Pathfinding Algorithm Visualizer

Non-Examination Assessment

Saeed Al Shrouf

Candidate No. 0926

Centre No. 74677

Kings' School
Al Barsha

Component Code
7517/C

Table of Contents

Analysis:

Technical Implementation:

Testing:

Evaluation:

Analysis

Project Overview

Problem Conspectus

Key Terms

Algorithm: Clear and precise steps which produce the correct output for any set of valid inputs – normally used to solve a problem/complete a task. An algorithm must have a point of termination.

Pathfinding Algorithms: Algorithms designed for the purpose of finding the shortest route between two points. Most commonly these search a graph by starting at one vertex and exploring adjacent nodes in a variety of methods.

Heuristic Methods: Methods of solving a problem employing algorithms that are not guaranteed to be optimal but are sufficient for the purpose. These prioritise speed over optimality, completeness, and accuracy.

Problem Identification

Client: Computer Science and Further Mathematics Departments

Pathfinding Algorithms and Heuristic Methods are prevalent in both the A-level Computer Science and Further Maths courses. However, from experience, the resources given by the exam board do not sufficiently convey their application. Even though books can explain the basic theory, a visual aid would assist greatly in helping the student to understand how pathfinding algorithms and heuristics function, which is the premise for the demand behind this project. Heuristic methods are also very poorly explained and the application of them can be understood much better with a system in place giving students freedom to explore the concepts.

Interview Investigation

Case 1 - Student Interview

Interviewee: Lucas Rietjens

Interviewee Capacity: Year 13 Computer Science Student

"What's your current method of learning about Pathfinding Algorithms and Heuristic Methods in Computer Science?"

LR: When learning it we just used the AQA A-Level Computer Science Book and a YouTube video that just explained the main specification points.

"What are the positives of learning Pathfinding Algorithms and Heuristic Methods with your current method?"

LR: The steps of the Dijkstra algorithm are simply explained with an example diagram to show a scenario of it happening. As for heuristic methods, the book is not very useful in explaining exactly how it works and only gives a basic overview.

"What are the downsides to your current method of learning Pathfinding Algorithms and Heuristic Methods?"

LR: The main downside to the Dijkstra chapter is that there's not much variety in examples of how the algorithm functions. It was also not the best in showing the actual usage of the algorithm in a different environment. As for heuristic methods, there was only a basic definition given, the book didn't explain how they could be used in a specific context.

"If I were to propose a different method of learning to you, what would feature would you most like to have?"

LR: I think that having a variety of scenarios to see the algorithm affect would be very useful in learning it. Any applications of the Algorithms would also be very useful to see. We should be able to make our own simulations in the program and see how the algorithm affects them.

"How many members of staff would utilise the software?"

LR: There are three Computer Science teachers and one Further Maths teacher.

"How would you like to be able to create these simulations?"

LR: It would be nice to have a software where I could create different types of interactions between nodes and their paths. Maybe by drawing on barriers between nodes or something of the sort using a UI.

Case 2 – Educator Interview

Interviewee: Rezaul Haque

Interviewee Capacity: A-Level Computer Science Teacher

"What is your current method of teaching Pathfinding Algorithms and Heuristic Methods in Computer Science?"

RH: I mainly teach the algorithm by the book; the steps are explained quite well, and some examples are given along with them to help students grasp the concept.

"What if a student doesn't understand the application of a Pathfinding Algorithm or Heuristic Method from the book's application?"

RH: I would probably try to find a video to help explain further as there's not many specifically concentrated resources on topics from the syllabus. Heuristic methods

especially aren't very common to appear and the info the student is required to know is very surface level.

"What would you want out of a new system to explain Pathfinding Algorithms, if you could decide its features?"

RH: As you mentioned earlier students don't always grasp the concept of the algorithm right away, as many people are visual learners. It would help to have a system to further consolidate the student's knowledge and show them the application of it. This is another thing lacking with the knowledge taught about Heuristic Methods.

"Could you further specify what sort of features you would desire in the project?"

RH: A system where a student would enter a menu screen and have the option presented to choose to run a pathfinding algorithm or open up some instructions on how to operate the software. The visualisation would have to allow the student to customise the sort of simulation they would like to make, and it would have to make it clear what sort of changes are happening within the algorithm to the user."

Interview Analysis and Objective Derivation

There are some key points which may be derived from the statements made by both the student and educator. Firstly, both the student and educator noted the need for there to be variation within the simulations, meaning that an ideal solution would allow the user to choose the type of simulation they would like to occur. Secondly, the student has mentioned the need for allowing the user to manipulate the nodes on the graph to allow them to place barriers. This was accompanied by the mention of the project having a User-Interface, which is a given – considering the nature of the project. As for the comments made by the educator, he noted that the software should contain a main menu and an instruction set. Additionally, he mentioned the necessity of the changes to the graph having to be clearly displayed to the user.

These points can be used to begin to derive the skeleton of the objectives necessary for the software to be successful. Based on the points mentioned by both the student and educator, some key objectives for the software can be derived. These include:

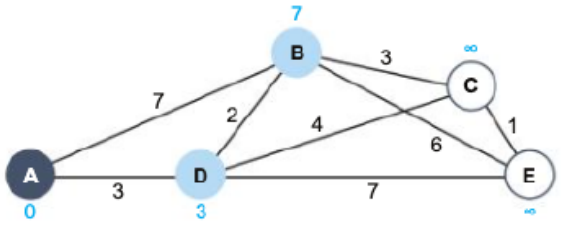
1. There must be a simplistic user interface.
2. The pathfinding algorithm must be visualised clearly on the graph.
3. The user must be able to select a node on the graph to be the starting position.
4. The user must be able to select a node on the graph to be the ending position.
5. The user must be able to create barrier nodes on the graph. (Nodes which the shortest route cannot pass through)
6. The user must be able to run the pathfinding algorithm.

Potential Platform Solutions

Case 0: Current Method of Pathfinding Visualisation

This system is very simple in that it explains the process of how the Algorithms function, but not anything beyond that. The book provides a very limited number of visual descriptors on how pathfinding algorithms function. The current system is far from ideal as it does not account for students who may not grasp the concept fully – because it then becomes a problem of the teacher searching for more examples of Pathfinding Algorithms online, which still doesn't give the student much freedom in learning the concept behind how they work.

Case 0 Visual



D = 3	B = 7	C = ∞	E = ∞	

Case 1: Web-Based Implementation of Pathfinding Visualisation

Timeframe Status: Sufficient

Resources: Sufficient

Case Positives

This system would allow any students looking to use the Visualiser easy access through the link. It would also be easily distributable for the teachers and in the promoting it to other environments outside of our Computer Science and Mathematics Departments. The programming would require me to learn Web Development, an aspect of Computer Science I have minimal experience in.

Case Negatives

Despite my timeframe and resources being sufficient for the implementation, it does not produce a result that is original and helpful to the department. There would also be many tasks that would additionally need to be completed to release a final Web-Based product. Areas such as security, establishing a domain, etc. are all tasks that I do not have experience in, meaning that they would require lots of trial and error before I had a sufficient product.

Timeframe Status: More than Sufficient

Resources: More than Sufficient

Case 2: Software Implementation of Pathfinding Visualisation

Case Positives

This system would forgo the need for securing it against external attacks as the software would be wholly client-side. It would also avoid the need for a domain to run, as well as an internet connection – after the initial download. The implementation of the system into the real world would be simple as it could be downloaded onto all of the Computer Science Lab computers, allowing it to be a tool that can be easily accessed whenever necessary.

Case Negatives

A Software-Based system would not be as easily distributable if the need for it occurred on personal computers – as it would not have the simplicity of a web link. People are largely hesitant to download dubious files onto their computers which could cause issues as people may be reluctant to download the software, but also poses the problem of any harmful individuals creating duplicates of the project with harmful code attached, which is a security risk to the school.

Potential Functionality Solutions

Case 1: Dijkstra Implementation of Pathfinding Visualisation

Timeframe Status: More than Sufficient

Resources: More than Sufficient

Case Positives

The Dijkstra implementation would sufficiently introduce the concept of pathfinding algorithms to students as it is the main algorithm covered in both the Computer Science and Further Mathematics Decision course. This framework would utilise buttons allowing the user to place start and end nodes, as well as barriers, on a grid, which would then apply the Dijkstra algorithm to it to visualise the algorithm's process. The metrics for the algorithm's runtime and distance covered would also be outputted to the user.

Case Negatives

Despite introducing the application of the Dijkstra algorithm in a visual capacity, it does not cover much more – making it a rather simple implementation. This Case ignores the potential for more educating that can be done with some extra work, which is possible as the timeframe and resources are beyond sufficient.

Case 2: A* Implementation of Pathfinding Visualisation

Timeframe Status: More than Sufficient

Resources: More than Sufficient

Case Positives

The A* implementation provides an, albeit of weaker connection, introduction to pathfinding algorithms. The main benefit of this project is that it provides a good understanding of Heuristic Methods to students – as that is the main feature of the A*

Algorithm, however, without a Dijkstra algorithm to compare it to, the impact of it is diminished. This framework would utilise buttons allowing the user to place start and end nodes, as well as barriers, on a grid, which would then apply the Dijkstra algorithm to it to visualise the algorithm's process. The metrics for the algorithm's runtime and distance covered would also be outputted to the user.

Case Negatives

The main drawback to this Case is that it does not sufficiently relate to the content within the Computer Science and Decision courses. This is because neither course covers the A* algorithm on its own. Therefore, it may even have the unintended effect of giving a student the wrong idea of how the Dijkstra algorithm functions – seeing as it does not provide a base case visualisation.

Case 3: Dijkstra and A* Implementation of Pathfinding Visualisation

Timeframe Status: More than Sufficient

Resources: More than Sufficient

This is the most ideal solution to the project as it utilises both Dijkstra and A* visualisations. As timeframe and resources are not relevant drawbacks in this scenario it is a near-perfect result, due to its ability to both demonstrate the algorithms correctly and provide the students a good idea of how Pathfinding and Heuristic Methods work. It allows students to cross-compare between the same simulation using the two different algorithms. This framework would utilise buttons allowing the user to place start and end nodes, as well as barriers, on a grid, which would then apply the Dijkstra algorithm to it to visualise the algorithm's process. The metrics for the algorithm's runtime and distance covered would also be outputted to the user.

Case 4: Any Prior Case with Maze Solutions Properties

Timeframe Status: Sufficient

Resources: More than Sufficient

Case Positives

Provides an idea on how Pathfinding Algorithms can be used within mazes to find paths, especially cross-comparing the algorithms to see which is more efficient and in which case-scenario. This would be implemented in a similar fashion to the other algorithms but with a selectable maze option, which would apply the algorithms to a randomly generated maze.

Case Negatives

This is a large addition to the project that does not provide much educational assistance whatsoever to the student – as Maze solutions are not prevalent within the specification at all. The cross-comparison between algorithms that can be done with mazes does not at all necessarily have to be done with mazes and the simple implementation suffices. In general, this provides an unnecessary reduction to timeframe with little to show for it.

Proposed Solution

Justification for Proposed Solution

Proposed Platform for Solution

Case 2 (Revised) – Software-Based Implementation with GitHub

The main reason behind me choosing this platform is that it does not require security against external attacks – such as SQL-Injection attacks – looking to take the service down. Additionally, the software-based implementation does not provide any endpoint vulnerabilities to the user, meaning that it does not provide any risk. In addition to this, the program will not require an internet connection to run.

Revisions

The disadvantages presented by the lack of a web-based implementation can be combatted by linking the project to a GitHub repository. This allows the project to have a link to export the project with, which would allow the usage of the project on personal computers as said link could be broadcasted by the Computer Science department for the relevant student cohort. The GitHub repository would also allow for me to explain the project in detail for those interested in downloading, making it seem less risky for them to download the project – as GitHub profiles are unique and the individual project's link cannot be duplicated.

Proposed Functionality of Solution

Case 3 – Dijkstra and A* Pathfinding Implementation

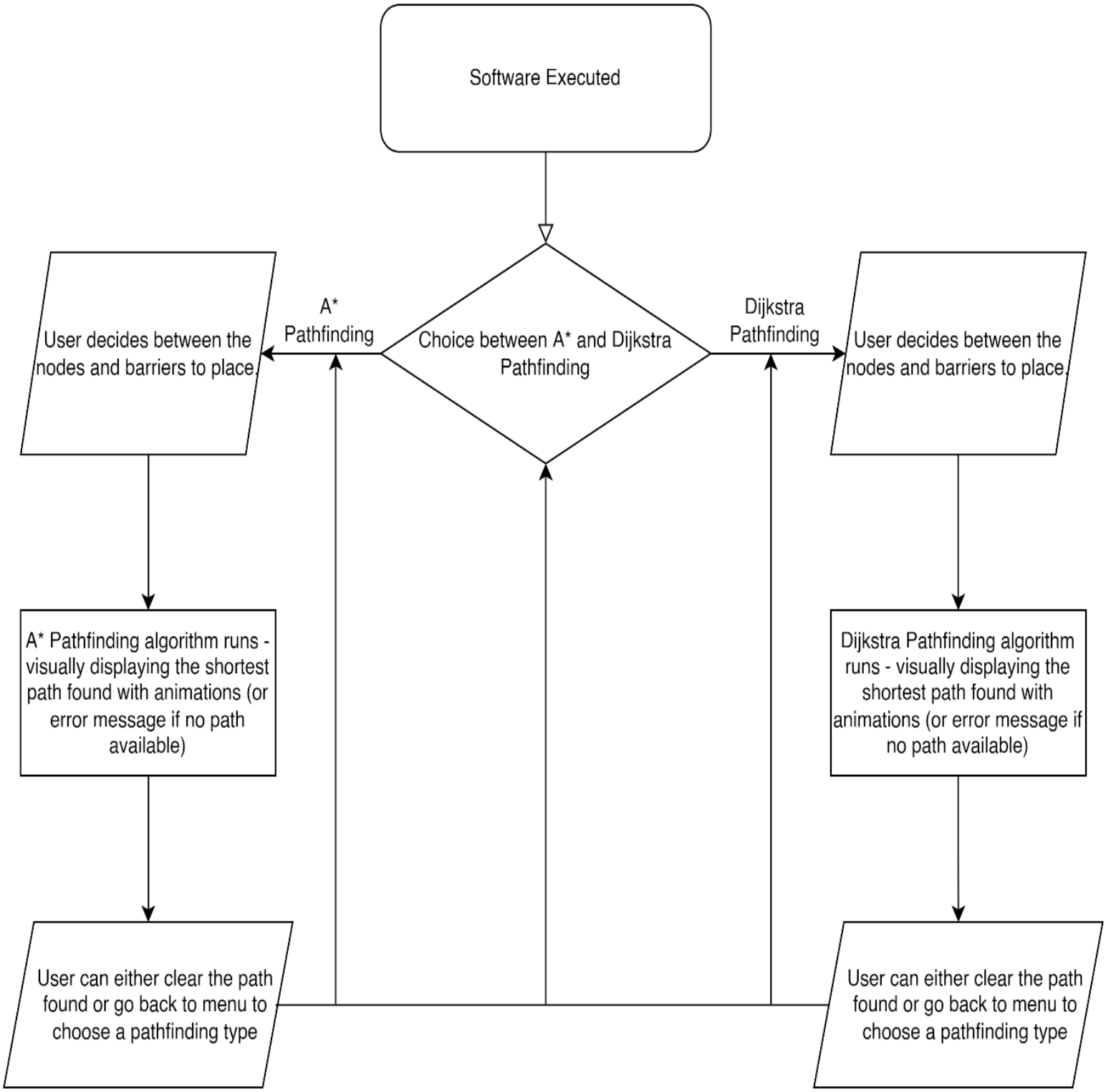
The solution I have come up with to combat this potential gap in students' knowledge is to design software that allows users to visualize both the Dijkstra and A* pathfinding. The software would first display a menu to the user, giving them the option to read the instructions or proceed to the visualiser – this is not considering other potential menu options such as a Contact page, information on pathfinding algorithms, etc.

The visualiser would function by allowing the user to press certain buttons to allow them to plot a start and end node on a 100x100 grid, as well as barriers in between them. The student will then be able to choose the method of Pathfinding and execute the algorithm. During the visualisation, the software will change the colour of every node on the grid that is affected by the pathfinding algorithm, allowing the user to see the effect of the algorithm on the grid. The student will then be able to keep that same previously drawn-out path and apply the other algorithm to it, which will give them the opportunity to analyse the differences and efficiencies of both algorithms in the same context – most probably on a larger scale than anything the book covers.

The student will also be able to refer to the 'Knowledge' section of the menu, which will provide an explanation of pathfinding algorithms and the relation of the heuristic technique

within A* Pathfinding, along with other applications of these algorithms. The combination of these tools within the software should allow teachers to implement a more engaging and informative system to teach this section of Computer Science and Decision Mathematics.

Proposal Data Flow Diagram



Assessing Validity of a Computational Solution

Significance of a Digital Environment

The solution I proposed warrants a computational solution due to a multitude of factors. This is because not only would the solution be in software form – meaning that it is in a digital environment, thus requiring a computer; but, implementing the algorithm and code will require the use of Python, as well as computational principles such as Abstraction.

Prevalent Abstraction Principles

Generalized Abstraction

The software itself can be considered a general abstraction as it breaks down the complex process of visual pathfinding that is present in real-world applications such as Google Maps, into a simplistic grid design. This abstraction can also more accurately represent the graph theory behind the algorithms that are being applied to the nodes on the grid.

Functional and Procedural Abstraction

This would be common throughout the code as I'm looking to utilize the Pygame module – meaning that I will have to use a variety of procedures such as the functions required to create and display the grid, the subroutines inside of the Node Objects, etc. Using this abstraction, I will be able to reference the subroutines more easily later whilst error-checking and maintaining the code – as I will not need to remember how to replicate each specific component of the code if I were to reference them again.

Data Abstraction

I am planning to use the Queues module in Python to assist me with the coding of the pathfinding algorithms. Queues are a form of abstract data type that will never be a concern to the user or relevant in the front-end of the project. This, as well as the fact that I will not require the knowledge of how the programming behind the Queue module was implemented in the first place, means that my usage of queues will be indicative of data abstraction.

Information Hiding

Information hiding is when data is not able to be directly accessed through subroutines. A common example of information hiding is object-oriented programming, which will be used in my program through the pathfinding nodes and Pygame features. On the front-end, the user will not be able to access any of the information in the code, which ties into the data abstraction.

Decomposition and Composition

Composition is going to be a large part of the project as I will be combining many different programming structures for the final product – Pygame design, queues to form the

pathfinding objects, a separate software to turn the code into an executable, etc. On a smaller scale, it would also be seen in the development of some subroutines, such as one to display the grid. This would need to both create the grid through a process, but also display it on the screen using a Pygame procedure. Decomposition is also seen in how I will code the Pathfinding Nodes – objects in which there will be subroutines to suit their different interactions. Breaking the process down into smaller functions is a combination of decomposition and functional abstraction.

Analysis of Existing Solutions

I researched existing pathfinding algorithm implementations to understand how users would interact with them and the issues that exist within the systems.

Web-Based Implementation Case Study

Project - https://gunjankadu.github.io/PathFindingVisualizer/

Author – Gunjan Kadu

Web-Based Implementation Drawbacks

1. The website is inaccessible without an internet connection, which may not always be available to a user.
2. Potential security risks due to it being web-based, meaning that the website should be secured.

Functionality Drawbacks

1. The weight of each path is pre-existing and random, meaning that the user is unable to set their own preferred settings for the visualisation, as the algorithm will function off these predetermined weights.
2. The animation style and graphics of the pathfinding make it difficult to see the order in which the path is being created.
3. A* Pathfinding is not available on the website, only Dijkstra is covered.

Drawback Analysis

The benefit of programming the project in the form of a software is that it allows the user to access the visualisation system at any point in time – with or without an internet connection - so long as it has been downloaded. The software implementation also removes the risk to the user and project as it is a client-side implementation, meaning that it is not at risk to attacks – for example, SQL Injection attacks to bring the website down, etc.

Due to the weights of each path being predetermined and random, the algorithm does not visualise a consistent solution to the same set of points every time. This can be eliminated if each block on the grid had a length of 1 and the distances were determined by summing the number of blocks between two points. The fade animation on the nodes is also not very clear in showing how the pathing is determined, which would be improved by using a solid fill. My software would also contain A* Pathfinding, which this program does not visualize.

Web-Based Implementation Positives

1. Website is easily accessible – given that there is an internet connection – and does not require much time to set up.
2. Project is easier to advertise and promote to more people, as it can be accessed with a simple link.

Functionality Positives

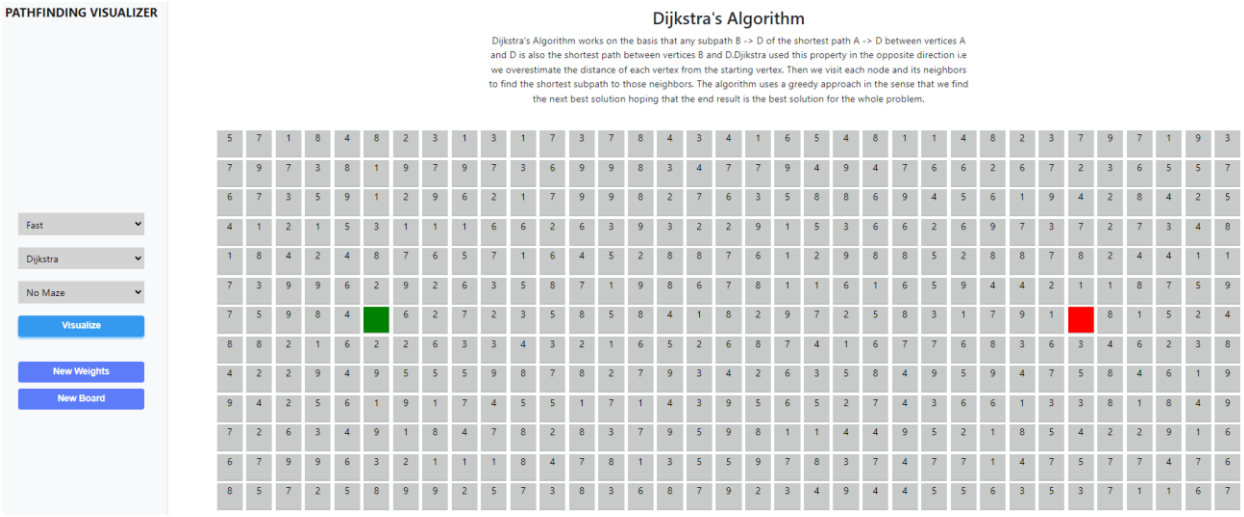
1. The website provides an instruction set on how to be used properly.
2. Dijkstra – along with other algorithms – is visualised correctly with freedom – bar the weighting of nodes - in how to structure the grid to be visualised.
3. Relatively simple UI with good readability, giving first-time users a straightforward user experience.

Positives Analysis

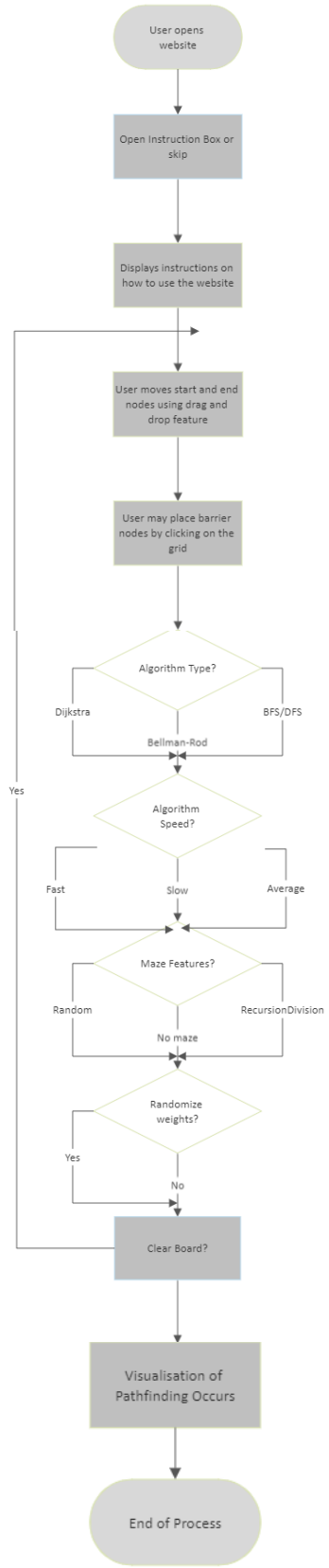
A substitute for the ease of being able to access the project with a link can be linking it to GitHub, which would give users a simple way of accessing the project's contents and reading the documentation regarding it.

The software implementation would have a set of easily readable instructions on how it can be used, along with a similar interpretation on how algorithms can be visualised. The use of Python would give me a chance to implement a similar project in a different language (as this was coded in HTML, CSS, and TypeScript) with the use of Pygame giving me an easy method of implementing a simple, interactive, UI.

Case Study Visual Example



Case Study System Flow Diagram



Software-Based Implementation Case Study

Project - https://github.com/xSnapi/cpp_pathfinding_using_sfml

Author - Matthew (xSnapi)

Software Implementation Drawbacks

1. The software must be downloaded to function.
2. A negative of this project in project is that the final export isn't compiled, meaning that when downloaded, results in multiple files coming out of the file extract, unnecessarily cluttering the folder.

Functionality Drawbacks

1. The biggest negative of this project is that it does not properly display how the algorithm visualises the path, meaning that it only shows the result of the algorithm, without all the nodes being affected. This defeats the entire purpose of the project as it is not instructive to the user at all.
2. The UI is quite poor, and the instructions are not sufficiently detailed as it took multiple attempts for me to realise that the placing of nodes required not only pressing the button but also clicking simultaneously, which was not mentioned.
3. No Dijkstra Algorithm Visualisation

Drawback Analysis

Using a software implementation does mean that the project requires a download, however, it is also a blessing in that the user will never need an internet connection to utilize it – as my implementation will not be utilizing any remote connections. As well as the fact that it is not a security risk, because it is a client-side app. My project will be compiled into a single file to decrease the extra clutter that came with this case study – i.e., the font file that was downloaded in a separate folder alongside it.

A major difference between my implementation and this case study will be the inclusion of the affected nodes being coloured, which was something left out by this project entirely. This change will aptly demonstrate the process carried out by the A* algorithm. Unlike this project I will also include the visualisation for the Dijkstra algorithm.

Software Implementation Positives

1. The project is client-side, meaning that it is secure from attacks.
2. Running the file does not require an internet connection, excluding the original download.

Functionality Positives

1. The time taken for the pathfinding algorithm to complete was outputted to the user, which is not a frequent function included in Pathfinding implementations.
2. A large grid was used, allowing many different variations of grids to be created by the user.

3. As C++ was the language used for the project, it has a fast runtime because it is compiled directly into the machine code, without the additional translation required upon runtime.

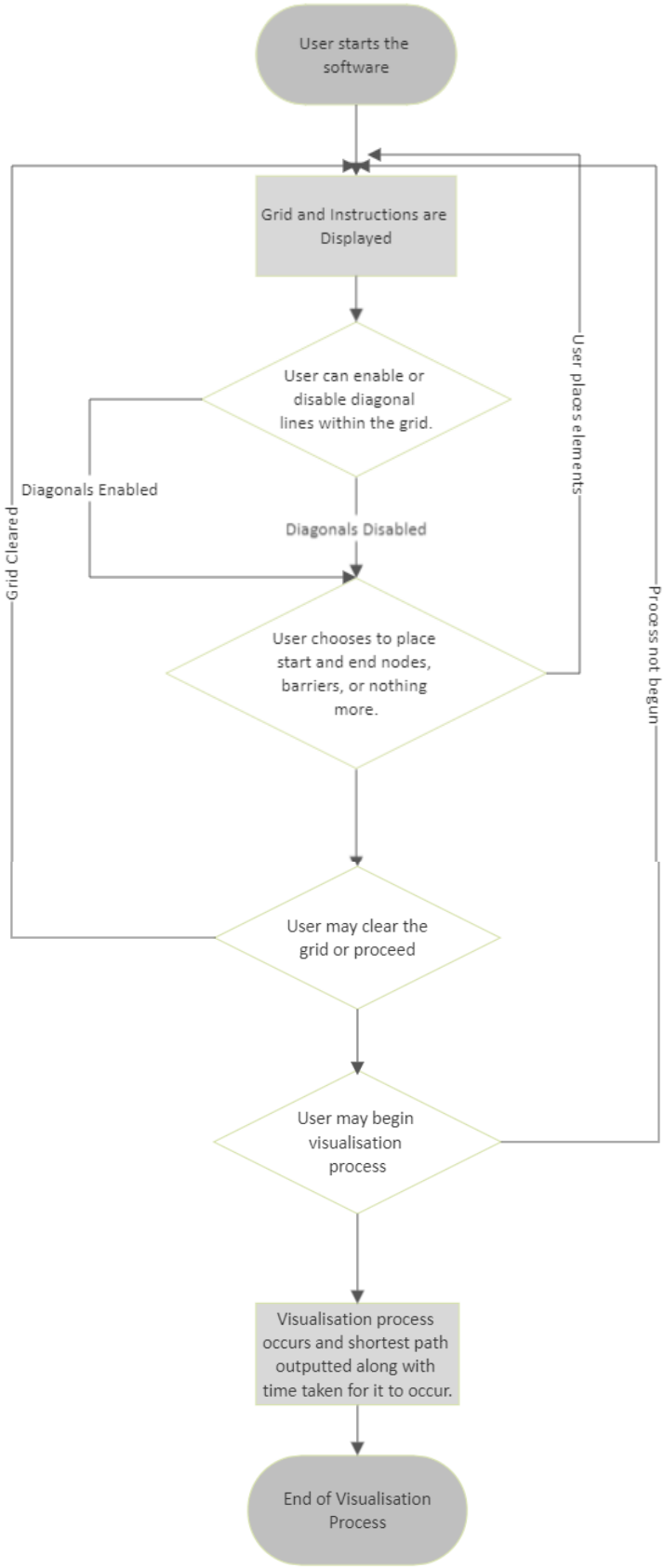
Positives Analysis

My implementation will take advantage of both software implementation positives as the project will have a similar conclusion – though it will be differently compiled.

If there are useful metrics – such as the distance travelled and time taken for the algorithm to complete – they will be outputted in a similar fashion, as it may provide some interest to students. Along with a similarly large grid, I will be implementing it in python, which means that my programming abilities will be tested in making the code as efficient as possible – due to Python having a slower execution speed, as it is an interpreted language, meaning that it must be translated upon running.

Case Study Visual Example

Case Study System Flow Diagram



Acceptable Limitations and Prospective Users

Analysis of Prospective Users

The user-base accessing this software is going to be made up of both students and teachers, despite these groups being considerably computer-literate, the project's exhibition page will include all the details on how to download the software, as well as utilise it. Instructions on the software's use will also be provided within, allowing for more ease-of-use, allowing the learning process to flow smoother.

Based on the current Department layout, there will be around 15-20 students utilising the software, along with 1-2 Teachers using it for instruction. Their computer literacy means that the complexity of the project will not be affected by the limitation of the user-base operating ability.

Constraints Affecting System Development

Accessibility

Hardware Requirements

Despite the number of computers easily available within the Computer Labs now being more limited with the removal of the PCs, the teachers all have computers and so do the students doing A-Levels, meaning that the project's exhibition page could be broadcasted through the Computer Science/Further Maths communication channels for ease of access to the students.

Software Requirements

There is also a limited amount of software that I can use to develop the project, due to the available funding – or rather, lack thereof. The software also must be able to accommodate my abilities and skills that are able to be developed within a reasonable time frame.

Computer Literacy

The user-base consists of A-Level students and Teachers who are all familiar with using different types of software on computers and working within digital environments. In the case of any confusion with installation, the exhibition page also contains instructions regarding how to download and operate the software – information regarding using the visualiser will also be programmed within.

Personal Skill and Knowledge

Ability

The project must be implemented with a method that is not completely foreign to me as I must fit it within the time constraint and the complexity is relatively high, meaning that using a language such as Python – that I have experience with – will be most beneficial.

Using Python is also advantageous as it gives me a chance to create a variation on this idea in a language that is not very frequently used for this, in my research. There is a variety of ways I can code the system within Python and deciding between methods of Object-Oriented and Functional Programming is a focus within the Design of the project. Finally, the driving force behind using Python additionally stems from the fact that the language enables the programmer to very swiftly prototype. This refers to Python's dynamic nature, making it easy to quickly prototype and test new ideas. This can be advantageous when developing a pathfinding algorithm visualizer as I will be looking to experiment with different types of algorithms and visualisation methods. Python's quick prototyping capabilities can make it easier to test out new ideas and iterate on the design of the visualizer.

Time

The project should be implemented within 2 months as it must be completed prior to the A-Level class's final revision for the exams. This will allow it to be useful to both the current set of students and incoming classes.

Necessary Programming Languages, Software, and Websites

Programming Language

As the proposed implementation for the project is a software, with some of the additional points of focus being a language I can utilise with relative familiarity and completion under a tight time constraint, I discerned that the most suitable programming language for the project was Python. Python hits both points of it being a language I have used regularly, and a relatively rare language used for this type of project, from my research.

The actual usage of Python will require me to use Visual Studio Code for a (subjectively) much more user-friendly development environment than the other common IDEs – such as Python IDLE, Pycharm, etc. This is due to it providing a more compact and expansive framework for working with multiple files and its ability to use an automatic Formatter and Intellisense (autofill extension).

My final product will be available for download as a compiled .exe file, for a multitude of benefits to the user – such as more compact installation, better run-time, and processing, etc. However, this will require me to use a compiling software to combine the graphics, audio files, and code, into a single, functional file.

Software

The graphics for the software will be completed within Adobe Illustrator, a software I currently have no experience in – meaning that this will also affect the time constraint I have to work with. I may implement a system of automating the creation of sprites within Adobe Illustrator using a script from PyAutoGUI, saving me a lot of valuable time to work on the other aspects of the project.

The audio used within the software will be created within Audacity, a sound processing software, which should allow me to create custom, original, sounds. These will enhance the user experience and add depth to the project.

Websites

I plan to base the software off a GitHub repository page, providing many advantages, such as the ability for the Departments to broadcast the project to all the relevant students, allowing them access to the project with easily readable instructions on the installation and operation of the product. This also eliminates the issue of false/potentially dangerous software being promoted in its stead – as the link/repository will be unique to the project, meaning that if the broadcasted link is correct and comes straight from the source – i.e., myself – there is no risk of a breach in security with a replica software. Hence, to be able to use all this functionality, I will be using the GitHub website; allowing me to display all the information about the project along with its files and information – making it open-source to those interested as well.

