Flask, using the Python programming language (See section 5.3.1 for details). Flask provides a development server, but this is not ideal for production for a number of reasons [26]. So instead, we'll use GUnicorn [25] [28] to create a proper, secure, WSGI (Web Server Gateway Interface) application. This application will be working in port 5000.

Beyond the front end and the back end, we have a number of services we are using to be visible on the web. Our domain was acquired through CloudFlare, which gives us access to DNS services and proxying, and our server will be hosted by Hostinger.

Requests to the server will go through a reverse proxy managed by nginx [29], any requests going to the /api region will be forwarded to port 5000, and the rest will go through React's server on port 3000. Note that nginx is, in fact, not serving any files directly, it is simply forwarding requests. The configuration file for nginx can be found in Appendix B.1.

Finally, we should also consider the user's browser, which is responsible for rendering the front end and executing the JavaScript.

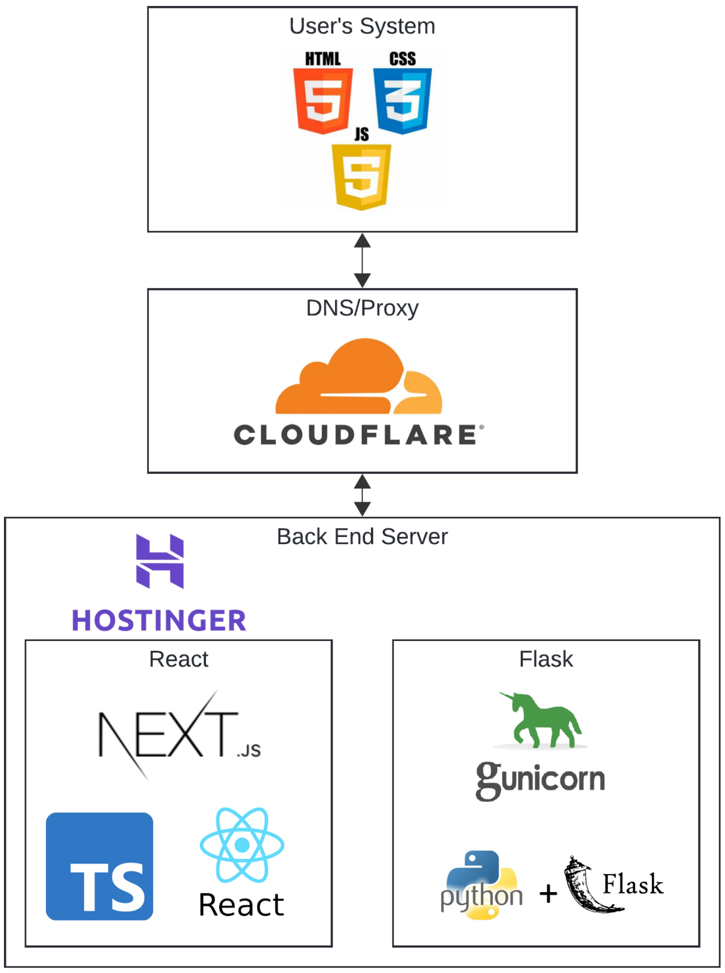


Figure 5.1: Clairvoyant Tech Stack by System

5.2 Detailed Design

The following is a deep dive into the data and class design of the application. While it is meant to be in-depth, it will go into detail into the intended functionality and design ideas rather than the strict implementation details. One notable exception is the Virtual-GraphEdge class, which poses an interesting approach to graph data design.

5.2.1. Graphs

Graphs are a fundamental part of both Graph Search and Adversarial Search. Graph Search deals with them more directly, as the *cases* are literal graphs and *algorithms* are meant to traverse them. However, Adversarial Search deals with Position trees. Where nodes are Positions and edges are actions. They are designed to work with the same broad implementation of graphs, using their data fields.

Figure 5.2 depicts the designed class diagram for Graphs and their related components, note that it omits certain internal management functions and mostly exposes functionality and relationships without going into implementation-level details. Also note that this is not static data representation meant to be used for database design, but rather the classes that are dynamically in use in the application itself, in memory. This is cemented by the use of certain JS types like Records or the dynamic "any" type.

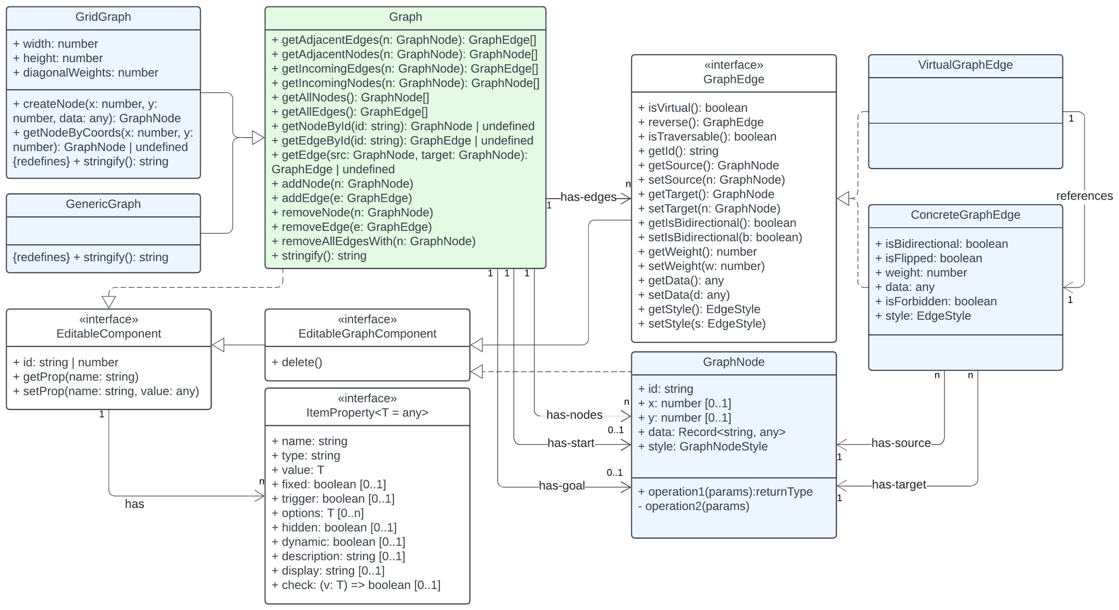

GridGraph
+ width: number
+ height: number
+ diagonalWeights: number
+ createNode(x: number, y: number, data: any): GraphNode
+ getNodeByCoords(x: number, y: number): GraphNode | undefined
{redefines} + stringify(): string
GenericGraph
{redefines} + stringify(): string
Graph
+ getAdjacentEdges(n: GraphNode): GraphEdge[]
+ getAdjacentNodes(n: GraphNode): GraphNode[]
+ getIncomingEdges(n: GraphNode): GraphEdge[]
+ getIncomingNodes(n: GraphNode): GraphNode[]
+ getAllNodes(): GraphNode[]
+ getAllEdges(): GraphEdge[]
+ getNodeById(id: string): GraphNode | undefined
+ getEdgeById(id: string): GraphEdge | undefined
+ getEdge(src: GraphNode, target: GraphNode): GraphEdge | undefined
+ addNode(n: GraphNode)
+ addEdge(e: GraphEdge)
+ removeNode(n: GraphNode)
+ removeEdge(e: GraphEdge)
+ removeAllEdgesWith(n: GraphNode)
+ stringify(): string
«interface» GraphEdge
+ isVirtual(): boolean
+ reverse(): GraphEdge
+ isTraversable(): boolean
+ getId(): string
+ getSource(): GraphNode
+ setSource(n: GraphNode)
+ getTarget(): GraphNode
+ setTarget(n: GraphNode)
+ getIsBidirectional(): boolean
+ setIsBidirectional(b: boolean)
+ getWeight(): number
+ setWeight(w: number)
+ getData(): any
+ setData(d: any)
+ getStyle(): EdgeStyle
+ setStyle(s: EdgeStyle)
VirtualGraphEdge
ConcreteGraphEdge
+ isBidirectional: boolean
+ isFlipped: boolean
+ weight: number
+ data: any
+ isForbidden: boolean
+ style: EdgeStyle
references
1
1
has-edges
1
n
«interface» EditableComponent
+ id: string | number
+ getProp(name: string)
+ setProp(name: string, value: any)
«interface» EditableGraphComponent
+ delete()
«interface» ItemProperty<T = any>
+ name: string
+ type: string
+ value: T
+ fixed: boolean [0..1]
+ trigger: boolean [0..1]
+ options: T [0..n]
+ hidden: boolean [0..1]
+ dynamic: boolean [0..1]
+ description: string [0..1]
+ display: string [0..1]
+ check: (v: T) => boolean [0..1]
has
1
n
GraphNode
+ id: string
+ x: number [0..1]
+ y: number [0..1]
+ data: Record<string, any>
+ style: GraphNodeStyle
+ operation1(params):returnType
- operation2(params)
1
1
1
has-nodes
n
has-start
0..1
has-goal
0..1
has-source
has-target
1
1
n
n

Figure 5.2: UML Class Diagram for Graphs

Note the presence of the VirtualGraphEdge class, whose only purpose is referencing a concrete GraphEdge. The `isRef()` method of GraphEdges returns true for VirtualGraphEdges and false for concrete ones. The following is an explanation on how the Graph class keeps its edges easily accessible and tractable, even for large graphs.

Graphs contain a map linking nodes to all edges whose source is that node. This is very useful in practice as getting all the neighbors of a node is a common operation, and this allows it to be executed in $O(1)$ time. However, it complicates getting the in-degree or the incoming edges of a node, which can be very useful when traversing a graph backward. Additionally, it complicates bidirectional edges. A bidirectional edge would have to become two separate edges, which could also make rendering more complicated.

The solution for this is the Virtual Edge. Virtual edges are not rendered and are always paired up with a concrete edge. They point in the opposite direction, so their source is their reference's target and vice versa. They can be traversable, but only if their referenced edge is *flipped* or *bidirectional*, and it is not *forbidden*. This logic allows us to grant the user easy modification of the graph without constantly creating and destroying objects and updating references. In fact, flipping edges, making them bidirectional, or preventing them from being traversed requires no such updates using this model. This creates a very stable environment where references to edges can be considered safe and the adjacency map doesn't need to be recalculated constantly. Instead, when adjacent nodes are requested, each of the candidate edges (some of which may be virtual), are checked for traversability and skipped if they are untraversable, this is an $O(1)$ operation for each edge.

In conclusion, the Virtual Edge makes it so edge operations are $O(1)$, as they only change local data to the concrete edge, and adjacency calculation operations are only $O(\text{in-degree} + \text{out-degree})$, which in general is very fast.

5.2.2. Graph Search

Graph search works based on an input graph, taking the role of the *case*, and an input solver, taking the role of the *algorithm*. When run, the solver is given the graph as an input to its `solve()` method, which is meant to generate a series of solution steps. Those solution steps are then internally converted to commands and stored in a command handler.

The command pattern has an *undo* and *redo* function, which allows us to execute small steps like marking a node as visited or unvisited without having to store the entire graph state for each step. When navigating to a step j from step i, steps are executed from $i + 1$ to j if $j > i$, or reverted from j to $i + 1$ if $j < i$. The execution of those commands make changes to the graph which are reflected in the GraphView component.

The Graph Notation Language

The notation for *cases* for the Graph Search problem type is what we'll call *Graph Notation*. Graph Notation is a command-based language with a relatively simple parser. It is meant to be familiar to instructors and flexible. As such, it uses syntax reminiscent of command-line interfaces.

With the exception of grid graph traversability setups, every command takes up one line. A command consists of a *verb*, a set of *options* or *flags*, some *arguments*, and a *data field*. The grammar for these commands is as follows:

$$\begin{aligned}
\langle command\rangle &:= \langle verb\rangle\ \langle arguments\rangle\ \langle options\rangle\ \langle data\rangle \\
\langle arguments\rangle &:= \langle argument\rangle\ \langle arguments\rangle \mid \epsilon \\
\langle argument\rangle &:= \langle word\rangle \mid \text{"}\langle string\rangle\text{"} \\
\langle options\rangle &:= \langle option\rangle\ \langle options\rangle \mid \epsilon \\
\langle option\rangle &:= \text{-}\langle char\rangle \mid \text{–}\langle word\rangle \\
\langle data\rangle &:= \langle \text{JSON Object}\rangle \mid \epsilon
\end{aligned}$$

Where $\langle char\rangle$ refers to a single character, $\langle word\rangle$ refers to a sequence of characters without whitespace $\langle string\rangle$ refers to any string, and $\langle\text{JSON Object}\rangle$ refers to a valid string in JavaScript Object Notation. The $\langle verb\rangle$ symbol is one of a specific subset of strings, which depends on the specific language design needs. Some possible examples of verbs are `NODE`, `EDGE`, `START`, or `GOAL`.

Every Graph Notation starts with the graph type, which currently can either be `GRID` or `GENERIC`.

In the case of "GRID" it is followed by the dimensions separated by an "x", for example, `GRID 4x10` indicates a 4 by 10 grid. Let the dimensions be $w \times h$. The following h lines are each w numbers which indicate whether the grid node at that position is traversable. 0 indicates untraversability, 1 indicates traversability. Grid graphs also support the `DIAGONAL` command, which takes one argument. If the argument is a number, the diagonalWeights property of the graph is set to that value, the constants `Manhattan`, `Euclidean`, and `Chevyshev` are also allowed as an argument.