Objectives for Proposed Solution

Set 1: Menu and Help State Requirements

1) The Program must contain a main menu state.
    a. The user must have the option to enter the Pathfinding state.
        i. The transition should be triggered by a button at the click of the user's mouse.
    b. The user must have the option to enter a 'help' state.
        i. The transition should be triggered by a button at the click of the user's mouse.
    c. The main menu state must have a simple user-interface design consisting of a simple background and button layout.
    d. The program must be able to be terminated during this state.

2) The Program must contain a 'help' state.
    a. This state must consist of clear and annotated instructions on how to operate the Pathfinding Visualiser.
        i. These consist of a picture of the Pathfinding state with annotations on what each button does and how to operate the software.
    b. There must be a 'back' button to return the user to the main menu.
    c. The program must be able to be terminated during this state.

Set 2: Visualisation State Requirements

1) There must be a simplistic user interface.
    a. Buttons for each function must be provided.
        i. Buttons must be clearly labelled with their functionality.
        ii. Buttons must be a unique colour for easier identification.
    b. The background should be simplistic – a near-solid colour.
    c. The graph should be large enough to easily identify individual nodes.
    d. If the Pathfinding Button is pressed.
        i. A new window with a user-interface must appear with an input for the possible graph dimension sizes.
        ii. Those graph dimension sizes must be passed into the Pathfinding state.
    e. There must be a 'back' button to allow the user to return to the main menu.
        i. This button must simultaneously reset the Pathfinding state – resetting the graph and all other elements within.
    f. The program must be able to be terminated during this state.

2) The pathfinding algorithm must be visualised clearly on the graph.
    a. Each node-type must be individually identifiable.
        i. Nodes will change colour depending on their type.

   b. As the algorithm runs, each node that it identifies must be coloured depending on its condition within the algorithm – for example, a node currently in the queue (refer to Design -> Dijkstra Algorithm Design) would be a different colour to one that has been popped from the queue.
      i. Once the shortest path has been identified, there should be a visual backtrack as each of the nodes in the path is coloured a new, unique, colour – to identify them as the result.

3) Information regarding the last performed pathfinding algorithm must be displayed.
   a. In its initial state – when no algorithms have been performed – all statistics must be set to 0.
   b. The information displayed must be clearly labelled.
   c. There must be a 'time-taken' statistic, which returns the amount of time it took for the algorithm to find the shortest path.
      i. This must be measured in seconds, with a resolution of up to 3 decimal places.
   d. There must be a 'distance' statistic, which returns the number of nodes that are traversed in the shortest path.
   e. There must be a 'total nodes accessed' statistic, which return the number of nodes checked by the algorithm.

4) The user must be able to switch between Dijkstra and A* Pathfinding modes.
   a. There must be a button which alternates between A* and Dijkstra Pathfinding when pressed.
   b. The button must be clearly labelled and easy to identify.
   c. This should be implemented with Boolean logic.

5) The user must be able to switch between Manhattan and Euclidean heuristic modes.
   a. There must be a button which alternates between Manhattan and Euclidean Heuristics when pressed.
   b. The button must be clearly labelled and easy to identify.
   c. This should be implemented with Boolean logic.

6) The user must be able to select a node on the graph to be the starting position.
   a. There must be a 'start node' button.
      i. This button allows the user to place the starting node on the graph.
      ii. The start node may be assigned at any point during the program's operation – excluding when the pathfinding algorithm is running.
      iii. The start node may be reassigned onto any other node after being placed, whether the destination node is being occupied or not.

7) The user must be able to select a node on the graph to be the ending position.
   a. There must be an 'end node' button.
      i. This button allows the user to place the end node on the graph.

ii. The start node may be assigned at any point during the program's operation – excluding when the pathfinding algorithm is running.
iii. The end node may be reassigned onto any other node after being placed, whether the destination node is being occupied or not.

8) The user must be able to create barrier nodes on the graph. (Nodes which the shortest route cannot pass through)
    a. There must be a 'barrier' button.
        i. This button allows the user to place barriers on the graph.
        ii. The barriers may be assigned at any point during the program's operation – excluding when the pathfinding algorithm is running.

9) The user must be able to clear the barriers that have been placed on the graph.
    a. There must be a 'clear barriers' button.
        i. This button resets the state of every node that was previously a barrier.
        ii. The barriers may be reset at any point during the program's operation – excluding when the pathfinding algorithm is running.
        iii. Clearing the graph when there are no barriers must result in a suitable error message being displayed.

10) The user must be able to reset the graph to its original state.
    a. There must be a 'reset' button.
        i. This button should allow the user to clear the graph of any nodes that have been changed.
        ii. The graph may be reset at any point during the program's operation – excluding when the pathfinding algorithm is running.
        iii. Clearing the graph when it is empty must result in a suitable error message being displayed.

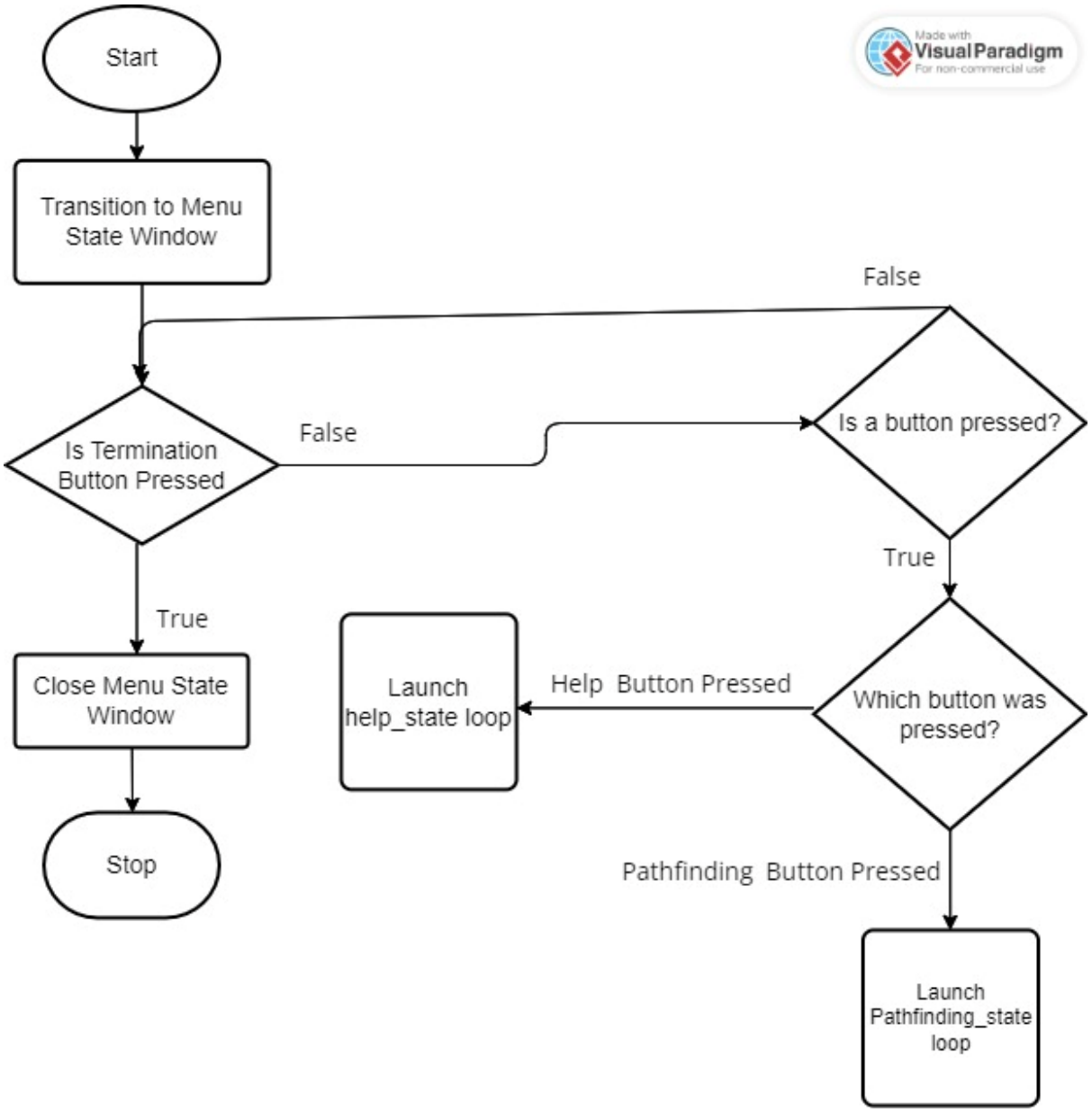
11) The user must be able to run the pathfinding algorithm.
    a. The 'Enter' key should run the pathfinding algorithm on the graph.
        i. If the conditions on the graph are insufficient to run the algorithm – either start or end node not identified – there must be a suitable error message displayed.
        ii. If the conditions to run the algorithm are met, the algorithm may be initiated at any point during the program's operation.

Design

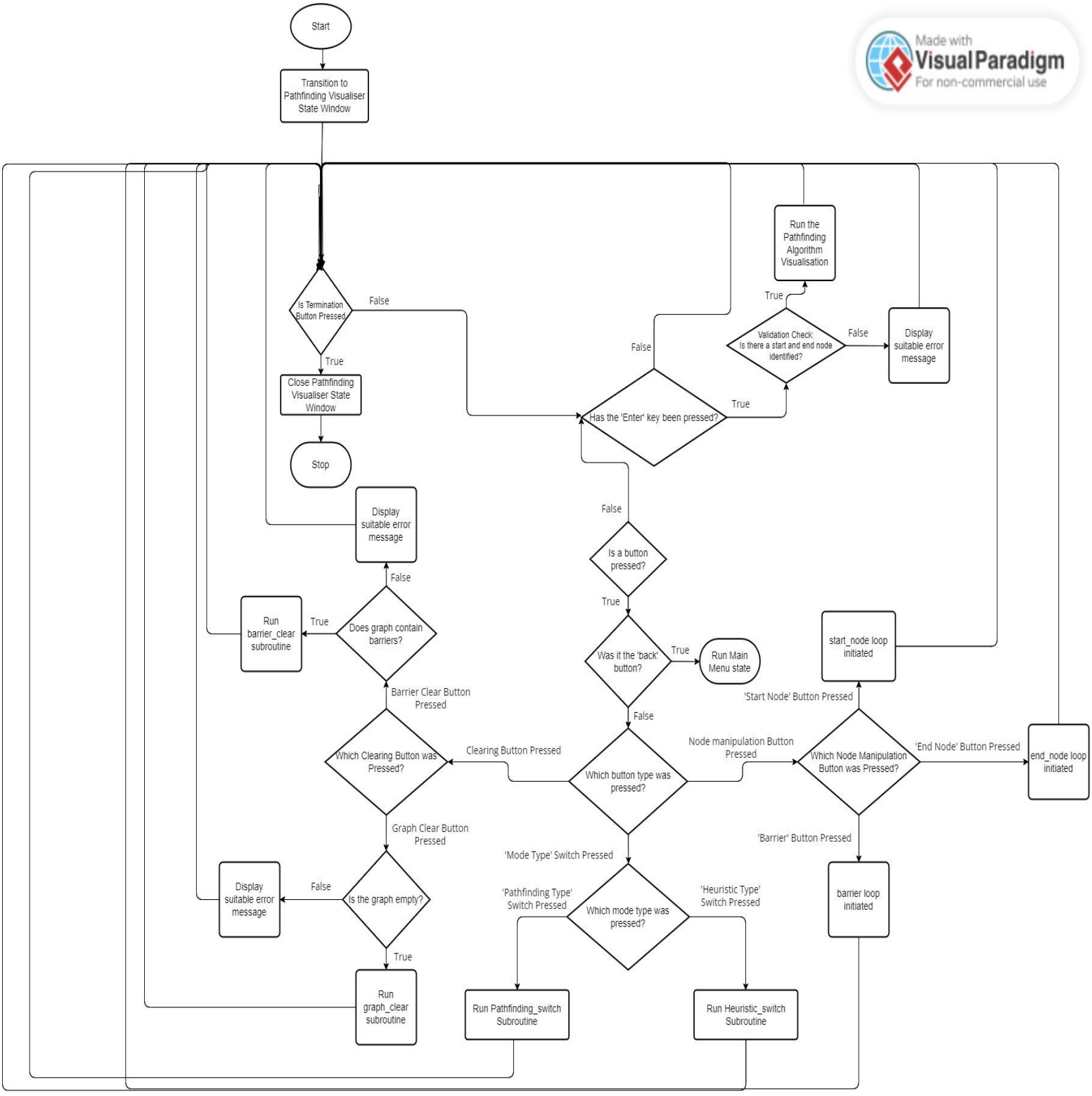
System Outline

System Flowchart

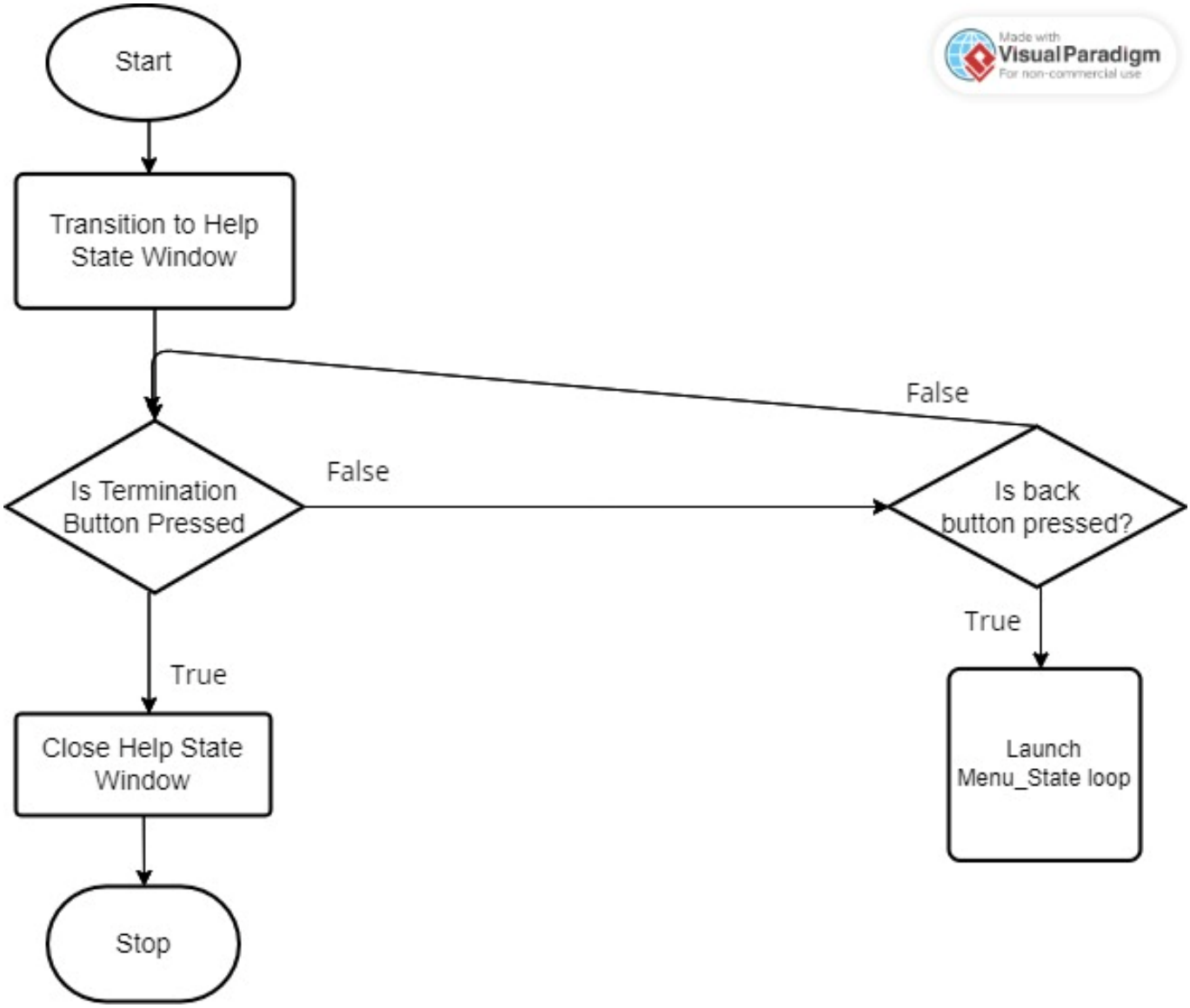
Main Menu Flowchart



Pathfinding Visualisation State Flowchart



Help State Flowchart



Stepwise Refinement

System Flow Refinement

This is a software development technique which I will use to break down the processes of the Pathfinding Algorithm Visualiser into sub-processes. These sub-processes will be more manageable and allow me to take a top-down approach to designing the implementation of these processes.

1) Main Menu State
    a. **If the termination button is clicked by the mouse pointer**
        i. The program shuts down and the window is closed.
    b. **If the 'help' button is clicked by the mouse pointer**
        i. The help_state loop must be initiated, which will trigger a transition of the window.
    c. **If the 'Pathfinding' button is clicked by the mouse pointer**
        i. A graph_dimensions input should be displayed to the user.
            - There must be four graph sizes displayed to the user
            - If the 10x10 button is clicked by mouse pointer
                i. Assign a size 10 to the graph dimensions variable, which will be used in initialising the graph.
            - If the 25x25 button is clicked by mouse pointer
                i. Assign a size 25 to the graph dimensions variable, which will be used in initialising the graph.
            - If the 50x50 button is clicked by mouse pointer
                i. Assign a size 50 to the graph dimensions variable, which will be used in initialising the graph.
            - If the 100x100 button is clicked by mouse pointer
                i. Assign a size 100 to the graph dimensions variable, which will be used in initialising the graph.
        ii. The pathfinding_state loop must be initiated, which will trigger a transition of the window.

2) Pathfinding Visualisation State
    a. **If the termination button is clicked by the mouse pointer**
        i. The program shuts down and the window is closed.
    b. **If the 'back' button is clicked by the mouse pointer**
        i. The main_state loop must be initiated, which will trigger a transition of the window.

c. If the 'start node' button is clicked by the mouse pointer

i. The start_node loop is initiated
- This is a loop which occurs during the Pathfinding Visualisation state
- The loop is used to ensure that any mouse click on the graph nodes whilst this button was the last button to have been clicked, changes the state of said node to a start node.

d. If the 'end node' button is clicked by the mouse pointer

i. The end_node loop is initiated
- This is a loop which occurs during the Pathfinding Visualisation state
- The loop is used to ensure that any mouse click on the graph nodes whilst this button was the last button to have been clicked, changes the state of said node to an end node.

e. If the 'barrier' button is clicked by the mouse pointer

i. The barrier loop is initiated
- This is a loop which occurs during the Pathfinding Visualisation state
- The loop is used to ensure that any mouse click on the graph nodes whilst this button was the last button to have been clicked, changes the state of said node to a barrier node.

f. If the 'Pathfinding Type' button is clicked by the mouse pointer

i. The pathfinding_swap subroutine is launched
- This is a subroutine which works by checking which Boolean type a current_pathfinding variable is and passing the variable through a NOT gate. True and False will both be assigned to a pathfinding type, determining what the current pathfinding type is.

ii. The visual graphic of the button interface must switch to mirror the new current pathfinding type.

g. If the 'Heuristic Type' button is clicked by the mouse pointer

i. The heuristic_swap subroutine is launched
- This is a subroutine which works by checking which Boolean type a current_heuristic variable is and passing the variable through a NOT gate. True and False will both be assigned to a pathfinding type, determining what the current pathfinding type is.

ii. The visual graphic of the button interface must switch to mirror the new current heuristic type.

h. If the 'barrier clear' button is clicked by the mouse pointer

i. The barrier_clear method is initiated

- This is a subroutine which contains a validation check for whether there are currently any nodes on the graph which are in the barrier state.
    i. If this validation check returns false, a relevant error message must pop-up to the user in the form of a new appearing window.
    ii. If this validation check returns true, the rest of the subroutine will run.
- This function loops through the graph iteratively to reset the state of each node to its original NoneType state.

i. If the 'clear graph' button is clicked by the mouse pointer

i. The graph_clear method is initiated

- This is a subroutine which contains a validation check for whether there are currently any nodes on the graph which are not in their original NoneType state.
    i. If this validation check returns false, a relevant error message must pop-up to the user in the form of a new appearing window.
    ii. If this validation check returns true, the rest of the subroutine will run.
- This function loops through the graph iteratively to reset the state of each node to its original NoneType state.

j. If the 'Enter' key has been pressed by the user

i. The current pathfinding method is initiated

- This will contain a subroutine which contains a validation check for whether there is both a start and end node on the graph.
    i. If this validation check returns false, a relevant error message must pop-up to the user in the form of a new appearing window.
    ii. If this validation check returns true, the rest of the subroutine will run.
- The output field labels must be updated with the information of the algorithm after its running. This changes the time, shortest path, and total nodes travelled labels, informing the user.

3) Help State

a. If the termination button is clicked by the mouse pointer

i. The program shuts down and the window is closed.

b. If the 'back' button is clicked by the mouse pointer

i. The main_state loop must be initiated, which will trigger a transition of the window.

Programming Paradigm Advantages and Disadvantages

Procedural Programming

Procedural programming is a paradigm which revolves around procedures/subroutines which perform designated tasks. These are akin to functional programming in Python and a direct implementation of Procedural Abstraction. These procedures may be reused throughout the code for the programmer's ease of use. Maintenance is also improved as it is a form of modular programming in terms of the subroutines. This would be implemented as structured programming which utilises the constructs of sequence, selection, iteration, and recursion. This paradigm stresses on utilising variables as they store data, these are not stateless and can change throughout the program's running.

Advantages	Disadvantages
This programming paradigm utilises modular code, as it is easier to maintain and understand the code – additionally can be reused within the code.	This paradigm is not suitable for complex applications requiring flexibility and dynamics – including GUIs, which is a prominent part of my project.
Procedural programming code is usually more efficient and faster than other paradigms as this is optimised for lower-level programming.	Procedural programming is not suited to a large project such as this as there is no encapsulation, due to there being a need for many different 'objects' such as nodes, which will be reused throughout the code.
The paradigm is easier to debug as the code is sequential, the functions can also be tested independently due to their isolation with Procedural Abstraction.	The procedural paradigm may have code redundancy in conjunction with having to recode chunks of main loop code due to this being the sort of code that does not belong in a subroutine.

Object-Oriented Programming

Object-Oriented Programming is a paradigm focused on the creation of objects which contain both attributes and methods. These are entities which follow the principle of abstraction as they work based on encapsulation. Encapsulation is the concept of data and methods being combined into a single unit – this being the class/object. These objects can be made to work in conjunction with each other due to the principles of inheritance and polymorphism. Inheritance relates to the creation of subclasses which 'inherit' characteristics from their superclass. However, subclasses may also have attributes and methods which override from their parent class, making them different to their parent class. Object-Oriented programming is modular, providing many advantages to this paradigm.

Advantages	Disadvantages
The modularity of object-oriented programming means that code can be	Object-oriented programming may perform slower than other paradigms due to the

reused in different parts of the program – leading to efficient code development and maintenance of the code.	overhead necessary in the creation, management, and instantiation of objects.
The encapsulation of object-oriented programming allows classes/objects to have both attributes and methods, preventing data being accidentally rewritten – especially as the attributes may be Private or Protected, meaning that it would take specific code to change the attributes.	The complexity of object-oriented programming makes it more difficult to implement than procedural programming, as class design and relationships must both be considered. A large depth of understanding is also necessary to take advantage of the benefits inheritance and polymorphism provide.
The paradigm allows you to not only make multiple instances of the same class, but also vary the classes and create subclasses, depending on the context.	This programming paradigm may induce over-engineering in the code, which occurs when the code becomes more complex than necessary to achieve a task. Possibly leading to slower development times and maintenance.

Programming Paradigm Conclusion

In conclusion, I believe that the Object-Oriented Programming paradigm is more suitable to the project. This is due to a variety of reasons:

1) OOP Property: Encapsulation
    a. Data and methods are combined into a single object.
        i. Within this project, I would be looking to create a Node object, which would have all the properties that a node within a graph does – the attributes including the node's state, position, neighbours, with its methods changing its state-type, drawing it onto the screen, etc.
    b. Modular programming in objects
        i. The code becomes more organised, easier to understand/modify, and maintain.
        ii. Attributes of the objects may be Public, Private, or Protected. Public attributes may be manipulated within any part of the code, whereas Protected may only be changed within the class's structural hierarchy – only within its own class and subclasses. Private attributes may only be changed within their own class.

2) OOP Property: Inheritance
    a. New classes may be created based on previously existing ones.
        i. For a pathfinding algorithm visualiser, we can create a specialised visualiser class for each algorithm. In example, the Dijkstra visualising class may be based on a general one, or we can base the A* visualising class on the Dijkstra – as it uses the same properties and

methods, excluding the heuristic function, which can be added to the subclass.

ii. Subclasses improve the code's reusability and reduces duplication of the code, as in the creation of the subclass, there is no need to rewrite the parent class's characteristics.

3) OOP Property: Polymorphism
    a. Objects may take on different forms depending on their context
        i. This allows us to implement both Dijkstra and A* algorithms as objects, giving the user the flexibility to choose which algorithm they would like to visualise with both being based on the same interface.
    b. Different heuristics can be implemented
        i. As we are implementing two different shortest-path algorithms, which differ in their use of heuristics – in that Dijkstra does not utilise one but A* pathfinding does, there may be a heuristic interface, which could allow the programmer to add more heuristic methods – such as Euclidean Distance at some point.

Algorithm Design

Pathfinding Overview

Graph Search Algorithms

These are algorithms which are used to traverse graphs for general discovery or a specific purpose – such as finding the shortest path between two nodes. In this context, graphs are considered structures which consist of sets of points connected. The edges between the vertices may be weighted or unweighted – meaning that there may be numerical values associated with them – such as the distance.

Node Analysis

Nodes are the vertices that are considered 'destinations' within pathfinding algorithms. A node is a data structure that contains the data of:

- Its state
- Its parents node – the node through which it was generated.
- The action applied to the parent to reach the current node.

Shortest Path Algorithms

These are graph search algorithms which have a start and end node. The algorithm finds the least costly path between the two vertices. The cost of the path is determined by the sum of the edge weights between the nodes used in traveling to the end node. In practice these weights can translate into anything measurable – such as distance, time, etc., but for the purpose of the project we are dealing with single weight edges between the nodes – therefore you get a path cost of n where n is the amount of nodes that have been traversed.

Nodes in Shortest Path Algorithms

Nodes in a graph used for shortest pathfinding have several features that are important for determining the optimal path. First, each node has a unique identifier or label that distinguishes it from other nodes in the graph. Second, each node has a set of edges or arcs connecting it to other nodes in the graph. Each edge has a weight or cost associated with it, which represents the distance or time required to travel from one node to another.

Another important feature of nodes in shortest pathfinding is their connectivity. The connectivity of a node refers to the number of edges connecting it to other nodes in the graph. Nodes with a higher degree of connectivity have more potential paths leading to them, making them more attractive to the pathfinding algorithm. In some cases, the connectivity of a node may be restricted, such as in a grid-based graph where nodes can only connect to their immediate neighbours.

Dijkstra Algorithm Design

Breadth-First Search Overview

Breadth-First search is an iterative algorithm which travels in multiple directions simultaneously, traveling an edge in all directions prior to continuing. This uses a 'first-in-first-out' queue data structure. The significance of this is that all the nodes adjacent to the start node are added to a queue, after which the start node is dequeued. The first node to have been explored is then dequeued, which means that its neighbouring nodes must then be added to the queue. The next node will then be dequeued and so on. The advantage of this algorithm is that it will always discover the most optimal route to be found, however this comes with the drawback of the algorithm having a run-time that is nearly guaranteed to be longer than the minimal discovery time (and may take the longest possible time to run at worst) – meaning that it is not very efficient. This can, however, be combatted using heuristic techniques, such as the one utilised by the A* pathfinding algorithm.

Breadth-First Search Pseudocode

Example of Breadth-First Pseudocode

```
// Create an empty queue to hold vertices to visit.
CREATE a QUEUE Q

// Create an empty set to keep track of which vertices have been
visited.
CREATE a SET visited, initially Empty

// Add the starting vertex to the queue and mark it as visited.
ADD StartVertex to Q
ADD StartVertex to visited

// Loop until all vertices reachable from the starting vertex have
been visited.
WHILE Q IS NOT Empty:

    // Dequeue the next vertex in the queue and mark it as the
current vertex.
    Current_node ← Dequeue Q
```

```
    // Iterate through all the neighbours of the current vertex.
    FOR each neighbour of Current_node:


        // If the neighbour has not been visited yet, mark it as
visited and add it to the queue.
        IF neighbour IS NOT IN visited:
            ADD neighbour to visited
            ADD neighbour to Q
```

In the context of algorithm visualisation, we would alter the algorithm by assigning colours to the nodes depending on whether they are unexplored, in the queue, and visited.

Breadth-First Search Test-Code Implementation

```
# A function that performs Breadth First Search algorithm starting
from a given vertex
def BreadthFirstSearch(StartVertex):
    # Create an empty Queue to store nodes to be visited
    Q = []
    # Create an empty set to store visited nodes
    visited = set()
    # Add the starting vertex to the Queue and to the visited set
    Q.append(StartVertex)
    visited.add(StartVertex)
    # While the Queue is not empty
    while Q:
        # Dequeue the current node from the Queue
        Current_node = Q.pop(0)
        # For each neighbor of the current node
        for neighbor in Current_node.neighbors:
            # If the neighbor has not been visited yet
            if neighbor not in visited:
                # Add the neighbor to the visited set and the Queue
                visited.add(neighbor)
                Q.append(neighbor)
```

Breadth-First Search Time Complexity

The Breadth-First Search Algorithm has a time complexity which is based on the representation of the graph that it is used on. Usually, the time complexity of the Breadth-

First Search algorithm is $O(V + E)$ for an adjacency list graph implementation and $O(V^2)$, where V is the number of vertices in the graph and E is the number of edges in the graph. The justification for this time complexity is that it visits each node a single time because visiting a single vertex and its adjacent edges takes a time complexity of $O(1)$ in an adjacency list and $O(V)$ in a matrix representation.

Breadth-First Search Space Complexity

The space complexity of the Breadth-First Search algorithm is dependent on the number of vertices in the graph it is used on. Therefore, it has a time complexity of $O(V)$, where V is the number of vertices in the graph. This is because each vertex is only visited a single time and the maximum number of vertices that can be within the queue at a single point is equal to the number of vertices at the maximum distance from the start node.

Dijkstra Algorithm Analysis

The Dijkstra Shortest-Path Algorithm is based on the Breadth-First Search Algorithm and varies it to include edge weights – allowing the shortest path to be found. In its initial state, the algorithm assigns distances to each node, the starting node distance being 0 with the distance of all other nodes being infinity. It then selects the node with the smallest distance and considers all its neighbours. For each neighbouring vertex, it computes the distance to that node by adding the distance to the current node and the weight of the edge between the two nodes. If the computed distance is smaller than the current distance of the neighbour, the neighbour's distance is updated to the new, smaller distance. The algorithm then iteratively selects the node with the smallest distance, considers its neighbours, and updates their distances until the destination node is reached or until all reachable nodes have been explored.

The Dijkstra pathfinding algorithm functions by incrementally discovering the shortest path to each node. The algorithm is considered 'greedy' as it chooses the most optimal distance every time. It also sets distances immutably – as the distance to a node will not be changed after first being assigned. The algorithm is not the most efficient when faced with large edge to node ratios though this can be improved upon using a heuristic technique as we will see in A* Pathfinding.

Dijkstra Algorithm Pseudocode

Example of Dijkstra Shortest-Path Pseudocode

```
// Create an empty set to keep track of visited vertices and an
empty set for unvisited vertices.

CREATE Empty SET visited

CREATE Empty SET unvisited
```

```
// Add the starting vertex to the unvisited set with a distance of
0.
ADD StartVertex to unvisited with distance 0


// Loop until all vertices have been visited.
WHILE unvisited IS NOT Empty:


    // Find the unvisited node with the smallest distance from the
start vertex.
    current ← node IN unvisited with smallest distance


    // Remove the current node from the unvisited set and add it to
the visited set.
    REMOVE current from unvisited AND ADD it to visited


    // Iterate through the neighbours of the current node.
    FOR each neighbour of current:


        // If the neighbour has not been visited yet:
        IF neighbour IS NOT IN visited:


            // Calculate the tentative distance from the start
vertex to the neighbour through the current node.
            tentative_distance ← distance[current] + distance
between current AND neighbour


            // If the tentative distance is smaller than the current
distance or the neighbour has not been added to the unvisited set
yet:
            IF tentative_distance < distance[neighbour] OR neighbour
NOT IN unvisited:
```

```
                // Add the neighbour to the unvisited set with the
tentative distance.
                ADD neighbour to unvisited with tentative_distance


                // Update the previous node of the neighbour to the
current node.
                UPDATE previous[neighbour] to current


                // Update the distance of the neighbour to the
tentative distance.
                UPDATE distance[neighbour] to tentative_distance


// Return the distances and previous nodes for each vertex.
RETURN distance, previous
```

Dijkstra Algorithm Test-Code Implementation

```
def Dijkstra(graph, start_vertex):
    # Create an empty set to keep track of visited vertices and an
empty set for unvisited vertices.
    visited = set()
    unvisited = set([(start_vertex, 0)])

    # Create dictionaries to keep track of the distances and
previous nodes for each vertex.
    distance = {start_vertex: 0}
    previous = {}

    # Loop until all vertices have been visited.
    while unvisited:

        # Find the unvisited node with the smallest distance from
the start vertex.
        current, current_distance = min(unvisited, key=lambda x:
x[1])

        # Remove the current node from the unvisited set and add it
to the visited set.
        unvisited.remove((current, current_distance))
        visited.add(current)
```

```
        # Iterate through the neighbours of the current node.
        for neighbour, neighbour_distance in graph[current].items():

            # If the neighbour has not been visited yet:
            if neighbour not in visited:

                # Calculate the tentative distance from the start
vertex to the neighbour through the current node.
                tentative_distance = current_distance +
neighbour_distance

                # If the tentative distance is smaller than the
current distance or the neighbour has not been added to the
unvisited set yet:
                if (
                    neighbour not in distance
                    or tentative_distance < distance[neighbour]
                ):

                    # Add the neighbour to the unvisited set with
the tentative distance.
                    unvisited.add((neighbour, tentative_distance))

                    # Update the previous node of the neighbour to
the current node.
                    previous[neighbour] = current

                    # Update the distance of the neighbour to the
tentative distance.
                    distance[neighbour] = tentative_distance

    # Return the distances and previous nodes for each vertex.
    return distance, previous
```

Dijkstra Algorithm Time Complexity

Dijkstra's pathfinding algorithm has a time complexity which is dependent on the data structure used to implement it as well as the data structure responsible for the graph. As the graph I am going to implement is based on a matrix structure – 2D array structure within Python – the time complexity would be dependent on the structure used to implement the Priority Queue for the Dijkstra Algorithm. As during the instance at which pathfinding will be available there will be a set graph dimension size, we will be able to calculate the exact time complexity of the algorithms for each dimension size once the priority queue structure is chosen. This is because the time complexity is based on how many vertices are within the

graph and number of edges available to each node. The edges value will always be constant at 4, because each node will have at most 4 neighbours.

Dijkstra Algorithm Space Complexity

The space complexity of Dijkstra's algorithm depends on a multitude of factors as there are many data structures used within the process. Firstly, a factor of the space complexity is the data structure used to represent the graph. In the likely case of representing the graph using a matrix, the space complexity will be $O(V^2)$, where V is the number of vertices within the graph. However, there may be larger space complexities within the operation so these must be checked as well. The priority queue used also has its own space complexity, which will depend on the type of data structure implementation I choose to base the queue on. Lastly, the hash table data structure I will be using to store the hashed open set nodes will also have its own space complexity. Which will have to be factored into the equation at the end.

A* Algorithm Design

Greedy Best-First Search Overview

This search method is an informed search algorithm, meaning that it utilises a heuristic function h(n) to determine which of the neighbouring nodes are closest to the end node. The efficiency of the algorithm is notably superior to that of uninformed algorithms such as Breadth-First Search (and Dijkstra as a by-product) but depends on the quality of the heuristic function. It works in a similar fashion to the Dijkstra algorithm in that it utilises a priority queue to choose the path that is best at that moment – making it greedy – but the nodes' priority is determined by their distance from the end node, which is found using the heuristic function.

A* Shortest-Path Algorithm Analysis

The A* algorithm builds upon the Greedy Best-First Search as it utilises not only the heuristic function h(n) – determining the distance of each node from the end node – but also the Dijkstra distance function, which allows the algorithm to determine the distance from the start node to the current node. The algorithm therefore has a more accurate way of giving priority to nodes as it determines the path. The algorithm tracks the current path's distance and the distance to travel, allowing it to backtrack if the path's cost exceeds that of a previous node's – allowing it to avoid inefficient paths. A* Pathfinding is guaranteed to find the shortest path so long as the heuristic function does not overestimate the true distance to the end node.

Manhattan Distance

Manhattan distance is a mathematical concept which counts the number of steps – in the (x, y) plane – it takes to travel from one node to another. This is estimated by taking both nodes' coordinate points and adding up the absolute distances between their x and y coordinates. This is commonly used as the heuristic technique in A* pathfinding algorithms and will be the technique I employ within my program.

Euclidean Distance

Euclidean distance is a measure of the distance between two points in Euclidean space. It is the straight-line distance between the two points, which is also known as the Pythagorean distance – calculated by $D = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$, where the two node coordinates are (x1, y1) and (x2, y2).

A* Algorithm Pseudocode

Example of A* Shortest-Path Pseudocode

```
open_list ← [StartVertex]  // Initialize the open list with the
starting node

closed_list ← []  // Initialize the closed list as empty
```

```
distance ← {StartVertex: 0}  // Initialize the cost from StartVertex
to each node as 0

f_score ← {StartVertex: manhattan_distance(StartVertex, EndVertex)}
// Initialize the total estimated cost as Manhattan distance from
StartVertex to EndVertex


WHILE open_list IS NOT Empty:

    current_node ← get_node_with_lowest_f_score(open_list, f_score)
// Find the node with the lowest f_score in the open list


    IF current_node == EndVertex: // If the current node is the
EndVertex, we have found the path

        RETURN reconstruct_path(StartVertex, EndVertex) // Return
the path from start to end


    open_list.remove(current_node)  // Remove the current node from
the open list since it has been visited

    closed_list.append(current_node) // Add the current node to the
closed list


    FOR neighbour IN current_node.neighbors: // For each neighbour
of the current node

        IF neighbour IN closed_list: // Skip if already visited

            CONTINUE


        tentative_distance ← distance[current_node] +
manhattan_distance(current_node, neighbour) // Calculate the
tentative distance from StartVertex to neighbour through the current
node


        IF neighbour NOT IN open_list: // If the neighbour is not in
the open list, add it

            open_list.append(neighbour)

        ELIF tentative_distance >= distance[neighbour]: //
Otherwise, if the tentative distance is not better, skip
```

```
            CONTINUE


        neighbour.came_from ← current_node  // Set the best current
path to the current node

        distance[neighbour] ← tentative_distance // Set the distance
to the tentative distance

        f_score[neighbour] ← tentative_distance +
manhattan_distance(neighbour, EndVertex) // Set the f score to the
sum of the tentative distance and the estimated distance from the
neighbour to the end vertex
```

A* Algorithm Test-Code Implementation

```
def a_star(start, end):
    open_list = [start]  # Initialize the open list with the
starting node
    closed_list = []  # Initialize the closed list as empty
    distance = {start: 0}  # Initialize the cost from StartVertex to
each node as 0
    f_score = {start: manhattan_distance(start, end)}  # Initialize
the total estimated cost as Manhattan distance from start to end

    while open_list:
        current_node = get_node_with_lowest_f_score(open_list,
f_score)  # Find the node with the lowest f_score in the open list

        if current_node == end:  # If the current node is the end,
we have found the path
            return reconstruct_path(start, end)

        open_list.remove(current_node)  # Move the current node from
open list to closed list
        closed_list.append(current_node)

        for neighbour in current_node.neighbors:  # For each
neighbour of the current node
            if neighbour in closed_list:  # Skip if already visited
                continue

            tentative_distance = distance[current_node] +
manhattan_distance(current_node, neighbour)  # Calculate the
tentative distance from start to neighbour through the current node
```

```
            if neighbour not in open_list:  # If the neighbour is
not in open list, add it
                open_list.append(neighbour)
            elif tentative_distance >= distance[neighbour]:  #
Otherwise, if the tentative distance is not better, skip
                continue

            neighbour.came_from = current_node  # Update the
neighbour's best current path
            distance[neighbour] = tentative_distance
            f_score[neighbour] = tentative_distance +
manhattan_distance(neighbour, end)  # Update the neighbour's f score

    # If we get here, there is no path from start to end
    raise ValueError("No path found")
```

A* Algorithm Time Complexity

The A* algorithm has a time complexity based on a variety of factors. Firstly, it is influenced by the quality of the heuristic function. The heuristic function has the potential to significantly reduce search space and therefore time complexity of the algorithm. Secondly, the time complexity is dependent on the data structure being used to represent the graph. In the case of a matrix representation, this complexity would be $O(V^2)$. Finally, the time complexity is also influenced by the number of nodes expanded during the search.

A* Algorithm Space Complexity

The space complexity of the A* algorithm is dependent on the data structure used to represent the graph – as before – as well as the number of nodes expanded (worst case scenario results in a complexity of $O(V)$ as every single vertex would have been visited), and the heuristic function's space complexity, along with the data structure used to implement the algorithm itself.

Data Structure Design

Priority Queue Design

Priority Queue Overview

Queues are complex data structures which are based on the 'First in First Out' principle. This means that the data which is appended to the queue first will be the first to be dequeued. These elements can only be dequeued from the front of the queue and elements are moved using front and rear pointers. Priority queues differ because they account for their elements' priorities. This means that the logical order of the elements inside of the queue depends on their priority – with the highest priority items moving to the front and the lowest priority moving to the back. I plan to design my own Priority Queue within the project to function within the pathfinding algorithms as Python does not have a non-external module Priority Queue data structure.

Default Priority Queue Algorithms

Algorithm	Input Parameters	Description	Returns
Push	• Item • Priority	This algorithm will be used to append the item to the priority queue with its priority.	None
Pop	None	This is the algorithm which will allow the highest priority item to be popped from the queue. However, it first checks if the list is empty, if so, it returns a ValueError.	Item with highest priority
Is_Empty	None	This is the algorithm which checks if the queue is empty and returns True if it is.	Boolean Value

Array-Based Priority Queue Pseudocode

```
DEF PriorityQueue():
    queue ← create QUEUE
    size ← 0
```

```
    DEF enqueue(item, priority):
        index ← size
        WHILE index > 0 AND queue[(index-1)//2][1] < priority:
            queue[index] ← queue[(index-1)//2]
            index ← (index-1)//2
        queue[index] ← [item, priority]
        size ← size + 1

    DEF dequeue():
        IF size == 0:
            RETURN "Queue is empty"
        root ← queue[0]
        size ← size - 1
        last ← queue[size]
        index ← 0
        WHILE index*2+1 < size:
            child ← index*2+1
            IF child+1 < size AND queue[child+1][1] >
queue[child][1]:
                child ← child+1
            IF last[1] >= queue[child][1]:
                BREAK
            queue[index] ← queue[child]
            index ← child
        queue[index] ← last
        RETURN root[0]
```

Array-Based Priority Queue Test-Code Implementation

```
class PriorityQueue:
    """
```

```
    Priority Queue is a class to implement a priority queue data
structure.
    """

    def __init__(self):
        """
        Initialize an empty queue.
        """
        self._queue = []

    def push(self, item, priority):
        """
        Pushes an item into the queue with a given priority.
        :param item: item to be added
        :param priority: priority of the item
        """
        entry = (priority, item)
        self._queue.append(entry)

    def pop(self):
        """
        Pops the item with the highest priority from the queue.
        :return: item with the highest priority
        """
        if len(self._queue) == 0:
            raise ValueError("Queue is empty")
        highest = 0
        for i in range(len(self._queue)):
            if self._queue[i][0] < self._queue[highest][0]:
                highest = i
        entry = self._queue[highest]
        del self._queue[highest]
        return entry[1]

    def is_empty(self):
        """
        Checks if the queue is empty.
        :return: True if the queue is empty, False otherwise
        """
        if len(self._queue) == 0:
            return True
```

Array-Based Priority Queue Space Complexity

The array-based priority queue does not have a particularly good space complexity as it depends on the number of elements currently in the queue. Therefore, in the worst-case scenario, an implementation such as this may have a space complexity of $O(N)$, where n is

the number of items added to the queue. This, however, fluctuates with how many items are currently in the queue as some may be popped over time, etc.

Array-Based Priority Queue Local Time Complexity

The array-based priority queue's time complexity depends on the methods within it. As such, we can assess the test implementation to see that the push method will have a complexity of $O(1)$, the is_empty method will also have a time complexity of $O(1)$, but the pop method will have a worst-case time complexity of $O(N)$, which is the size of the queue. The true time complexity of the method, however, may be less in practice.

Array-Based Priority Queue Global Time Complexity

The array-based priority queue also influences the algorithms which use it to function. In the case of this project, that is the Dijkstra and A* algorithms. Observing the queue structure's effect on the algorithm shows that it is unsuitable to the task/may be optimised heavily, as it results in the algorithms having a time complexity of $O(V^2)$, where V is the number of vertices. The reason for this is that this implementation uses a linear search to find the vertex with the highest priority, which takes O(V) time. Since the algorithm needs to extract the vertex with the minimum distance from the priority queue in each iteration, and there are V iterations, the total time complexity becomes O(V^2). Therefore, we must find a new alternative to this priority queue implementation.

Priority Queue Alternative Structure Analysis

Binary Heap

This is a tree-based data structure that can be used to implement a priority queue efficiently. It has a logarithmic time complexity for insertion and deletion of elements, and a constant time complexity for accessing the highest-priority element.

Advantages:

1. Simple to implement and easy to understand.
2. Space-efficient, as it only needs to store the elements in an array.
3. Fast access time for the highest-priority element.

Disadvantages:

1. Insertion and deletion operations have logarithmic time complexity in the worst case.
2. May not be the most efficient implementation for certain types of operations, such as merging or decreasing the priority of an element.
3. Not well-suited for dynamic resizing.

Fibonacci Heap

A more advanced tree-based data structure that can also be used to implement a priority queue efficiently. It has very good time complexities specifically for operations that involve merging or decreasing the priority of an element.

Advantages:

1. Better time complexity for certain operations than a binary heap.
2. Good for handling large amounts of data, as it can handle many insertions and deletions without incurring significant performance degradation.
3. Can handle dynamic resizing better than a binary heap.

Disadvantages:

1. More complex to implement and understand than a binary heap.
2. Higher constant factors and overhead than a binary heap.
3. May not perform as well as other data structures in certain scenarios, such as in the case of small or sparse data sets.

Pairing Heap

Another type of tree-based data structure that can be used to implement a priority queue. It has an amortized constant time complexity for insertion and deletion of elements, and a logarithmic time complexity for accessing the highest-priority element.

Advantages:

1. Constant time complexity for insertion and deletion operations on average.
2. Simple to implement and understand.
3. Good for handling dynamic resizing.

Disadvantages:

1. Slower access time for the highest-priority element compared to other heap data structures.
2. Higher constant factors and overhead than other heap data structures.
3. May not perform as well as other data structures in certain scenarios, such as in the case of small or sparse data sets.

Conclusion

In conclusion, I believe that a Binary Heap design would be the most suitable for the project. It will be a relatively straightforward implementation and cause a noticeable improvement in the time complexity of the algorithms. The Fibonacci heap was also a potential option but the complexity in programming it is not worth the limited advantage it has over the Binary Heap.

Binary Heap Design

Binary Heap Overview

As mentioned earlier, this is a binary tree data structure. Hence, each node inside of the heap has at most two children with the tree always being complete. This means that all levels of the tree must be filled – other than the final level, which may be filled from left to right. It is normally implemented through an array system with the parent to child relationship being represented by, for example, the left child of a node at index n being located at index 2n + 1, which the right child at 2n + 2. The father of the node at index n would be located at index floor($\frac{n-1}{2}$), where floor is a function that intakes a real number and outputs a number that is the greatest integer less than or equal to the input.

The heap itself is maintained through a series of swaps after the rear-insertion of the newest element. The swaps ensure that the root of the tree has the minimum (or potentially maximum) value. The removal of the root element is done by swapping it with the last element in the array, removing the last element, and then completing a series of swaps which restore the heap. The structure has a time complexity of $O(\log N)$ for the insert and remove operations, where n is the number of elements inside.

Binary Heap Algorithms

Algorithm	Input Parameters	Description	Returns
Push	• Item • Priority	This algorithm adds an item with a priority to the queue. It takes an item and a priority as input parameters, creates a tuple of (priority, item) and appends it to the _queue. The sift_up method is called with the index of the last item in the list to maintain the heap property.	None
Pop	None	This is the algorithm that removes and returns the item with the highest priority from the queue. It raises a ValueError if the queue is empty. The item at the root	Item with highest priority

		of the heap (the first item in the _queue list) is removed and stored in a variable entry. If the _queue list is not empty, the last item in the list is moved to the root position and sift_down method is called with the index of the root to maintain the heap property. The item stored in entry is returned.	
Is_Empty	None	This is the algorithm which checks if the queue is empty and returns True if it is.	Boolean Value
Sift_Up	Index	The algorithm which performs the sift-up operation on the heap. It takes an index as an input parameter and compares the priority of the item at that index with its parent node. If the priority is less than the parent node, the two nodes are swapped, and the index is updated to its parent's index. This continues until the parent has a lower priority than the current node or the root of the heap is reached.	None
Sift_Down	Index	The algorithm that performs the sift-down operation on the heap. It takes an index as an input parameter and	None

		compares the priority of the item at that index with its child nodes. The smallest child node is swapped with the current node, and the index is updated to the smallest child's index. This continues until the current node has a smaller priority than its child nodes or it is a leaf node.	

Binary Heap Priority Queue Structure Pseudocode

```
CLASS PriorityQueue
    FUNCTION __init__()
        queue ← []
    END FUNCTION

    FUNCTION push(item, priority)
        entry ← (priority, item)
        queue.ADD(entry)
        SIFT_UP(queue, LEN(queue) - 1)
    END FUNCTION

    FUNCTION update_priority(item, new_priority)
        FOR i FROM 0 TO LEN(queue)-1
            IF queue[i][1] EQUALS item
                old_priority ← queue[i][0]
                queue[i] ← (new_priority, item)
                IF new_priority < old_priority
                    SIFT_UP(queue, i)
                ELSE
```

```
                SIFT_DOWN(queue, i)
            END IF
        END IF
    END FOR
END FUNCTION

FUNCTION pop()
    IF LEN(queue) EQUALS 0
        RAISE ValueError("Queue is empty")
    END IF
    entry ← queue[0]
    last_entry ← queue.POP()
    IF LEN(queue) > 0
        queue[0] ← last_entry
        SIFT_DOWN(queue, 0)
    END IF
    RETURN entry[1]
END FUNCTION

FUNCTION is_empty()
    IF LEN(queue) EQUALS 0
        RETURN True
    ELSE
        RETURN False
    END IF
END FUNCTION

FUNCTION _sift_up(queue, index)
    parent_index ← (index - 1) DIV 2
    WHILE index > 0 AND queue[index][0] < queue[parent_index][0]
```

```
            queue[index], queue[parent_index] ← queue[parent_index],
queue[index]
            index ← parent_index
            parent_index ← (index - 1) DIV 2
        END WHILE
    END FUNCTION

    FUNCTION _sift_down(queue, index)
        left_child_index ← 2 * index + 1
        right_child_index ← 2 * index + 2
        smallest_child_index ← index
        IF left_child_index < LEN(queue) AND
queue[left_child_index][0] < queue[smallest_child_index][0]
            smallest_child_index ← left_child_index
        END IF
        IF right_child_index < LEN(queue) AND
queue[right_child_index][0] < queue[smallest_child_index][0]
            smallest_child_index ← right_child_index
        END IF
        IF smallest_child_index ≠ index
            queue[index], queue[smallest_child_index] ←
queue[smallest_child_index], queue[index]
            _sift_down(queue, smallest_child_index)
        END IF
    END FUNCTION
END CLASS
```

Binary Heap Priority Queue Test-Code Implementation

```
class BinaryHeap:
    def __init__(self):
        self.heap = []
```

```
    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def insert(self, item):
        self.heap.append(item)
        self.heapify_up(len(self.heap) - 1)

    def heapify_up(self, i):
        while i > 0 and self.heap[i] < self.heap[self.parent(i)]:
            self.swap(i, self.parent(i))
            i = self.parent(i)

    def pop(self):
        if len(self.heap) == 0:
            raise ValueError("Heap is empty")

        self.swap(0, len(self.heap) - 1)
        item = self.heap.pop()
        self.heapify_down(0)

        return item

    def heapify_down(self, i):
        while i < len(self.heap):
            min_index = i

            left = self.left_child(i)
            if left < len(self.heap) and self.heap[left] <
self.heap[min_index]:
                min_index = left

            right = self.right_child(i)
            if right < len(self.heap) and self.heap[right] <
self.heap[min_index]:
                min_index = right

            if i != min_index:
                self.swap(i, min_index)
                i = min_index
```

```
            else:
                break
```

Binary Heap Priority Queue Test Flaws

1. It doesn't include a priority queue interface, making it unclear how to use this implementation to create a priority queue.
2. It doesn't allow for duplicate values, as it assumes that each value is unique.
3. The pop() method can raise an IndexError if called on an empty heap, but the raise ValueError("Heap is empty") statement will raise a ValueError instead.
4. The heapify_down() method should be checking the value at min_index against the value at i to determine whether to continue swapping values.
5. The heapify_down() method doesn't need to continue swapping values if the minimum value is already at the root of the subtree (i.e., if i == min_index).
6. It does not currently function for A* pathfinding as it does not provide the support to change an element's priority. In the current implementation it would take a full queue search to change an element's priority, which would result in a time complexity of $O(N)$.

These are all key points which I will take into consideration when programming the final binary heap which will be used within my implementation. I will aim to eliminate all these flaws. It will also be necessary to modify the binary heap so that it updates an element's priority.

Binary Heap Priority Queue Local Time Complexity

The Binary Heap priority queue implementation has a time complexity of $O(\log N)$ for the insertion and removal of elements. During the insertion, an element is added to the bottom of the heap and must have its priority repeatedly compared to its parent's – resulting in a swap if required. The process takes $O(\log N)$ time as the height of the binary heap is $\log N$, where n is the number of elements inside the heap.

During removal, the max/min element is swapped with the last element and then continuously has its priority compared against its children's – swapping them when necessary. The process takes $O(\log N)$ time as the height of the binary heap is $\log N$, where n is the number of elements inside the heap.

The total time complexity of a binary heap is, however, dependent on its application. As will be noted within the global time complexity section.

Binary Heap Priority Queue Global Time Complexity

The two applications of the Binary Heap Priority Queue that I will be looking at are for the Dijkstra and A* pathfinding algorithms, as those are the two relevant ones within the project. For the both the Dijkstra and A* algorithms, the time complexity when it utilises this data structure implementation is $O((E + V) \log V)$, where E represents the number of edges available to each node and V represents the number of vertices within the graph. The time complexity of the binary heap priority queue for push and pop operations is $O(\log N)$, where N is the number of elements in the queue. In each iteration of the algorithms, one vertex is popped from the priority queue and its adjacent vertices are updated with the minimum distance. Since each vertex is pushed and popped from the priority queue at most once, the total number of push and pop operations is proportional to the number of vertices V.

Binary Heap Priority Queue Space Complexity

A binary heap priority queue structure has a space complexity of $O(N)$, where N is the number of elements inside of the heap. This is due to it being implemented as a binary tree, requiring $O(N)$ space to store N elements.

Binary Heap/Array-Based P-Queue Time Complexity Comparison

Noting that I have decided to allow the user to choose between 4 graph dimension sizes – 10x10, 25x25, 50x50, and 100x100; I can now do a concrete comparison between the two priority queue implementations' time complexities using the numbers.

10x10 Calculation

Array-Based Time Complexity

$O(V^2)$ where V represents the number of vertices

$10 \times 10 = 100$ vertices

Therefore,
$O(100^2)$ = $O(10{,}000)$

Binary Heap Time Complexity

$O((E + V) \log V)$ where V represents the number of vertices and E represents the number of edges

$10 \times 10 = 100$ vertices

Calculating Graph Edges:

For a graph sized DxD, $E = 2D^2 - 2D$

Therefore,

$E = 2 \times 100 - 20 = 180$ Edges

Using it in the equation:

$O((180 + 100) \log 100) = O(560)$

Calculating % Difference:

$$\frac{(10{,}000 - 560)}{560} \times 100 = 1685.70$$

As can be seen, there is approximately a 1690% difference in the time complexity of the array-based priority queue in comparison to the binary heap priority queue.

25x25 Calculation

Array-Based Time Complexity

$O(V^2)$ where V represents the number of vertices

$25 \times 25 = 625$ vertices

Therefore,
$O(625^2) = O(390{,}625)$

Binary Heap Time Complexity

$O((E + V) \log V)$ where V represents the number of vertices and E represents the number of edges

$25 \times 25 = 625$ vertices

Calculating Graph Edges:

For a graph sized DxD, $E = 2D^2 - 2D$

Therefore,

$E = 2 \times 625 - 50 = 1{,}200$ Edges

Using it in the equation:

$O((1{,}200 + 625) \log 100) = O(3{,}650)$

Calculating % Difference:

$$\frac{(390{,}625 - 3{,}650)}{3{,}650} \times 100 = 10{,}602.05$$

As can be seen, there is approximately a 10,603% difference in the time complexity of the array-based priority queue in comparison to the binary heap priority queue.

50x50 Calculation

Array-Based Time Complexity

$O(V^2)$ where V represents the number of vertices

$50 \times 50 = 2{,}500$ vertices

Therefore,
$O(2{,}500^2) = O(6{,}250{,}000)$

Binary Heap Time Complexity

$O((E + V) \log V)$ where V represents the number of vertices and E represents the number of edges

$50 \times 50 = 2{,}500$ vertices

Calculating Graph Edges:

For a graph sized DxD, $E = 2D^2 - 2D$

Therefore,

$E = 2 \times 2{,}500 - 100 = 4{,}900$ Edges

Using it in the equation:

$O((4{,}900 + 2{,}500) \log 100) = O(14{,}800)$

Calculating % Difference:

$$\frac{(6{,}250{,}000 - 14{,}800)}{14{,}800} \times 100 = 42{,}129.73$$

As can be seen, there is approximately a 42,130% difference in the time complexity of the array-based priority queue in comparison to the binary heap priority queue.

100x100 Calculation

Array-Based Time Complexity

$O(V^2)$ where V represents the number of vertices

$100 \times 100 = 10{,}000$ vertices

Therefore,
$O(10{,}000^2)$ = $O(100{,}000{,}000)$

Binary Heap Time Complexity

$O((E + V) \log V)$ where V represents the number of vertices and E represents the number of edges

$100 \times 100 = 10{,}000$ vertices

Calculating Graph Edges:

For a graph sized DxD, $E = 2D^2 - 2D$

Therefore,

$E = 2 \times 10{,}000 - 200 = 19{,}800$ Edges

Using it in the equation:

$O((19{,}800 + 10{,}000) \log 100) = O(59{,}600)$

Calculating % Difference:

$$\frac{(100{,}000{,}000 - 59{,}600)}{59{,}600} \times 100 = 167{,}685.23$$

As can be seen, there is approximately a 167,685% difference in the time complexity of the array-based priority queue in comparison to the binary heap priority queue.

Conclusion

From the results that have been obtained, I believe there is a stark contrast between how well the two Priority Queue design structures scale. This scalability – especially for the 100x100 graph, is a major justification for the necessity behind implementing a Binary Heap Priority Queue structure. A 167,685% increase in time complexity is staggeringly large. This, of course, is not completely felt as the true time complexity of the algorithms is based on more than just the data structure time complexity, but this is still an integral part of the final result's optimisation.

Hashing Table Design

Hashing Table Overview

A Hashing Table is a data structure which is used to map keys to values to increase the efficiency of looking up and retrieving data. It combines an array with an algorithm which intakes a key and returns a destination index for the item being inserted into the array. When an item needs to be retrieved from the table, the algorithm is run again, and the value stored at the index in the table is returned. In the context of pathfinding algorithms, hashing tables are normally used to keep track of which nodes are inside of the open set, etc. The justification for why a custom Hashing Table design is required within this program is that due to my graph sizes being limited – e.g., 100x100 – there is a limited number of nodes that may be stored inside. This means that by using programming a minimal perfect hashing algorithm, I can ensure that no data collisions may occur within the table, whilst minimising the space complexity of the table to be exactly fitting to the number of items required.

Default Hashing Table Algorithms

Algorithm	Input Parameters	Description	Returns
Hashing_Algorithm	Item	This is the hashing algorithm which is used on the item to find an index for the item inside of the table.	Hashed_Value
Insert	Item	This is the algorithm responsible for adding the item to the table. It utilises Hashing_Algorithm to find the index at which the item should be placed and appends it to the Table in that location. In imperfect systems collisions may occur, meaning that this method may require a collision-avoidance system.	None
Remove	Item	This is the algorithm used to remove an item from the Table.	None

		It first utilises Hashing_Algorithm to find the index of the item, and then removes it.	

Minimal Perfect Hash Algorithm

This is an algorithm which maps keys – usually the item's attributes, etc – to a set of unique integers. This means that no collisions occur, hence the 'perfect'. It is a minimalistic algorithm as it minimises the amount of memory being used to store the items. The algorithm is meant to achieve the state of being able to produce a hash function which can be evaluated in $O(1)/Constant\ Time$ for any element of the set.

Implementing this for vertices within a graph could be done by using a conjunction of the node's x and y coordinates – as the combination of the two will be completely unique to the node and hashing it with relation to the size of the table, which will match the dimensions of the current graph at that time.

Collision Avoidance

The method I am interested in using to avoid collisions and maintain the minimal perfect hash is one such that we form a tuple out of the x and y coordinates of a vertex. This tuple is then passed in and we extract the x and y coordinates from it. The x value would be multiplied by the square root of the hash table's size, the value of the y coordinate would then be added to this. To ensure that the resulting index is within the bounds of the hash table, a remainder of the result divided by the size of the hash table is taken.

Hashing Table Pseudocode

```
DEF PriorityQueue():
    queue ← create QUEUE
    size ← 0

    DEF enqueue(item, priority):
        index ← size
        WHILE index > 0 AND queue[(index-1)//2][1] < priority:
            queue[index] ← queue[(index-1)//2]
            index ← (index-1)//2
        queue[index] ← [item, priority]
```

```
        size ← size + 1

    DEF dequeue():
        IF size == 0:
            RETURN "Queue is empty"
        root ← queue[0]
        size ← size - 1
        last ← queue[size]
        index ← 0
        WHILE index*2+1 < size:
            child ← index*2+1
            IF child+1 < size AND queue[child+1][1] >
queue[child][1]:
                child ← child+1
            IF last[1] >= queue[child][1]:
                BREAK
            queue[index] ← queue[child]
            index ← child
        queue[index] ← last
        RETURN root[0]

        DEF HashTable(rows) -> None:
        """
        Initialize the hash table with a given number of rows.
        :param rows: number of rows
        """
        size ← next_prime(rows**2)
        table ← [[] FOR _ IN RANGE(size)]
```

```
    DEF hash(tup) -> int:
        """
        Hash a given tuple to a corresponding index in the hash
table.
        :param tup: the given tuple
        :return: the corresponding index in the hash table
        """
        hash_val ← INT((tup.x * size**0.5 + tup.y) % size)

        RETURN hash_val

    DEF insert(tup) -> None:
        """
        Insert a given tuple into the hash table.
        :param tup: the tuple to be inserted
        """
        hash_val ← hash(tup)
        table[hash_val].APPEND(tup)

    DEF remove(tup) -> None:
        """
        Remove a given tuple from the hash table.
        :param tup: the tuple to be removed
        """
        hash_val ← hash(tup)
        IF tup IN table[hash_val]:
            table[hash_val].REMOVE(tup)
```

Hashing Table Time Complexity

The 3 main operations within the Hashing Table form the options for its time complexity. Firstly, the Hashing_Algorithm will have a time complexity of $O(1)$, as it performs a constant number of arithmetic operations to calculate the hash value. Secondly, the insert algorithm will also have a time complexity of $O(1)$, as due to the minimal perfect hash algorithm there will be no collisions and each vertex in the graph will be mapped to its own individual index in the hash table. Lastly, the time complexity of the remove method will be much the same as the insert function – also gaining a time complexity of $O(1)$ due to the minimal perfect hash algorithm. This means that the overall time complexity of the hash table should be constant time, which is very beneficial as it is used within both pathfinding algorithms in the project.

Hashing Table Space Complexity

The space complexity of the hashing table implementation for a pathfinding graph will be $O(V)$, where V is the number of vertices in the graph. This is because in a worst-case situation, the hashing table must be able to store every node of the graph at once, this is reflected in the space complexity.

Visualisation Algorithm Design

Displaying Node State Shifts

I plan to give nodes a state attribute, which will allow me to store the phase of the algorithm each node is in. This means that by assigning colours to each stage of the algorithm I can visualise the nodes shifting colours as they are appended and dequeued from the priority queue.

The colour assignment would be as follows:

Stage	Colours
Start Node	Green
End Node	Red
Barrier Node	Black
Dequeued Node	Light Blue
Queued Node	Dark Blue
Determined Shortest Path Node	Yellow

Implementing Visualisation

I plan to visualise the graph using the Pygame module in Python. Pygame is a Python module which is commonly used to program games and provide graphical user interfaces. Pygame's modules provide functionality to draw graphics and handle user input. The user input functions will allow me to make the visualiser interactive, allowing the user to change the states of nodes, etc.

For the graph dimensions input and pathfinding statistics I will be using the Tkinter module in Python. This is a standard GUI package for Python which provides tools to program user interfaces into your software. As it has pre-existing error message boxes and user input handling, I will be working around these features to make a system that suits my needs for the project.

Object-Oriented Programming Class Design

Software State Objects Identification

Menu_State Class

Overview

The Menu_State class is the object responsible for the state of the software during which the user is in the Main Menu. This is the state at which the user has two option buttons – to enter the Pathfinding state or the Help Menu state. This is the default state that will be presented to the user upon launching, the user may come back to this state to swap between the Pathfinding and Help Menu states.

Attributes

Attribute	Data Type	Purpose
x	Integer	This is the x coordinate position on the screen display which correlates to the coordinate at which a button should be placed to be in the centre of the screen.
y	Integer	This is the y coordinate position on the screen display which correlates to the coordinate at which a button should be placed to be in the centre of the screen.
Button_Pathfinding_Object	Class Object	This is the object responsible for allowing the user to enter the Pathfinding_State from the Menu_State. It displays a png image on the screen and completes an action when clicked.
Button_Help_Object	Class Object	This is the object responsible for allowing the user to enter the Help_State from the Menu_State. It displays a png image on the screen and completes an action when clicked.

Method Details

Method	Arguments	Return
@staticmethod get_position	• Button_Width • Button_Height	Tuple of (x, y)
main_loop	• Self	Boolean Value

Method Purposes

Method	Purpose
@staticmethod get_position	This is a method that calculates what x and y position the buttons should be placed at to be displayed in the centre. This is done through a calculation which considers the width and height of the buttons, as well as the size of the software window. The x and y positions calculated are returned and assigned to become the class attributes within the Menu_State.
main_loop	This is the menu loop which will run in the main script to initialise the actual iteration that the user will be inside of whilst the user is on the Main Menu screen. It will be inside of this function that we place the two buttons for the user options. The implementation I plan to use is to have the main_loop return a Boolean value depending on whether the user chooses to enter the Pathfinding state or the Help state.

Help_State Class

Overview

The Help_State class is the object responsible for the state of the software during which the user is in the Help Menu. This state is where the user will have an image of the instructions to operate the pathfinding software displayed to them. It will also have a 'Back' button, which will destroy the Help_State class when clicked. Following this, the user will be taken back into the Menu_State object.

Attributes

Attribute	Data Type	Purpose
Button_Back_Object	Class Object	This is the object responsible for allowing the user to enter the Menu_State from the Help_State. It displays a png image on the screen and completes an action when clicked.

Method Details

Method	Arguments	Returns
help_loop	• Self	None

Method Purpose

Method	Purpose
help_loop	This is the help loop which is responsible for the iteration during which the user is in the Help Menu state of the software. Within this function the Back_Button_Object is displayed and causes the object to self-destruct upon mouse click. This will return the user back to the Menu_State. For the purpose of the self-deletion, an UnboundLocalError must be excepted to continue the program's operation.