The rest of the parsing is very similar for both graphs, the only difference is that Grid graphs disallow the creation of new nodes and node targeting is done by coordinates instead of IDs, for example, instead of targeting node `2_4` (on the 3rd row and 5th column), one would use `2 4`.

The `NODE` command creates or alters a node, it takes 1 node identifier as arguments. It merges the command's data field with the node's data.

The `EDGE` command creates or alters an edge, it takes 2 node identifiers as arguments. It merges the command's data field with the edge's data and sets the edge's bidirectional field to `true` if the `-b` flag is set.

The `START` command sets the graph's start node to the one in the argument field.

The `END` command sets the graph's goal node to the one in the argument field.

The Solver Class

The Solver class works as the base for the *algorithm* provided by the user. It provides useful methods like `visit()`, `expand()`, `log()`, or `alter()` (which works with the property subsystem).

The user is meant to implement a `constructor` (for any setup tasks, this constructor takes a reference to the Graph) and a `solve()` implementation, which is meant to execute the intended algorithm. The aforementioned methods modify an internal Command-Handler by appending commands that modify the look of the graph. For example, the `visit()` method adds a GraphCommand to the end of the Handler's queue which, on execution, colors a node yellow, and on revert, it sets its color to whatever the property held beforehand. This makes use of an internal property map which the solver provides transparently and is updated as commands are added to the queue. The UML Class diagram for the Solver is provided in Figure 5.3.

The GraphView Component

The GraphView component works as an interface between our local representation of Graphs (Figure 5.2) and the visualization system based on Vis.js's Network module[16].

We will be using a wrapper[17] of Vis.js for React. This is critical because we don't want to constantly rerender the graph, we only want to do this when React's component framework detects effectual changes that require a rerender.

The GraphView component has a number of tasks it needs to fulfill. First, and most importantly, it must provide an interactive view of the graph. To do this, it will convert the local graph into a vis.js graph, applying the necessary styles in the process.

Secondly, it must feed information back to the system about selected nodes or edges (for the sake of displaying *property inspectors*), or any interaction that may require effects such as creating new nodes or edges. The GraphView component is, in turn, the core of the user's ability to visually edit the graph.

5.2.3. Adversarial Search

Adversarial Search is more complicated to design than Graph Search for a few reasons. Firstly, the problem size is massive, and often infinite. The total number of possible play sequences for a given game is often astronomical, even Tic-Tac-Toe gets too large to properly fully visualize. Secondly, the search process generates a lot of metadata, such as utility values, alpha, beta, total matches, etc; depending on the *algorithm* being run.

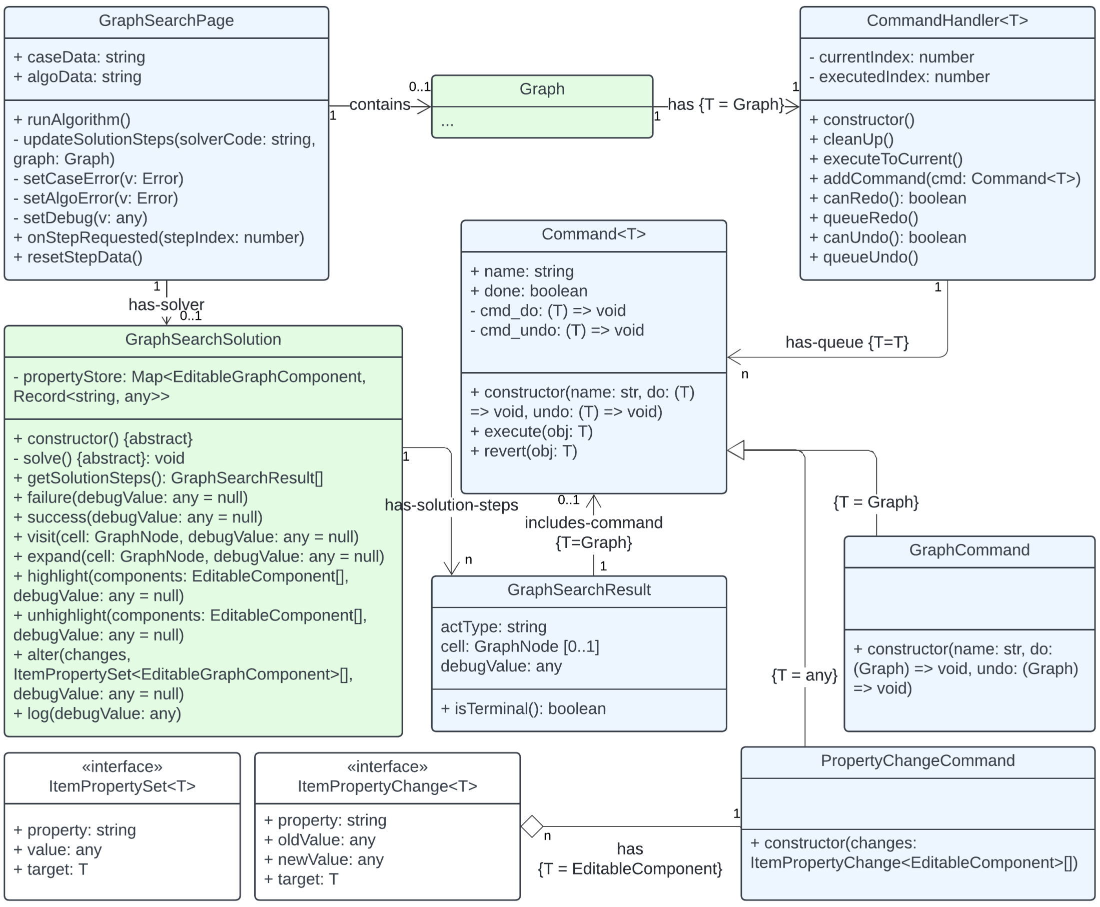

GraphSearchPage
+ caseData: string
+ algoData: string
+ runAlgorithm()
- updateSolutionSteps(solverCode: string, graph: Graph)
- setCaseError(v: Error)
- setAlgoError(v: Error)
- setDebug(v: any)
+ onStepRequested(stepIndex: number)
+ resetStepData()
contains
1
0..1
Graph
...
has {T = Graph}
1
1
CommandHandler<T>
- currentIndex: number
- executedIndex: number
+ constructor()
+ cleanUp()
+ executeToCurrent()
+ addCommand(cmd: Command<T>)
+ canRedo(): boolean
+ queueRedo()
+ canUndo(): boolean
+ queueUndo()
has-solver
1
0..1
GraphSearchSolution
- propertyStore: Map<EditableGraphComponent, Record<string, any>>
+ constructor() {abstract}
- solve() {abstract}: void
+ getSolutionSteps(): GraphSearchResult[]
+ failure(debugValue: any = null)
+ success(debugValue: any = null)
+ visit(cell: GraphNode, debugValue: any = null)
+ expand(cell: GraphNode, debugValue: any = null)
+ highlight(components: EditableComponent[], debugValue: any = null)
+ unhighlight(components: EditableComponent[], debugValue: any = null)
+ alter(changes, ItemPropertySet<EditableGraphComponent>[], debugValue: any = null)
+ log(debugValue: any)
Command<T>
+ name: string
+ done: boolean
- cmd_do: (T) => void
- cmd_undo: (T) => void
+ constructor(name: str, do: (T) => void, undo: (T) => void)
+ execute(obj: T)
+ revert(obj: T)
has-queue {T=T}
1
n
has-solution-steps
1
n
includes-command {T=Graph}
0..1
1
GraphSearchResult
actType: string
cell: GraphNode [0..1]
debugValue: any
+ isTerminal(): boolean
{T = Graph}
GraphCommand
+ constructor(name: str, do: (Graph) => void, undo: (Graph) => void)
{T = any}
PropertyChangeCommand
+ constructor(changes: ItemPropertyChange<EditableComponent>[])
«interface» ItemPropertySet<T>
+ property: string
+ value: any
+ target: T
«interface» ItemPropertyChange<T>
+ property: string
+ oldValue: any
+ newValue: any
+ target: T
has {T = EditableComponent}
n
1

Figure 5.3: Graph Search UML Class Diagram

Another challenge we will need to tackle is that visualization makes the usual approach for Adversarial Search (Iterative Deepening), not as effective, as we need to keep all positions in memory to represent them in a tree. This makes the problem size an even larger issue.

The approach we will use to keep algorithms from running forever is limiting the amount of positions to be explored. To do this, we will allow the implementation of two functions, one which can expand nodes, and one which can't and simply runs the *algorithm* on the existing nodes.

Both of those functions can be very long-running, so we will need to use a system that allows us to surrender control back to the application to process events and prevent the tab from crashing. We can use JavaScript Generators for this purpose. Each of these functions will be intended to run as a generator which regularly `yields` back to the Adversarial Search page. We will use this mechanism to limit the amount of time spent on processing.

The user will have to provide code for a *case* (derived from GameBase) and an *algorithm* class, derived from AdversarialSearchSolution. The game will, in turn, generate an initial position, which will be derived from the BasePosition class. This initial position will be the root of the game tree.

With those details out of the way, the intended design for this problem type is to have an expansion generator and an algorithm step generator, the user can run either either step-by-step or automatically as fast as possible. The expansion generator will cause the tree to have new positions, where the algorithm step generator will cause the *algorithm* to make progress by calculating node utilities.

The Game Class

Let's consider the Game Class to be the class of the object resulting from the execution of the *case* for Adversarial Search. The Game class requires the following methods to be implemented by the user, these methods are abstract in the base class.

- `getInitialPosition(): AdversarialSearchPosition`: This function should return a reference to the initial position of the game, this will function as the root of the game tree and also allow the preliminal tests to check that the derived position class implements the required methods.

- `getActions(pos: AdversarialSearchPosition): Action[]`: This function returns all possible actions resulting from a given position. It is up to the programmer to decide how to represent actions for a given game, but at least a `name` field is required to differentiate the actions in the game tree.

- `getResult(pos: AdversarialSearchPosition, action: Action`: AdversarialSearchPosition: This function returns the resulting position from taking a given action from a source position.

  The Action interface is a class with a string field "name", an optional string field "display", and a free [key=string]: any field that allows it to have any other attributes at the programmer's whim.

The game class must be derived from the base class AdversarialSearchCase, the base class provides no helper methods but it implements the EditableComponent interface to utilize the property system. A user can optionally override the `get properties()` getter, along with `setProp`, to allow the user to change game properties like player piece colors using a convenient inspector.

The Position Class

The position class must be derived from AdversarialSearchPosition; it is the class that holds information about each individual position, and one that will likely have hundreds of instances as expansions are run. The position class is responsible for providing not only a way to be rendered, but also must inform a lot of the logic on how to handle the position. It must provide whether it is terminal, what its score is if so, whose player's turn it is (if any, as there is a chance positions exist with random outcomes), and must hold all data required for the Game class to get any possible subsequent positions.

The position class must implement the following abstract methods:

- `isTerminal(): boolean`; returns true if the node is terminal, i.e. no moves can be played and the result of the game is determined.
- `getScore(): number`; returns a value from -1 to 1 if the node is terminal; the higher it is, the better the result is for the first player.
- `render(ctx: CanvasRenderingContext2D): void`; given the canvas context, draws the position graphically.
- `getId(): string`; returns a unique ID for the given position. Equivalent positions should return the same ID, this allows the tree to detect convergent play sequences and avoids unnecessary recalculations and re-expansions.
- `getPlayer(): number`; returns -1 if it's the minimizing player's turn, 0 if it's a random step turn, and 1 if it is the maximizing player's turn.

It can also optionally provide a `getHeuristic(): number` method, which can be used by algorithms to calculate a heuristic utility for non-terminal nodes.

To aid the user in rendering, the position class provides a reference to the game class, in case any properties need to be accessed, and the following method:

- drawHelper(ctx: CanvasRenderingContext2D): provides a helper which can perform some common rendering operations like drawing grids, lines, circumferences, or text without having to use the minimal rendering context methods.

The AdversarialSearchPosition class also implements the EditableComponent interface, so, much like with the game class, a user can define their own properties, which will be inspectable from an inspector window in the TreeView when a node is selected.

The Solver Class

The solver class must be derived from AdversarialSearchSolution; it is the class that holds information about how to run the *algorithm* and keeps a lot of the internal data needed for visualization. A core example is the *game tree*, which is a Graph where all nodes have a `position` field in their data pointing to a position, and all edges have an `action` field in theirs.

The solver class must implement the following abstract methods:

- `runExpansion()`, which must return a Generator of Expansions, in turn, Expansions can and should be obtained from the `expand` method in the base class.
- `runAlgorithm()`, which must return a Generator of AlgoSteps, in turn, AlgoSteps can and should be obtained from the `algoStep` method in the base class.

The AdversarialSearchSolution base class provides the following helper methods:

- `expand(position: AdversarialSearchPosition): Expansion`, tries to expand `position` and populates its internal `moves` array. It uses the internal game tree as a cache and generates any yet-ungenerated actions and resulting positions as edges and nodes respectively. Decreases the expansion budget.
- `algoStep(debugValue: any = undefined): AlgoStep`, decreases the algorithm step budget.

The return value of these methods is used by the graph search visualizer to interpret what has happened and what it must render.

The TreeView Component

The TreeView component works in a very similar way to the GraphView component. The main difference (along with some minor differences with how inspectors are managed, since these are now targeting the "position" field in data rather than the node themselves), is that it has a "collapsed" ID `Set<string>`. Whenever this set is altered, the TreeView starts at the root of the tree and starts a depth-first search with a visitation list. On each iteration, it adds the node it is traversing to the vis.js graph object for rendering. Whenever it finds a node whose ID is in the collapsed set, it backtracks instead of expanding further.

In terms of styling, it also changes quite significantly. Score is used as a color, negative values are rendered as red, and positive as green, linearly interpolated. This becomes the fill of terminal nodes. After the algorithm is run, all nodes with a "utility" value use that utility color as a thick border.

Terminal nodes are rendered as circles. Non-terminal nodes are rendered with different shapes based on the active player (up arrows for the maximizer, down arrows for the minimizer), chance nodes are drawn as circles as well.

Lastly, actions are drawn in green for actions taken by the maximizer and red for the minimizer, with a pastel color for suboptimal moves and a saturated color for those considered "best moves" by the algorithm. This is taken from the `bestMoves: Action[]` field in the position, where the action name is used for comparison.

(a) C# with ASP.NET **(b)** Python with Flask

(c) JavaScript with Node.js **(d)** PHP

Figure 5.4: Back End Technology Options

5.3 Technologies

This section will not go in-depth into how the technologies were used, but how and why they were chosen. Their specific usage in practice will be discussed in Chapter 6.

There are three main goals we are trying to pursue when making these decisions. Firstly, we want to use technology that the team is familiar with, so as to reduce the necessity to learn new technologies or languages. Secondly, we want to ensure that potential future collaborators are also comfortable with this technology; to achieve this, our choices should be among well-known and supported technologies. Lastly, we want to use technology that is easy to use, fast to develop with, and provides a good level of maintainability and support; both external (in the form of third-party packages or plug-ins) and internal (in the form of built-in functionalities or official packages).

5.3.1. Back End Framework

There are many options out there for back end frameworks. The options to be considered are laid out in Figure 5.4.

Each of them have their pros and cons. Here's a brief description of those for each technology:

ASP.NET has a wealth of support in the form of C# nuget packages and benefit from .NET's expansive ecosystem. Furthermore, its project architecture works well with very large and expandable projects. However, it requires a lot of setup and involves some significant level of boilerplate just to get it running. It is safe to say that, given the requirements of the application. Using ASP.NET would be an instance of over-engineering the back end.

Flask uses Python, which is one of the most used programming languages, especially in the realm of Artificial Intelligence. The setup for Flask is simple and straightforward

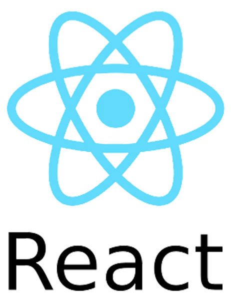
[18]. However, none of the members of the team have used it before, so it would require learning the technology first.

Node.js uses JavaScript, which is the same language used for the *cases* and *algorithms* for the application, and TypeScript is used in the front end. This makes it easier to connect the two sides into one. However, setting up a server, while still less complex than with ASP.NET, still requires more work than with Flask.

PHP is the quintessential back end scripting language. It works out-of-the-bag with a lot of webserver providers and it has a lot of functionality. However, none of the members of the team have used PHP before with this capacity and it requires extra work to connect it in the usual way to front-end web frameworks.

Overall, given the extensive knowledge of Python in the community, and the ease of development with it, the final decision on how to develop the front end was to use **Flask**. However, note that the back end is very minimal as far as the specifications outline, so it would be possible to change this later in the project life cycle if something requires it to change. Most of the responsibilities of the back end are serving files, and those files are really where the design complexity lies. The entire code for the back end in Python is in Appendix C.1.

(a) Angular



(b) React **(c)** Vue

Figure 5.5: Front End Technology Options

5.3.2. Front End Framework

Front-end frameworks are much more important in this instance. This application is going to be large in terms of front end and it will require complex functionalities that would be extremely time-consuming to refactor into a different framework.

With that in mind, this decision is critical for the development of the application, the options to be considered are depicted in Figure 5.5.

Much like with back-end options. These front-end frameworks have their own pros and cons.

Angular is a mature technology that offers much of the expected functionality of a modern front-end framework. It is also one that the team has worked with extensively before. It has a fair amount of 3rd party support such as Angular Material [19]. However, it requires a lot of files for each component and in doing so bogs down development and makes it harder to gauge interactions.

React is a powerful tool that manages to condense components into simple TypeScript (or JavaScript) functions. Over time, React has been reducing the amount of boilerplate required to develop with it. It has rich support from third party libraries such as Material UI [20] and specific ports of JavaScript graph visualization libraries like vis.js with react-vis [21], used by Uber. However, none of the team is familiar with react, so it will impose a learning curve on the project.

Vue is about as new as React and has had a lot of praise for its simplicity and performance [22]. It is better than react in a lot of aspects. However, its community is significantly smaller (Figure 5.6). This will have two downsides. The availability of 3rd party support will be more lacking, and more potential contributors would have to use a technology that they're statistically less likely to know.

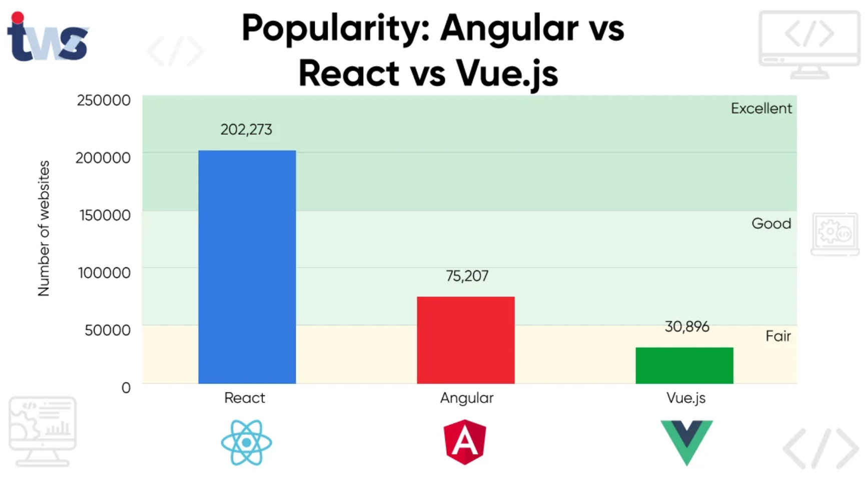


Figure 5.6: Usage Rates of Various Front End Frameworks [23]

Overall, while popularity by no means equals quality, in this case, with us having a project where people getting involved is critical, React seems to be the better choice. The added 3rd-party module and library support will also be critical in this endeavor.

CHAPTER 6

Development and Deployment

This chapter talks about the tools used for development and, in the extent that it wasn't discussed in the previous chapters, the reason for their use in development. It goes more in-depth into the practical aspect of using these tools.

Section 6.2 within this chapter explains the deployment process in detail, along with the services used to achieve a successful solution.

6.1 Development Tools

6.1.1. Visual Studio Code

Visual Studio Code (VS Code) was the IDE of choice for this project. Visual Studio Code is free and has a number of features that made development much easier. Namely; git integration, accessible project folder views, and critically, extensions.

VS Code Extensions

Extensions in VS Code are extremely powerful tools. They allow you to add functionality to an otherwise simple and lightweight tool without many built-in features. This allows one to decide what features they need and they don't.

The number of extensions used for this project is actually very high, but here are some highlights:

- Python extension package: Adds Intellisense and syntax highlighting options for python.
- HTML extension package: Adds features for creating HTML code fast, like auto-closing tags and DOM Component attribute Intellisense.
- Tailwind Intellisense: reduces the need to search the documentation and automatically detects both default and custom class names in react className property fields.

6.1.2. React

The decision on using React is explained in Section 5.3.2. In development, React is used for fast iteration on the front end. With near-instantaneous and automatic hot reloading, changes made in the project files are immediately reflected on the testing application hosted on `localhost`. Using React's development server made development much more dynamic and fast, without having to wait for builds after every change.

Tailwind CSS

Tailwind is an alternative to CSS styles. Note that CSS styles still work as normal. Tailwind, however, provides a vast library of default classes which are automatically created on project compilation and skips the need to create specialized CSS classes or inline styles, which can often be long or clash with the HTML syntax and general shape of the code. It even provides access to pseudoclasses like `hover` or media-queries like the user's preferred theme or window size with just classes.

Most importantly, Tailwind, especially with its VS Code extension, makes theming and creating the wanted visual structure much faster overall.

Alternatives to Tailwind exist; bootstrap is a relatively popular, but not nearly as flexible alternative.

6.1.3. Flask

The decision on using Flask is explained in Section 5.3.1. In development, Flask was used as a development back-end server, used without any WSGI application such as gunicorn. It works with React's front end to provide a suitable API that can be hosted right from the development computer.

6.1.4. Git and GitHub

The project management aspects of Git and GitHub are mentioned in Section 3.2. Git is also used as a development tool. With GitHub as the storage system, we can use Git to work from various computers or collaboratively. Branches can also be useful when various collaborators are working on separate features or, after deployment is complete, to have a live version in the main branch and in-development features in their own branches or in an unstable testing branch before being pushed to production.

This is also very useful for collaborators post-release; as Git is what will allow us to collaborate with open source contributors (Section 4.5).

6.2 Deployment

The deployment process started with the acquisition of a Virtual Private Server (VPS) that could be used to externally keep the necessary services running at all times. Section 6.3 goes over those.

Once a VPS was acquired, SSH was used to connect to it, using PuTTY to connect to an ssh service from a Windows device. The following is a series of tasks that needed to be completed to set up the server.

- Set up SFTP: `sudo apt-get install openssh-server`. Since we want to log into SFTP with root, we needn't set up a new user and set overrides. But this would be posssible in the `/etc/ssh/sshd_config` configuration file.
- Install Git with `sudo apt-get install git`, then configure the appropriate username and email with `git config -global user.name NAME` and `git config -global`

user.email EMAIL. Lastly, navigate to the user folder and clone the project repository. Note that this requires a GitHub access token since password authentication is no longer supported for security reasons.

- Install Python, pip, and Node.js, and nginx. All of these can be done quite easily with a sudo apt-get install command.

With that, we have the initial setup and everything we need to have installed, successfully installed. All that is left is to get the necessary services up and running to actually host the website.

As a general guide, a tutorial [25] outlining the steps of deploying a flask back-end with react was followed. Note, however, that it assumes a static deployment of the react back end, which is not the case for us, so some adjustments needed to be made, notably with the clairvoyant_server service outlined later.

6.2.1. Flask Setup

The setup for the Flask back end is rather simple, all we have to do is install the necessary pip modules (one of which is gunicorn) and then run the gunicorn module on our main flask script.

To do this, and to avoid bloating the global python instance with modules, a virtual environment was created using the venv package, so, after installing with pip install venv, creating a new environment in the same directory with python -n venv venv, and activating it with ./venv/scripts/activate, we can use the requirements.txt file to install all necessary modules in this new virtual environment.

Running pip install -r requirements.txt will install all necessary modules and also create a gunicorn executable in the venv/scripts directory. We will use this executable to define the service.

All that is left is setting up and configuring the service. The specific configuration is available in appendix B.2. It sets the service to run the gunicorn executable targeting our main.py script and listening on port 5000.

The resulting service will be named clairvoyant_api.

6.2.2. React Server Setup

By contrast, the react server setup is much simpler. All we really have to do is install all necessary node modules using our package.lock file with the npm i command, then run the production build process with npm run build. It is critical that before this step is run, the build directory is set properly in the react configuration (See appendix B.4).

Much like with the back end, we will use a service for the front end, as we are not creating a static webpage. The process for this is very similar to Flask's, except the command we run is npm run start from the front end directory. By default, this leads to the front end server running on port 3000.

The resulting service will be named clairvoyant_server.

6.2.3. Nginx

The last step is to set up a reverse proxy so that we don't need to specify a port every time we make a request. We want to use the HTTP protocol on port 80 to make all our requests. We want anything going to the `/api` region to be forwarded to Flask on port 5000, and everything else to go to port 3000 with React's front end server.

The solution for this is nginx, a reverse proxy which is standard across the industry and has recently won out over Apache in terms of usage [27]. Nginx allows us to very easily set up a reverse proxy with a very versatile configuration.

All we have to do is create a new `clairvoyant.nginx` configuration file in nginx's `sites-available` directory (See appendix B.1 for the file contents), and symlink that into the `sites-enabled` directory.

After restarting nginx with `systemd restart nginx`, the reverse proxy is up and running.

6.2.4. Updating

Pushing updates to the front end can be quite a process. To mitigate this, an auto-update script was created. This allows us to update the website and back end to the latest commit on GitHub with a single command.

The update process goes as follows:

- Pull all changes from the GitHub repository.
- Update all node packages to the required version.
- Rebuild the react front end.
- Restart clairvoyant services.

The resulting bash script file is available in appendix B.3.

6.3 Services Used

The service option used for domain acquisition and DNS management was **CloudFlare**, and in terms of a VPS hosting solution, **Hostinger** was used.

(a) Hostinger **(b)** CloudFlare

Figure 6.1: Deployment Services Used

CloudFlare offers competitive prices with many built-in features such as automatic proxying and DDoS protection.

Hostinger offers very low prices for long-term plans and grants complete control over a VPS with decent specs. Given that we are not running a traditional LAMP setup, we cannot rely on webserver solutions like cPanel. Instead, we need to connect via ssh and create our own services.

The cost of these services is specified in section 4.4.

CHAPTER 7

Testing

Testing is an essential step in the process of creating a product. Before any feature can be considered as complete, it must be thoroughly tested and checked for expected functionality.