Pathfinding_State Class

Overview

The Pathfinding_State class is the object responsible for the state of the software during which the user is in the Pathfinding state. This state is where the user will have all the pathfinding options and graph on the screen. This object encapsulates the graph data structure, as well as all the algorithm and data structure objects within the Pathfinding state.

Attributes

Attribute	Data Type	Purpose

Graph_Dimensions_Input	Integer	This integer is responsible for the dimensions of the graph that the Pathfinding will occur on. It is an attribute that is passed in as an argument when Pathfinding_State is initialised.
Button_Start_Object	Class Object	This is the object responsible for displaying the Start Node png image on the screen. It completes an action upon button click.
Button_End_Object	Class Object	This is the object responsible for displaying the End Node png image on the screen. It completes an action upon button click.
Button_Barrier_Object	Class Object	This is the object responsible for displaying the Barrier Node png image on the screen. It completes an action upon button click.
Button_Barrier_Clear_Object	Class Object	This is the object responsible for displaying the Barrier Clear png image on the screen. It completes an action upon button click.
Button_Clear_All_Object	Class Object	This is the object responsible for displaying the Clear All png image on the screen. It completes an action upon button click.
Button_Back_Object	Class Object	This is the object responsible for displaying the Back png image on the screen. It completes an action upon button click.
Button_Pathfinding	Class Object	This is the object responsible for displaying a png that is passed in as a parameter – in this case this is the Dijkstra Pathfinding image. It completes an action upon button click as well as a boolean operation.

Button_Heuristic	Class Object	This is the object responsible for displaying a png that is passed in as a parameter – in this case this is the Manhattan Distance image. It completes an action upon button click as well as a boolean operation.
Graph	Class Object – Complex Data Structure	This class is responsible for the representation of the complex Graph data structure which will encapsulate the entirety of the pathfinding algorithms' operations.

Method Details

Method	Arguments	Return
position_check	• Self • Pos	None
draw_start	• Self	None
draw_end	• Self	None
draw_barrier	• Self	None
pathfinding_loop	• Self	None

Method Purpose

Method	Purpose
position_check	This method is one that is used to patch a major user quality-of-life issue that arises from the type of iteration system I used to implement the node state changes. When one of the node button changes is pressed, it enters an inner loop which is within its own function. The user can leave the method by pressing off the graph on the screen – as if the user presses a graph on a node, they are still looking to change nodes to whatever state they desire. The issue that arose with this system was that if the user pressed on a different button – for example, from Start Node placing loop they press on End Node placing button – all that

	would happen would be the leaving of the original loop, the second loop would not be entered. position_check is used in conjunction with the draw methods to ensure that when the first loop is returned (it returns the last position of the cursor), the method checks whether it was on the same location as the buttons were placed. If so, that draw method for that button is then initiated, effectively removing the need for a double click.
draw_start	This method draws the Button_Start_Object onto the screen with a conditional that if the button is pressed, the graph pathfinding for the placing of the start node is initiated and will return the mouse click's last position when ending. Prior to ending, the function runs a try-exception TypeError on the position to ensure that the program continues running if the position clicked is outside of the graph area.
draw_end	This method draws the Button_End _Object onto the screen with a conditional that if the button is pressed, the graph pathfinding for the placing of the start node is initiated and will return the mouse click's last position when ending. Prior to ending, the function runs a try-exception TypeError on the position to ensure that the program continues running if the position clicked is outside of the graph area.
draw_barrier	This method draws the Button_Barrier _Object onto the screen with a conditional that if the button is pressed, the graph pathfinding for the placing of the start node is initiated and will return the mouse click's last position when ending. Prior to ending, the function runs a try-exception TypeError on the position to ensure that the program continues running if the position clicked is outside of the graph area.
pathfinding_loop	This is the main loop that the user is in during the Pathfinding_State. It is inside of this function that all the buttons are 'drawn' onto the screen and gain the

	conditional statements for what they should complete upon mouse click. Aside from the buttons, there is also a conditional statement checking if the 'Enter' key is pressed by the user at any point inside of the loop. If so, the graph's running_algorithm is initiated. There is an UnboundLocalError exception within the function as the Pathfinding_State will self-destruct upon the click of the 'Back' button.

Data Structure Objects Identification

Graph Class

Overview

The Graph object represents a graph data structure that encapsulates all the pathfinding and other data structure objects. It is responsible for drawing itself on the screen as well as the pathfinding algorithm operations that occur within. It also handles all the user inputs for the algorithmic operations.

Attributes

Attribute	Data Type	Purpose
dimensions	Integer	This attribute represents the size of the rows and columns within the graph. For example, if the value of this attribute is 50, the graph will be 50x50 sized.
size	Integer	This represents the pixel size of how much space the Graph will take place on the software display. It is initialised with the WINDOW_HEIGHT constant as I want the graph to be the size of the display's height.
node_size	Integer	This is the attribute which represents the pixel size of how large the size of each node should be when displayed on the window. It is assigned by the integer division calculation of the size attribute being divided by the graph dimensions.
graph_background	Pygame Object	This attribute will be the background of the Graph that is displayed on the software. It is a Pygame surface object that makes up a white background of the graph.

start	Class Object	This attribute is initialised as a NoneType, which is then changed to the Node object to represent which node is currently considered the start node on the graph.
end	Class Object	This attribute is initialised as a NoneType, which is then changed to the Node object to represent which node is currently considered the end node on the graph.
Dijkstra	Class Object	The Dijkstra attribute represents the Dijkstra_Pathfinding algorithm object implementation which is within the Graph data structure. This will be used to perform the pathfinding methods on the graph's infrastructure.
Astar	Class Object	The Astar attribute represents the Astar_Pathfinding algorithm object implementation which is within the Graph data structure. This will be used to perform the pathfinding methods on the graph's infrastructure.
grid	Two-Dimensional Array	This is a two-dimensional array which consists of all the nodes that are used to represent the graph in a matrix representation of the graph. It is initialised with the make_grid method.

Method Details

Method	Arguments	Return
@property path_length	• Self	self.Dijkstra.path_length + self.Astar.path_length
@property nodes_accessed	• Self	self.Dijkstra.nodes_accessed + self.Astar.nodes_accessed

@property elapsed_time	• Self	self.Dijkstra.elapsed_time + self.Astar.elapsed_time
get_path_length	• Self	self.path_length
get_nodes_accessed	• Self	self.nodes_accessed
get_elapsed_time	• Self	self.elapsed_time
make_grid	• Self	2-Dimensional Array
draw_grid	• Self	None
draw	• Self	None
get_clicked_pos	• Self • Pos	• row • col
running_algorithm	• current_algorithm • current_heuristic	None
clear_graph	• Self	None
clear_barriers	• Self	None
pathfinding	• mode • current_algorithm • current_heuristic	Last Mouse Position

Method Purpose

Method	Purpose
@property path_length	This is a method that contains that @property class decorator. This is so that the Graph's path_length attribute automatically updates when the components of its return statement change. In this case, when one of the algorithms' path_length attribute changes, it calculates the sum of the two algorithms' path_length(s) to find what last path_length attribute is. This works because when one algorithm's path_length is set, the other's is reset.
@property nodes_accessed	This is a method that contains that @property class decorator. This is so that the Graph's nodes_accessed attribute automatically updates when the components of its return statement change. In this case, when one of the algorithms' nodes_accessed attribute changes, it calculates the sum of the two algorithms' nodes_accessed to find what last nodes_accessed attribute is. This works because when one algorithm's

	nodes_accessed is set, the other's is reset.
@property elapsed_time	This is a method that contains that @property class decorator. This is so that the Graph's elapsed_time attribute automatically updates when the components of its return statement change. In this case, when one of the algorithms' elapsed_time attribute changes, it calculates the sum of the two algorithms' elapsed_time(s) to find what last elapsed_time attribute is. This works because when one algorithm's elapsed_time is set, the other's is reset.
get_path_length	This is a getter_method which is used to return the graph's current path_length property attribute.
get_nodes_accessed	This is a getter_method which is used to return the graph's current nodes_accessed property attribute.
get_elapsed_time	This is a getter_method which is used to return the graph's current elapsed_time property attribute.
make_grid	This is the function which is responsible for the creation of the grid of the graph. Within this function, we calculate the node_size, which will be the size of each node when it is displayed on the screen. The return function then has a list comprehension statement which creates a self.dimensions number of arrays inside of another array. These arrays have self.dimensions number of Node objects within them. Therefore, you get a Matrix representation of a dimensions-by-dimensions graph. When the Node objects are dynamically instantiated, the nodes' row and column numbers are assigned – depending on the index of the iteration that they are within – in the list comprehension.
draw_grid	This is a method which draws onto the screen all the lines which are responsible to display to the user the splitting of the grid. It passes in the pixel size of each of the nodes, which is done through the Pygame.draw.line method whilst

	specifying the colour and location of them. All of this is contained within a nested loop structure.
draw	When the draw method is called, it fills the graph_background surface object with a white colour. It then uses a nested list to draw each node onto the screen. Additionally, it runs the draw_grid method. All in all, this method is used to display the graph to the user.
get_clicked_pos	The get_clicked_pos method takes in the current position of the mouse and calculates which node that mouse position is on.
running_algorithm	The running_algorithm function is responsible for the actual operation of the pathfinding algorithms. It first updates all the neighbours of each node inside of the graph, followed by checking which current_algorithm is being used – through a Boolean check. Depending on which algorithm is used, the other algorithm's path statistics are reset. It also runs a validation check to see whether the algorithm returns false or not – if so, a 'No Path Found' error is returned to the user, otherwise the path's statistics are displayed to the user in a new window. If the pathfinding type is A*, the current_heuristic is passed in so that it reflects the user's chosen heuristic.
clear_graph	This method starts off with an error validation check. This is done by initialising an Error_Count variable at 0. A nested loop is then run, which checks if each node in the graph is in the None state. If a node is in fact in the None state Error_Count is increased. The Error_Count is then checked against the number of nodes on the graph. This is done to see if every single node on the graph is empty. If so, the "Graph is already clear" error is displayed to the user. If the graph is not already clear, a nested loop is run, which results in each node being reset to its None state. Prior

	to its ending, the start and end attributes are reset to None, and the draw method is also initiated, displaying the newly cleared graph to the user.
clear_barriers	This function first runs an error validation check. The check starts by setting a local Error variable to True. It then runs a nested loop that searches through the graph to see to see if a single node is in a barrier state, if so, it will set the Error variable to False. If the variable is set to True, an error message explaining that no barriers on the graph exist will appear. Otherwise, another nested loop operation will occur, during which every node gets reset to its None state.
pathfinding	This is a method which contains the 'inner loop' of the Pathfinding state. At its start, it sets a local Error variable to False. A nested loop check is then initiated which checks if a single node on the graph is in its Path state. If so, the Error variable is set to True. A conditional statement then runs checking if Error is true. If so, an error appears which asks the user to reset the graph prior to pressing any more buttons. Otherwise, it records the current position of the mouse and checks if the position is within the area that the graph covers. If so, it will identify the node at that location and then depending on the 'mode' that has been passed in as an argument (for example, 'Start'), it will change that node to the state that the mode represents. If the position of the mouse, however, is not within the graph, the algorithm will return the mouse's current position. The algorithm method will also contain the conditional statement checking if the 'Enter' key has been pressed – then running the insufficient_nodes_error check and if that passes, initiating the running_algorithm method.

Binary_Heap Class

Overview

This class defines a Binary_Heap data structure, which is a type of heap data structure where the parent nodes are always greater (or smaller) than their child nodes. The Binary_Heap class contains two private methods, _sift_up() and _sift_down(), which are used to maintain the heap property while adding and removing elements. Overall, the Binary_Heap class provides an efficient way to maintain a collection of elements where the elements with the highest (or lowest) priority can be quickly accessed and removed.

Method Details

Method	Arguments	Return
sift_up	• Self • Index	None
sift_down	• Self • Index	None

Method Purpose

Method	Purpose
sift_up	The _sift_up method takes an index as input and moves the element at that index up the heap until the heap property is satisfied. Specifically, it compares the element with its parent and swaps them, if necessary, until the element is in a position where it is greater (or smaller) than its parent.
sift_down	The _sift_down method takes an index as input and moves the element at that index down the heap until the heap property is satisfied. Specifically, it compares the element with its children and swaps them, if necessary, until the element is in a position where it is smaller (or greater) than its children.

Priority_Queue Class

Overview

This class PriorityQueue is a subclass of the Binary_Heap class and implements a priority queue data structure using a binary heap. The priority queue is a data structure where each element has a priority assigned to it and elements with higher priority are dequeued before those with lower priority.

Attributes

Attribute	Data Type	Purpose
_queue	Array	The purpose of the private queue attribute is to store all the elements within the Priority Queue. This is implemented by an array structure within Python.

Method Details

Method	Arguments	Return
push	• Self • Item • Priority	None
update_priority	• Self • Item • New_Priority	None
pop	• Self	Item
is_empty	• Self	Boolean Value

Method Purpose

Method	Purpose
push	The push method adds an item with a given priority to the queue by creating a tuple with the priority and item and appending it to the queue. The _sift_up method is then called to maintain the heap property.
update_priority	The update_priority method updates the priority of an item in the queue by searching for the item in the queue and updating its priority. If the new priority is higher, the _sift_up method is called to maintain the heap property, and if the new priority is lower, the _sift_down method is called.

pop	The pop method removes the item with the highest priority from the queue and returns it. The _sift_down method is called to maintain the heap property after removing the element.
is_empty	The is_empty method checks if the queue is empty or not by seeing whether the length of the queue is equal to zero.

Hash_Table Class

Overview

The Hash_Table class is a Python implementation of a hash table data structure. This object provides a way to store and retrieve data using a hash table data structure, which may provide fast access to data, especially using a minimal perfect hash algorithm. This will minimise the size of the table and provide no collisions.

Attributes

Attribute	Data Type	Purpose
size	Integer	This attribute is responsible for determining the size of the hash table. The size of the hash table is determined using its next_prime method. This considers the dimensions argument when Hash_Table is first instantiated. It calculates the next prime number after value of $dimensions^2$ as making it this value is good hashing practice.
table	2-Dimensional Arrays	The table attribute represents the hash table's table. It is assigned upon instantiation using a list comprehension statement. This utilises an iterative statement creating an array inside of an array self.size number of times.

Method Details

Method	Arguments	Return
@staticmethod next_prime	• Number	next_prime_number
hash	• Self • Item	hash_value
insert	• Self • Item	None
remove	• Self • Item	None

Method Purpose

Method	Purpose
@staticmethod next_prime	The next_prime method finds the next prime number after a given number by generating a list of numbers to check, and then iterating through each number in the list and checking if it is prime. If it is prime, it is added to a list of prime numbers, and the minimum prime number is returned.
hash	The hash method takes a tuple of the Node's (x,y) coordinates as input and returns the index in the hash table where the tuple should be inserted. It does this by computing a hash value using a hashing function and then taking the modulus of that hash value with the size of the hash table. This is an implementation of the minimal perfect hashing algorithm.
insert	The insert method takes a tuple as input and inserts it into the hash table by computing the hash value of the tuple and appending it to the list at that index in the hash table.
remove	The remove method takes a tuple as input and removes it from the hash table by computing the hash value of the tuple, checking if the tuple exists in the list at that index in the hash table, and then removing it if it exists.

Pathfinding Objects Identification

State Class

Overview

The State class represents the state of a node in a graph. It has an attribute state that stores the current state of the node. The possible states are "Open", "Closed", "Barrier", "Start", "End", and "Path". The class provides methods to get and set the state and colour of the node, and to check if the node is in a certain state, such as being a barrier or being part of the path.

Attributes

Attribute	Data Type	Purpose
state	String	This attribute is responsible for storing the current state of the node. The states are determined by a dictionary which stores colours corresponding to each state. It is initialised as a NoneType.

Methods

Method	Arguments	Return
@property colour	• Self	states_dictionary.get(self.state)
get_colour	• Self	Self.colour
get_state	• Self	Self.state
set_colour	• Self	None
set_state	• Self	None
is_closed	• Self	Boolean Value
is_open	• Self	Boolean Value
is_barrier	• Self	Boolean Value
is_start	• Self	Boolean Value
is_end	• Self	Boolean Value
is_path	• Self	Boolean Value

Method	Purpose
@property colour	The colour method is a property method that returns the colour corresponding to the current state of the node provided as

	an argument. This is based off the dictionary which stores colours corresponding to each state.
get_colour	The get_colour method returns the same result as the colour method.
get_state	The get_state method is a getter method which returns the current state of the node.
set_colour	. The set_colour method is a setter method which sets the colour of the node based on the current state.
set_state	The set_state method sets the state of the node to a new state.
is_closed	The is_closed method checks if the node is in the closed state. This method returns True if the node is in the specified state, otherwise returning False.
is_open	The is_open method checks if the node is in the open state. This method returns True if the node is in the specified state, otherwise returning False.
is_barrier	The is_barrier method checks if the node is in the barrier state. This method returns True if the node is in the specified state, otherwise returning False.
is_start	The is_start method checks if the node is in the start state. This method returns True if the node is in the specified state, otherwise returning False.
is_end	The is_end method checks if the node is in the end state. This method returns True if the node is in the specified state, otherwise returning False.
is_path	The is_path method checks if the node is in the path state. This method returns True if the node is in the specified state, otherwise returning False.

Node Class

Overview

The Node class represents a node in a grid, which is used in a pathfinding visualization application. It stores information about the node's row and column position, its coordinates on the window, its state, its neighbours, its width, and the total number of rows in the grid.

Attributes

Attribute	Data Type	Purpose
row	Integer	This attribute is used to store the row that the node is on.
col	Integer	This attribute is used to store the column that the node is on.
x	Integer	This attribute is used to store the x coordinate that the node is located on.
y	Integer	This attribute is used to store the y coordinate that the node is location on.
state	Class Object	This attribute is used to store the node's current state.
neighbours	Array	This attribute stores an array of the node's neighbouring nodes.
width	Integer	This attribute stores the width of the node.
total_rows	Integer	This attribute is used to store the total rows of the graph.

Method Details

Method	Arguments	Return
draw	•	
get_pos	•	
reset	•	
make_start	•	
make_closed	•	
make_open	•	
make_barrier	•	
make_end	•	
make_path	•	
update_neighbours	•	
__lt__	•	

Method Purpose

Method	Purpose
draw	The draw function is responsible for drawing the node onto the screen display. This is done using the Pygame.draw.rect method.
get_pos	The get_pos method returns the row and column position of the node.
reset	The reset method sets the state of the node to None, which essentially resets the node's state.
make_start	The make_start method sets the state of the node to "Start", indicating that this node is the start node for a pathfinding algorithm.
make_closed	The make_closed method sets the state of the node to "Closed", indicating that the node has been visited by the algorithm but has not been fully explored.
make_open	The make_open method sets the state of the node to "Open", indicating that the node is open for exploration.
make_barrier	The make_barrier method sets the state of the node to "Barrier", indicating that the node cannot be passed through.
make_end	The make_end method sets the state of the node to "End", indicating that this node is the end node for a pathfinding algorithm.
make_path	The make_path method sets the state of the node to "Path", indicating that the node is part of the path that was found by the pathfinding algorithm.
update_neighbours	The update_neighbours method updates the neighbours of the node by checking if its surrounding nodes are barriers or not. If the neighbouring node is not a barrier, then it is added to the list of neighbours of the current node. This method is used for finding the shortest path between two nodes.

__lt__	The __lt__ method is used for sorting objects and returns false in this case.

Dijkstra_Pathfinding Class

Overview

The Dijkstra_Pathfinding class implements Dijkstra's pathfinding algorithm to find the shortest path between two points on a grid. The class uses a priority queue to store nodes, dictionaries to hold the cost of the path from the start node to a node and the sum of g_score and h_score (heuristic) for a node, and a hash table to keep track of which nodes are in the open set. The algorithm loops through the neighbours of the current node, calculates the tentative g_score for the neighbour node, and updates the g_score and f_score of the neighbour node if the tentative g_score is lower than the current g_score. The path is reconstructed from the end node to the start node using parent nodes, and the elapsed time, path length, and number of nodes accessed are recorded.

Attributes

Attribute	Data Type	Purpose
open_set	Class Object	This attribute holds the nodes to be explored in priority order, based on their f_score. The priority queue data structure allows for efficient access to the node with the lowest f_score.
came_from	Dictionary	This attribute is a dictionary that keeps track of the parent node of each node visited in the algorithm. It is used for path reconstruction at the end of the algorithm.
g_score	Dictionary	This attribute is a dictionary that stores the cost of the shortest path found so far from the start node to a given node. The value of g_score for each node is initially set to infinity, except for the start node which is set to 0.

f_score	Dictionary	This attribute is a dictionary that stores the sum of g_score and the heuristic function h_score for each node. The heuristic function is used to estimate the cost of the shortest path from a node to the end node. The value of f_score for each node is initially set to infinity, except for the start node which is set to 0.
open_set_hash	Class Object	This attribute is a hash table that is used to keep track of which nodes are in the open set. This is used to check whether a node has already been explored or not, and to efficiently add or remove nodes from the open set.
path_length	Integer	This attribute is an integer that represents the length of the shortest path found by the algorithm. It is updated during path reconstruction.
nodes_accessed	Integer	This attribute is an integer that represents the number of nodes visited by the algorithm. It is used for performance analysis and optimization.
elapsed_time	Float	This attribute is a float that represents the time taken for the algorithm to run. It is used for performance analysis and optimization.

Method Details

Method	Arguments	Return
reconstruct_path	• current_node • draw	None
algorithm	• draw • grid	Boolean Value

	• start_node • end_node	

Method Purpose

Method	Purpose
reconstruct_path	The reconstruct_path method of the Dijkstra_Pathfinding class reconstructs the path from the end node to the start node using parent nodes and draws the path. The method initializes the path length variable and loops through the parent nodes until the start node is reached. For each node, the make_path method is called to draw the updated path, and the path length is incremented.
algorithm	The algorithm method of the Dijkstra_Pathfinding class implements Dijkstra's pathfinding algorithm. The method initializes the g_score and f_score dictionaries, adds the start node to the open set with priority 0, and loops until the open set is empty. For each node, the method gets the node with the lowest f_score from the open set, removes the current node from the hash table, and checks if the current node is the end node. If the current node is the end node, the reconstruct_path method is called to reconstruct the path and return True. Otherwise, the method loops through the neighbours of the current node, calculates the tentative g_score for the neighbour node, and updates the g_score and f_score of the neighbour node if the tentative g_score is lower than the current g_score. If the neighbour node is not in the open set, the neighbour node is added to the open set with priority f_score and added to the hash table. The make_open method is called to visualize the neighbour node as an open node. The method updates the nodes accessed variable, records the

	elapsed time, and draws the updated path. Finally, if no path is found, False is returned.

AStar_Pathfinding Class

Overview

The Astar_Pathfinding class is a subclass of Dijkstra_Pathfinding and implements the A* pathfinding algorithm. The constructor of the class takes the number of rows in the grid as an argument and calls the constructor of the superclass. There are three methods in this class. It returns a Boolean value depending on whether or not the path is found.

Attributes

All but one of the attributes within the AStar_Pathfinding class are inherited from the Dijkstar_Pathfinding class. The rows attribute is passed in as an argument to the Dijkstra class, which requires it to also be passed in to the Astar_Pathfinding class. The attributes are inherited using the super() class method in Python.

Method Details

Method	Arguments	Return
@staticmethod manhattan_distance	• Start Node coordinate tuple • End Node coordinate tuple	Distance
@staticmethod euclidean_distance	• Start Node coordinate tuple • End Node coordinate tuple	Distance
algorithm	• Self • Draw • Grid • Start • End • Current_heuristic	Boolean Value

Method Purpose

Method	Purpose
@staticmethod manhattan_distance	The manhattan_distance method calculates the Manhattan distance between two points. The method takes two points as arguments and returns the

	absolute difference between the x coordinates plus the absolute difference between the y coordinates.
@staticmethod euclidean_distance	The euclidean_distance method calculates the Euclidean distance between two points. The method takes two points as arguments and returns the square root of the sum of the squares of the differences between the x and y coordinates.
algorithm	The algorithm method implements the A* algorithm. The method takes five arguments: draw, grid, start, end, and current_heuristic. The draw argument is a function that draws the grid, the grid argument is the grid object that the algorithm will traverse, the start and end arguments are the start and end nodes respectively, and the current_heuristic argument is a Boolean value that determines which heuristic to use. The method starts by adding the start node to the open set and marking it as visited. It then sets the g score for each node to infinity and the g score for the start node to 0. It also sets the f score for each node to infinity and the f score for the start node based on the chosen heuristic. The method loops until the open set is empty. In each iteration, it removes the node with the lowest f score from the open set and checks if it is the end node. If it is, it reconstructs the path from the start node to the end node and returns True. Otherwise, it loops through the neighbours of the current node and calculates the tentative g score for each neighbour. If the tentative g score is lower than the current g score for the neighbour, it updates the g score, f score, and adds the neighbour to the open set. Finally, it returns False if a path was not found.

User Interface Objects Identification

UI_Button Class

Overview

The UI_Button class is a simple implementation of a graphical user interface (GUI) button in Python using the Pygame library. It allows the creation of a button object with given properties such as its position, image, and scale factor. The button can be drawn on a screen, and the draw method returns a Boolean indicating whether the button was clicked or not. The button's click-ability is determined by checking the position of the mouse, and if it is over the button and clicked, the method returns True, indicating that the button was clicked.

The constructor of the UI_Button class initializes the button object by taking five arguments, namely x and y (the position of the button), image (the image of the button), scale (the scale factor of the button), and screen (the screen to draw the button on). The dimensions of the image are obtained, and the image is scaled according to the given scale factor. The position of the button is set based on the dimensions of the scaled image, and the screen is set to draw the button on.

Attributes

Attribute	Data Type	Purpose
width	Integer	This is the width of the button based on the width of the chosen image to represent it visually.
height	Integer	This is the height of the button based on the height of the chosen image to represent it visually.
image	Pygame image object	This is a scaled version of the image you wanted the button to be to fit whatever parameters u set in the instantiation of the button.
rect	Pygame Rectangle object	This stored the image as a rectangle object to assist in collision detection later.
rect.topleft	Tuple in format (Integer, Integer)	This stores the top left (x,y) position of the Pygame rectangle object.
clicked	Boolean	This stores whether the button has been clicked.

Method Details

Method	Arguments	Return
draw	• Self	Boolean Value

Method Purpose

Method	Purpose
draw	The draw method of the UI_Button class is responsible for drawing the button on the screen and determining whether it was clicked or not. It first initializes the action variable as False. It then gets the position of the mouse using the Pygame mouse module. The method then checks if the mouse is over the button and if it is clicked. If the mouse is over the button and clicked, the action variable is set to True. The button is then drawn on the screen using the blit method of the Pygame Surface class. Finally, the method returns the action variable indicating whether the button was clicked or not.

Boolean_Button Class

Overview

The BooleanButton class is a subclass of the UI_Button class that allows for toggling between two images by changing the Boolean value of the button. It has an additional instance variable alternate_image, which is the image to be displayed when the boolean_condition is False. The BooleanButton class has two methods: update and get_boolean_condition, it additionally inherits the attributes and method from the UI_Button class.

Attributes

Attribute	Data Type	Purpose
Boolean_Condition	Boolean	This is responsible for the state of the button that it is in. It is initially set to True, but this will change

		dependent on the update method.
Count	Integer	The count attribute is responsible to track how to swap between states. It is used in an even/odd check that will change the state of the button dependent on the count attribute.
Alternage_image	Composite Image Object	This is the second image that the button will display when the Boolean_Condition is equal to False.
Temporary_image	Composite Image Object	This is the image that the button will display when it is in the primary state/when Boolean_Condition is equal to True.

Method Details

Method	Arguments	Return
update	• Self	Self.Boolean_Condition
get_boolean_condition	• Self	Self.Boolean_Condition

Method Purpose

Method	Purpose
update	The update method updates the image of the button and the boolean_condition based on the count of updates. The method changes the image of the button every other time it is called. If the count is even, it sets the image to the default temporary_image and sets the boolean_condition to True. Otherwise, it sets the image to alternate_image and sets the boolean_condition to False. The update method then calls the draw method of the UI_Button class to draw the button on the screen and returns the Boolean value of the button.
get_boolean_condition	The get_boolean_condition method simply returns the Boolean value of the

	button, which is True if boolean_condition is True, and False otherwise. This method can be used to check the state of the button without updating the image.

Error_Message Class

Overview

The ErrorMessage class is designed to handle different types of error messages in the algorithm. The class has an __init__ method that initializes an instance of the class with a given error type. It also has a show_error_message method that displays the appropriate error message based on the error type specified. Overall, the ErrorMessage class is a useful tool for providing users with informative error messages in the algorithm. By displaying relevant error messages, it can help users to understand what has gone wrong and take appropriate action to resolve the issue.

Attributes

Attribute	Data Type	Purpose
error_type	String	This is the parameter that decides what type of error the Error_Message class is going to be representing using the show_error_message method.

Method Details

Method	Arguments	Return
show_error_message	• error_type	None

Method Purpose

Method	Purpose
show_error_message	The show_error_message method has several if statements to determine which type of error message to display. For instance, if the error is related to clearing the graph, it checks whether the

	error type is "clear_graph_error" or "clear_barriers_error" and displays the appropriate error message accordingly. If there are insufficient nodes to run the algorithm, it displays a message asking the user to place both start and end nodes. If there is an attempt to place more nodes without clearing the graph, it shows a message asking the user to clear the graph first. Finally, if there is no possible path between the start and end nodes, it displays a message informing the user that there is no possible path between the start and end nodes.

Graph_Dimensions_Input Class

Overview

The Graph_Dimensions_Input class is a class in a Python script that creates a Tkinter window with buttons to prompt the user to select a graph size. Upon initialization, an instance of the class creates a window with four buttons for different graph sizes: 10x10, 25x25, 50x50, and 100x100. When the user selects one of the buttons, the window is closed, and the selected graph size is returned.

Attributes

Attribute	Data Type	Purpose
root	Tkinter Class Object	The root attribute refers to the main window of the Tkinter application. In the Graph_Dimensions_Input class, it is initialized with a tk.Tk() instance and configured with a size of 300x200 pixels, a title of "Menu Window" and a background color of "#325971".
label	Tkinter Class Object	The label attribute is a tk.Label instance that displays the text "Choose Graph Dimensions:" on the root window. It is configured with a

		foreground color of "#C8B782", a background color of "#325971", and a font of "Helvetica" with a size of 16.
row1	Tkinter Frame Instance	The row1 attribute is a tk.Frame instance that serves as a container for the Button_1, Button_2, Button_3, and Button_4 widgets. They are configured with a background color of "#325971" and are packed vertically on the root window.
row2	Tkinter Frame Instance	The row2 attribute is a tk.Frame instance that serves as a container for the Button_1, Button_2, Button_3, and Button_4 widgets. They are configured with a background color of "#325971" and are packed vertically on the root window.
Button_1	Tkinter Class Object	This is a tk.Button instance that represents the graph dimension that the user can choose from. It is configured with a 10x10 size and a common foreground color of "#325971" and background color of "#C8B782". When clicked, they call the close_window_and_return method with a value corresponding to their respective dimensions.
Button_2	Tkinter Class Object	This is a tk.Button instance that represents the graph dimension that the user can choose from. It is configured with a 25x25 size and a common foreground color

		of "#325971" and background color of "#C8B782". When clicked, they call the close_window_and_return method with a value corresponding to their respective dimensions.
Button_3	Tkinter Class Object	This is a tk.Button instance that represents the graph dimension that the user can choose from. It is configured with a 50x50 size and a common foreground color of "#325971" and background color of "#C8B782". When clicked, they call the close_window_and_return method with a value corresponding to their respective dimensions.
Button_4	Tkinter Class Object	This is a tk.Button instance that represents the graph dimension that the user can choose from. It is configured with a 100x100 size and a common foreground color of "#325971" and background color of "#C8B782". When clicked, they call the close_window_and_return method with a value corresponding to their respective dimensions.
return_value	Integer	The return_value attribute is initially set to None and is updated by the close_window_and_return method to store the user's selected graph dimension. It can be accessed using the get_return_value method.

Method Details

Method	Arguments	Return
close_window_and_return	• Self • return_value	None
get_return_value	• Self	self.return_value

Method Purpose

Method	Purpose
close_window_and_return	The close_window_and_return method takes a single parameter return_value and sets the instance variable return_value to the value of the parameter. Then, it closes the Tkinter window using the destroy method of the Tkinter root window.
get_return_value	The get_return_value method returns the value selected by the user when the close_window_and_return method was called. The method returns None if the user did not select any graph size before closing the window.

Output_Last_Path_Statistics Class

Overview

The InfoWindow class is an object that creates a graphical user interface (GUI) window with information about the last path taken by a pathfinding algorithm. The window contains three labels, each with a corresponding value label. The labels provide information about the elapsed time, total number of affected nodes, and shortest path length of the last path.