For this project, most features were tested by hand immediately after implementation. The reason for this is twofold; since this is a visualizer, most of the features that were implemented create some kind of visual effect, which automated tests have a hard time catching the nuance of. Secondly, the back end is rather simple (in terms of functionality only, the files it serves are rather complicated), so it didn't require any specialized testing.

There are, however, notable exceptions. The graph API needs to be tested, as it is a core part of the project developed for the purpose of supporting all graphs in the application, and it contains a lot of functions that interact with one another in potentially unexpected ways. Those will be explained in Section 7.1.

7.1 Unit Testing

While we are not using a methodology that hinges around testing (unlike approaches like extreme programming do [30]), it is still important that those features that are prone to error or can easily break on modification are thoroughly tested.

Unit testing allows errors that would otherwise be hard to catch to be caught automatically. Throughout the design process of certain features, we can write tests that describe specifically the expected behaviour of what we are planning to develop.

In our case, the graph API was one of the most critical parts of the application to keep testing, as the methods kept changing and getting more complicated to accommodate the increasing complexity and requirements of the application's visualization needs and specific features for more complex *algorithms* to utilize.

To get unit tests working, we used the Jest testing framework [31]; or, more specifically, ts-jest (its TypeScript counterpart). Jest allows us to create test files anywhere in our project (either by putting them inside a `__tests__` folder or by using the `.test.ts` extension. A few illustrative examples of such tests are available in appendix C.2.1.

With a bit of configuration, we can then run the tests with a simple npm command: `npm run test`. The tests we write use jest-specific methods such as `expect`, which allows us to use matches which are much more informative when a test fails.

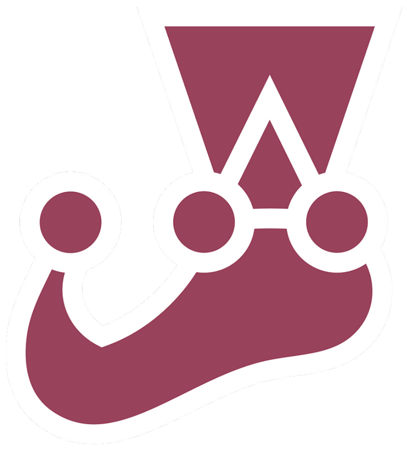


Figure 7.1: Jest Testing Framework

While the use of unit tests wasn't extreme, it is possible that as the application grows larger and more complicated, more and more tests are required.

Automated testing is also an extremely helpful tool when it comes to open source contributions. GitHub allows automated tests to run on a proposed pull request and will automatically tell us whether a given request satisfies all tests. Adding such automated testing is mentioned under further improvements in section 8.3.

7.2 Integration Testing

Integration testing is the process of testing how certain components of a large project work with other components at large. The main difference between integration testing and unit testing is that unit testing focuses on very small steps, whereas integration testing takes the application as a whole as a basis.

Integration testing has some significant limitations. Firstly, it is very complicated to design good integration tests, as there are often many moving parts, and designing a test that can properly locate the specific piece of code that is failing can often be very difficult.

Indeed, figuring out why new features were not working and debugging them was one of the most complicated parts of the project, though using an agile methodology where features were designed, implemented, and tested one at a time made it much easier to pinpoint errors rather than the alternative of testing the whole thing all at once after everything is implemented.

It is hard to specify a methodology for integration testing, as different parts of the system work in very different ways. Most of the testing was done on inbuilt *algorithms* (for graph search and adversarial search) and *cases* (specifically for adversarial search). The process was effectively running through the standard use case of the application using those *cases* and *algorithms*. One such step-by-step example for each problem type is given in section 7.3.

7.3 Example Scenarios

As part of integration testing, a full use of the application is ideal. The following is not only a test case, but also a showcase of the way the application is meant to work in a more direct and practical sense.

Since each problem type works in vastly different ways, we will showcase one use case for each of the problem types.

7.3.1. Graph Search Scenario

Our Graph Search scenario is rather simple. Our goal is to load up one of the default generic graph *cases*, and run a simple DFS algorithm. We will step through the solution steps and check which nodes are expanded and in what order before the goal is reached. Lastly, we will add some nodes and edges to the graph using the visual editor, and run DFS again.

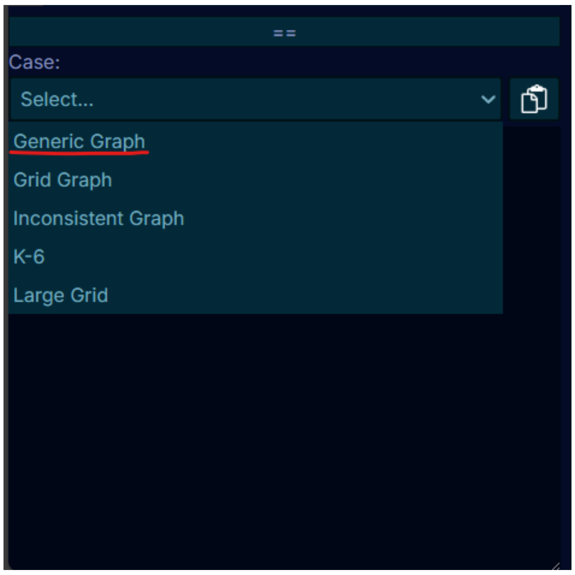
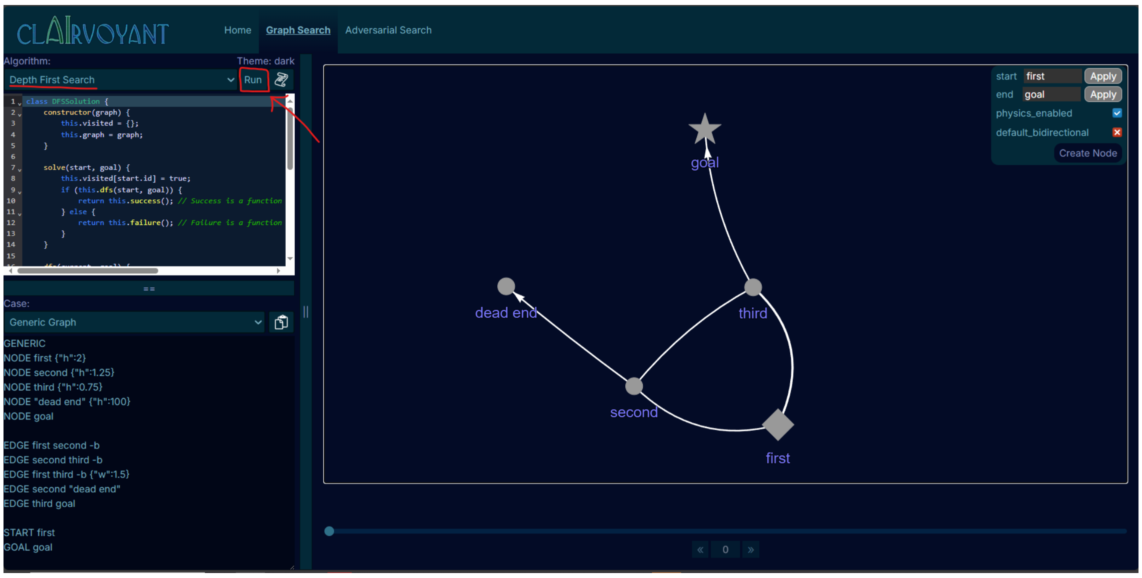


Figure 7.2: Graph Search Scenario: Step 1

Figure 7.2 shows the *case* editor with the dropdown menu open. The options here are populated from a request to the back-end server made once when the page first loads. Clicking on the highlighted "Generic Graph" option loads the default *case* and displays it on the text and visual editors.

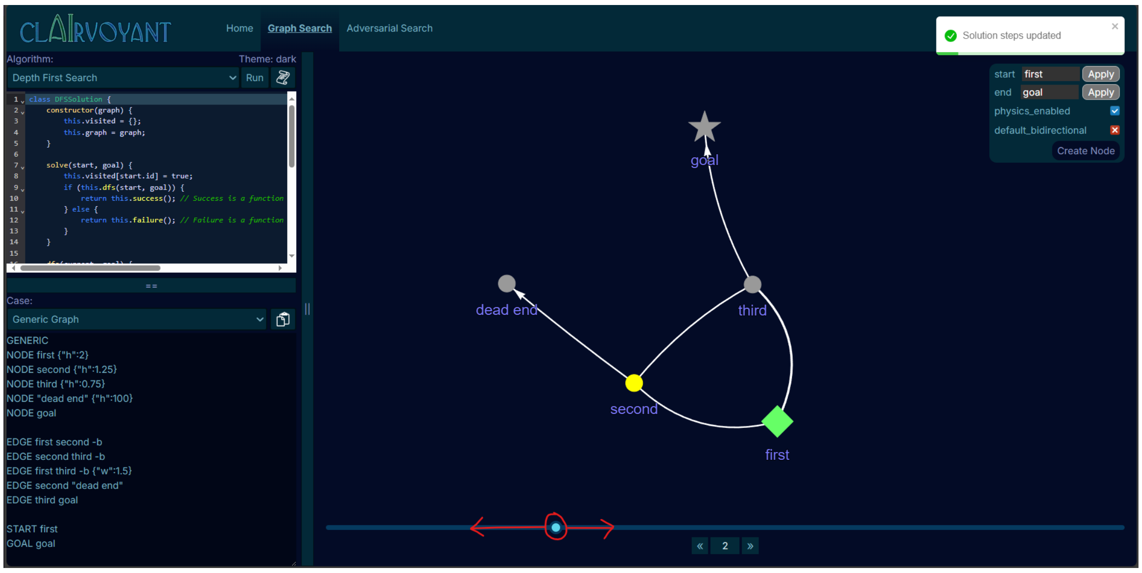


```
class DFSSolution {
    constructor(graph) {
        this.visited = {};
        this.graph = graph;
    }

    solve(start, goal) {
        this.visited[start.id] = true;
        if (this.dfs(start, goal)) {
            return this.success(); // Success is a function
        } else {
            return this.failure(); // Failure is a function
        }
    }
```

```
GENERIC
NODE first {"h":2}
NODE second {"h":1.25}
NODE third {"h":0.75}
NODE "dead end" {"h":100}
NODE goal

EDGE first second -b
EDGE second third -b
EDGE first third -b {"w":1.5}
EDGE second "dead end"
EDGE third goal

START first
GOAL goal
```

Figure 7.3: Graph Search Scenario: Step 2

Figure 7.3 shows the result of loading the first *case* and similarly selecting "Depth First Search" as the *algorithm* to run. It also shows the graphical representation of the selected default *case*.



Algorithm: Depth First Search
Theme: dark

```
class DFSSolution {
    constructor(graph) {
        this.visited = {};
        this.graph = graph;
    }

    solve(start, goal) {
        this.visited[start.id] = true;
        if (this.dfs(start, goal)) {
            return this.success(); // Success is a function
        } else {
            return this.failure(); // Failure is a function
        }
    }

```

Case: Generic Graph

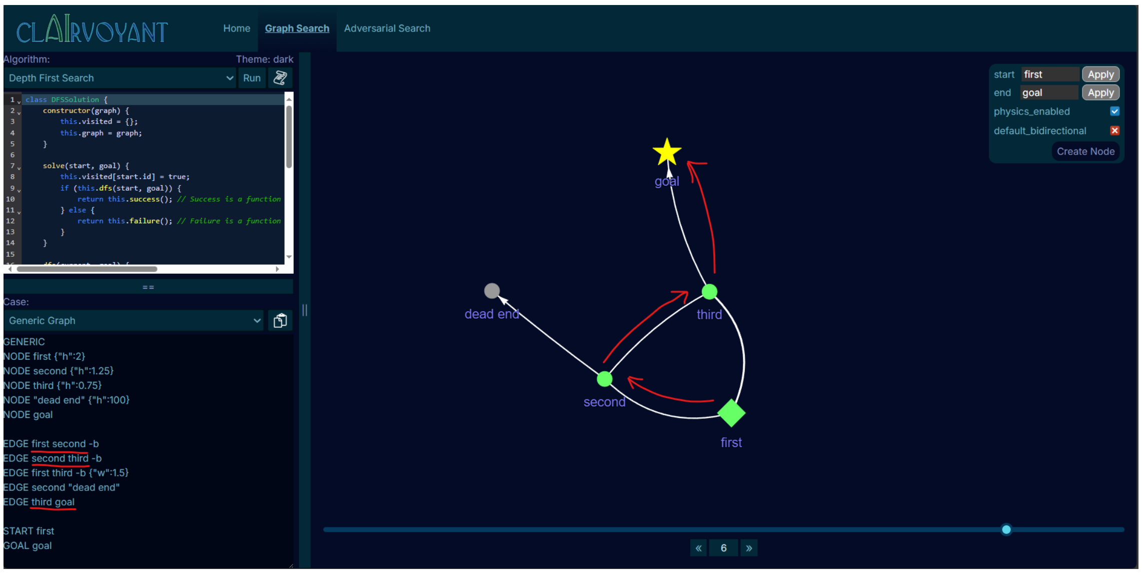
```
GENERIC
NODE first {"h":2}
NODE second {"h":1.25}
NODE third {"h":0.75}
NODE "dead end" {"h":100}
NODE goal

EDGE first second -b
EDGE second third -b
EDGE first third -b {"w":1.5}
EDGE second "dead end"
EDGE third goal

START first
GOAL goal
```

Solution steps updated

start first Apply
end goal Apply
physics_enabled ✓
default_bidirectional ✗
Create Node

Figure 7.4: Graph Search Scenario: Step 3

After pressing "Run", and moving the solution step slider to step 3, we see the result depicted in figure 7.4. The top right of the screen displays the graph inspector with general information about the given generic graph. The node labeled "first" is highlighted in green, indicating that it has been expanded. The node labeled "second" is highlighted in yellow, indicating that it has been visited.



Algorithm: Depth First Search
Theme: dark

```
class DFSSolution {
    constructor(graph) {
        this.visited = {};
        this.graph = graph;
    }

    solve(start, goal) {
        this.visited[start.id] = true;
        if (this.dfs(start, goal)) {
            return this.success(); // Success is a function
        } else {
            return this.failure(); // Failure is a function
        }
    }

```

Case: Generic Graph

```
GENERIC
NODE first {"h":2}
NODE second {"h":1.25}
NODE third {"h":0.75}
NODE "dead end" {"h":100}
NODE goal

EDGE first second -b
EDGE second third -b
EDGE first third -b {"w":1.5}
EDGE second "dead end"
EDGE third goal

START first
GOAL goal
```

start first Apply
end goal Apply
physics_enabled ✓
default_bidirectional ✗
Create Node

Figure 7.5: Graph Search Scenario: Step 4

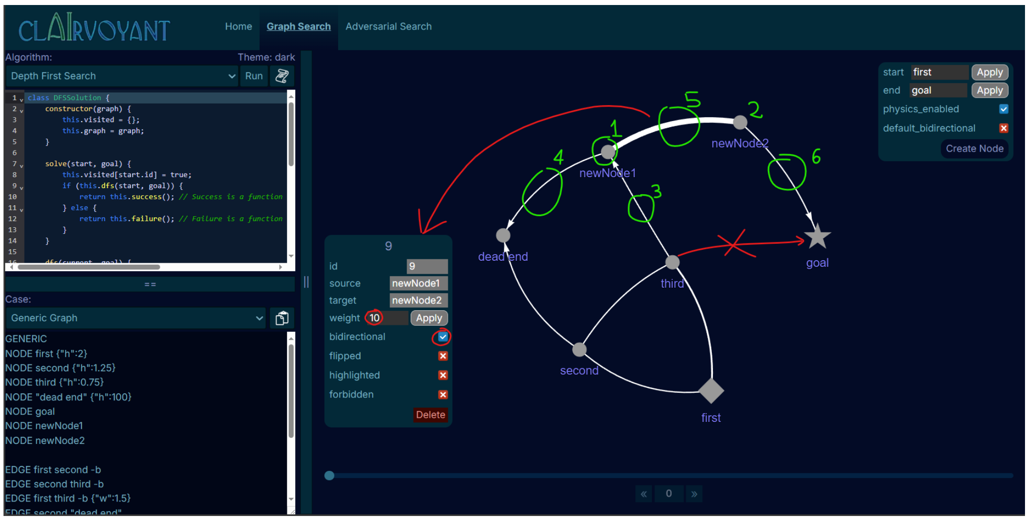
After sliding the solution slider to the second to last step (the last is simply the success state, which highlights the whole tree in blue), we can see figure 7.5. The arrows indicate the order in which each node was visited and subsequently expanded. Note that this is deterministic. The order in which the edges are declared (visible in the graph notation on the bottom left) is always the order in which they are returned from the various inbuilt graph functions.

Editing the graph is a very simple process. Let's go over the changes necessary in order to get the result visible in figure 7.6. First, the edge connecting "third" and "goal" has been severed by pressing the "Delete" button in the edge inspector.

Next, "newNode1" and "newNode2" were created. Node creation can be achieved by double-clicking on an empty space in the graph or by clicking the "Create Node" button on the graph inspector. The id is entered into a modal field.

Lastly, 3 new edges were created. First, one connecting third and newNode1, then one connecting newNode1 and "dead end", then one connecting newNode1 and newNode2, and lastly one connecting newNode2 and goal. The order in which this happened is also important; since this determines the order in which they are declared. To create an edge, one can hold click on an existing node and then drag the mouse to another existing node, then release. This, by default, creates a directed edge, though this behaviour can be changed by toggling the "default bidirectional" property on the graph inspector.

For good measure, the edge connecting newNode1 and newNode2 was altered through the edge inspector. It was made bidirectional and its weight was set to 10.



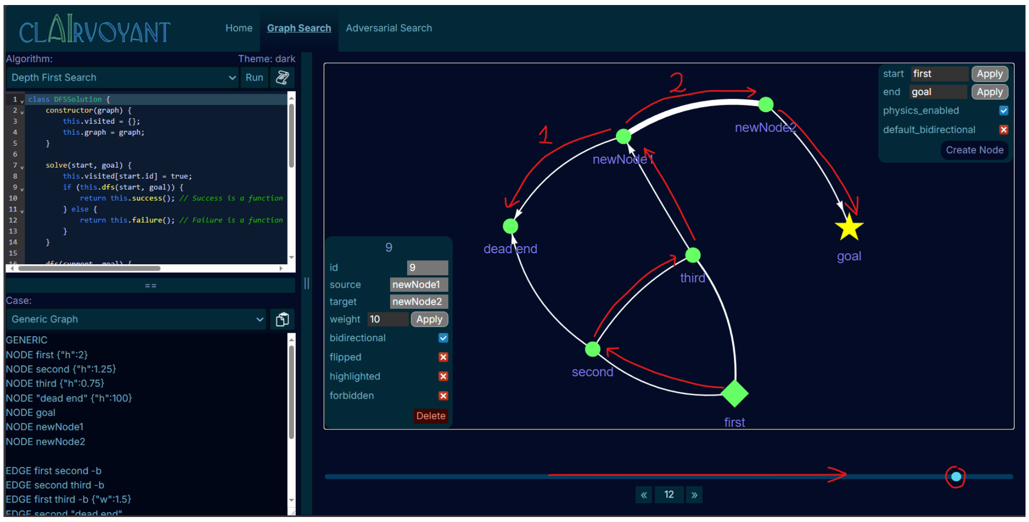
```
class DFSSolution {
    constructor(graph) {
        this.visited = {};
        this.graph = graph;
    }

    solve(start, goal) {
        this.visited[start.id] = true;
        if (this.dfs(start, goal)) {
            return this.success(); // Success is a function
        } else {
            return this.failure(); // Failure is a function
        }
    }

    dfs(current, goal) {
```

```
GENERIC
NODE first {"h":2}
NODE second {"h":1.25}
NODE third {"h":0.75}
NODE "dead end" {"h":100}
NODE goal
NODE newNode1
NODE newNode2

EDGE first second -b
EDGE second third -b
EDGE first third -b {"w":1.5}
EDGE second "dead end"
```

id	9
source	newNode1
target	newNode2
weight	10
bidirectional	✓
flipped	✗
highlighted	✗
forbidden	✗

start	first
end	goal
physics_enabled	✓
default_bidirectional	✗

Figure 7.6: Graph Search Scenario: Step 5



```
class DFSSolution {
    constructor(graph) {
        this.visited = {};
        this.graph = graph;
    }

    solve(start, goal) {
        this.visited[start.id] = true;
        if (this.dfs(start, goal)) {
            return this.success(); // Success is a function
        } else {
            return this.failure(); // Failure is a function
        }
    }

    dfs(current, goal) {
```

```
GENERIC
NODE first {"h":2}
NODE second {"h":1.25}
NODE third {"h":0.75}
NODE "dead end" {"h":100}
NODE goal
NODE newNode1
NODE newNode2

EDGE first second -b
EDGE second third -b
EDGE first third -b {"w":1.5}
EDGE second "dead end"
```

id	9
source	newNode1
target	newNode2
weight	10
bidirectional	✓
flipped	✗
highlighted	✗
forbidden	✗

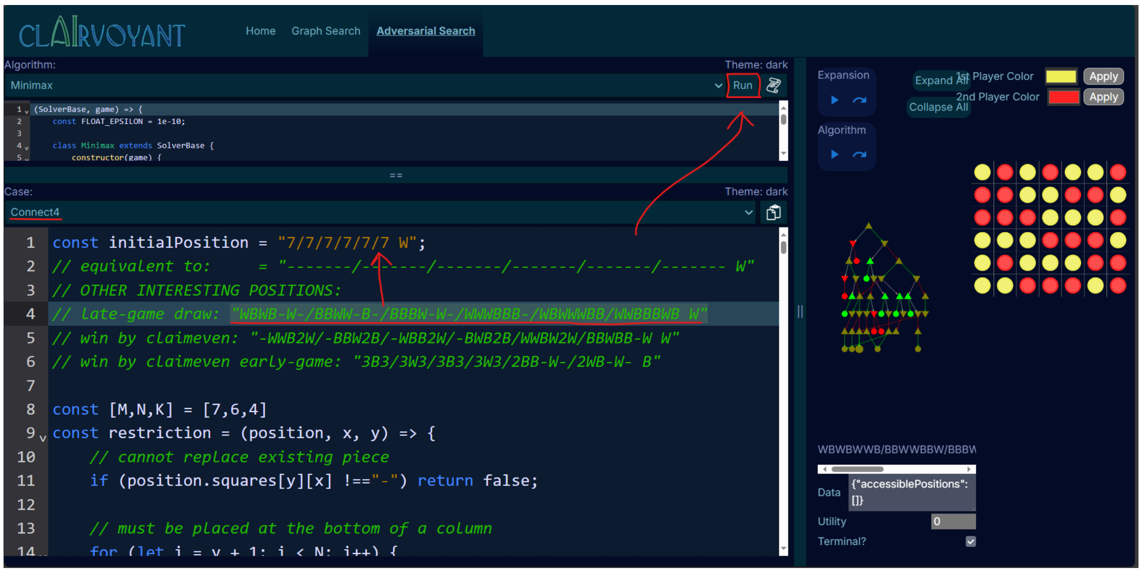
start	first
end	goal
physics_enabled	✓
default_bidirectional	✗

Figure 7.7: Graph Search Scenario: Step 6

Much like with the default case, this modified case was run through a DFS algorithm. Figure 7.7 shows the resulting visualization on the second to last step. As expected given the order in which we created the edges, "dead end" was visited before newNode2 from newNode1. The result is the least efficient possible path, reached after expanding every single node in the graph, a distinct possibility with the DFS algorithm.

7.3.2. Adversarial Search Scenario

Our Adversarial Search scenario will showcase *case* editing via the code editor. The first step will be loading up the Connect 4 default *case*. We will change the initial position to be one of a puzzle a few moves away from resolution. We will run the expansion algorithm and show all nodes to see the possible games that stem from the initial position. Finally, we will run the minimax algorithm to see if this position is a win, draw, or loss for player 1, and quickly step through an optimal move sequence.



```
(SolverBase, game) => {
    const FLOAT_EPSILON = 1e-10;

    class Minimax extends SolverBase {
        constructor(game) {
```

```
const initialPosition = "7/7/7/7/7/7 W";
// equivalent to:      = "-------/-------/-------/-------/-------/------- W"
// OTHER INTERESTING POSITIONS:
// late-game draw: "WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWWWBB/WWBBBWB W"
// win by claimeven: "-WWB2W/-BBW2B/-WBB2W/-BWB2B/WWBW2W/BBWBB-W W"
// win by claimeven early-game: "3B3/3W3/3B3/3W3/2BB-W-/2WB-W- B"

const [M,N,K] = [7,6,4]
const restriction = (position, x, y) => {
    // cannot replace existing piece
    if (position.squares[y][x] !=="-") return false;

    // must be placed at the bottom of a column
    for (let i = y + 1; i < N; i++) {
```

WBWBWWB/BBWWBBW/BBBW

Data	{"accessiblePositions": []}
Utility	0
Terminal?	✓

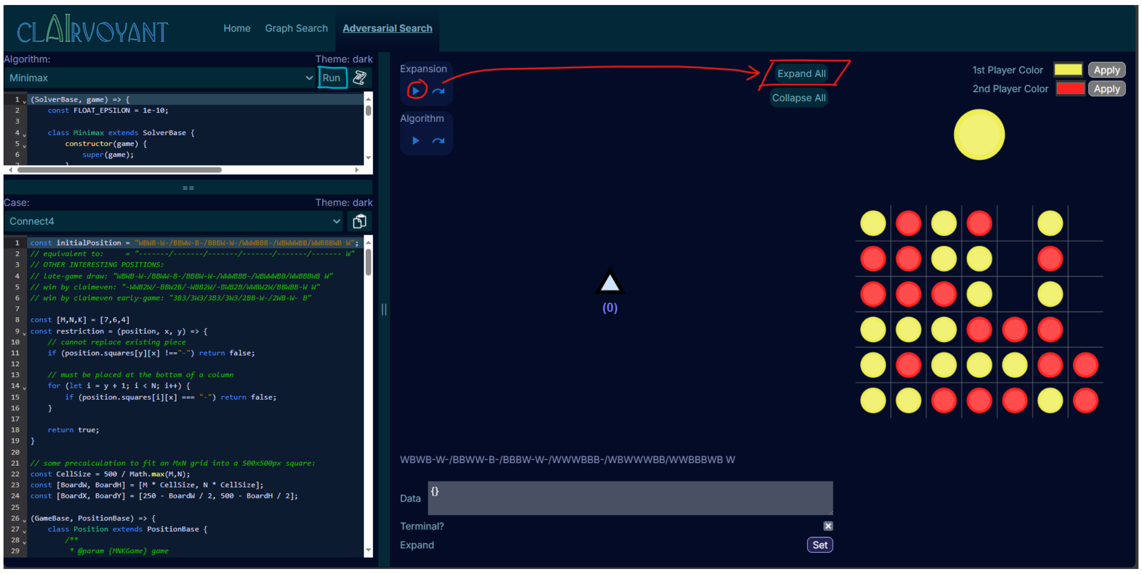
Figure 7.8: Adversarial Search Scenario: Step 1

Choosing from the default *cases* and *algorithms* is identical to Graph Search for Adversarial Search. Figure 7.8 shows the *case* editor after loading the Connect 4 default *case* and being expanded to take up a large amount of the screen using the frame resizing limiters on top and on its right side. Additionally, its font size was significantly increased using the mouse wheel while holding down the control key. This way, it is made readable. A similar technique could be used when editing live code in a classroom environment.