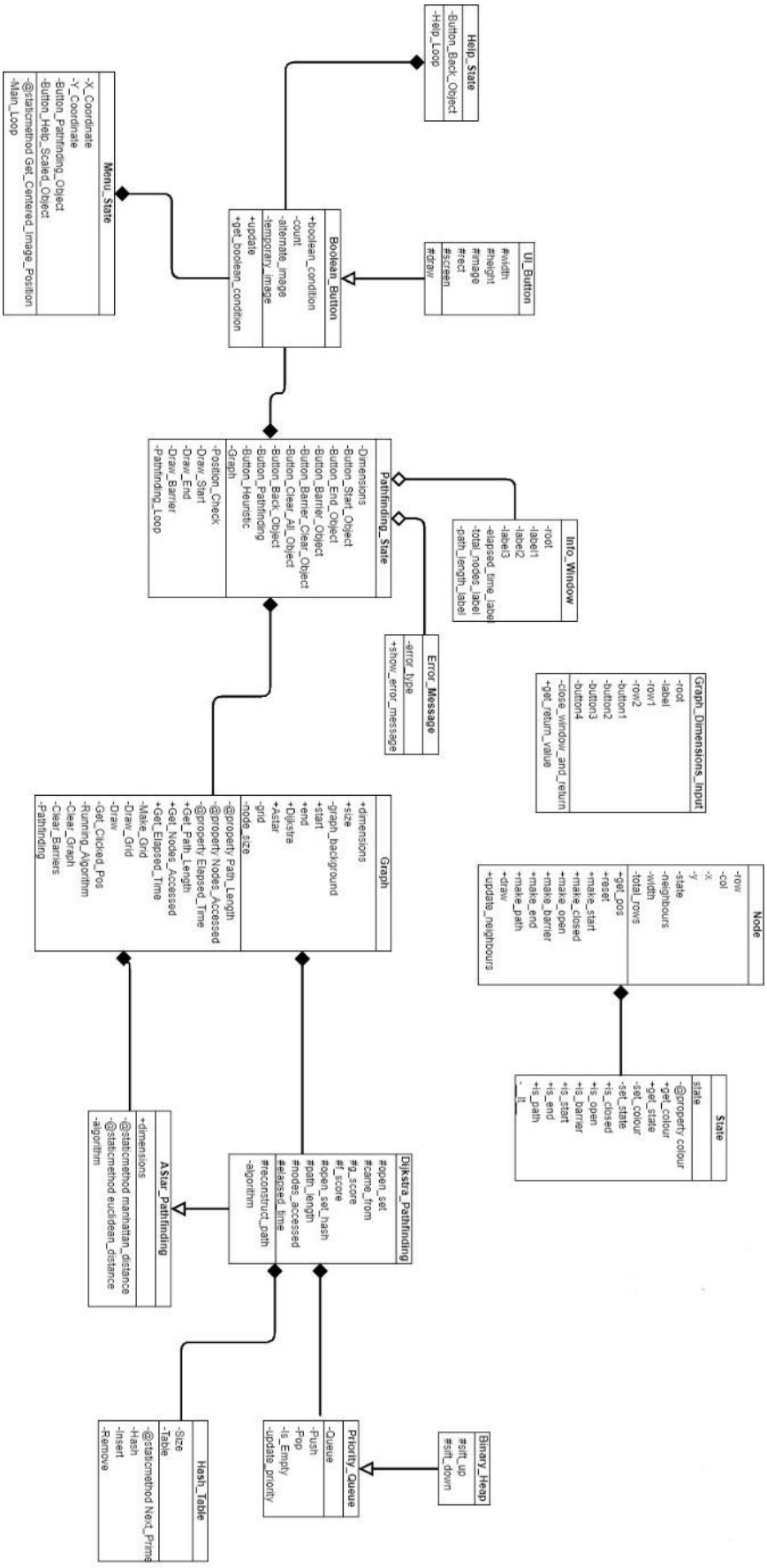
Attributes

Attribute	Data Type	Purpose
root		The root attribute in the Output_Last_Path_Statistics class is an instance of the Tkinter Tk class, which represents the main window of the user interface. This attribute is created and configured in the __init__ method of the class. It is used to display all the graphical elements of

		the window, such as labels and buttons.
label_1		The label_1 attribute is an instance of the Tkinter Label class that displays the text "Pathfinding Elapsed Time:". This label is created and configured in the __init__ method of the class, and its formatting includes colour, font, and alignment. The pack method is called to add some padding and positioning to the label before it is displayed in the window.
label_2		The label_2 attribute is another instance of the Tkinter Label class that displays the text "Total Nodes Affected:". Like label_1, this label is created and configured in the __init__ method of the class, and its formatting includes colour, font, and alignment. The pack method is also called to add padding and positioning before it is displayed.
label_3		The label_3 attribute is yet another instance of the Tkinter Label class that displays the text "Shortest Path Length:". This label is created and configured in the __init__ method of the class, and its formatting includes colour, font, and alignment. The pack method is called again to add some padding and positioning before it is displayed in the window.
elapsed_time_label		The elapsed_time_label attribute is an instance of the Tkinter Label class that

		displays the elapsed time of the pathfinding algorithm. This label is created and configured in the __init__ method of the class, and its text is passed in as an argument to the class. The pack method is called to add padding and positioning to the label before it is displayed in the window.
nodes_accessed_label		The nodes_accessed_label attribute is another instance of the Tkinter Label class that displays the total number of nodes accessed by the pathfinding algorithm. Like elapsed_time_label, this label is created and configured in the __init__ method of the class, and its text is passed in as an argument to the class. The pack method is called to add padding and positioning to the label before it is displayed.
path_length_label		The path_length_label attribute is yet another instance of the Tkinter Label class that displays the shortest path length found by the pathfinding algorithm. This label is created and configured in the __init__ method of the class, and its text is passed in as an argument to the class. The pack method is called to add padding and positioning to the label before it is displayed.

Unified Modelling Language Diagram



User-Interface Design

Interface Elements Analysis

Button Object Class: This object contains a visual element label, which will have the Contents text at the forefront.

Composite (xbm): This refers to the typing of the background image – it will be imported as a bitmap image.

Main Menu Interface Elements

Element	Data/Object Type	Contents
Button_1	Button Object Class	'Help'
Button_2	Button Object Class	'Pathfinding'
Background_1	Composite (xbm)	Background Bitmap Image with Title: 'Pathfinding Visualiser'

Pathfinding State Interface Elements

Element	Data/Object Type	Contents
Graph/Grid	Array/Complex Data Structure	Array consisting of objects
Button_3	Button Object Class	'Start Node'
Button_4	Button Object Class	'End Node'
Button_5	Button Object Class	'Barrier'
Button_6	Button Object Class	'Clear Barrier'
Button_7	Button Object Class	'Clear All'
Button_8	Button Object Class	Current Pathfinding Type
Button_9	Button Object Class	'Back'
Label_1	Integer	Shortest distance found
Label_2	Integer	Total nodes covered
Label_3	Real	Time taken for algorithm to run
Background_2	Composite (xbm)	Solid Background Bitmap Image with Title: 'Pathfinding'

Button_8 displays the type of pathfinding that is currently being employed. Upon click, this shifts to the other pathfinding type.

Help Menu Interface Elements

Element	Data/Object Type	Contents
Annotated Diagram	Composite (xbm)	'Image of Pathfinding state with annotations on how to operate the software'
Button_10	Button Object Class	'Back'

Graph Dimensions Input Window Interface Elements

Element	Data/Object Type	Contents
Input_1	String	'10x10'
Input_2	String	'25x25'
Input_3	String	'50x50'
Input_4	String	'100x100'
Title_Input	String	'Choose Graph Dimensions'

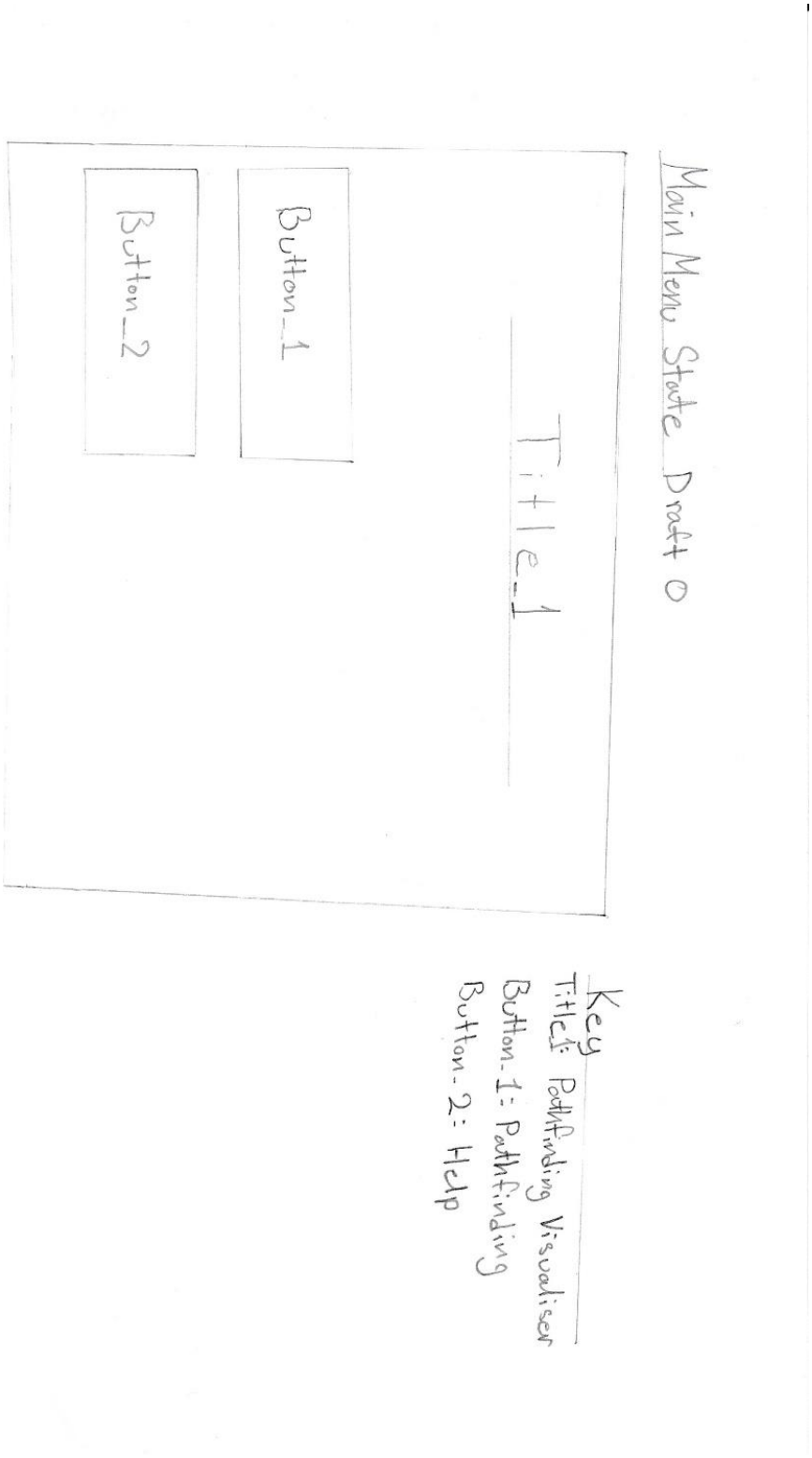
Pathfinding Statistics Window Interface Elements

Element	Data/Object Type	Contents
Label_1	String	'Path Length: '
Label_2	String	'Nodes Accessed: '
Label_3	String	'Elapsed Time: '
Title_Stats	String	'Pathfinding Statistics'

Initial Designs

Main Menu Designs

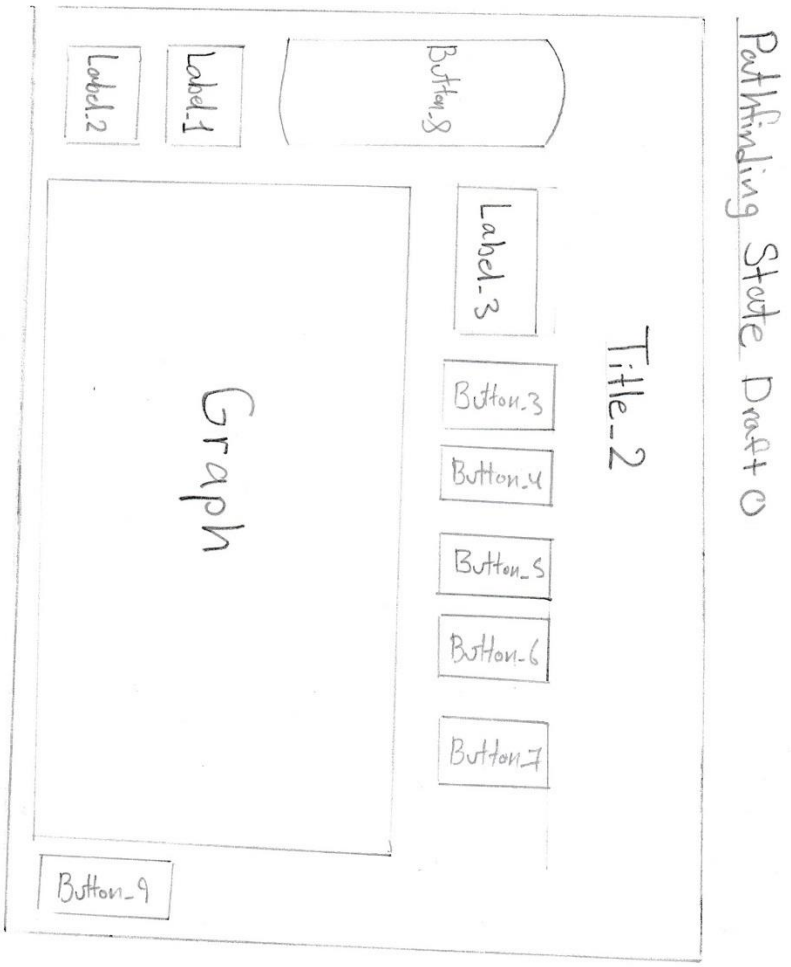
Design 0:



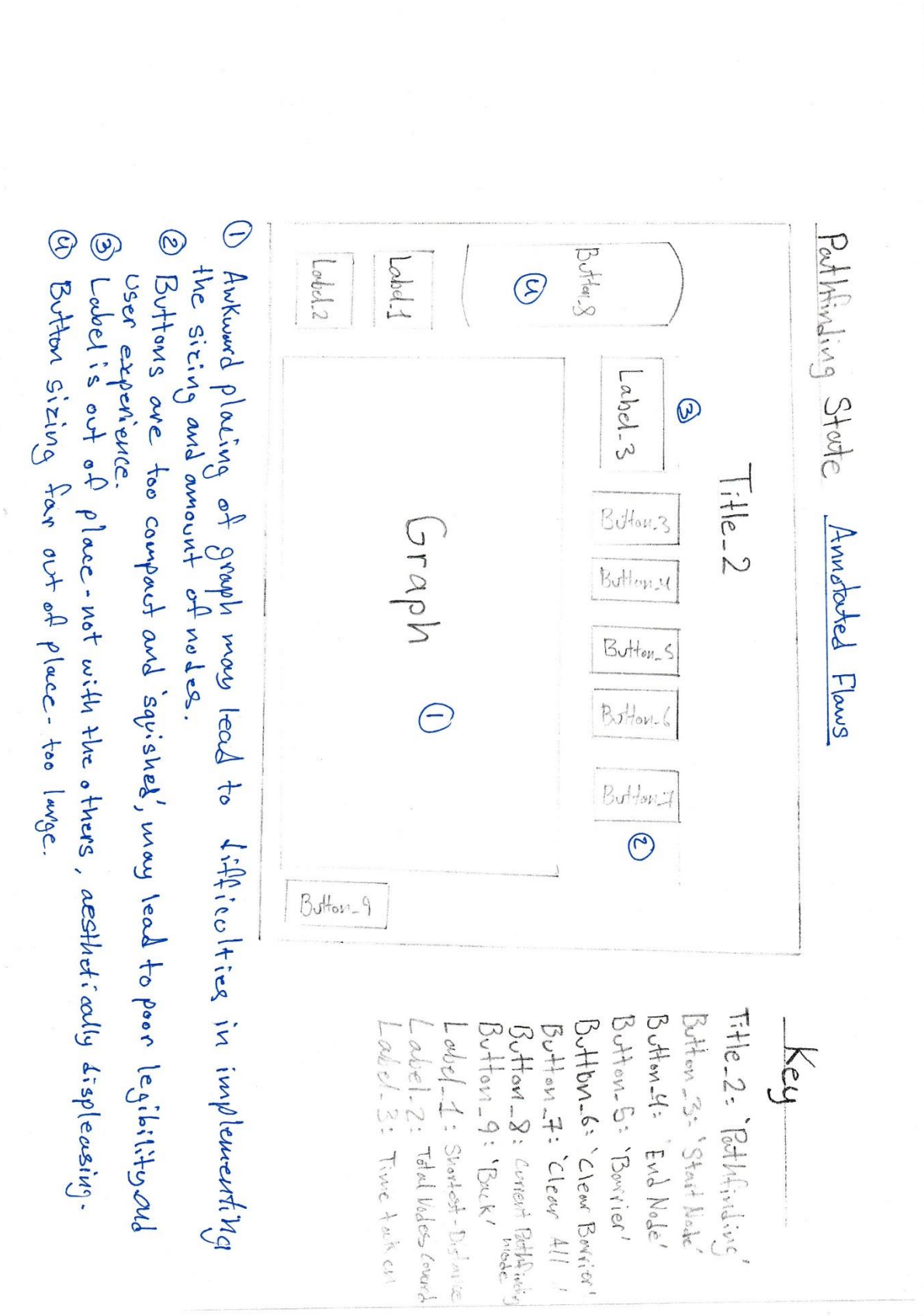
Annotated Design 0:

Pathfinding State Designs

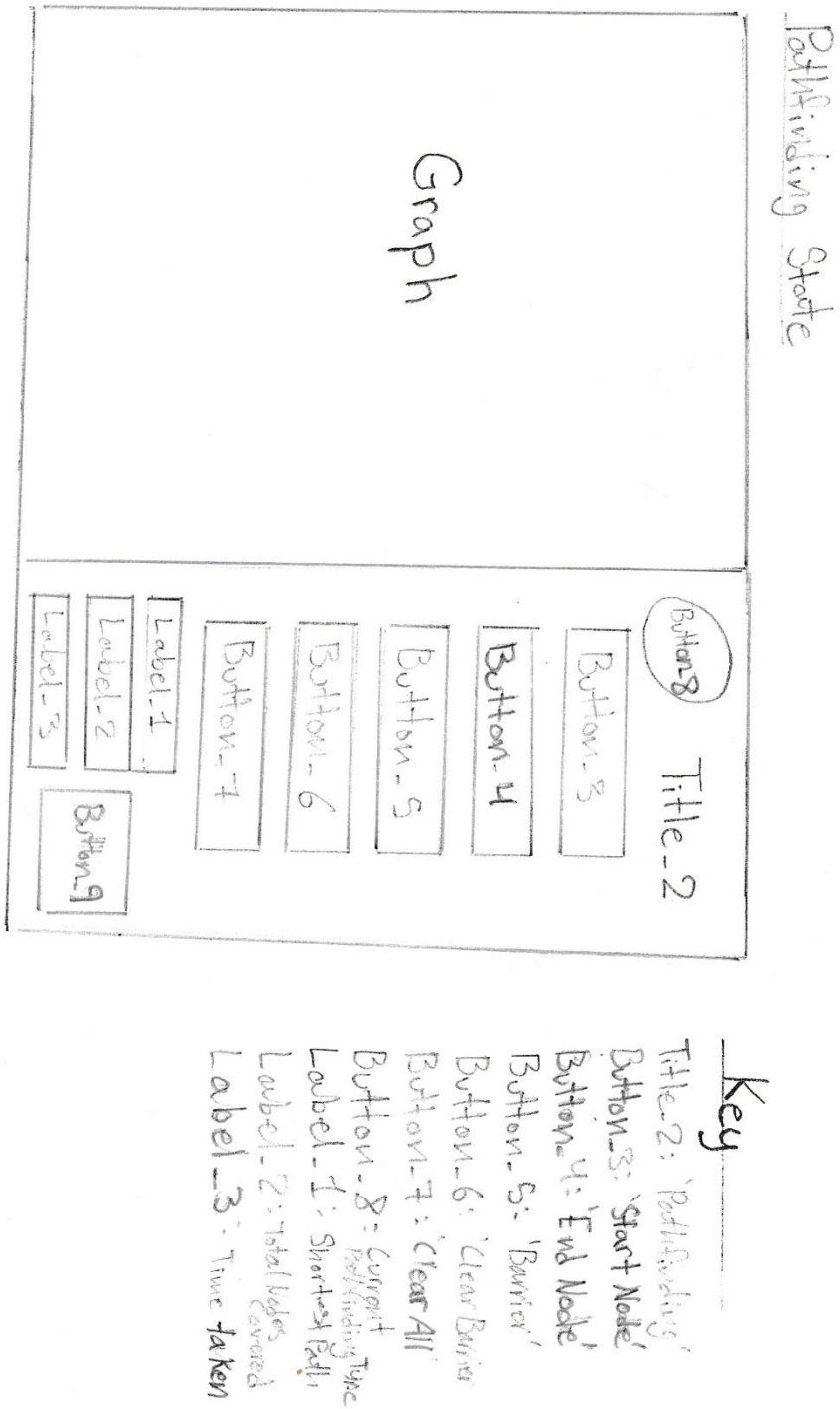
Design 0:



Annotated Design 0:

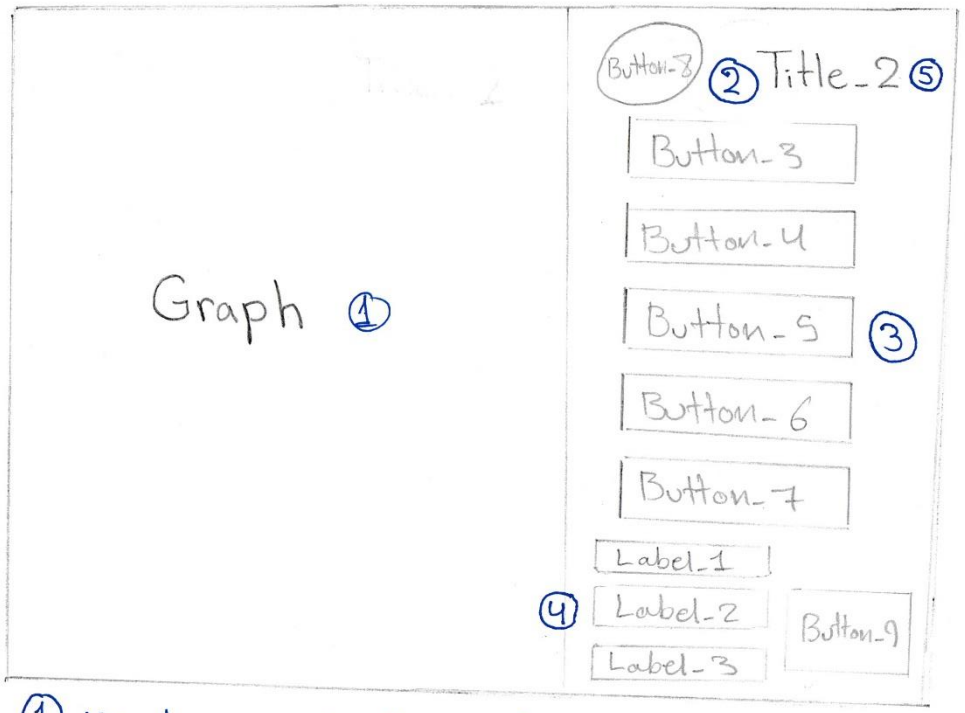


Design 1:



Annotated Design 1:

Pathfinding State Final Design



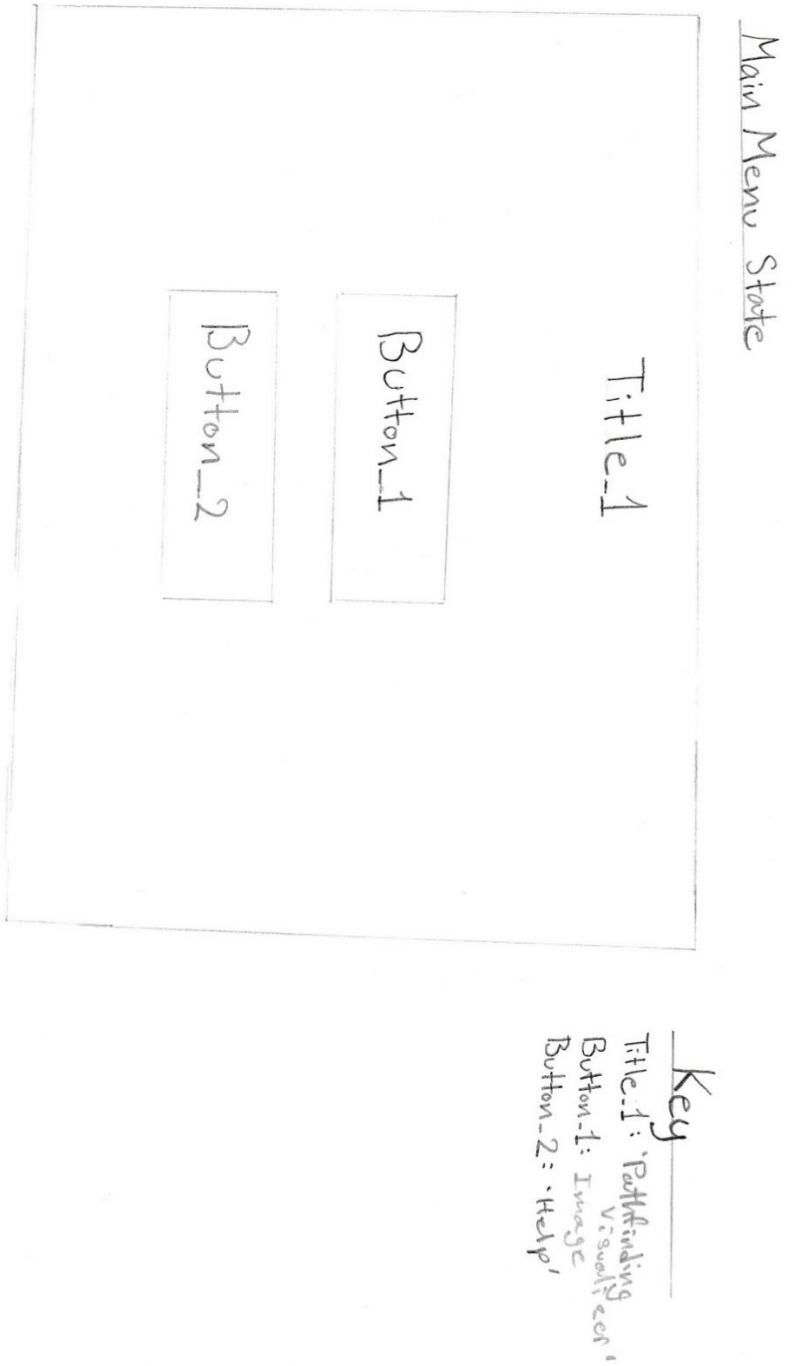
Key

Title_2: 'Pathfinding'
Button_3: 'Start Node'
Button_4: 'End Node'
Button_5: 'Barrier'
Button_6: 'Clear Barrier'
Button_7: 'Clear All'
Button_8: Current Pathfinding Type
Label_1: Shortest Path
Label_2: Total Nodes Covered
Label_3: Time taken

(1) Very large graph allowing for greater legibility of the algorithm functioning, potentially offering flexibility to the user in how large they would want the grid to be, and offers a capacity for more nodes.
(2) Current Pathfinding type switch placed next to 'Pathfinding' Title for greater clarity. Button is circular so as to be distinguished.
(3) All buttons arrayed together with sufficient sizing for legibility.
(4) Labels not in the way of user experience but sufficiently sized to provide info.
(5) Title not an individual element in implementation, part of background.