Before pressing "Run", a quick edit is made to change the `initialPosition` from the default empty position to one labeled "late-game draw" in a comment.

Upon pressing the "Run" button, we can see (Figure 7.9) the editor with a tree containing one single node (the starting position). Its upright triangle shape indicate that this is the maximizer's turn. A black outline indicates an unknown utility (which is concordant with the fact that the algorithm hasn't yet been run).

On the right side, we can see a visual representation of the initial position, which is selected by default. The inspector also shows some general information about the position, including its ID and an "Expand" button that can be used to manually expand the position. The rendering of the position is given by the `render` function in the *case*'s `Position` class.



```
(SolverBase, game) => {
    const FLOAT_EPSILON = 1e-10;

    class Minimax extends SolverBase {
        constructor(game) {
            super(game);
```

```
const initialPosition = "WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWWWBB/WWBBBWB W";
// equivalent to:     = "-------/-------/-------/-------/-------/------- W"
// OTHER INTERESTING POSITIONS:
// late-game draw: "WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWWWBB/WWBBBWB W"
// win by claimeven: "-WWB2W/-BBW2B/-WBB2W/-BWB2B/WWBW2W/BBWBB-W W"
// win by claimeven early-game: "3B3/3W3/3B3/3W3/2BB-W-/2WB-W- B"

const [M,N,K] = [7,6,4]
const restriction = (position, x, y) => {
    // cannot replace existing piece
    if (position.squares[y][x] !=="-") return false;

    // must be placed at the bottom of a column
    for (let i = y + 1; i < N; i++) {
        if (position.squares[i][x] === "-") return false;
    }

    return true;
}

// some precalculation to fit an MxN grid into a 500x500px square:
const CellSize = 500 / Math.max(M,N);
const [BoardW, BoardH] = [M * CellSize, N * CellSize];
const [BoardX, BoardY] = [250 - BoardW / 2, 500 - BoardH / 2];

(GameBase, PositionBase) => {
    class Position extends PositionBase {
        /**
         * @param {MNKGame} game
```

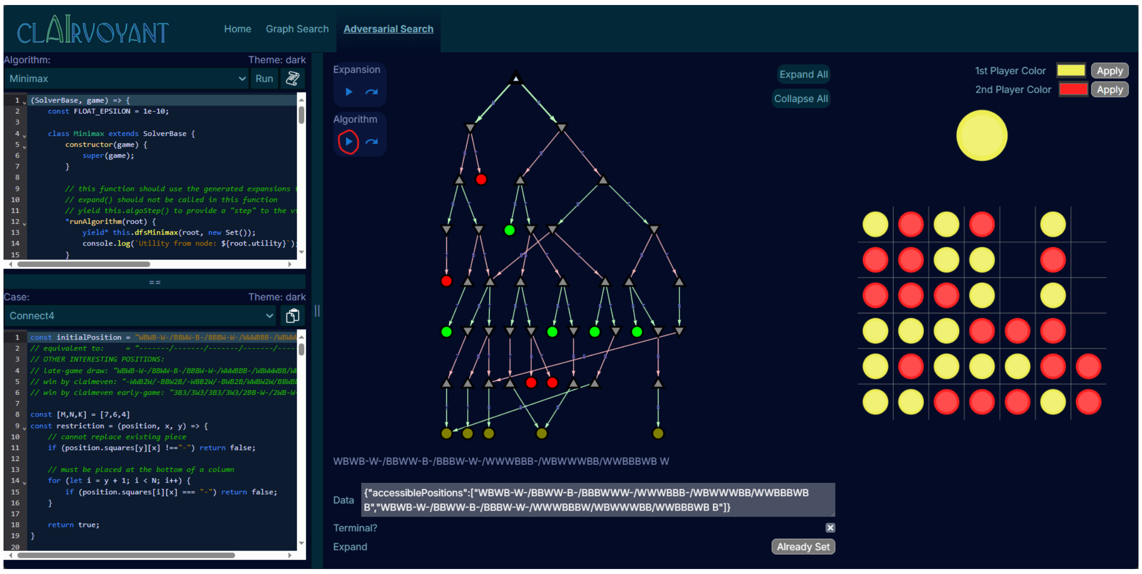
WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWWWBB/WWBBBWB W

Data {}

Figure 7.9: Adversarial Search Scenario: Step 2

Figure 7.10 showcases the result of running the expansion algorithm to completion by pressing the play button on the top left control labeled "Expansion"; then pressing the "Expand All" button on the top right to show the whole tree.

The tree shows some upright triangles, some upside down triangles (minimizer), and terminal states depicted as colored circles. The greener the color, the higher the resulting score is, red indicates the opposite, and a yellow (in the middle between the two) indicates a score closer to a draw. The outer color of each node indicates the utility, which is black because it is yet to be calculated.



```
(SolverBase, game) => {
    const FLOAT_EPSILON = 1e-10;

    class Minimax extends SolverBase {
        constructor(game) {
            super(game);
        }

        // this function should use the generated expansions
        // expand() should not be called in this function
        // yield this.algoStep() to provide a "step" to the
        *runAlgorithm(root) {
            yield* this.dfsMinimax(root, new Set());
            console.log(`Utility from node: ${root.utility}`)
        }
```

```
const initialPosition = "WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWW
// equivalent to:     = "-------/-------/-------/-------/----
// OTHER INTERESTING POSITIONS:
// late-game draw: "WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWWWBB/WW
// win by claimeven: "-WWB2W/-BBW2B/-WBB2W/-BWB2B/WWBW2W/BBWBB
// win by claimeven early-game: "3B3/3W3/3B3/3W3/2BB-W-/2WB-W-

const [M,N,K] = [7,6,4]
const restriction = (position, x, y) => {
    // cannot replace existing piece
    if (position.squares[y][x] !=="-") return false;

    // must be placed at the bottom of a column
    for (let i = y + 1; i < N; i++) {
        if (position.squares[i][x] === "-") return false;
    }

    return true;
}
```

WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWWWBB/WWBBBWB W

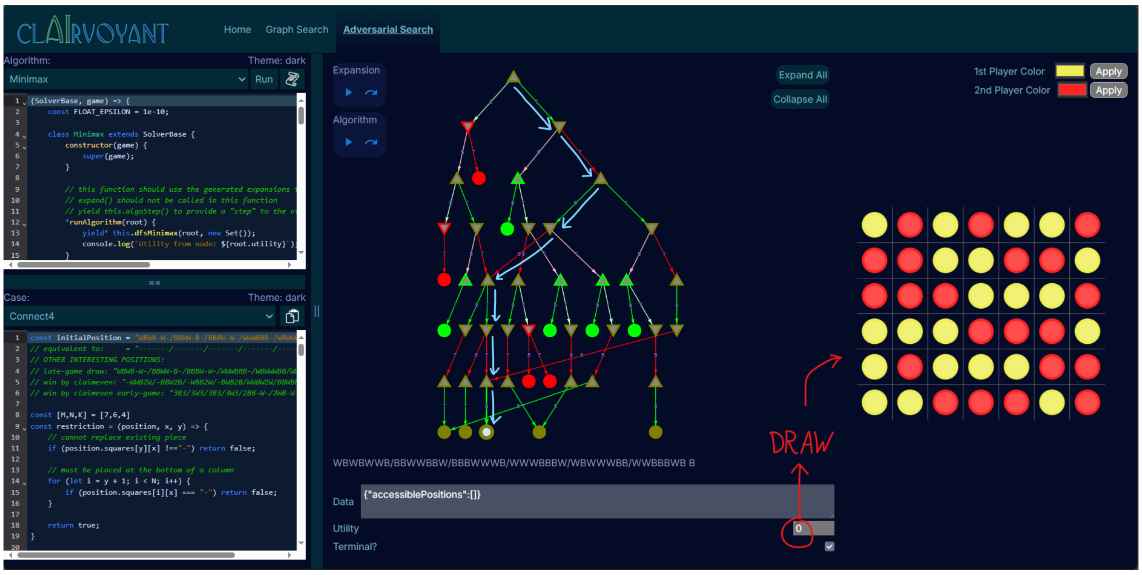
Data {"accessiblePositions":["WBWB-W-/BBWW-B-/BBBWWW-/WWWBBB-/WBWWWBB/WWBBBWB B","WBWB-W-/BBWW-B-/BBBW-W-/WWWBBBW/WBWWWBB/WWBBBWB B"]}

Figure 7.10: Adversarial Search Scenario: Step 3

To calculate it, we can run the selected *Algorithm* by pressing the play button on the Algorithm control on the top left. This results in Figure 7.11.

As we can see, the node matching the initial position has a yellow outline, indicating that, with optimal play, the result is a draw. One such sequence of moves fitting optimal play is highlighted in light blue over the tree. Note that the edges denoting these optimal moves are a darker color than suboptimal moves.

Clicking on any of the nodes in the tree renders the resulting position and shows the properties of that position in the position inspector on the bottom left. Here, the value "0" for utility on the final position indicates that this is a draw. Additionally, the checkmark on the "Terminal" property indicates that this is a terminal node.



Algorithm: Minimax — Theme: dark

```
(SolverBase, game) => {
    const FLOAT_EPSILON = 1e-10;

    class Minimax extends SolverBase {
        constructor(game) {
            super(game);
        }

        // this function should use the generated expansions t
        // expand() should not be called in this function
        // yield this.algoStep() to provide a "step" to the v
        *runAlgorithm(root) {
            yield* this.dfsMinimax(root, new Set());
            console.log(`Utility from node: ${root.utility}`);
        }
```

Case: Connect4 — Theme: dark

```
const initialPosition = "WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWW
// equivalent to:     = "-------/-------/-------/-------/----
// OTHER INTERESTING POSITIONS:
// late-game draw: "WBWB-W-/BBWW-B-/BBBW-W-/WWWBBB-/WBWWWBB/W
// win by claimeven: "-WWB2W/-BBW2B/-WBB2W/-BWB2B/WWBW2W/BBWB
// win by claimeven early-game: "3B3/3W3/3B3/3W3/2BB-W-/2WB-W-

const [M,N,K] = [7,6,4]
const restriction = (position, x, y) => {
    // cannot replace existing piece
    if (position.squares[y][x] !=="-") return false;

    // must be placed at the bottom of a column
    for (let i = y + 1; i < N; i++) {
        if (position.squares[i][x] === "-") return false;
    }

    return true;
}
```

WBWBWWB/BBWWBBW/BBBWWWB/WWWBBBW/WBWWWBB/WWBBBWB B

Data: {"accessiblePositions":[]}

Utility: 0

Terminal? ✓

Figure 7.11: Adversarial Search Scenario: Step 4

CHAPTER 8

Conclusions and Potential Improvements

Overall, the development of this application was a great success. The learned technologies and techniques will be discussed along with the related coursework in section 8.1.

Given the general feedback from the tutors overseeing this project, it seems it has potential to meet its goal and become a very useful tool in AI education.

It is important to note that this project was more than just a proof of concept. A significant investment was made in UX-oriented design, and the application was fully deployed live.

The development process was full of challenges. I had to learn React from scratch (which I had never used before), I had experience with Angular, which is similar in some ways but very different in others. I had already learned TypeScript from working in Angular before outside of University courses. React also has its own way of updating and rerendering components; which came up a lot in the form of components not refreshing when internal data changed. This was one of the most frequent problems I had to deal with.

Debugging was very challenging. Courses taken university had gone over Unit Testing and how to set those up, but converting those ideas into a mostly front-end environment was complicated. Indeed, testing was one of the parts of the project I would have liked to flesh out further.

I expected the use of and integration of external APIs such as visgraph would be more complicated, but it turned out not to be too complicated. It took a bit of trial and error to find the right library to use for a React project.

Another part I had never done before was fully deploying an application. I have dealt with managing already-deployed applications using ssh, but I had never deployed one myself. While Network courses were somewhat useful, I had to basically learn everything from scratch.

Debugging took a fair amount of time, finding the right techniques to do debugging over a front-end application was complicated; and also something that didn't really fit with the IDE-based debugging I had experienced in courses like Software Engineering.

Overall, I believe I not only learned a lot from developing this project; but I also managed to create something useful to help others learn some of the very techniques that allowed me to create this in the first place.

8.1 Aspects Related to Taken Coursework

This project has two main avenues through which it capitalized on my specific education throughout my university studies. They will be broken down into two subsections; computer science and software engineering.

Each one has a table with a breakdown on subjects and their contributions. The "Location" column in that table may reference universities. For that purpose; UPV refers to the "Polytechnic University of Valencia", also known as "Universitat Politècnica de València", specifically the Vera Campus; and NU refers to "Northeastern University", specifically the Boston Campus.

8.1.1. Computer Science

This subsection explains the way the project relates to computer science topics is in terms of what it is meant to teach. After all, the product developed here is meant to be used exactly in those kinds of classes. Table 8.1 has a breakdown by subjects on their specific contributions.