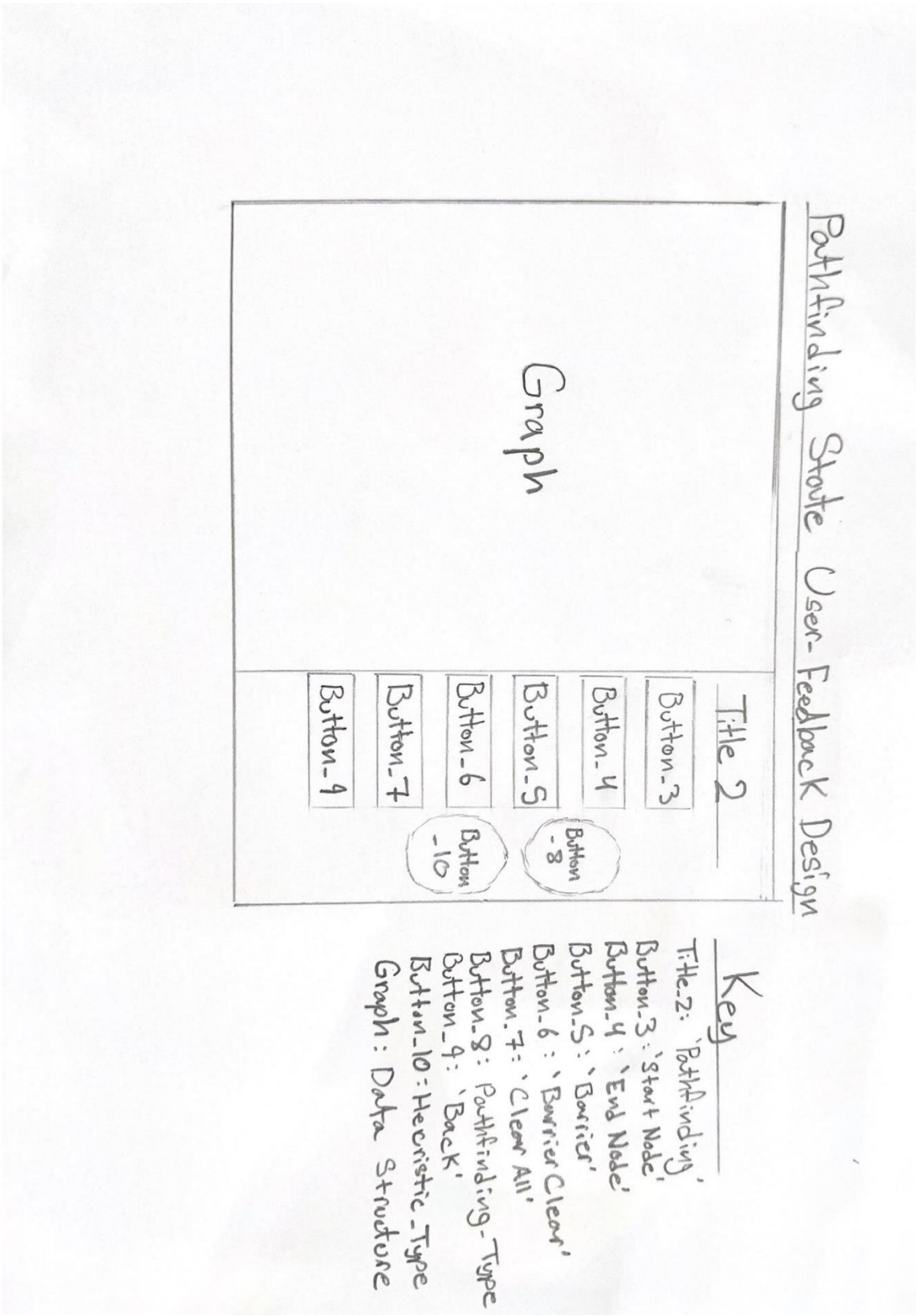
Final Designs

Main Menu Design

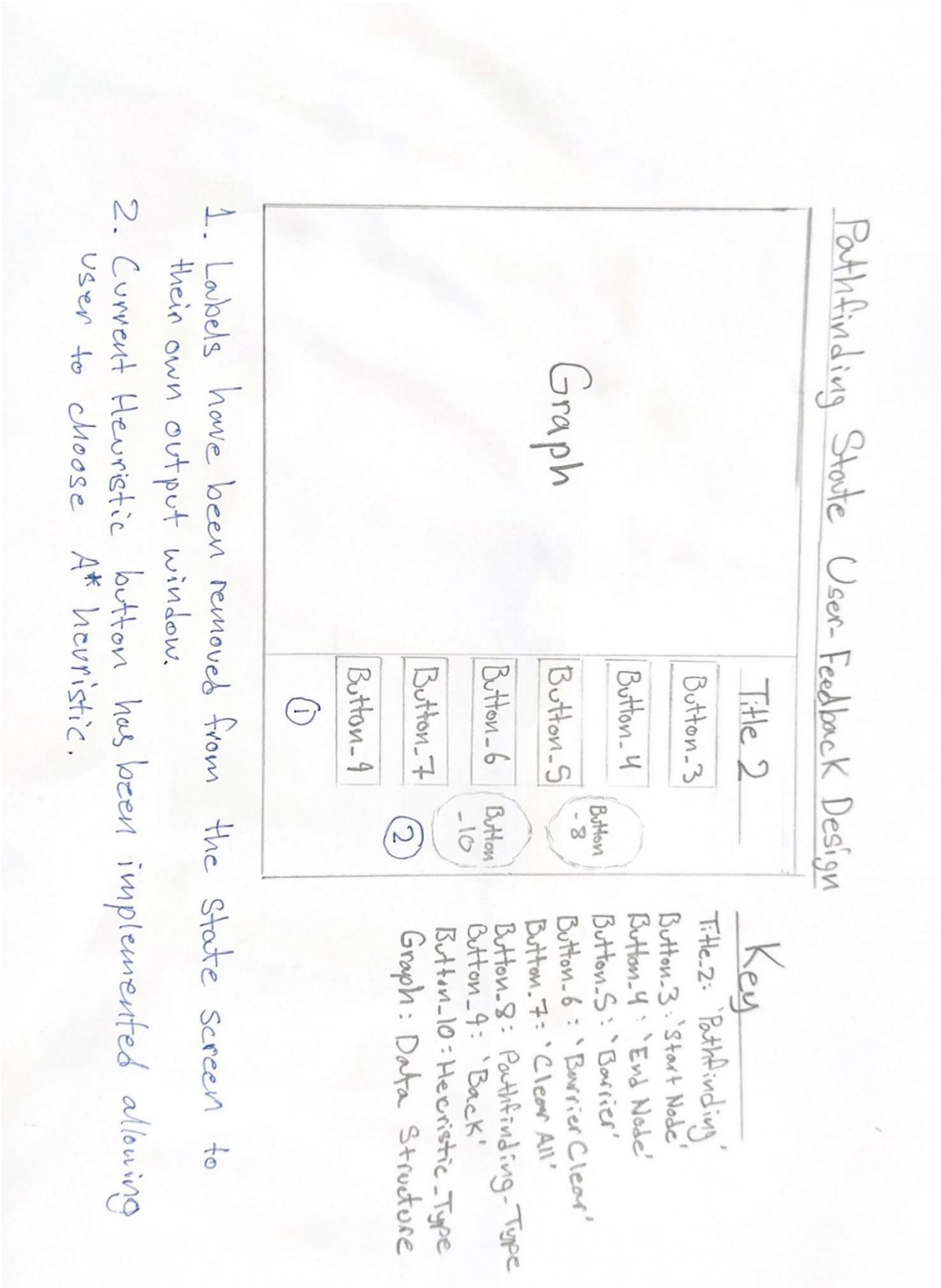


Annotated Main Menu Design

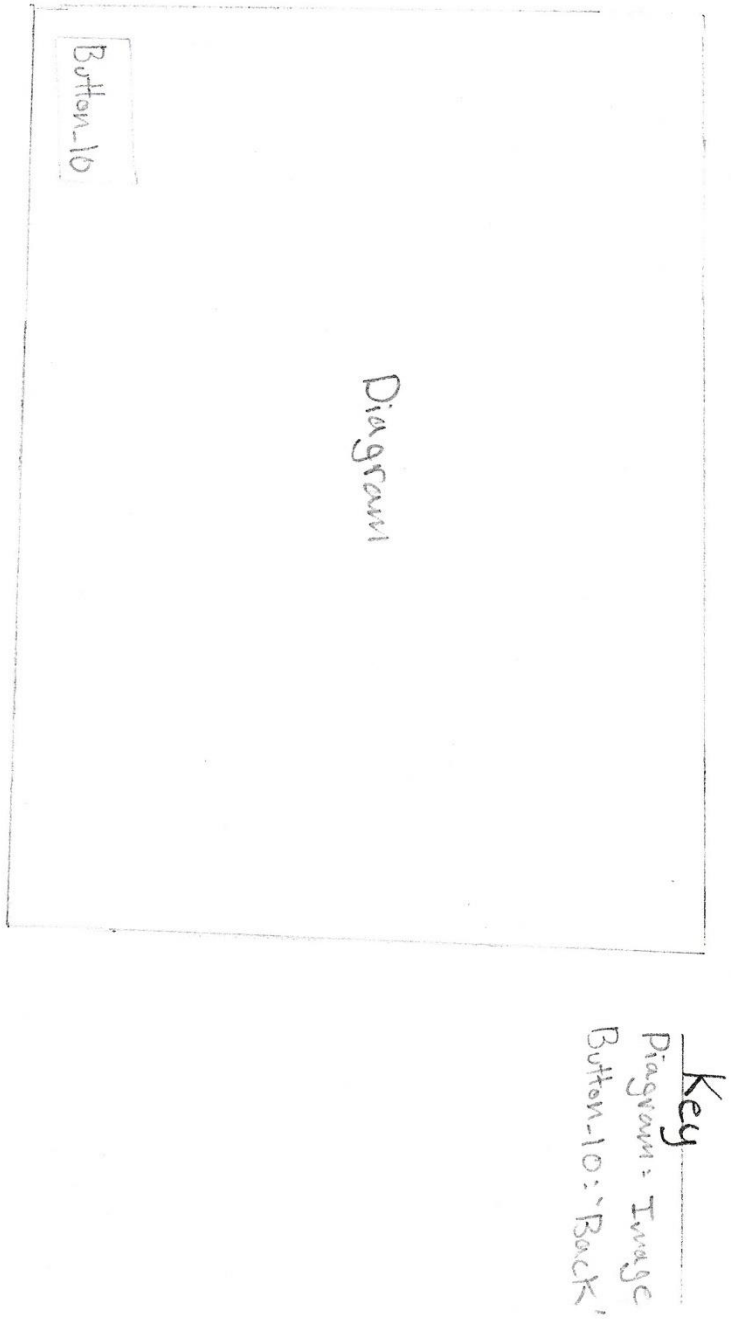
Pathfinding State Design



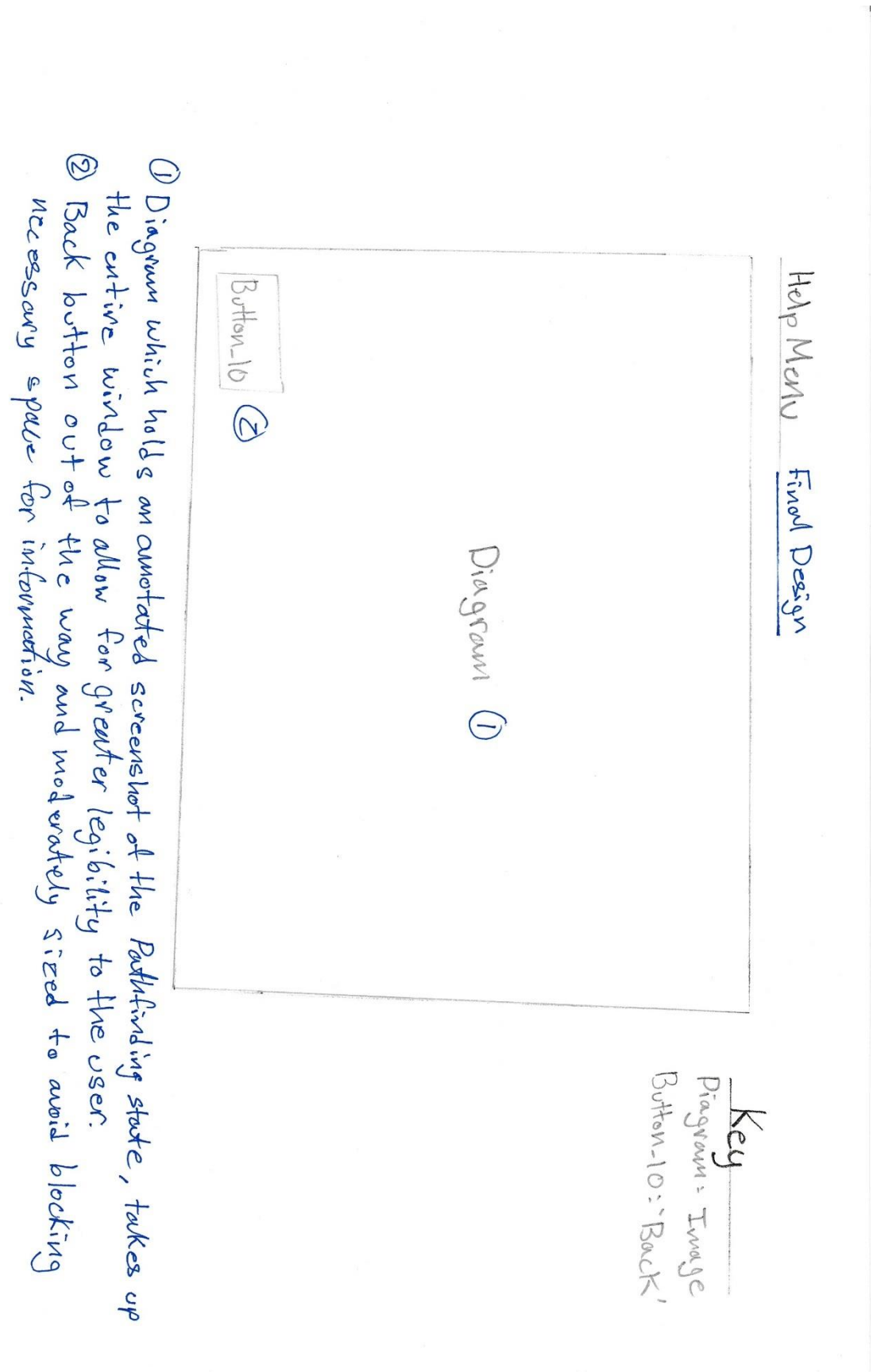
Annotated Pathfinding State Design



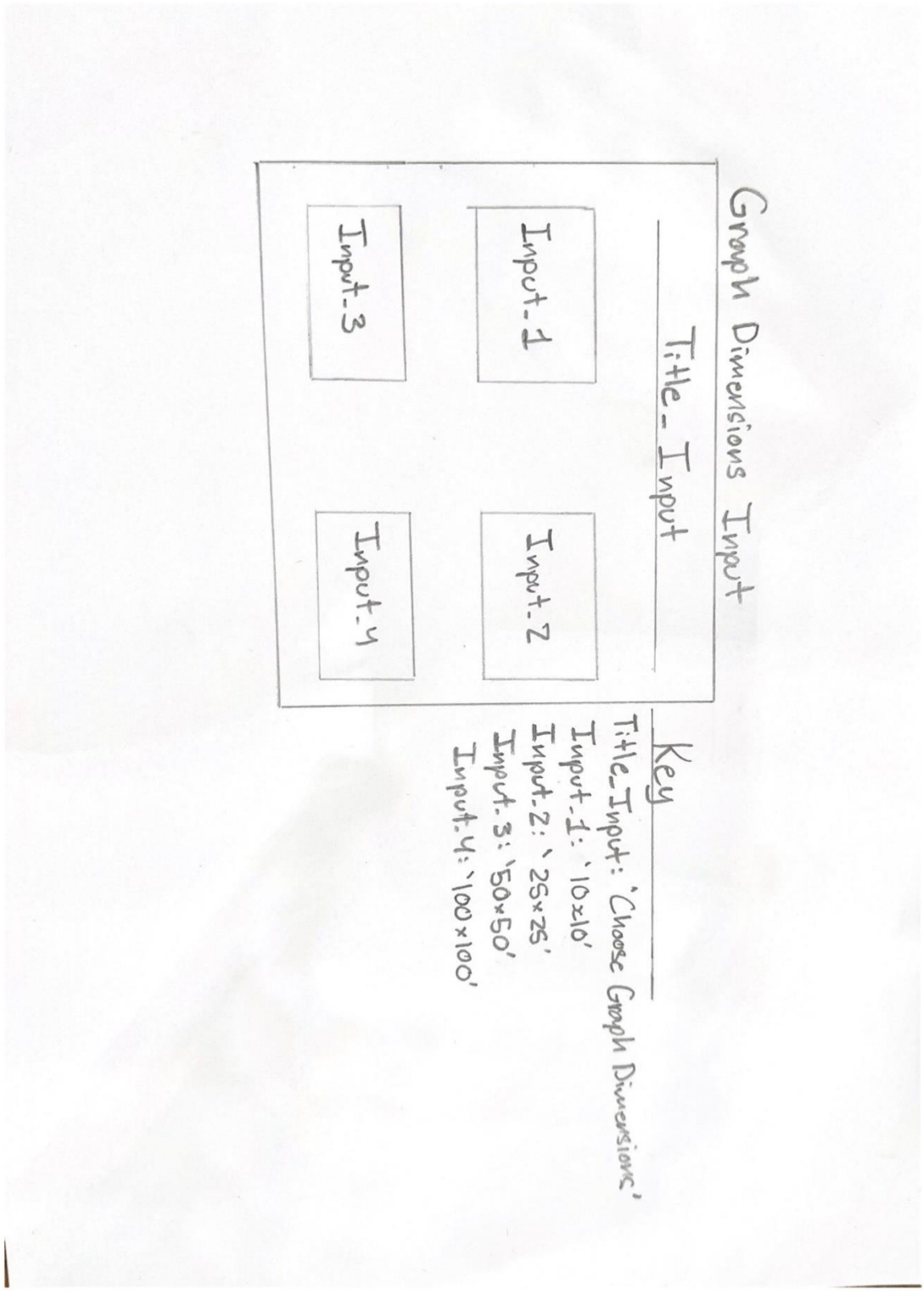
Help Menu Design



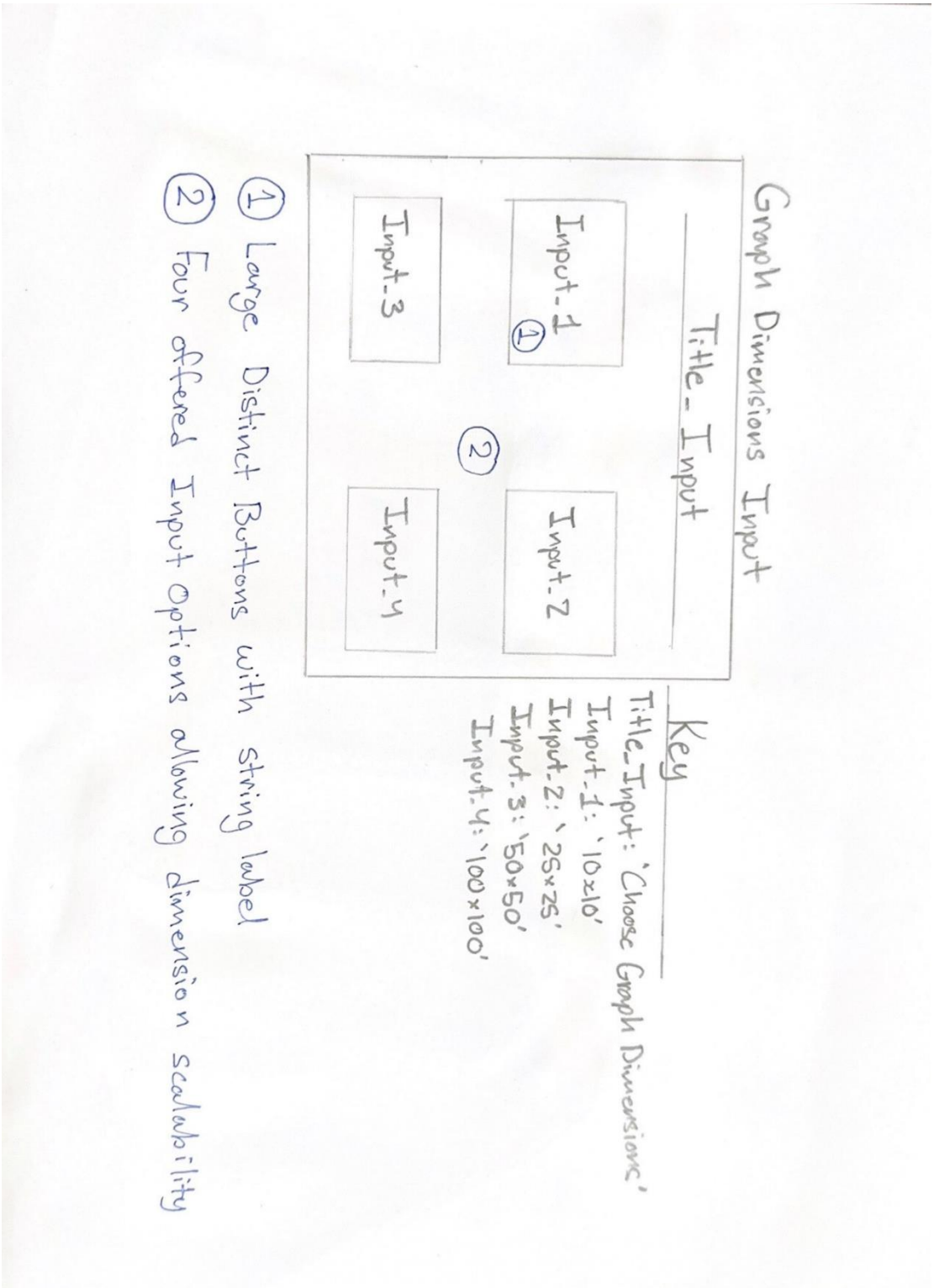
Annotated Help Menu Design



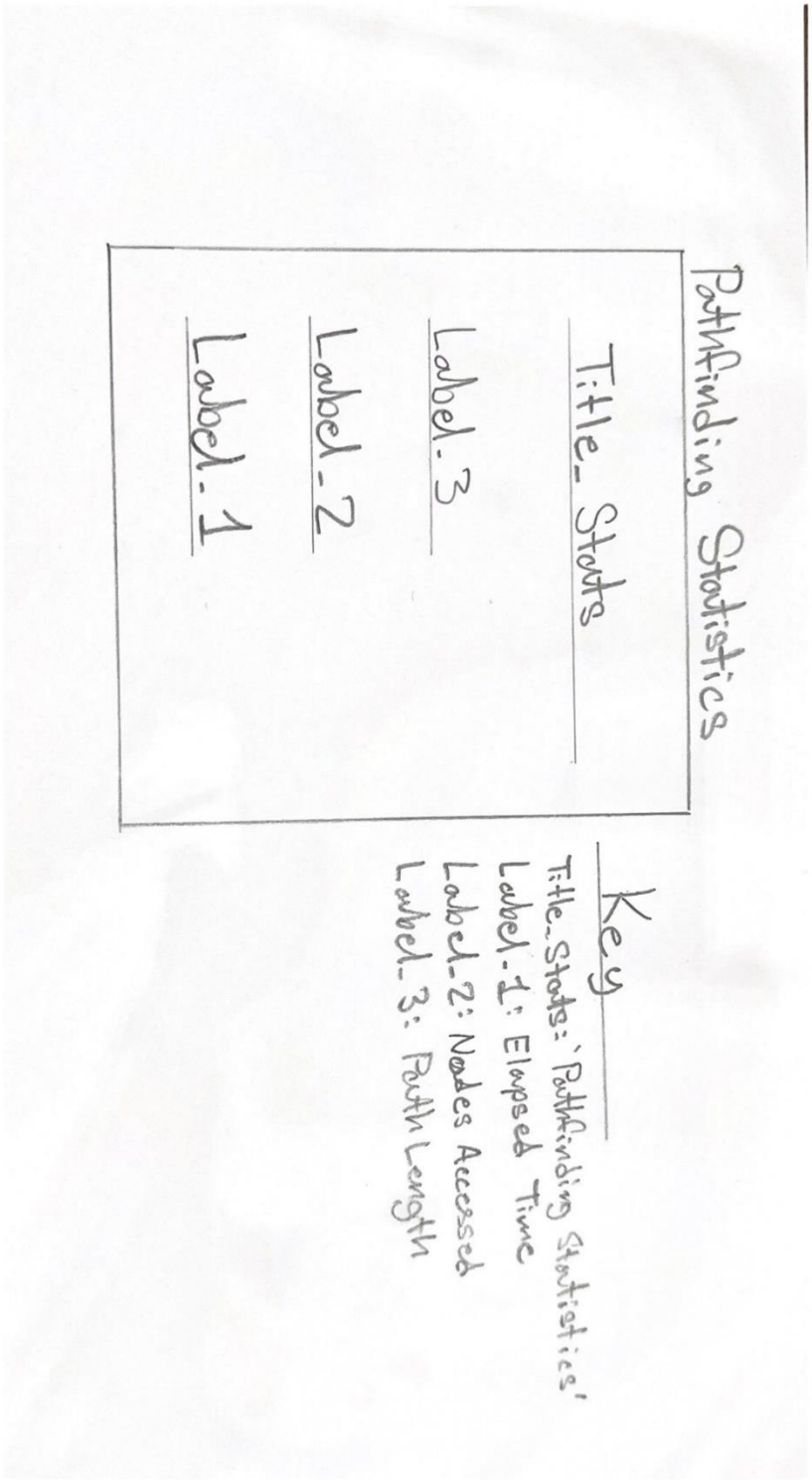
Graph Dimensions Input Design



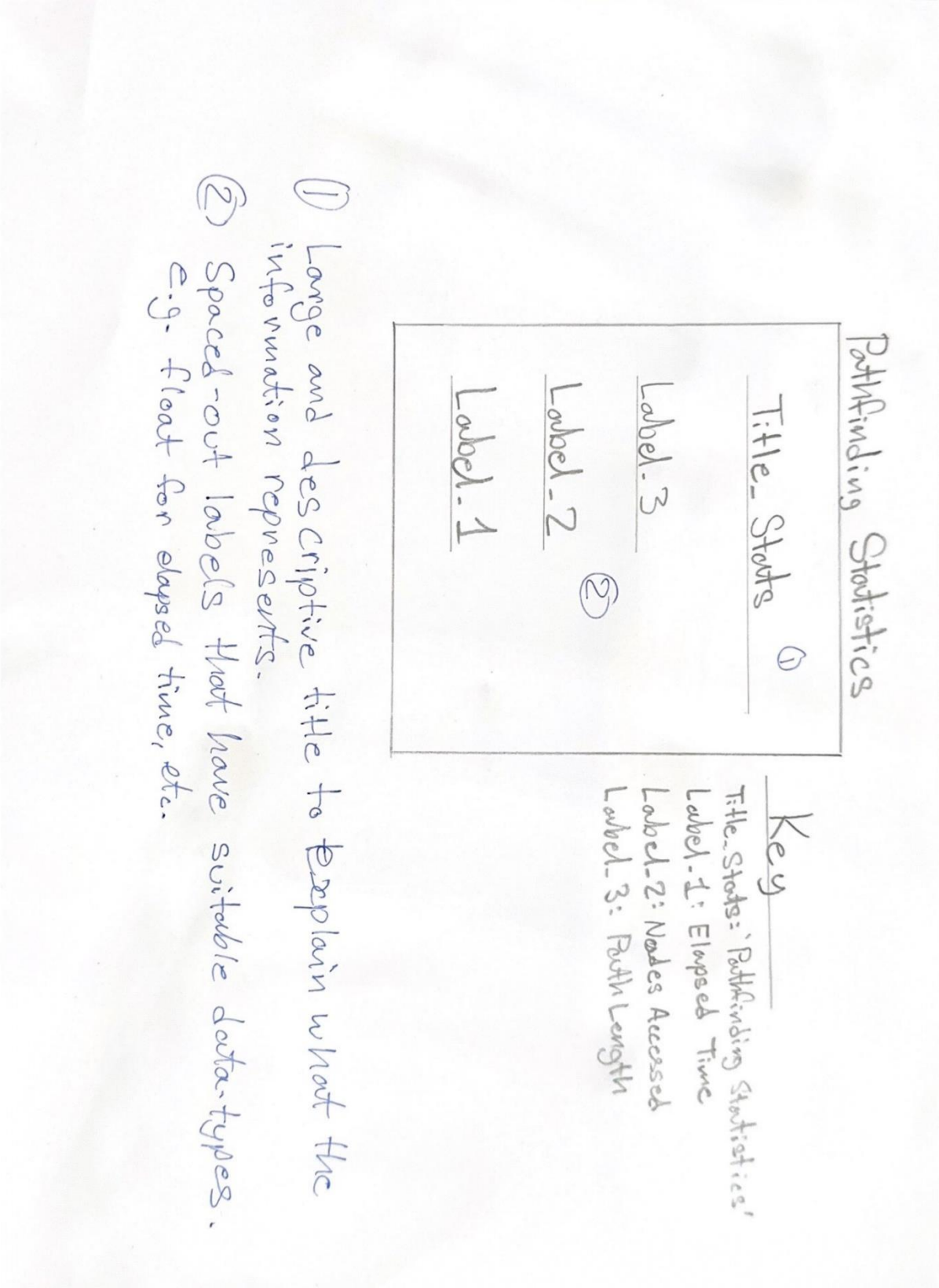
Annotated Graph Dimensions Input Design



Pathfinding Statistics Design



Annotated Pathfinding Statistics Design



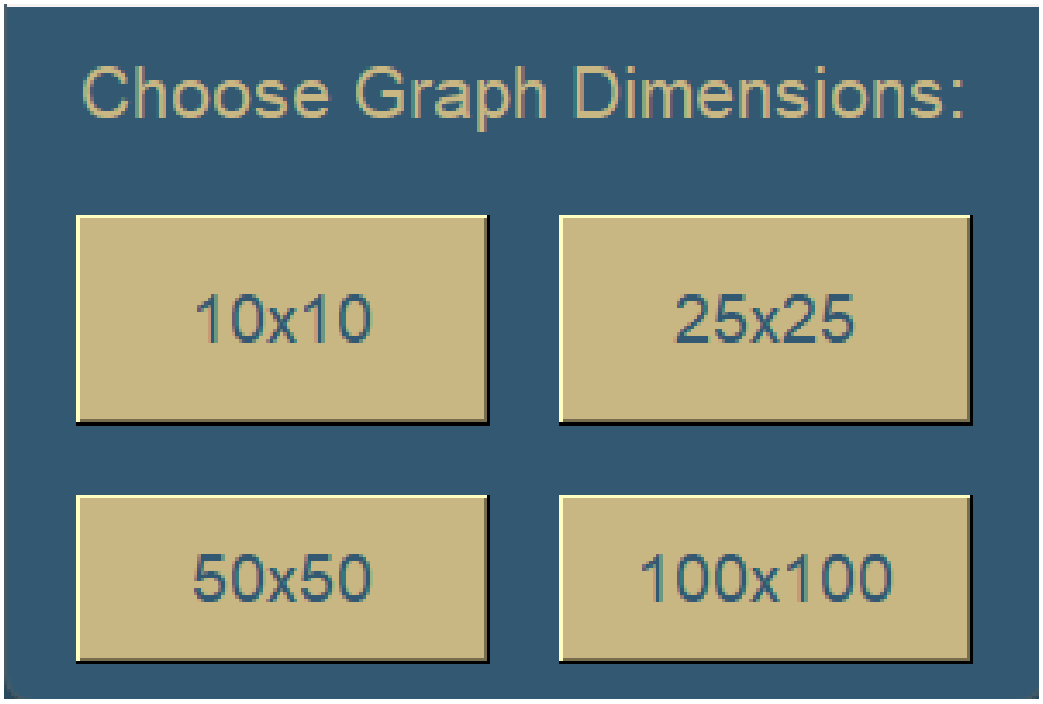
Implemented Interface Overview

Individual Element Visuals

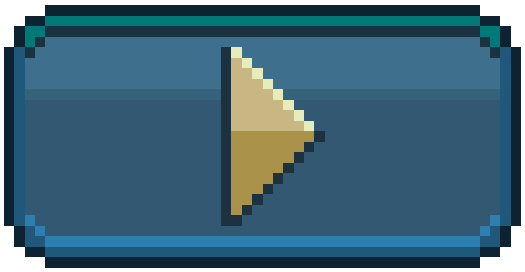
Background_1:

Background_2:

Graph_Dimensions_Input:



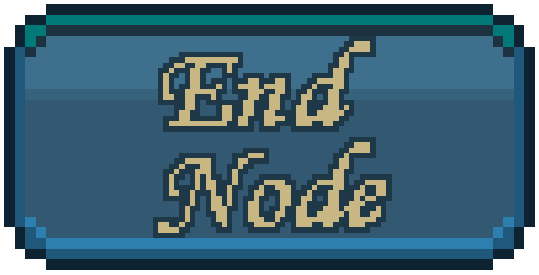
Button_1:



Button_2:

Button_3:

Button_4:



Button_5:

Button_6:

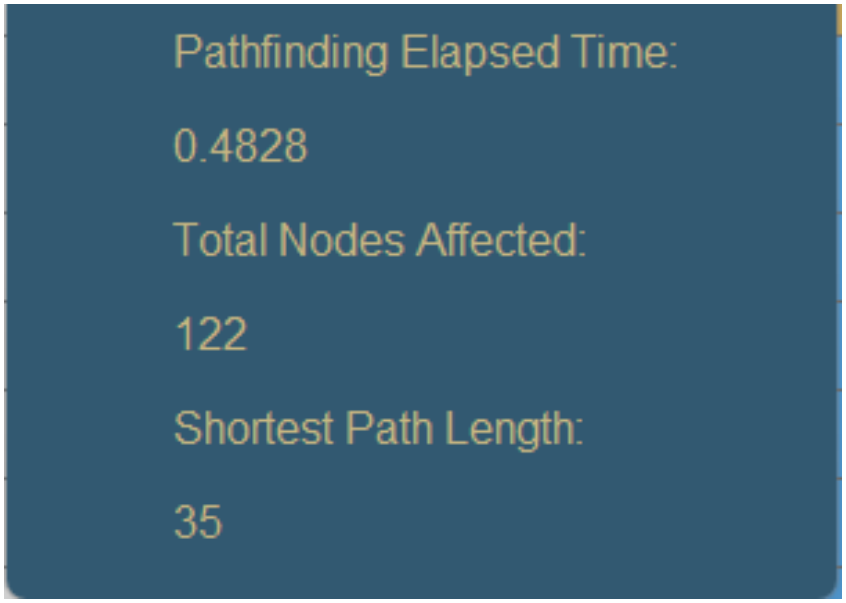
Button_7:

Button_8 (Dijkstra Phase):

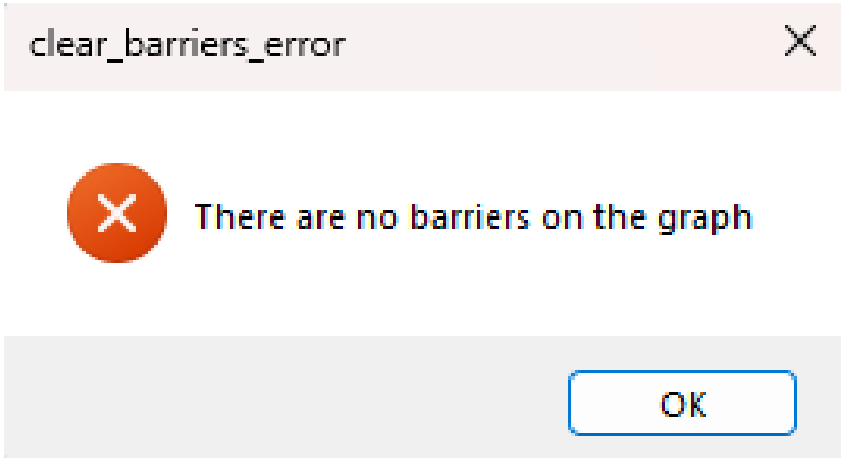
Button_8 (A* Phase):

Button_9:

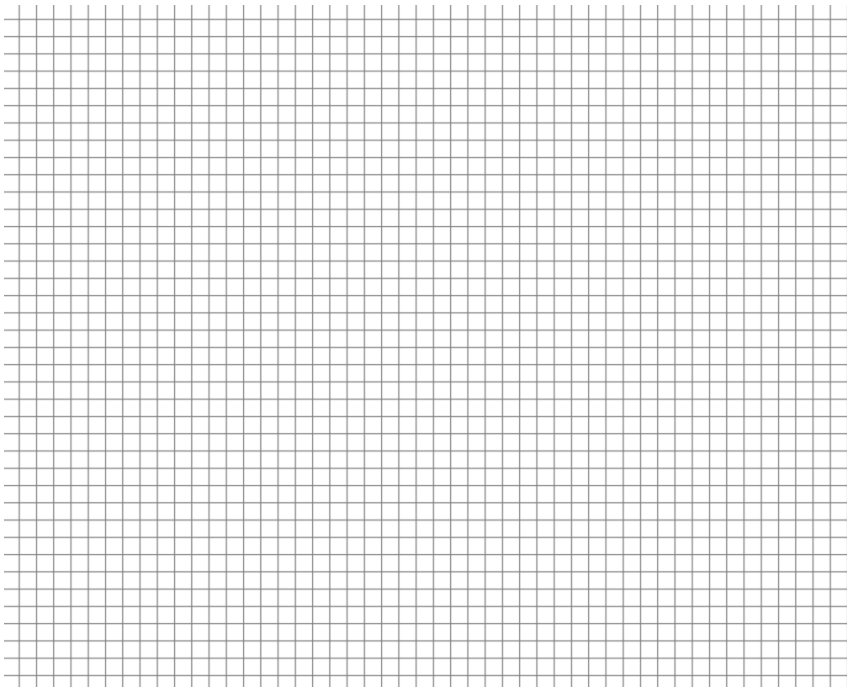
Output_Last_Path_Statistics:



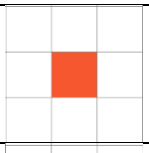
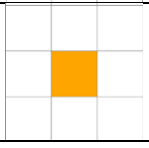
Error_Message:

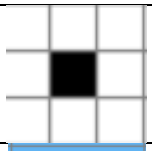
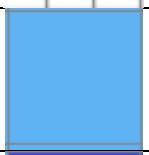
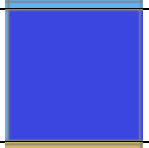
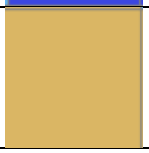


Graph:



Graph Visualisation Elements

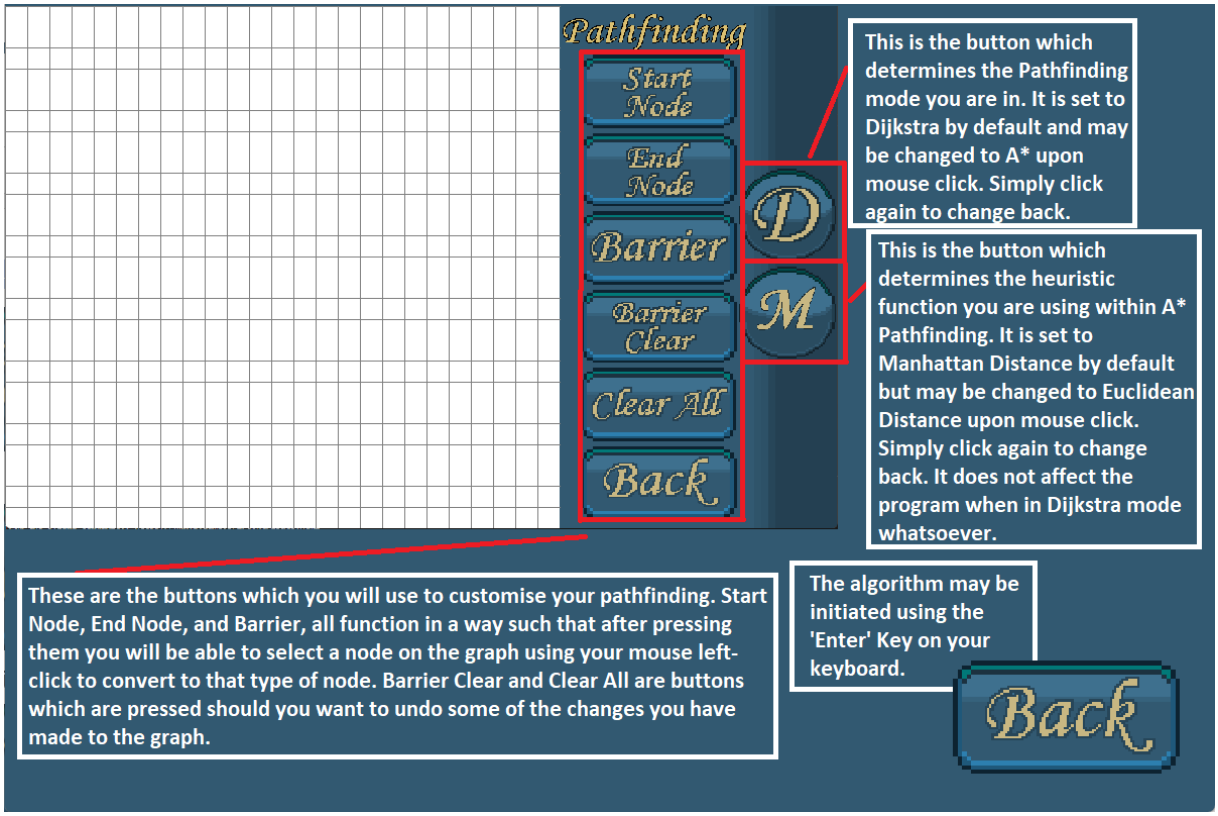
Stage	Graphic
Start Node	
End Node	

Barrier Node	
Dequeued Node	
Queued Node	
Determined Shortest Path Node	

Implemented Complete Interface Overview

Menu_State:

Help_State:

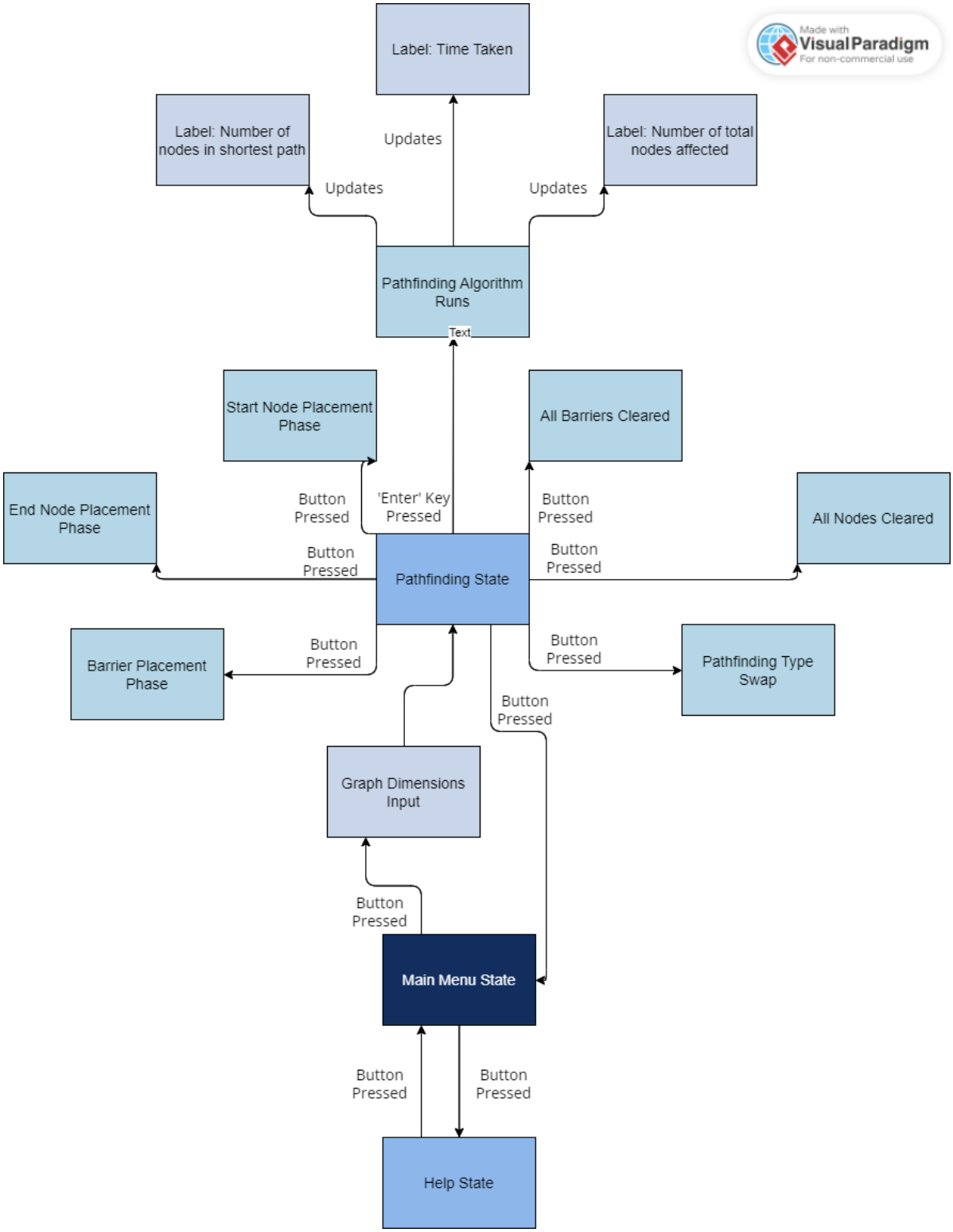


Pathfinding_State:

Form Navigation Design Diagram

This navigation diagram is used to display how users navigate between the different states of graph manipulation, as well as how the visual elements are updated due to user input. This branches out from the main menu state – the initial state of the program.