In general, the general way in which AI works, both from the more algorithm-oriented version such as that taken in Game AI, Advanced Algorithms or Data Structures and Algorithms, or in terms of how machine learning works from courses like AI or Intelligent Systems.

Code	Subject Name	Location	Contribution
11551	Data Structures and Algorithms	UPV	First introduction to mainline computer science algorithms like graph search or array sorting. Time and space complexity analysis.
11560	Intelligent Systems	UPV	AI Algorithms and the main framework for the kind of class that the product is supposed to support.
CS4100	Artificial Intelligence	NU	Similar to Intelligent Systems, but touching more on Machine Learning from a practical and technical perspective.
CS4150	Game Artificial Intelligence	NU	A look into algorithms for AI in games, crucially including pathfinding and the more practical applications of efficient graph search.
CS4800	Advanced Algorithms	NU	A highly technical look at advanced algorithms. Helpful for understanding how to break down and teach otherwise complex algorithms.

Table 8.1: Related Coursework Subject Breakdown — Computer Science

8.1.2. Software Engineering

This subsection is all about the practical aspects of designing, developing, and deploying a full application. The various subjects mentioned here are not about the elements that the application is meant to teach, but rather the aspects related to completing the project itself from the ground up. Table 8.2 has a breakdown by subjects on their specific contributions.

Code	Subject Name	Location	Contribution
11553	Computer Architecture and Engineering	UPV	Methods for managing and organizing agile software projects. Common design patterns for software applications. UML diagrams.
11554	Project Management	UPV	Methods for managing both the analysis and documentation required for large projects. Technical details such as formal requirement specifications.
11556	Human-Computer Interfaces	UPV	Theory around UX and user-oriented interface design.
12990	Computer Networks	UPV	Mechanisms of Web APIs and HTTP requests.

Table 8.2: Related Coursework Subject Breakdown – Software Engineering

8.2 Requirement Fulfillment Analysis

This section will go over all requirements, starting with mandatory functional requirements, then moving down the degree of necessity (DoN) level.

8.2.1. Functional Requirements

- FR1 is met, the code that achieves this in the back end is available in appendix C.1. This allows clients using the front end to retrieve a list of all available *algorithms* and *cases*, as well as their specific contents.
- FR2 is met, though at the moment, only one *algorithm* is completed for Adversarial Search. I would like to get Alpha-Beta Pruning, and ideally Expectiminimax and Monte Carlo Tree Search. However, those are not mandatory for the product to work, and any user could, theoretically, program those algorithms themselves.
- FR3 is met, all one needs to do is alter the files available in the `/api/problems/` subfolders. This is also automatically pulled from the GitHub repository's main branch through the auto-update handler (See Appendix B.3).
- FR4 is met, with the help of the ACE editor [24], which automatically implements all the required functionality. It would have been interesting to develop something like this by ourselves, but it would have resulted in both a less usable product and a dramatic bloat of the scope of the project. We tried our best not to reinvent the wheel on this project and instead focus on innovating and creating otherwise unavailable functionality.
- FR9 is met, some restrictive decisions needed to be made to ensure that grid graphs were intuitive to edit and still kept a simple command-based structure. Notably, the ability to remove or create new nodes or edges in grid graphs was removed. The `EDGE` and `NODE` commands therefore edit components instead of creating new ones.
- FR10 is met, using visgraph [16] [17], and converting our local graph representation (See figure 5.2) into visgraph's network object representation, we can let vis handle the rendering and user interactions.

- FR12 and FR13 are met, this functionality can be seen in action in section 7.3.1. The code is evaluated from raw JavaScript, some checks are executed to ensure that the implementation is compatible with requirements, and then the resulting class is constructed and the algorithm is run. All steps are then stored in a list which can be stepped through using something similar to the command design pattern.
- FR14 is met, in a very similar way to FR12. The only difference is the requirements for the code, the fact that both the *algorithm* and the *case* are code (which are checked sequentially), and the fact that the result is a set of class instances rather than a list of steps to move through.
- FR15 is met, in an almost identical way to graph visualization for FR10. The way position data is converted into graph node and edge styles differs significantly, and the graph is represented using visgraph's hierarchal layout, which models trees.
- FR16 is met, by passing the *case*'s `Position`'s `render()` function to a react canvas, which is rendered next to the tree visualizer.
- FR17 is met, by simply running the generator functions provided by the *algorithm* once it is run. This is also visible in action in section 7.3.2.

Those are all the mandatory requirements. The following are desirable functional requirements which are not essential to the general functionality of the application, but are good to have.

- FR5 is met, with stack trace display `<div>`s which render below the affected components. A potential improvement on this is to truncate stack traces to user code, avoiding stepping through the entire react framework function calls and providing more succinct information to the user. However, Providing a full stack trace is relatively standard, and hiding information from experienced users might also be problematic.
- FR6 is met, documentation exists under the `/docs` section. A general documentation for cross-problem APIs is available directly under that URI, and problem-specific documentation is available under `/docs/problem-type`.
- FR7 is met, the home page provides a general overview of all components and the general way to use them. However, it would be nice to have problem-specific tutorials. This could be another potential improvement for the future to improve usability.
- FR8 is met, property inspectors ended up being way more powerful and useful than I initially considered. I believe this to be a particularly strong success in this project and a feature I can see myself implementing in other projects.

This project has no requirements with a Degree of Necessity of *optional*.

8.2.2. Non-functional requirements

- NFR1 is met, as can be seen in Appendix C.1, the code for the backend is incredibly simple. Later features may require adjustments to this requirement (for example, certain requests related to accounts may require a lot of processing time in extraordinary circumstances). All current backend code runs in $O(n)$ processing time (where n is the number of *cases*, *algorithms*, or the size of a specific *case* or *algorithm* file, depending on the request).

- NFR3 is met, the back end server is protected with a strong password and is only accessible via SSH and SFTP using secrets which are not accessible anywhere in the project.
- NFR4 is met, our host is Hostinger, a professional hosting service. They guarantee 99.9% uptime, agreeing even to refund 5% of the monthly hosting fee in case of the uptime falling below this already very high threshold.
- NFR5 has not been tested, the testing in general is something that unfortunately could not have a lot of time invested in. However, given the fast response time from NFR1, the excellent performance of the server, and how infrequent requests are under normal use, it should not be an issue to handle 1000 simultaneous users. Regardless, in practice, during the beginning stages of this project, a scalability to this level is unnecessary. It might become necessary later on in the life cycle. This requirement is inherently reactive, as it depends on the actual use of the application.
- NFR8 is met in the same way as 4 is met thanks to the use of the ACE code editor [24].

The following requirements are *desirable* instead of *mandatory*.

- NFR2 is met, this is most obvious when running long adversarial search cases. The application still responds without issue, allowing users to see different positions while the *algorithm* is still running. However, the issue of infinite loops in user code is unavoidable, even if none of the default *algorithms* have this issue. It is also possible for the browser to run out of memory from running very large cases for an extended period of time.
- NFR6 and NFR7 are met, the general behaviour of page scaling handles most components. The only exception is the ACE editor, which keeps its font size. To solve this, ACE editors listen to scroll events, and if the control key is pressed, they adjust their font size in the theme.
- NFR10 is met. When the application was provided to some prospective users, along with a series of tasks to complete, they were able to easily follow the instructions without issue. Further testing is required for the learning curve in programming.
- NFR11 is met. The application is fully functional in desktop computers. It also works well on laptops with modern resolutions and specifications. The application loads in mobile devices, though some issues have been detected with tap-based inputs which may need to be refined. This is currently on the development backlog on low priority.

The only optional requirement is NFR9 on themes, this is met. Though the only way to currently switch between application-wide themes is to do so through the browser preferences, as the general theme responds to the user's `prefers-theme` media query. The theme for a given editor can be easily switched with the press of a button on the top right of the editor.

8.3 Improvements and Future Work

The following subsections outline a couple of improvements that could be achieved later on in the life cycle of the application, but which were excessive for the scope of the project.

8.3.1. Accounts and Authentication

As mentioned in section 5.3.1, the back end doesn't have a lot to do. The lack of personal accounts has a lot to do with this. Personal accounts would allow for a wide range of quality of life and user experience improvements.

Some potential ideas that could be unlocked with such a feature would be: saved *cases* and *algorithms*; community boards, forums, or help pages; and certain limited-access features such as donator-exclusive features.

The list goes on, but those are some illustrative examples of what could be achieved with this. Ideally, the account management side would be left to a third party, allowing authentication with accounts like Google or GitHub. This is both good for user experience and for security. Of course, it complicates implementation as we have to connect to third-party APIs.

8.3.2. Localization

Our analysis of the target audience (See the User Characteristics section in the SRS, Section 4.1) left out language; however, typically in the field of Computer Science, English is the driving language. While that is the case and Computer Science students and professors are largely expected to be able to use the language, localization could add a layer of personal comfort and ease of use that would otherwise be missing.

8.3.3. Responsive Design

The application was developed with computers and classroom projectors in mind. However, with significant tweaks, developing a version of the application that can run on mobile devices is an achievable goal. The complexity of the application doesn't easily lend itself to small-screen, portable devices. But with certain compromises, a decent user experience is definitely possible.

8.3.4. Automated Testing with GitHub Actions

As with any open source application (See section 4.5), dealing with pull requests can take up a significant amount of time. Bugs can easily be introduced from an open source pipeline, and they can be hard to track and fix. GitHub provides a framework for automated testing, which could be used to mitigate this problem and provide contributors immediate feedback on what is wrong. It would also allow us as maintainers to immediately weed out dangerous pull requests.

This would not only require the set up of GitHub actions for testing, but would also require the introduction of a large test suite into the project. Our testing is currently not on that scale, as is mentioned in chapter 7.

Bibliography

[1] Tore Dybå and Torgeir Dingsøyr. *Information and Software Technology*, 50:9:833-859, 2008.

[2] IEEE Guide for Software Requirements Specifications. *IEEE Std 830-1984*, 1-26, 1984.

[3] Ziauddin, Shahid Kamal Tipu, Shahrukh Zia. *Advances in Computer Science and its Applications*, 2:1:314-324, 2012

[4] Pathfindout 2D Grid Pathfinding Algorithm Visualizer. Accessible at https://pathfindout.com/

[5] Visualgo DFS and BFS Algorithm Visualizer. Accessible at https://visualgo.net/en/dfsbfs.

[6] Graph Search Visualizer on Codepen. Accessible at https://codepen.io/geekytime/pen/DENvYr

[7] Netlify Grid Graph Search Visualizer. Accessible at https://graphalgorithms.netlify.app/

[8] Brilliant.org Learning Platform. Accessible at https://brilliant.org

[9] Brilliant'org's article on the A* algorithm. Accessible at https://brilliant.org/wiki/a-star-search/

[10] Datacamp Learning Platform. Accessible at https://www.datacamp.com/

[11] Khan Academy's Computing Section. Accessible at https://www.khanacademy.org/computing

[12] Codecademy. Accessible at https://www.codecademy.com/

[13] Python's sqlite3 package. Accessible at https://docs.python.org/3/library/sqlite3.html

[14] Di Penta, Massimiliano and German, Daniel M. and Guéhéneuc, Yann-Gaël and Antoniol, Giuliano. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 1:145-154, 2010.

[15] Statista, Worldwide Leading Open Source Licenses. Accessible at https://www.statista.com/statistics/1245643/worldwide-leading-open-source-licenses/

[16] Documentation for Vis.js's Network Module. Accessible at https://visjs.github.io/vis-network/docs/network/

[17] Wokstym, React-vis-graph-wrapper on GitHub. Accessible at https://github.com/Wokstym/react-vis-graph-wrapper

[18] Flask 3.0 Documentation. Accessible at https://flask.palletsprojects.com/en/3.0.x/

[19] Angular Material. Accessible at https://material.angular.io/

[20] Material UI, Material for React. Accessible at https://mui.com/material-ui/

[21] React-vis on GitHub. Accessible at https://uber.github.io/react-vis/

[22] React vs Vue Comparison at Sitepoint. Accessible at https://www.sitepoint.com/vue-vs-react/

[23] Angular vs React vs Vue usage analysis on Reddit from 2020, by SimilarTech. Accessible at https://www.reddit.com/r/vuejs/comments/ik04dv/angular_vs_react_vs_vue_a_complete_comparison/

[24] ACE Code editor home page. Accessible at https://ace.c9.io/

[25] Miguel Grinberg, How to deploy a React Flask project. Accessible at https://blog.miguelgrinberg.com/post/how-to-deploy-a-react--flask-project

[26] Vladislav Supalov, Flask is not your production server. Accessible at https://vsupalov.com/flask-web-server-in-production/

[27] W3Tech statistics on Nginx usage. Accessible at https://w3techs.com/technologies/details/ws-nginx

[28] Green Unicorn (gunicorn) home page. Accessible at https://gunicorn.org/

[29] Nginx home page. Accessible at https://nginx.org/en/

[30] Crispin, Lisa and House, Tip. Testing Extreme Programming. Chapter 13. 2003

[31] Jest Testing Framework Documentation. Accessible at https://jestjs.io/docs/getting-started

APPENDIX A

Sustainable Development Goals

A.1 Degree of relationship of this project with Sustainable Development Goals (SDGs)

Sustainable Development Goals	High	Medium	Low	None
SDG 1. **End of poverty.**				✓
SDG 2. **Zero hunger.**				✓
SDG 3. **Health and well-being.**				✓
SDG 4. **Quality Education.**	✓			
SDG 5. **Gender Equality.**			✓	
SDG 6. **Clean water and sanitation.**				✓
SDG 7. **Affordable and non-polluting energy.**				✓
SDG 8. **Decent work and economic growth.**			✓	
SDG 9. **Industry, innovation, and infrastructure.**		✓		
SDG 10. **Reduction of inequality.**			✓	
SDG 11. **Cities and sustainable communities.**				✓
SDG 12. **Responsible production and consumption.**				✓
SDG 13. **Climate action.**				✓
SDG 14. **Marine life.**				✓
SDG 15. **Life in terrestrial ecosystems.**				✓
SDG 16. **Peace, justice, and solid institutions.**				✓
SDG 17. **Alliances to achieve goals.**				✓

A.2 Reflection on the relationship of the Capstone Project with Sustainable Development Goals

Of the sustainable development goals outlined above, quality education is the most closely related goal. This product is meant to be used primarily as an instruction and learning aide. Clairvoyant can effectively help professors prepare for classes on the topics it covers by serving as a fast and versatile framework to work off of. Similarly, it grants students a way to interact with otherwise fiddly and complex examples intuitively and interactively; with or without a professor or tutor being present.