Navigation Diagram



Technical Implementation

Complexities Overview

Technical Skills Evidenced

Group	Model (including data model/structure)	Page number(s) evidenced in my NEA	Algorithms	Page number(s) evidenced in my NEA
A	Hash Table - Data Structure	177	Graph/Tree Traversal	169, 171, 175
	Binary Heap – Data Structure	175		
	Priority Queue – Data Structure	176	List Operations	160
	Graph – Data Structure	159	Stack/Queue Operations	169, 176
	Complex user-defined use of object orientated programming (OOP) model, e.g., classes, inheritance, composition, polymorphism, interfaces[1]	167, 171, 180	Hashing	169, 177
			Recursive Algorithms	148, 149
			Dynamic generation of objects based on complex user-defined use of OOP model[2]	148, 160

Group	Model (including data model/structure)	Page number(s) evidenced in my NEA	Algorithms	Page number(s) evidenced in my NEA
B	Multi-dimensional arrays	160	Simple user defined algorithms (e.g., a range of mathematical/statistical calculations)	171, 177
	Dictionaries	145, 169, 172	Generation of objects based on simple OOP model	148
	Simple scientific/mathematical /robotics/control/business model[3]	Explained Below		

Group	Model (including data model/structure)	Page number(s)	Algorithms	Page number(s)

		evidenced in my NEA		evidenced in my NEA
C	Single-dimensional arrays	167, 168	Simple mathematical calculations (e.g., average)	151
	Appropriate choice of simple data types	145, 185		

Further Evidence on Technical Skills

1. Complex user-defined use of object orientated programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces:
    a. Inheritance: The AStar_Pathfinding class inherits the attributes and methods of the Dijkstra_Pathfinding class.
    b. Overriding: The AStar_Pathfinding class overrides the algorithm method to incorporate its heuristic methods.
    c. Interfaces: The methods within the classes have clear interfaces which use a multitude of arguments and attributes within – such as the algorithm method.
    d. Composition: There are many instances of composition within the code, for example when the Node class is composed of the State class, as well as the Dijkstra_Pathfinding class being composed of the Priority_Queue and Hash_Table classes.

2. Dynamic generation of objects based on complex user-defined use of OOP model:
    a. User-Defined input for the generation: the Pathfinding_Dimensions object which then passes the value into the Pathfinding_State object.
    b. Dynamic Generation: Within the Graph class, when the make_grid method is used, the grid size is dependent on the dimensions attribute, which was the previous user input. As the grid consists of n x n nodes – where n is the dimensions attribute, that many node objects are generated. When the Pathfinding_State is closed, the Graph object is deleted, which leads to the deletion of the node objects.

3. Simple scientific/mathematical /robotics/control/business model
    a. The code itself is a simple mathematical model because it represents a real-world problem which may be solved using mathematical concepts.
    b. In this case, the problem is finding the shortest path between two points in a graph, which are represented mathematically as a set of vertices and edges. The software uses mathematical algorithms such as A* and Dijkstra to find the shortest path between the two points, and the choice between the Manhattan and Euclidean distance heuristics allows the user to choose the appropriate algorithm for their specific problem. Additionally, the Hash Table employs the use of a minimal perfect hash algorithm, which is a

mathematical solution that allows the hash table to be as space efficient as possible, whilst maintaining a design resulting in no collisions in the data.

c. The software also allows the user to choose the size dimensions of the graph, which is another mathematical concept. By providing a simple interface for the user to interact with, the software can be used to explore and experiment with different graph configurations and algorithms, which is a key feature of mathematical modelling.

Coding Practices Evidenced

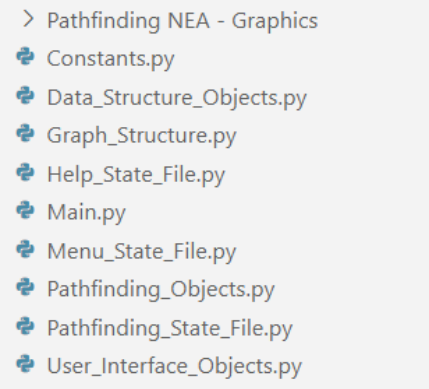
Style	Characteristic	Page number(s) evidenced in my NEA
Excellent	Modules (subroutines) with appropriate interfaces - by interfaces we mean sensible identifier name, appropriate parameters, and return values	151, 171
	Loosely coupled modules (subroutines) – module code interacts with other parts of the program through its interface only. This means simply calling the module and passing arguments to make the program function	160, 165
	Cohesive modules (subroutines) – module code does just one thing.	171
	Modules (collections of subroutines) – subroutines with common purpose grouped.	171
	Defensive programming	148
	Good exception handling.	153, 156
Good	Well-designed user interface	150, 155
	Modularisation of code. Putting subroutines into separate files and then importing the module and calling the functions	162, 181
	Good use of local variables	151
	Minimal use of global variables	*Not used
	Managed casting of types	178
	Use of constants	145
	Appropriate indentation	164
	Self-documenting code.	171
	Consistent style throughout (expanded on below in my coding practices)	166
	File paths parameterised	145
Basic	Meaningful identifier names for variables and subroutines	151
	Annotation used effectively where required.	160

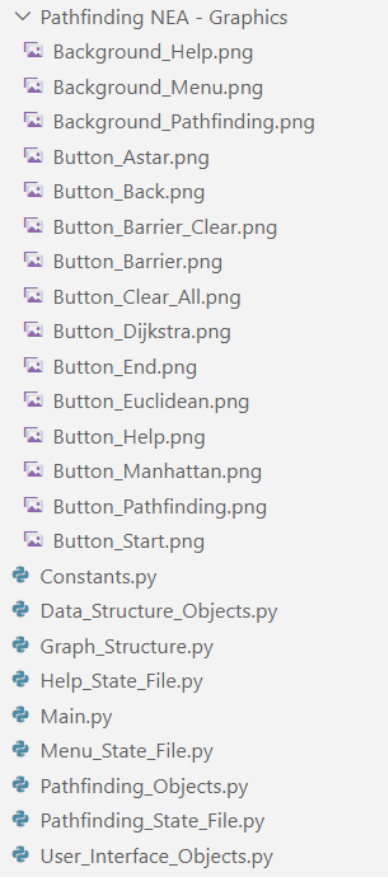
* Constants have global scope but are unchanging variables.

Code Modules

Working File Directory

This is the working file directory of the program. All the Python modules must be within this directory – the graphics must be within their own file inside of this directory as well. As they are in the same directory, they may be easily imported into the necessary files. As the modules are class-based, these classes may be accessed within the program after being imported. Additionally, when the file is converted into an executable, the directory will consist of a single executable file along with its graphics file.





Constants.py

I classified all variables that I will be using more than once as non-temporary variables. As such, they will be represented within this file as 'constants'. Though, officially, constants do not exist within the version of Python I am currently working in (3.10.8), I have implemented them with the Python convention of being fully uppercase. They can be used from anywhere within the program but never change their value throughout its running, hence the term, constant.

```
import pygame

WINDOW_HEIGHT = 800
WINDOW_WIDTH = 1200

# create a Pygame display with the given dimensions
WINDOW = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))

# define colours using RGB values
RED = (247, 87, 47)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
ORANGE = (255, 165, 0)
GREY = (128, 128, 128)
DARK_BLUE = (59, 70, 225)
LIGHT_BLUE = (96, 179, 243)
GOLD = (218, 182, 100)


# define the states with the corresponding colours
STATES = {
    None: WHITE,
    "Open": DARK_BLUE,
    "Closed": LIGHT_BLUE,
    "Start": RED,
    "End": ORANGE,
    "Barrier": BLACK,
    "Path": GOLD,
}

# load background images
MENU_BACKGROUND_IMAGE = pygame.image.load("Pathfinding NEA -
Graphics\Background_Menu.png")
PATHFINDING_BACKGROUND_IMAGE = pygame.image.load("Pathfinding NEA
Graphics\Background_Pathfinding.png")
HELP_BACKGROUND_IMAGE = pygame.image.load("Pathfinding NEA -
Graphics\Background_Help.png")

# load button images
```

Appropriate choice of simple data types

Use of Constants

Dictionaries

File paths parameterised

```
BUTTON_PATHFINDING_IMAGE = pygame.image.load("Pathfinding NEA -
Graphics\Button_Pathfinding.png")
BUTTON_BACK = pygame.image.load("Pathfinding NEA -
Graphics\Button_Back.png")
BUTTON_HELP = pygame.image.load("Pathfinding NEA -
Graphics\Button_Help.png")
BUTTON_DIJKSTRA = pygame.image.load("Pathfinding NEA -
Graphics\Button_Dijkstra.png")
BUTTON_ASTAR = pygame.image.load("Pathfinding NEA -
Graphics\Button_Astar.png")
BUTTON_MANHATTAN = pygame.image.load("Pathfinding NEA -
Graphics\Button_Manhattan.png")
BUTTON_EUCLIDEAN = pygame.image.load("Pathfinding NEA -
Graphics\Button_Euclidean.png")
BUTTON_START_NODE = pygame.image.load("Pathfinding NEA -
Graphics\Button_Start.png")
BUTTON_END_NODE = pygame.image.load("Pathfinding NEA -
Graphics\Button_End.png")
BUTTON_BARRIER_NODE = pygame.image.load("Pathfinding NEA -
Graphics\Button_Barrier.png")
BUTTON_BARRIER_NODE_CLEAR = pygame.image.load(
    "Pathfinding NEA - Graphics\Button_Barrier_Clear.png"
)
BUTTON_CLEAR_ALL = pygame.image.load("Pathfinding NEA -
Graphics\Button_Clear_All.png")

# scale button images
BUTTON_DIJKSTRA_SCALED = pygame.transform.scale(
    BUTTON_DIJKSTRA,
    (BUTTON_DIJKSTRA.get_width() * 0.7, BUTTON_DIJKSTRA.get_height() *
0.7),
)
BUTTON_ASTAR_SCALED = pygame.transform.scale(
    BUTTON_ASTAR, (BUTTON_ASTAR.get_width() * 0.7,
BUTTON_ASTAR.get_height() * 0.7)
)
BUTTON_MANHATTAN_SCALED = pygame.transform.scale(
    BUTTON_MANHATTAN,
    (BUTTON_MANHATTAN.get_width() * 0.7, BUTTON_MANHATTAN.get_height() *
0.7),
)
BUTTON_EUCLIDEAN_SCALED = pygame.transform.scale(
    BUTTON_EUCLIDEAN,
    (BUTTON_EUCLIDEAN.get_width() * 0.7, BUTTON_EUCLIDEAN.get_height() *
0.7),
)
BUTTON_PATHFINDING_IMAGE_SCALED = pygame.transform.scale(
```

```
    BUTTON_PATHFINDING_IMAGE, (BUTTON_PATHFINDING_IMAGE.get_width() *
0.75, BUTTON_PATHFINDING_IMAGE.get_height() * 0.75)
)
BUTTON_HELP_SCALED = pygame.transform.scale(
    BUTTON_HELP, (BUTTON_HELP.get_width() * 0.75, BUTTON_HELP.get_height()
* 0.75)
)
```

Main.py

This is the most vital file within the implementation of my proposed solution. This is the only file within the directory which is a script. This means that it is the file being run – as it controls and implements all the other modules.

It is based on a recursive main loop system. This means that it contains the main software loop, which has a main state of the Menu_State object. As can be seen, when the Pathfinding_State or Help_State objects are destroyed – when the 'back' button is pressed, the main state will simply be called again, and you will return to the menu. This is not the same as exiting the program, however, as that can be done from any point within – destroying the full system.

*There is an implementation of defensive programming within this script as the user input for the graph's dimensions is a separate Tkinter window. As such, I had to implement as fail-safe in the case of the user destroying the input window. As can be seen, in this situation the main function simply recurs, giving the user the Menu_State screen again.

```
# import required modules
from Pathfinding_State_File import Pathfinding_State
from Menu_State_File import Menu_State
from User_Interface_Objects import Graph_Dimensions_Input
from Help_State_File import Help_State

# Define the main function
def software_loop():
    # Create a new Menu_State object
    Menu = Menu_State()

    # Check the value returned by the main loop method of the Menu_State
object
    if Menu.main_loop():
        # If the return value is True, create a new Graph_Dimensions_Input
object and retrieve its return value
        Pathfinding_Dimensions = Graph_Dimensions_Input()


        if Pathfinding_Dimensions.return_value == None:
            return software_loop ()


        Current_State =
Pathfinding_State(Pathfinding_Dimensions.return_value)
        Current_State.pathfinding_loop()


    else:
```

Generation of objects based on simple OOP model

Dynamic generation of objects based on complex user-defined use of OOP model[1]

*Defensive programming

Recursive Algorithms

```
            Current_State = Help_State()
            Current_State.help_loop()

        # Call the main function again
        return software_loop()


# Call the main function if this script is run as the main program
if __name__ == "__main__":
    software_loop()
```

Recursive Algorithms

Menu_State_File.py

This is the module responsible for the Menu_State object. It encompasses the entirety of the class. The Menu_State object will return a Boolean value depending on the outcome of the main_loop method. This will reflect the user's choice on whether to go to the Pathfinding state or Help Menu state. The Menu_State object contains the staticmethod get_centered_image_position to allow me to place the buttons in such positions on the screen that the user has the best experience with the interface.

```
# import required modules

import pygame, sys
from User_Interface_Objects import UI_Button
from Constants import *


class Menu_State:
    """
    This class is responsible for the Menu state of the software
    """

    def __init__(self):
        # Initialize pygame
        pygame.init()

        # Set the caption and icon of the game window
        pygame.display.set_caption("Main Menu")
        pygame.display.set_icon(BUTTON_ASTAR)

        # Set the background image of the game window
        WINDOW.blit(MENU_BACKGROUND_IMAGE, (0, 0))

        # Calculate the x,y coordinates for the two buttons
        self.x, self.y = self.get_centered_image_position(
            BUTTON_HELP_SCALED.get_width(),
BUTTON_HELP_SCALED.get_height()
        )

        # Create a play button
        self.Button_Pathfinding_Object = UI_Button(
            *self.get_centered_image_position(
                BUTTON_PATHFINDING_IMAGE_SCALED.get_width(),
BUTTON_PATHFINDING_IMAGE_SCALED.get_height()
            ),
            BUTTON_PATHFINDING_IMAGE_SCALED,
            1,
            WINDOW
        )
```

Well-designed user interface

```
        # Create a help button
        self.Button_Help_Object = UI_Button(
            self.x,
            self.y + self.Button_Pathfinding_Object.height + 40,
            BUTTON_HELP_SCALED,
            1,
            WINDOW,
        )

    @staticmethod
    def get_centered_image_position(image_width, image_height):
        """Return the x,y coordinates of the top-left corner of an image
with size m by n centered on a WINDOW with size p by q."""
        # Calculate the x,y coordinates of the top-left corner of the
image to be centered
        x = (WINDOW_WIDTH - image_width) / 2
        y = (WINDOW_HEIGHT - image_height) / 2
        return (x, y)

    def main_loop(self):
        # The main loop of the game
        while True:
            for event in pygame.event.get():
                # Check if the user closed the window
                if event.type == pygame.QUIT:
                    sys.exit()

            # Check if the help button is clicked
            if self.Button_Help_Object.draw() == True:
                return False

            # Check if the play button is clicked
            elif self.Button_Pathfinding_Object.draw() == True:
                return True

            # Update the display
            pygame.display.update()
```

Meaningful identifier names for variables and subroutines

Modules (subroutines) with appropriate interfaces.

Simple mathematical calculations (eg average)

Good use of local variables

Help_State_File.py

This is the module responsible for the Help_State object. This object is instantiated when the user would like to gain understanding on how to operate the software. The help_loop method contains an exception for an UnboundLocalError as it self-destructs upon the 'Back' button being pressed. This is to ensure that the program does not enter any unnecessary loop structures as the user switches between states.

```
# import required modules
import pygame, sys
from User_Interface_Objects import UI_Button
from Constants import *

# Defining a class for the help menu
class Help_State:
    # Initializing the class and creating the window and button object
    def __init__(self):
        # Initializing pygame
        pygame.init()
        # Setting the caption of the window
        pygame.display.set_caption("Help Menu")

        # Creating an object for the back button
        self.Button_Back_Object = UI_Button(940, 650, BUTTON_BACK, 0.5,
WINDOW)

        # Displaying the help menu background image
        WINDOW.blit(HELP_BACKGROUND_IMAGE, (0, 0))

    # Creating the main loop for the help menu
    def help_loop(self):
        try:
            # Running an infinite loop until the user closes the window or
clicks the back button
            while True:
                # Checking for events
                for event in pygame.event.get():
                    # If the user clicks the close button, exit the
program
                    if event.type == pygame.QUIT:
                        sys.exit()
                    # If the user clicks the back button, delete the
object and return to the previous menu
                    if self.Button_Back_Object.draw():
                        del self

                # Updating the display
                pygame.display.update()
```

```
        # Catching an error that occurs when the object is deleted before
exiting the loop
        except UnboundLocalError:
            pass
```

Good Exception Handling

Pathfinding_State_File.py

This module is responsible for the Pathfinding_State object. This object is the 'outer loop' of the Pathfinding state. In essence, during operation, the user may be within the Graph object – which is the 'inner loop', or this one. As such, there are some complexities within the code which were required to allow it to function smoothly. For example, when the software enters the Graph object to manipulate the graph, it must leave said object upon user mouse-click outside of the graph area. However, this will render the first mouse-click irrelevant as it is still within the Graph object. Therefore, I had to implement the position_check and draw method systems to allow the state to function as necessary.

When the user clicks out of the graph, the mouse position is returned as the graph loop closes, this allows me to use the position_check method to see whether this occurs on a coordinate position that coincides with the Start_Node, End_Node, and Barrier_Node buttons. If so, the draw method for that button will be run, which will set the mode for the type of graph manipulation that will be occurring within the graph.pathfinding method. This is a solution that renders the need for a double click outside of the graph to be unnecessary – effectively improving the user-experience.

The Pathfinding_State object also contains an UnboundLocalError exception due to it deleting itself upon the 'Back' button being pressed. This avoids unnecessary object loops and simply shifts the software back to the Menu_State object within the main software_loop method.

```
# import required modules
import pygame, sys
from Graph_Structure import Graph
from Constants import *
from User_Interface_Objects import *

# define a class for pathfinding state
class Pathfinding_State:
    def __init__(self, graph_dimensions_input):

        pygame.init()

        # set the title of the window
        pygame.display.set_caption("Shortest Path Visualiser")


        self.graph_dimensions_input = graph_dimensions_input
        self.Button_Start_Node_Object = UI_Button(830, 75,
BUTTON_START_NODE, 0.5, WINDOW)
        self.Button_End_Node_Object = UI_Button(830, 195, BUTTON_END_NODE,
0.5, WINDOW)
```

```
        self.Button_Barrier_Node_Object = UI_Button(830, 315,
BUTTON_BARRIER_NODE, 0.5, WINDOW)
        self.Button_Barrier_Node_Clear_Object = UI_Button(
            830, 435, BUTTON_BARRIER_NODE_CLEAR, 0.5, WINDOW
        )
        self.Button_Clear_All_Object = UI_Button(
            830, 555, BUTTON_CLEAR_ALL, 0.5, WINDOW
        )
        self.Button_Back_Object = UI_Button(830, 675, BUTTON_BACK, 0.5,
WINDOW)
        self.Button_Pathfinding_Type_Swap = Boolean_Button(
            1058, 250, BUTTON_DIJKSTRA_SCALED, 1, WINDOW,
BUTTON_ASTAR_SCALED
        )
        self.Button_Heuristic = Boolean_Button(
            1058, 400, BUTTON_MANHATTAN_SCALED, 1, WINDOW,
BUTTON_EUCLIDEAN_SCALED
        )
        self.graph = Graph(self.graph_dimensions_input)

        # set the background image
        WINDOW.blit(PATHFINDING_BACKGROUND_IMAGE, (0, 0))

        # draw the graph
        self.graph.draw()

    def position_check(self, pos):
        # check which button was pressed
        if self.Button_Start_Node_Object.rect.collidepoint(pos):
            self.draw_start()
        elif self.Button_End_Node_Object.rect.collidepoint(pos):
            self.draw_end()
        elif self.Button_Barrier_Node_Object.rect.collidepoint(pos):
            self.draw_barrier()

    def draw_start(self):
        # if start button is clicked
        if self.Button_Start_Node_Object.draw():

            # get the position of the button clicked
            pos = self.graph.pathfinding(
                "Start",
                self.Button_Pathfinding_Type_Swap.get_boolean_condition(),
                self.Button_Heuristic.get_boolean_condition(),
            )

            # try to check the position
            try:
```

Well-designed user interface

```
                self.position_check(pos)
            except TypeError:
                pass

    def draw_end(self):
        # if end button is clicked
        if self.Button_End_Node_Object.draw():

            # get the position of the button clicked
            pos = self.graph.pathfinding(
                "End",
                self.Button_Pathfinding_Type_Swap.get_boolean_condition(),
                self.Button_Heuristic.get_boolean_condition(),
            )

            # try to check the position
            try:
                self.position_check(pos)
            except TypeError:
                pass

    def draw_barrier(self):
        # if barrier button is clicked
        if self.Button_Barrier_Node_Object.draw():

            # get the position of the button clicked
            pos = self.graph.pathfinding(
                "Barrier",
                self.Button_Pathfinding_Type_Swap.get_boolean_condition(),
                self.Button_Heuristic.get_boolean_condition(),
            )

            # try to check the position
            try:
                self.position_check(pos)
            except TypeError:
                pass

    def pathfinding_loop(self):
        try:
            while True:
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        sys.exit()

                    self.draw_start()

                    self.draw_end()
```

Good Exception Handling

```
                    self.draw_barrier()

                    if self.Button_Back_Object.draw():
                        del self

                    elif self.Button_Barrier_Node_Clear_Object.draw():
                        self.graph.clear_barriers()

                    elif self.Button_Clear_All_Object.draw():
                        self.graph.clear_graph()

                    elif self.Button_Pathfinding_Type_Swap.draw():
                        self.Button_Pathfinding_Type_Swap.update()

                    elif self.Button_Heuristic.draw():
                        self.Button_Heuristic.update()

                    if event.type == pygame.KEYDOWN:

                       if (
                            event.key == pygame.K_RETURN
                            and self.graph.start
                            and self.graph.end
                       ):
                            self.graph.running_algorithm(
                                self.Button_Pathfinding.get_boolean_condition(
),
                                self.Button_Heuristic.get_boolean_condition(),
                            )
                       elif (
                            event.key == pygame.K_RETURN
                            and self.graph.start
                            and not self.graph.end
                       ):
                            Insufficient_Nodes_Error = ErrorMessage(
                                "insufficient_nodes_error"
                            )
                            Insufficient_Nodes_Error.show_error_message()
                       elif (
                            event.key == pygame.K_RETURN
                            and self.graph.end
                            and not self.graph.start
                       ):
                            Insufficient_Nodes_Error = ErrorMessage(
                                "insufficient_nodes_error"
                            )
                            Insufficient_Nodes_Error.show_error_message()
```

```
                    elif (
                        event.key == pygame.K_RETURN
                        and not self.graph.start
                        and not self.graph.end
                    ):
                        Insufficient_Nodes_Error = ErrorMessage(
                            "insufficient_nodes_error"
                        )
                        Insufficient_Nodes_Error.show_error_message()

                pygame.display.update()

        except UnboundLocalError:
            pass

    def __del__(self):
        pass
```

Graph_Structure.py

This is the module for the Graph object. It is responsible for the 'inner loop' of the Pathfinding_State and is entered when one of the buttons in Pathfinding_State is pressed. It contains three properties which automatically update to track the last path's statistics/details. All these properties contain 'getter' methods, removing the need for the object to be Public in the pursuit of good modular coding practice.

The make_grid method dynamically generates objects based on the dimensions attribute of the graph. This is done using list comprehension, which is a compact and consistent method of operating on and creating lists. The Graph object, however, mainly operates through the pathfinding method, which intakes all the arguments required for the manipulation of the graph that is required at that point in time – mainly through the 'mode' argument, which specifies which type of node is being currently placed onto the graph. There are methods for clearing the barriers and all nodes on the graph, allowing the buttons within the Pathfinding_State object to bypass the need of entering the 'inner loop'.

```
# Import the required modules
import pygame, sys

# Import the required classes from other files
from Constants import *
from Data_Structure_Objects import *
from Pathfinding_Objects import *
from User_Interface_Objects import Error_Message,
Output_Last_Path_Statistics


class Graph:
    """
    This is the Graph data structure object
    """

    def __init__(self, dimensions):
        # Initialize the dimensions and size properties
        self.dimensions = dimensions
        self.size = WINDOW_HEIGHT
        self.node_size = self.size // self.dimensions

        # Initialize the graph background as a surface of size (size,
size)
        self.graph_background = pygame.Surface((size, size))
        # Initialize the start and end points as None
        self.start = None
        self.end = None
        # Initialize a Dijkstra_Pathfinding object and an
AStar_Pathfinding object
        self.Dijkstra = Dijkstra_Pathfinding(self.dimensions)
```

Graph – Data Structure

```
        self.Astar = AStar_Pathfinding(self.dimensions)
        # Initialize the grid property by calling the make_grid method
        self.grid = self.make_grid()

    # Define a property to return the total path length
    @property
    def path_length(self):
        return self.Dijkstra.path_length + self.Astar.path_length

    # Define a property to return the total number of nodes accessed
    @property
    def nodes_accessed(self):
        return self.Dijkstra.nodes_accessed + self.Astar.nodes_accessed

    # Define a property to return the total elapsed time
    @property
    def elapsed_time(self):
        return self.Dijkstra.elapsed_time + self.Astar.elapsed_time

    # Define methods to return the path length, number of nodes accessed,
and elapsed time
    def get_path_length(self):
        return self.path_length

    def get_nodes_accessed(self):
        return self.nodes_accessed

    def get_elapsed_time(self):
        return self.elapsed_time

    # Define a method to create the grid of nodes
    def make_grid(self):


        # Create a 2D list of nodes with each node having a size of
node_size x node_size
        return [
            [Node(i, j, self.node_size, self.dimensions) for j in
range(self.dimensions)]
            for i in range(self.dimensions)
        ]

    def draw_grid(self):

        # Draw horizontal lines on the screen
        for i in range(self.dimensions):
```

Loosely coupled modules (subroutines) – module code interacts with other parts of the program through its interface

Dynamic generation of objects based on complex user-defined use of OOP model[1]

List Operations

Multi-dimensional Arrays

Annotation used effectively where required.

```
            pygame.draw.line(WINDOW, GREY, (0, i * self.node_size),
(self.size, i * self.node_size))
            # Draw vertical lines on the screen
            for j in range(self.dimensions):
                pygame.draw.line(WINDOW, GREY, (j * self.node_size, 0), (j
* self.node_size, self.size))

    # Define a method to draw the graph
    def draw(self):
        # Fill the background with white color
        self.graph_background.fill(WHITE)
        # Draw each node in the grid
        for row in self.grid:
            for node in row:
                node.draw(WINDOW)
        # Draw the grid lines
        self.draw_grid()
        # Update the display
        pygame.display.update()

    # Define a method to get the row and column of the node that was
clicked
    def get_clicked_pos(self, pos):

        # Get the x and y coordinates of the mouse click
        y, x = pos
        # Calculate the row and column of the node that was clicked
        row = y // self.node_size
        col = x // self.node_size

        return row, col

    def running_algorithm(self, current_algorithm, current_heuristic):
        """
        Runs the selected pathfinding algorithm (Dijkstra or A*) and shows
the pathfinding statistics window.
        """
        for row in self.grid:
            for node in row:
                node.update_neighbors(self.grid)

        if current_algorithm == True:
            # If A* is selected, reset path length, nodes accessed and
elapsed time to 0
            self.Astar.path_length = 0
            self.Astar.nodes_accessed = 0
            self.Astar.elapsed_time = 0
```

```
            # Run Dijkstra's algorithm, if it returns False show a
Path_Error error message
            if (
                self.Dijkstra.algorithm(
                    lambda: self.draw(), self.grid, self.start, self.end
                )
                == False
            ):
                Path_Error = Error_Message("Path_Error")
                Path_Error.show_error_message()
                return None
            else:
                # Show the pathfinding statistics window
                Pathfinding_Statistics = Output_Last_Path_Statistics(
                    self.get_elapsed_time(),
                    self.get_nodes_accessed(),
                    self.get_path_length(),
                )
                return None

        else:
            # If Dijkstra's algorithm is selected, reset path length,
nodes accessed and elapsed time to 0
            self.Dijkstra.path_length = 0
            self.Dijkstra.nodes_accessed = 0
            self.Dijkstra.elapsed_time = 0
            # Run A* algorithm, if it returns False show a Path_Error
error message
            if (
                self.Astar.algorithm(
                    lambda: self.draw(),
                    self.grid,
                    self.start,
                    self.end,
                    current_heuristic,
                )
                == False
            ):
                Path_Error = Error_Message("Path_Error")
                Path_Error.show_error_message()
                return None
            else:
                # Show the pathfinding statistics window
                Pathfinding_Statistics = Output_Last_Path_Statistics(
                    self.get_elapsed_time(),
                    self.get_nodes_accessed(),
                    self.get_path_length(),
                )
```

Putting subroutines into separate files and then importing the module and calling the functions

```
                return None

    def clear_graph(self):
        """
        Clears the entire graph and resets the start and end points.
        """
        Error_Count = 0
        for row in self.grid:
            for node in row:
                if node.state.get_state() == None:
                    Error_Count += 1

        if Error_Count == self.dimensions**2:
            # If there are no nodes on the grid, show a Clear_Graph_Error
error message
            Clear_Graph_Error = Error_Message("clear_graph_error")
            Clear_Graph_Error.show_error_message()
            Error_Count = 0
        else:
            # Reset all nodes to their initial state, set start and end
points to None, and draw the grid
            for row in self.grid:
                for node in row:
                    node.reset()

            self.start = None
            self.end = None

            self.draw()

    def clear_barriers(self):
        """
        Clears all barrier nodes from the grid.
        """
        Error = True
        # Checks if any node is a barrier node.
        for row in self.grid:
            for node in row:
                if node.state.is_barrier():
                    Error = False

        # If no node is a barrier node, shows an error message. Otherwise,
clears all barrier nodes.
        if Error:
            Clear_Barrier_Error = Error_Message("clear_barriers_error")
            Clear_Barrier_Error.show_error_message()
        else:
            for row in self.grid:
```

```
                for node in row:
                    if node.state.is_barrier():
                        node.reset()
            self.draw()

    def pathfinding(self, mode, current_pathfinding, current_heuristic):
        """
        Conducts pathfinding according to the given mode and pathfinding
algorithm.

        :param mode: the current mode (Start, End, or Barrier)
        :param current_pathfinding: the current pathfinding algorithm
(Dijkstra or A*)
        :param current_heuristic: the current heuristic function
(Manhattan or Euclidean)
        """
        while True:
            self.draw()
            Error = False
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()

                for row in self.grid:
                    for node in row:
                        if node.state.is_path():
                            Error = True

                # If any node is a path node, shows an error message and
returns None
                if Error:
                    Reset_Graph_Error = Error_Message("reset_graph_error")
                    Reset_Graph_Error.show_error_message()
                    return None
                else:
                    if pygame.mouse.get_pressed()[0]:  # LEFT
                        pos = pygame.mouse.get_pos()
                        if pos[0] <= self.size:
                            row, col = self.get_clicked_pos(pos)

                            node = self.grid[row][col]

                            if mode == "Start":

                                if self.start == None:
                                    if node.state.is_end() == True:
                                        self.end = None
                                    self.start = node
```

Appropriate Indentation

```
                            self.start.make_start()
                        else:
                            if node.state.is_end() == True:
                                self.end = None
                            self.start.reset()
                            self.start = node
                            self.start.make_start()
                    elif mode == "End":
                        if self.end == None:
                            if node.state.is_start() == True:
                                self.start = None
                            self.end = node
                            self.end.make_end()
                        else:
                            if node.state.is_start() == True:
                                self.start = None
                            self.end.reset()
                            self.end = node
                            self.end.make_end()

                    elif mode == "Barrier":
                        if node.state.is_start() == True:
                            self.start = None
                        elif node.state.is_end() == True:
                            self.end = None
                        node.make_barrier()

                else:

                    return pos

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RETURN:
                    if self.start and self.end:
                        self.running_algorithm(
                            current_pathfinding, current_heuristic
                        )
                    else:
                        Insufficient_Nodes_Error = Error_Message(
                            "insufficient_nodes_error"
                        )
                        Insufficient_Nodes_Error.show_error_messag
e()

                    # A NoneType is returned as we must go back to the
main loop
                    return None
```

Loosely coupled modules (subroutines) – module code interacts with other parts of the program through its interface only.