The second most closely related is innovation; given that this is a product meant to help students better learn the fundamentals of a modern technology, we would hope that down the line, those very students can more effectively come up with innovative ideas in industry.

The rest of the relationships are arguably tenuous and stem from the core concept of accessible education. Regardless of whether one is in a university course, studying by themselves, or anywhere in between; Clairvoyant is a platform they have full access to, one they can experiment with at their leisure and with no cost or initial commitment.

Going item by item, gender equality is marked with a low degree of relationship with this project because it provides an introduction to what is a traditionally male-dominated field to anyone, regardless of gender. The general reduction of inequality point plays out with very similar logic, though focusing on the acquisitive and economic power and stressing that the product is completely free to use.

Lastly, on the topic of economic growth, it follows from all previous points that allowing anyone to access this framework would have a positive effect on their skillset and, thus, on their power to both bargain for better job accommodations or higher salaries.

APPENDIX B
System Configuration

This appendix will briefly describe various configuration options that were required to get the system running properly and efficiently.

B.1 NGINX

The NGINX reverse proxy configuration has a very simple task, it has to forward /api requests to port 5000 and any other requests to port 3000. Note that this configuration would slightly change when moving to HTTPS with encryption. Port 443 would also be in use.

```
# /etc/nginx/sites-available/clairvoyant.nginx

server {
    listen 80;
    root /home/ender/clairvoyant/ai-visualization/build;
    index index.html;

    location / {
        include proxy_params;
        proxy_pass http://localhost:3000;
    }

    location /api {
        include proxy_params;
        proxy_pass http://localhost:5000;
    }
}
```

B.2 Service Configuration

The processes hosting the front end and back end in the server are not started directly from an ssh instance, as that would be quite a fragile deployment. Instead, they must be able to restart automatically either when they crash for any reason, or when the server itself restarts.

Conveniently, Linux provides services for exactly such a purpose, through the `systemctl` command. These commands set the state of services configured in `/etc/systemd/system`, they run as background processes that start with a given command.

```
# /etc/systemd/system/clairvoyant_api.service
[Unit]
Description=A service that provides a back-end API for the
    Clairvoyant Web Application. Backed by Flask and
    GreenUnicorn.
After=network.target

[Service]
User=ender
WorkingDirectory=/home/ender/clairvoyant/api
ExecStart=/home/ender/clairvoyant/api/venv/bin/gunicorn -b
    127.0.0.1:5000 main:app
Restart=always

[Install]
WantedBy=multi-user.agent
```

```
# /etc/systemd/system/clairvoyant_server.service
[Unit]
Description=A service that runs the React Next server required
     to provide the front end elements of the Clairvoyant App
After=network.target

[Service]
User=ender
WorkingDirectory=/home/ender/clairvoyant/ai-visualization
ExecStart=npm run start
Restart=always

[Install]
WantedBy=multi-user.agent
```

B.3 Auto-updating

For convenience, and to ease the process of updating from the back-end, an auto-updating script was created and placed by the local application directory. This script automatically pulls the latest changes from github, rebuilds the application, and restarts the relevant services. All one must do to run it is run `~/clairvoyant_update.sh`.

```
# /home/ender/clairvoyant_update.sh

# Pull all relevant changes
cd /home/ender/clairvoyant
git pull

# Ensure node modules are up to date
cd ./ai-visualization
npm i

# Rebuild front end application
npm run build

# Restart services with new files
systemctl restart clairvoyant_server
systemctl restart clairvoyant_api
```

B.4 React Configuration

To properly get the previous services to work, it was important to set some minimal but critical configuration in the react project. The `distDir` option sets the build directory, which is critical for our services to work, as they need to know where the build is located, and this can't change between builds.

The `images` options is simply there to ensure that we can load external resources properly from the React server and that webp is accepted as an image format and properly recognized as an internal resource.

Lastly, an option that isn't present, is the `output` option, which allows the `export` value. This, instead of creating the output of a react server started with `npm run start`, it configured react in such a way that building produces a completely static site. This imposes some limitations on the react side, as it forbids use of react-server-specific tools like `getServerSideProps`, for example. While we briefly experimented with this option, the default server-form export was chosen.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
    distDir: 'build',
    images: {
        domains: [],
        formats: ['image/webp'],
    }
}

module.exports = nextConfig
```

APPENDIX C
Code

This appendix contains extensive chunks of code that are considered relevant for understanding certain aspects of the scope and design of the application.

C.1 Back End

The entire back end code is rather brief; as was mentioned in previous sections. The flask application is meant to be expandable (See section 8.3), but it doesn't have many responsibilities at the moment. All it needs to do is serve files and provide a list of available files. The files themselves are technically part of the back end and show a more significant level of complexity.

```python
# api/main.py
from flask import Flask, json, jsonify

from os import listdir, path

app = Flask(__name__)

def OK(data):
    response = jsonify(data)
    response.status_code = 200
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response

def NotFound(data):
    response = jsonify(data)
    response.status_code = 404
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response

@app.route("/api/v1/<problem>/algorithms", methods=["GET"])
def get_algorithms(problem: str):
    directory = path.join("./problems", problem, "algorithms")
    if not path.exists(directory): return NotFound("Problem
       name not found.")
    result = []
    for file in listdir(directory):
        if file.endswith(".js"): result.append(file[:-len(".js")])
    return OK(result)

@app.route("/api/v1/<problem>/cases", methods=["GET"])
def get_cases(problem: str):
    directory = path.join("./problems", problem, "cases")
    if not path.exists(directory): return NotFound("Problem name not found.")
    result = []
    for file in listdir(directory):
        if file.endswith(".txt"): result.append(file[:-len(".txt")])
        elif file.endswith(".js"): result.append(file[:-len(".js")])
    print(f"returning {json.dumps(result)}")
    return OK(result)

@app.route("/api/v1/<problem>/algorithms/<algorithm>", methods=["GET"])
def get_algorithm(problem: str, algorithm: str):
    file = path.join("./problems", problem, "algorithms", f"{algorithm}.js")
    if not path.exists(file): return NotFound("Problem or algorithm name not found.")
    with open(file, "r") as f:
        content = f.readlines()
    return OK("".join(content))

@app.route("/api/v1/<problem>/cases/<case>", methods=["GET"])
def get_case(problem: str, case: str):
    for ext in [".txt", ".js"]:
        file = path.join("./problems", problem, "cases", f"{case}{ext}")
        if path.exists(file): break
    else:
        return NotFound("Problem or algorithm name not found.")
    with open(file, "r") as f:
        content = f.readlines()
    return OK("".join(content))

if __name__ == '__main__':
    app.run("localhost", 5000)
```

C.2 Front End

The code base for the front end is massive and it would be unreasonable to include the entire base here, however, a few highlights have been made to showcase some of the more interesting programming interactions in the application.

C.2.1. Front End Tests

```
import {expect, jest, test} from '@jest/globals';
import { GenericGraph, Graph, GraphEdgeSimple, GraphNode,
   GridGraph } from '../graph';

// test parsing
test("Generic Parsing", () => {
    const graph = GenericGraph.fromNotation(`GENERIC
        NODE A {"a": "hello world"}
        NODE B
        NODE C
        NODE D

        EDGE A B {"w": 2, "edgeTest": true}
        EDGE B C -b
        EDGE B D

        START A
        GOAL C

        # Extra edges
        EDGE B D -b {"w": 3, "secondary": true}

        # Data!
        NODE A {"x": 1}
    `);
    // basic contents
    expect(graph.getAllNodes()).toHaveLength(4);
    expect(graph.getAllEdges()).toHaveLength(4);
    // node fetching
    let nodeA = graph.getNodeById("A");
    let nodeB = graph.getNodeById("B");
    let nodeC = graph.getNodeById("C");
    let nodeD = graph.getNodeById("D");
    expect(nodeA).toBeDefined();
    expect(nodeB).toBeDefined();
    expect(nodeC).toBeDefined();
    expect(nodeD).toBeDefined();
    // edges
    let edgeBC = graph.getEdge(nodeB!, nodeC!);
    expect(edgeBC).toBeDefined();
    expect(edgeBC!.isBidirectional).toBeTruthy();
    let edgeBA = graph.getEdge(nodeB!, nodeA!, true);
    expect(edgeBA).toBeDefined();
    expect(edgeBA!.traversable()).toBeFalsy();
    expect([...graph.getEdges(nodeB!, nodeD!)]).toHaveLength(2);
    // start, end, and referential equality
    expect(Object.is(graph.startNode, nodeA)).toBe(true);
    expect(Object.is(graph.endNode, nodeC)).toBe(true);
    // data
    expect(nodeA?.data["a"]).toEqual("hello world");
    expect(nodeA?.data["x"]).toEqual(1);
    expect(edgeBA!.data["edgeTest"]).toEqual(true); // note that it is the opposite orientation as that given in the parse!
    expect(edgeBA!.weight).toEqual(2);
})

test("Grid Parsing", () => {
    const graph = GridGraph.fromNotation(`GRID 3x2
        1 0 1
        1 1 1
        DIAGONAL MANHATTAN
        START 0 0
        GOAL 2 0

        # Merges data to existing components
        NODE 0 0 {"test": "isStart"}
        EDGE 2 1 2 0 {"w": 3.5}
    `);

    expect(graph.getAllNodes()).toHaveLength(6);
    expect(graph.getAllEdges()).toHaveLength(11); // 7 orthogonal + 4 diagonal

    let nodeTL = graph.getNodeByCoords(0, 0);
    let nodeTC = graph.getNodeByCoords(1, 0);
    let nodeTR = graph.getNodeByCoords(2, 0);
    let nodeBL = graph.getNodeByCoords(0, 1);
    let nodeBC = graph.getNodeByCoords(1, 1);
    let nodeBR = graph.getNodeByCoords(2, 1);
    [nodeTL, nodeTC, nodeTR, nodeBL, nodeBC, nodeBR].forEach(n => {
        expect(n).toBeDefined();
    })

    // traversability
    expect(nodeTL!.traversable).toBeTruthy();
    expect(nodeTC!.traversable).toBeFalsy();

    // edges and diagonal weights
    let edgeTLTC = graph.getEdge(nodeTL!, nodeTC!, true);
    expect(edgeTLTC).toBeDefined();
    expect(edgeTLTC!.traversable()).toBeFalsy();
    expect(edgeTLTC!.weight).toEqual(1);
    let edgeTLBC = graph.getEdge(nodeTL!, nodeBC!);
    expect(edgeTLBC).toBeDefined();
    expect(edgeTLBC!.traversable()).toBeTruthy();
    expect(edgeTLBC!.weight).toEqual(2);

    // start, end, and referential equality
    expect(Object.is(graph.startNode, nodeTL)).toBe(true);
    expect(Object.is(graph.endNode, nodeTR)).toBe(true);

    // custom data
    expect(nodeTL!.data["test"]).toEqual("isStart");
    let edgeBRTR = graph.getEdge(nodeBR!, nodeTR!, true);
    expect(edgeBRTR).toBeDefined();
    expect(edgeBRTR!.weight).toEqual(3.5);
})

function expectGraphAdjacencies(graph: Graph, node: GraphNode, adjacent: number, incoming: number) {
    expect([...graph.getAdjacentNodes(node)]).toHaveLength(adjacent);
    expect([...graph.getAdjacentEdges(node)]).toHaveLength(adjacent);
    expect([...graph.getIncomingEdges(node)]).toHaveLength(incoming);
    expect([...graph.getIncomingNodes(node)]).toHaveLength(incoming);
}

test("Graph Adjacencies", () => {
    const graph = new GenericGraph();
    const nodeA = new GraphNode(graph, "A");
    const nodeB = new GraphNode(graph, "B");
    const nodeC = new GraphNode(graph, "C");
    graph.addNode(nodeA);
    graph.addNode(nodeB);
    graph.addNode(nodeC);
    const edgeAB = new GraphEdgeSimple(1, nodeA, nodeB, true);
    const edgeBC = new GraphEdgeSimple(2, nodeB, nodeC, false);
    graph.addEdge(edgeAB);
    graph.addEdge(edgeBC);
    // adjacencies

    expectGraphAdjacencies(graph, nodeA, 1, 1); // adj to B, accessible from B
    expectGraphAdjacencies(graph, nodeB, 2, 1); // adj to A and C, accessible from A
    expectGraphAdjacencies(graph, nodeC, 0, 1); // adj to none, accessible from B
})
```