Pathfinding_Objects.py

The Pathfinding_Objects module holds all the classes directly responsible for the pathfinding operations and graph traversal which occurs within the Graph and thereby the Pathfinding_State. It contains the Node class, which is responsible for the functionality of each node/vertex created on the graph. This Node class is composed of the State class, which has a property that automatically updates the node's colour depending on the node's current state.

The module also contains the Dijkstra_Pathfinding and AStar_Pathfinding classes. The A* object inherits the methods and attributes excluding the dimensions attribute and algorithm method. The dimensions attribute must be overridden as the Dijkstra object itself takes this as a parameter and is considered independent of the A* object in this instance. This requires the dimensions to also be passed into the A* object. The algorithm method, however, must be overridden as it incorporates the heuristic methods.

```
# import required modules
from Constants import *
from Data_Structure_Objects import *
import time


class State:
    """
    A class to represent the state of a node.
    """

    def __init__(self):
        self.state = None

    @property
    def colour(self):
        return STATES.get(self.state)

    def get_colour(self):
        return self.colour

    def get_state(self):
        return self.state

    def set_colour(self):
        self.colour = STATES.get(self.state)

    def set_state(self, newState):
        self.state = newState

    def is_closed(self):
        return self.state == "Closed"
```

Consistent Style Throughout

```
    def is_open(self):
        return self.state == "Open"

    def is_barrier(self):
        return self.state == "Barrier"

    def is_start(self):
        return self.state == "Start"

    def is_end(self):
        return self.state == "End"

    def is_path(self):
        return self.state == "Path"


class Node:
    """
    A class to represent a node in the grid.
    """

    def __init__(self, row, col, width, total_dimensions):
        self.row = row
        self.col = col
        self.x = row * width
        self.y = col * width
        self.state = State()
        self.neighbours = []
        self.width = width
        self.total_dimensions = total_dimensions

    def get_pos(self):
        return self.row, self.col

    def reset(self):
        self.state.set_state(None)

    def make_start(self):
        self.state.set_state("Start")

    def make_closed(self):
        self.state.set_state("Closed")

    def make_open(self):
        self.state.set_state("Open")

    def make_barrier(self):
        self.state.set_state("Barrier")
```

Complex user-defined use of object orientated programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces

Single-dimensional arrays

```
    def make_end(self):
        self.state.set_state("End")

    def make_path(self):
        self.state.set_state("Path")

    def draw(self, win):
        """
        Draw the node onto the window.
        """
        pygame.draw.rect(
            win, self.state.get_colour(), (self.x, self.y, self.width,
self.width)
        )

    def update_neighbours(self, grid):
        """
        Update the neighbours of the node.
        """
        self.neighbours = []
        if (
            self.row < self.total_dimensions - 1
            and not grid[self.row + 1][self.col].state.is_barrier()
        ):  # Update the bottom side neighbour
            self.neighbours.append(grid[self.row + 1][self.col])

        if (
            self.row > 0 and not grid[self.row -
1][self.col].state.is_barrier()
        ):  # Update the top side neighbour
            self.neighbours.append(grid[self.row - 1][self.col])

        if (
            self.col < self.total_dimensions - 1
            and not grid[self.row][self.col + 1].state.is_barrier()
        ):  # Update the right side neighbour
            self.neighbours.append(grid[self.row][self.col + 1])

        if (
            self.col > 0 and not grid[self.row][self.col -
1].state.is_barrier()
        ):  # Update the left side neighbour
            self.neighbours.append(grid[self.row][self.col - 1])

    def __lt__(self, other):
        return False
```

Single-dimensional arrays

```
class Dijkstra_Pathfinding:
    """
    This class implements Dijkstra's pathfinding algorithm
    """

    # Initialize the class with a priority queue, dictionaries for g_score
and f_score, and other variables
    def __init__(self, dimensions):
        self.open_set = Priority_Queue()  # priority queue to hold node
        self.came_from = {}  # dictionary to hold parent nodes
        self.g_score = {}
        # dictionary to hold the cost of the path from the start node to a
node
        self.f_score = {}
        # dictionary to hold the sum of g_score and h_score (heuristic)
for a node
        self.open_set_hash = Hash_Table(
            dimensions
        )  # hash table to keep track of which nodes are in the open set
        self.path_length = 0
        self.nodes_accessed = 0
        self.elapsed_time = 0

    # Reconstruct the path from the end node to the start node using
parent nodes and draw the path
    def reconstruct_path(self, current, draw):
        self.path_length = 0
        while current in self.came_from:
            current = self.came_from[current]
            current.make_path()
            self.path_length += 1
            draw()

    # Dijkstra's pathfinding algorithm
    def algorithm(self, draw, grid, start, end):
        initial_time = time.time()  # record start time
        self.open_set.push(start, 0)
        self.open_set_hash.insert(start)

        self.g_score = {node: float("inf") for row in grid for node in
row}
        self.g_score[start] = 0
        self.f_score = {node: float("inf") for row in grid for node in
row}
        self.f_score[start] = 0

        # Loop until the open set is empty
        while not self.open_set.is_empty():
```

Stack/Queue Operations

Dictionaries

Hashing

Graph/Tree Traversal

```
            current = (
                self.open_set.pop()
            )  # get node with lowest f_score from the open set
            self.nodes_accessed += 1
            self.open_set_hash.remove(
                current
            )

            if (
                current == end
            ):  # if the current node is the end node, reconstruct the
path and return True
                self.reconstruct_path(end, draw)
                end.make_end()
                return True

            for neighbour in current.neighbours:

                temp_g_score = (
                    self.g_score[current] + 1
                )


                if (
                    temp_g_score < self.g_score[neighbour]
                ):
                    self.came_from[
                        neighbour
                    ] = current  # update the parent node of the neighbour
node
                    self.g_score[
                        neighbour
                    ] = temp_g_score  # update the g_score of the
neighbour node
                    self.f_score[
                        neighbour
                    ] = temp_g_score  # update the f_score of the
neighbour node
                    if neighbour not in [
                        element
                        for sublist in self.open_set_hash.table
                        for element in sublist
                    ]:
                        # add the neighbour node to the open set with
priority f_score and add it to the hash table
                        self.open_set.push(neighbour,
self.f_score[neighbour])
```

```
                    self.open_set_hash.insert(neighbour)
                    neighbour.make_open()

            draw()

            if current != start:
                current.make_closed()
            final_time = time.time()
            self.elapsed_time = final_time - initial_time

        # if no path is found, False is returned
        return False


class AStar_Pathfinding(Dijkstra_Pathfinding):
    """
    This class implements A* pathfinding algorithm
    """


    def __init__(self, dimensions):

        super().__init__(dimensions=dimensions)


    @staticmethod
    def calculate_manhattan_distance(node1_position, node2_po

        x1, y1 = node1_position
        x2, y2 = node2_position

        return abs(x1 - x2) + abs(y1 - y2)


    @staticmethod
    def calculate_euclidean_distance(node1_position, node2_po

        x1, y1 = node1_position
        x2, y2 = node2_position

        return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

    # Define a method that implements the A* algorithm.
    def algorithm(self, draw, grid, start, end, current_heuristic)

        # Record the time at the start of the algorithm.
        initial_time = time.time()
```

Complex user-defined use of object orientated programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces

Cohesive modules (subroutines) – module code does just one thing.

Simple user defined algorithms (eg a range of mathematical/statistical calculations)

Modules (subroutines) with appropriate interfaces.

Modules(collections of subroutines) – subroutines with common purpose grouped.

Self-documenting code

Graph/Tree Traversal

Complex user-defined use of object orientated programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces

```
        # Add the start node to the open set and mark it as visited.
        self.open_set.push(start, 0)
        self.open_set_hash.insert(start)

        # Set the g score for each node to infinity.
        self.g_score = {node: float("inf") for row in grid for node in
row}

        # Set the g score for the start node to 0.
        self.g_score[start] = 0

        # Set the f score for each node to infinity.
        self.f_score = {node: float("inf") for row in grid for node in
row}

        # Set the f score for the start node based on the chosen
heuristic.
        if current_heuristic:
            self.f_score[start] = self.calculate_manhattan_distance(
                start.get_pos(), end.get_pos()
            )
        else:
            self.f_score[start] = self.calculate_euclidean_distance(
                start.get_pos(), end.get_pos()
            )

        # Loop until the open set is empty.
        while not self.open_set.is_empty():

            # Check for quit event.
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()

            # Remove the node with the lowest f score from the open set.
            current = self.open_set.pop()

            # Increment the nodes_accessed counter.
            self.nodes_accessed += 1

            # Remove the current node from the hash table.
            self.open_set_hash.remove(current)

            # Check if the current node is the end node.
            if current == end:
                # Reconstruct the path from the start node to the end
node.
                self.reconstruct_path(end, draw)
```

Dictionaries

```
                # Mark the end node as the end node.
                end.make_end()

                # Return True to indicate that a path was found.
                return True

            # Loop through the neighbours of the current node.
            for neighbour in current.neighbours:

                # Calculate the tentative g score for the neighbour.
                temp_g_score = self.g_score[current] + 1


                # Check if the tentative g score is lower than the current
g score for the neighbour.
                if temp_g_score < self.g_score[neighbour]:
                    self.came_from[neighbour] = current
                    self.g_score[neighbour] = temp_g_score
                    if current_heuristic:
                        self.f_score[neighbour] = temp_g_score +
self.calculate_manhattan_distance(
                            neighbour.get_pos(), end.get_pos()
                        )
                    else:
                        self.f_score[neighbour] = temp_g_score +
self.calculate_euclidean_distance(
                            neighbour.get_pos(), end.get_pos()
                        )

                    if neighbour not in [
                        element
                        for sublist in self.open_set_hash.table
                        for element in sublist
                    ]:
                        self.open_set.push(neighbour,
self.f_score[neighbour])
                        self.open_set_hash.insert(neighbour)
                        neighbour.make_open()

            draw()

            if current != start:
                current.make_closed()

            # Record the time at the end of the algorithm
            final_time = time.time()
```

```
            # Set the elapsed time attribute to the amount of time the
algorithm took
            self.elapsed_time = final_time - initial_time

        # Return False if path was not found
        return False
```

Data_Structure_Objects.py

This module contains the Binary_Heap, Priority_Queue and Hash_Table objects, which are data structures integral to the shortest path algorithms' functioning. The Priority_Queue object is based on the Binary Heap data structure and as such inherits the methods from that class. It also contains a slight modification from the usual implementation as it automatically considers an item to be of higher relative priority if it is pushed into the queue prior to another with the same base priority. The Hash_Table class utilises a minimal perfect hash algorithm to ensure the least memory consumption by the queue whilst maintaining no collisions.

```
class Binary_Heap:
    """
    Binary Heap is a class to implement a binary heap data structure.
    """

    def _sift_up(self, index):
        parent_index = (index - 1) // 2
        while index > 0 and self._queue[index][0] <
self._queue[parent_index][0]:
            self._queue[index], self._queue[parent_index] = (
                self._queue[parent_index],
                self._queue[index],
            )
            index = parent_index
            parent_index = (index - 1) // 2

    def _sift_down(self, index):
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2
        smallest_child_index = index
        if (
            left_child_index < len(self._queue)
            and self._queue[left_child_index][0] <
self._queue[smallest_child_index][0]
        ):
            smallest_child_index = left_child_index
        if (
            right_child_index < len(self._queue)
            and self._queue[right_child_index][0] <
self._queue[smallest_child_index][0]
        ):
            smallest_child_index = right_child_index
        if smallest_child_index != index:
            self._queue[index], self._queue[smallest_child_index] = (
                self._queue[smallest_child_index],
                self._queue[index],
            )
```

Binary Heap – Data Structure

Graph/Tree Traversal

```
            self._sift_down(smallest_child_index)

class Priority_Queue(Binary_Heap):
    """
    Priority Queue is a class to implement a priority queue data structure
using a binary heap.
    """

    def __init__(self):
        """
        Initialize an empty queue.
        """
        self._queue = []

    def push(self, item, priority):
        """
        Pushes an item into the queue with a given priority.
        :param item: item to be added
        :param priority: priority of the item
        """
        entry = (priority, item)
        self._queue.append(entry)
        self._sift_up(len(self._queue) - 1)

    def update_priority(self, item, new_priority):
        """
        Updates the priority of an item in the queue.
        :param item: item whose priority is to be updated
        :param new_priority: new priority of the item
        """
        for i in range(len(self._queue)):
            if self._queue[i][1] == item:
                old_priority = self._queue[i][0]
                self._queue[i] = (new_priority, item)
                if new_priority < old_priority:
                    self._sift_up(i)
                else:
                    self._sift_down(i)

    def pop(self):
        """
        Pops the item with the highest priority from the queue.
        :return: item with the highest priority
        """
        if len(self._queue) == 0:
            raise ValueError("Queue is empty")
        entry = self._queue[0]
        last_entry = self._queue.pop()
```

Priority Queue – Data Structure

Stack/Queue Operations

```
        if len(self._queue) > 0:
            self._queue[0] = last_entry
            self._sift_down(0)
        return entry[1]

    def is_empty(self):
        """
        Checks if the queue is empty.
        :return: True if the queue is empty, False otherwise
        """
        if len(self._queue) == 0:
            return True
```

Hash Table – Data Structure

Hashing

```
class Hash_Table:
    """
    Hash_Table is a class to implement a hash table data structure.
    """

    def __init__(self, dimensions):
        """
        Initialize the hash table with a given number of dimensions.
        :param dimensions: number of dimensions
        """
        self.size = self.next_prime(dimensions**2)
        self.table = [[] for _ in range(self.size)]

    @staticmethod
    def next_prime(n):
        """
        Find the next prime number after a given number.
        :param n: the given number
        :return: the next prime number
        """
        not_prime = []
        isprime = []

        for i in range(n + 1, n + 200):
            not_prime.append(i)

        for j in not_prime:
            val_is_prime = True
            for x in range(2, j - 1):
                if j % x == 0:
                    val_is_prime = False
                    break
            if val_is_prime:
                isprime.append(j)
```

Simple user defined algorithms (eg a range of mathematical/statistical calculations)

```
        return min(isprime)

    def hash(self, tup):
        """
        Hash a given tuple to a corresponding index in the hash table.
        :param tup: the given tuple
        :return: the corresponding index in the hash table
        """
        a, b = tup.x, tup.y
        hash_val = int((a * self.size**0.5 + b) % self.size)
        return hash_val

    def insert(self, tup):
        """
        Insert a given tuple into the hash table.
        :param tup: the tuple to be inserted
        """
        hash_val = self.hash(tup)
        self.table[hash_val].append(tup)

    def remove(self, tup):
        """
        Remove a given tuple from the hash table.
        :param tup: the tuple to be removed
        """
        hash_val = self.hash(tup)
        if tup in self.table[hash_val]:
            self.table[hash_val].remove(tup)
```

Managed casting of types

User_Interface_Objects.py

The User_Interface_Objects module contains all of the objects which provide the infrastructure to the elements which are visually displayed on the screen – excluding the background images. This includes two Pygame implemented objects, which consist of a main UI_Button class that places a button onto the screen and allows it to perform an action when clicked, as well as a Boolean_Button, which inherits from the UI_Button class with the functionality of switching its image and Boolean condition based on an Odd/Even check system that I implemented within the update method.

The Tkinter objects consist of one user input and two information outputs.

- The Graph_Dimensions_Input object provides four button options to the user, allowing them to choose an integer which will then be responsible for the dimensions of the graph.
- Output_Last_Path_Statistics is self-explanatory in that it displays a window which outputs two integer numbers – the number of nodes accessed and length of last path, as well as a floating-point number – the time taken for the algorithm to run.
- Error_Message is a system which I created to display an error message to the user using the Tkinter module. The type of error displayed is dependent on the error_type parameter which is passed when the object is initialised.

```
# importing required modules
import tkinter as tk
from tkinter import Tk, messagebox
import pygame
from Contants import *


class UI_Button:
    def __init__(self, x, y, image, scale, screen):
        """
        Initializes a UI button object with given properties.

        :param x: The x-coordinate of the button.
        :param y: The y-coordinate of the button.
        :param image: The image of the button.
        :param scale: The scale factor of the button.
        :param screen: The screen to draw the button on.
        """

        # Get the dimensions of the image.
        self.width = image.get_width()
        self.height = image.get_height()

        # Scale the image.
        self.image = pygame.transform.scale(
            image, (int(self.width * scale), int(self.height * scale))
```

```
        )

        # Set the position of the button.
        self.rect = self.image.get_rect()
        self.rect.topleft = (x, y)

        # Set the screen to draw the button on.
        self.screen = screen

    def draw(self):
        """
        Draws the button on the screen and returns whether the button was
clicked.

        :return: A boolean indicating whether the button was clicked.
        """

        action = False
        # Get the position of the mouse.
        pos = pygame.mouse.get_pos()

        # Check if the mouse is over the button and if it is clicked.
        if self.rect.collidepoint(pos):
            if pygame.mouse.get_pressed()[0] == 1:
                action = True

        # Draw the button on the screen.
        self.screen.blit(self.image, (self.rect.x, self.rect.y))

        return action


class Boolean_Button(UI_Button):
    def __init__(self, x, y, image, scale, screen, alternate_image):
        """
        Initializes a Boolean_Button object with a given x and y position,
image, scale, screen, and alternate image.

        Args:
            x (int): x position of the button
            y (int): y position of the button
            image (pygame.Surface): default image of the button
            scale (float): scaling factor of the image
            screen (pygame.Surface): screen to draw the button on
            alternate_image (pygame.Surface): image of the button when
boolean_condition is False
        """
        self.boolean_condition = True
        self.count = 0
```

Complex user-defined use of object orientated programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces

```
        self.alternate_image = alternate_image
        self.temporary_image = image
        super().__init__(x, y, image, scale, screen)

    def update(self):
        """
        Updates the image of the button and the boolean condition based on
the count of updates.

        Returns:
            bool: True if boolean_condition is True, False otherwise
        """
        self.count += 1

        if self.count % 2 == 0:
            self.image = self.temporary_image
            self.boolean_condition = True
        else:
            self.image = self.alternate_image
            self.boolean_condition = False

        self.draw()

        return self.boolean_condition

    def get_boolean_condition(self):
        """
        Returns the boolean condition of the button.

        Returns:
            bool: True if boolean_condition is True, False otherwise
        """
        return self.boolean_condition


class Error_Message:
    def __init__(self, error_type):
        """
        Initializes an instance of the Error_Message class.

        Args:
            error_type (str): The type of error to display.
        """
        self.error_type = error_type

    def show_error_message(self):
        """
        Displays an error message based on the type of error.
        """
```

Putting subroutines into separate files and then importing the module and calling the functions

```
        if "clear" in self.error_type:
            # If error is related to clearing the graph
            if self.error_type == "clear_graph_error":
                error_message = "non-default nodes"
            elif self.error_type == "clear_barriers_error":
                error_message = "barriers"
            # Show an error message with the specific error type and
message
            tk.Tk().wm_withdraw()
            messagebox.showerror(
                self.error_type,
                f"There are no {error_message} on the graph",
            )

        # If there are insufficient nodes to run the algorithm
        elif self.error_type == "insufficient_nodes_error":
            tk.Tk().wm_withdraw()
            # Show an error message with the specific error type and
message
            messagebox.showerror(
                self.error_type,
                f"Start or End node missing, place both to run the
algorithm.",
            )

        # If there is an attempt to place more nodes without clearing the
graph
        elif self.error_type == "reset_graph_error":
            tk.Tk().wm_withdraw()
            # Show an error message with the specific error type and
message
            messagebox.showerror(
                self.error_type,
                f"Please 'Clear Graph' before continuing to place more
nodes.",
            )

        else:
            # If there is no possible path between the start and end nodes
            tk.Tk().wm_withdraw()
            # Show an error message with the specific error type and
message
            messagebox.showinfo(
                self.error_type,
                f"There is no possible path between the start and end
nodes to be found.",
            )
```

```
class Graph_Dimensions_Input:
    def __init__(self):
        """
        Initializes a Graph_Dimensions_Input Tkinter object

        """

        # Instantiating properties
        self.root = tk.Tk()
        self.root.geometry("300x200")
        self.root.title("Menu Window")
        self.root.config(bg="#325971")

        # Set label for graph dimensions
        self.label = tk.Label(
            self.root,
            text="Choose Graph Dimensions:",
            fg="#C8B782",
            bg="#325971",
            font=("Helvetica", 16),
        )
        self.label.pack(pady=10)

        # Create buttons for different dimensions
        self.row1 = tk.Frame(self.root, bg="#325971")
        self.row1.pack()
        self.row2 = tk.Frame(self.root, bg="#325971")
        self.row2.pack()

        self.button_1 = tk.Button(
            self.row1,
            text="10x10",
            command=lambda: self.close_window_and_return(10),
            height=2,
            width=10,
            bg="#C8B782",
            fg="#325971",
            font=("Helvetica", 14),
        )
        self.button_2 = tk.Button(
            self.row1,
            text="25x25",
            command=lambda: self.close_window_and_return(25),
            height=2,
            width=10,
            bg="#C8B782",
            fg="#325971",
            font=("Helvetica", 14),
```

```
        )

        self.button_3 = tk.Button(
            self.row2,
            text="50x50",
            command=lambda: self.close_window_and_return(50),
            height=2,
            width=10,
            bg="#C8B782",
            fg="#325971",
            font=("Helvetica", 14),
        )
        self.button_4 = tk.Button(
            self.row2,
            text="100x100",
            command=lambda: self.close_window_and_return(100),
            height=2,
            width=10,
            bg="#C8B782",
            fg="#325971",
            font=("Helvetica", 14),
        )

        # Pack buttons in the window
        self.button_1.pack(side=tk.LEFT, padx=10, pady=10)
        self.button_2.pack(side=tk.LEFT, padx=10, pady=10)
        self.button_3.pack(side=tk.LEFT, padx=10, pady=10)
        self.button_4.pack(side=tk.LEFT, padx=10, pady=10)

        self.return_value = None

        # Run Tkinter window
        self.root.mainloop()

    def close_window_and_return(self, return_value):
        # Close the window and return selected value
        self.return_value = return_value
        self.root.destroy()

    def get_return_value(self):
        # Returns the value selected by the user
        return self.return_value


class Output_Last_Path_Statistics:
    # Initialize the Output_Last_Path_Statistics class with three values
    def __init__(self, elapsed_time, total_nodes, path_distance):
        """
```

```
        Initializes an Output_Last_Path_Statistics object which takes in
the output info from the last path.

        Args:
            elapsed_time (float): Last path's elapsed time
            total_nodes (int): Last path's total affected nodes
            path_distance (int): Last path's length
        """
        # Create a new window
        self.root = tk.Tk()

        self.root.title("Pathfinding Statistics")

        self.root.geometry("300x215")

        self.root.config(bg="#325971")

        # Create the first label with some text, color, font, and
alignment
        self.label_1 = tk.Label(
            self.root,
            text="Pathfinding Elapsed Time:",
            fg="#C8B782",
            bg="#325971",
            font=("Helvetica", 12),
            width=20,
            anchor="w",
        )
        self.label_1.pack(pady=5, padx=5)

        # Create the first value label with the first value passed to the
class, some text, color, font, and alignment
        self.elapsed_time_label = tk.Label(
            self.root,
            text=f"{elapsed_time:.3f}",
            fg="#C8B782",
            bg="#325971",
            font=("Helvetica", 12),
            width=20,
            anchor="w",
        )
        self.elapsed_time_label.pack(pady=5, padx=5)


        self.label_2 = tk.Label(
            self.root,
            text="Total Nodes Affected:",
            fg="#C8B782",
```

Appropriate choice of simple data types

```
        bg="#325971",
        font=("Helvetica", 12),
        width=20,
        anchor="w",
    )
    self.label_2.pack(pady=5, padx=5)


    self.total_nodes_label = tk.Label(
        self.root,
        text=f"{total_nodes}",
        fg="#C8B782",
        bg="#325971",
        font=("Helvetica", 12),
        width=20,
        anchor="w",
    )
    self.total_nodes_label.pack(pady=5, padx=5)


    self.label_3 = tk.Label(
        self.root,
        text="Shortest Path Length:",
        fg="#C8B782",
        bg="#325971",
        font=("Helvetica", 12),
        width=20,
        anchor="w",
    )
    self.label_3.pack(pady=5, padx=5)


    self.path_distance_label = tk.Label(
        self.root,
        text=f"{path_distance}",
        fg="#C8B782",
        bg="#325971",
        font=("Helvetica", 12),
        width=20,
        anchor="w",
    )
    self.path_distance_label.pack(pady=5, padx=5)


    self.root.mainloop()
```

Implementation Packaging

Compiling Files to Executable

Compilation Process

To package multiple Python files into one executable file using PyInstaller, I first installed the PyInstaller package using pip in the command prompt by typing "pip install pyinstaller". Next, I navigated to the directory containing the Python files I wanted to package using the "cd" command in the command prompt. Once I was in the correct directory, I used the PyInstaller command to create the executable file by typing "pyinstaller --onefile Pathfinding-NEA.py" in the command prompt. PyInstaller then created a new directory called "dist" in the same directory as the Python files, where the executable file was located.

Benefits of Compiled Software

Firstly, an executable file provides portability, making it easy to share the program with others. With an executable file, you can distribute your program to others without requiring them to have Python or any additional packages installed on their computer. This is especially beneficial if the target audience is not familiar with Python or if you want to ensure that your program will run correctly on different systems.

Secondly, an executable file can help protect your code from being easily accessible or modified by others. The code is compiled and packaged into a single file, making it difficult for others to view or modify the code.

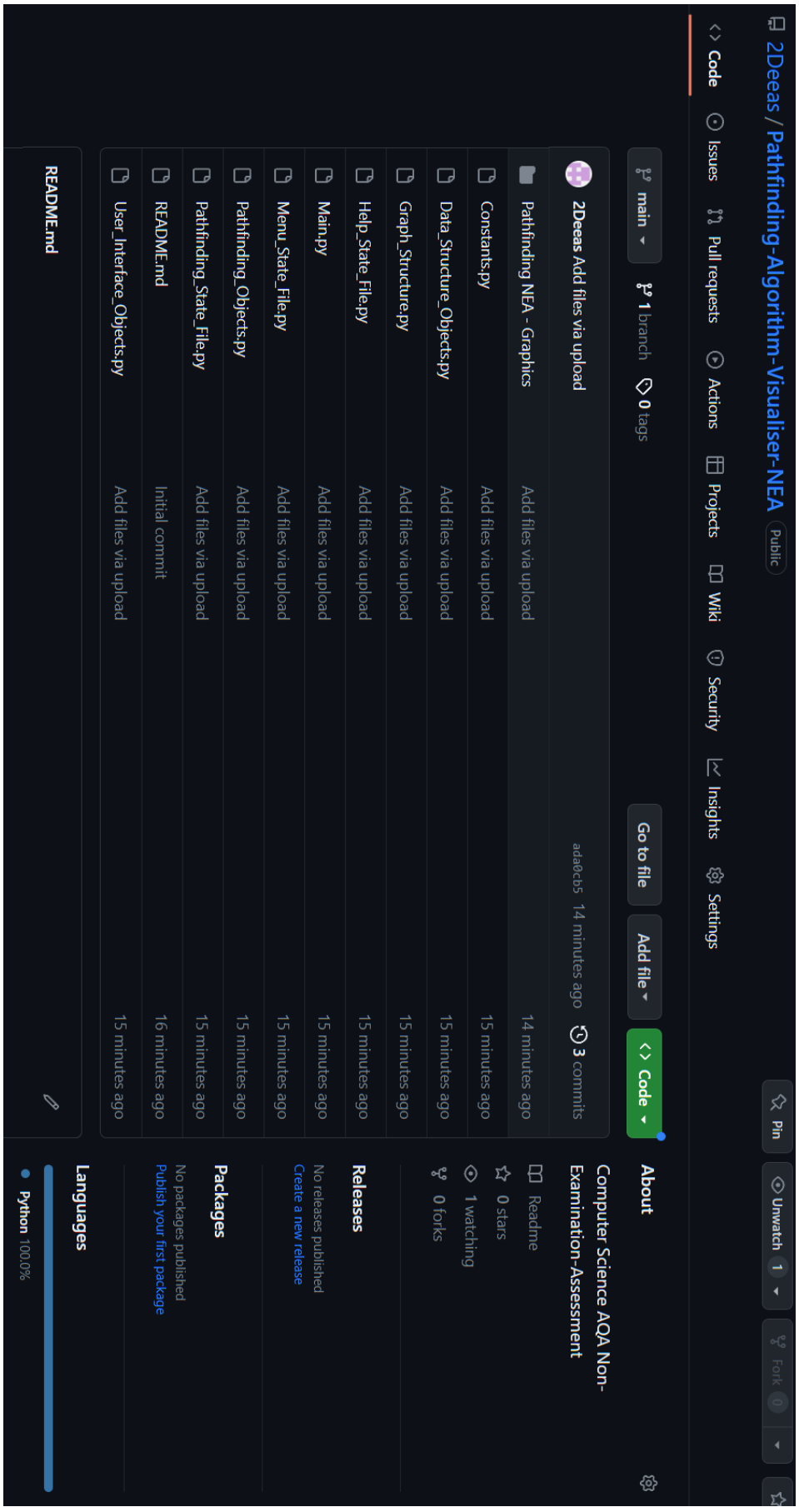
Lastly, an executable file can also improve the performance of the program. By packaging all the necessary code into a single file, the program can run more efficiently and quickly. This can be especially important for larger programs or those that require complex computations.

GitHub Representation

GitHub Repository

I have already created a repository for the code, which will allow me to distribute this project more easily and provide all the benefits mentioned in the Analysis. The following image is a picture of the repository. The reason I am not linking it is for the purpose of the NEA's security, as I would prefer not to have the project leaked online prior to submitting it to the exam board. However, as soon as the project is submitted, I will be making the

repository public.



Testing

System Testing

This part of the document describes the comprehensive testing carried out on the final version of the software. The testing was performed on all modules and was aimed at verifying that all the requirements have been met. As the program doesn't have a lot of input data, the testing mainly focused on checking that each feature functions as intended. The tests are well-structured and presented in a detailed testing table. Additional information has been provided under each test case that failed or needed further explanation. The evidence of the testing will be provided in the form of a video, which can be found in the references section, and specific timestamps of the tests will be given whenever necessary.

This is the link to the testing evidence video: https://youtu.be/H8rN7bNoaJU - though all evidence within the table is hyperlinked to the timestamp within.

Test Cases

*For non-dimension-size related testing, I will be using a default dimension size of 25 for consistency.

Test no.	Objective no.	Test Type	Purpose of test*	Test conditions / Input Data	Expected output	Evidence	Pass / Fail
1	Set 1: 1	Accepted	Test that main script functions correctly.	Launch the software.	The software GUI must appear on the screen set to the Main Menu state.	0:00	Pass
2	Set 1: 1	Accepted	Test whether the exits from the loops function correctly no matter what state the software is in.	Attempt to exit the software from the Menu state.	The software must close, and the window must be destroyed.	0:03	Pass
3	Set 2: 1	Accepted	Test whether the exits from the loops	Navigate to the Pathfinding state and	• The Pathfinding button must function and take	0:05	Pass

			function correctly no matter what state the software is in.	attempt to exit the software.	the user to the Pathfinding state. • The software must close, and the window must be destroyed.		
4	Set 1: 2	Accepted	Test whether the exits from the loops function correctly no matter what state the software is in.	Navigate to the Help state and attempt to exit the software.	• The Help state button must function and take the user to the Help state. • The software must close, and the window must be destroyed.	0:07	Pass
5	Set 2: 1	Accepted	Test to see whether this causes an error in the main script - if the window closing causes an unexpected error.	After pressing the Pathfinding state button, attempt to close the graph dimensions input window. Finally, press the Pathfinding button again.	The user must stay in the Menu state and have the option to press the Pathfinding button again to choose a new dimensions size.	0:09	Pass
6	Set 2: 6	Accepted	Test to see whether all the methods and class related to the start button functioning works.	Within the Pathfinding state, click on start node button and set a node on the graph to a start node state.	The start node button must function as programmed – the selected node on the graph must change colour to indicate it being the start node.	0:18	Pass
7	Set 2: 7	Accepted	Test to see whether all the methods and class related to the end button functioning works.	Within the Pathfinding state, click on the end node button and set a node on the graph to an end node state.	The start node button must function as programmed – the selected node on the graph must change colour to indicate it being the end node.	0:21	Pass

8	Set 2: 8	Accepted	Test to see whether all the methods and class related to the barrier button functioning works.	Within the Pathfinding state, click on the barrier node button and set a node on the graph to a barrier node state.	The barrier node button must function as programmed – the selected node on the graph must change colour to indicate it being the barrier node.	0:24	Pass
9	Set 2: 9	Accepted	Test to see whether the Barrier Clear button works as intended – to see if any of the functions in the process contain an error.	Within the Pathfinding state, set a barrier node, followed by clicking the clear barriers button.	The barrier node that had been set must change its colour back to white to indicate that it has been reset to its None state.	0:28	Pass
10	Set 2: 10	Accepted	Test to see whether the Clear All button works as intended – to see if any of the functions in the process contain an error.	Within the Pathfinding state, set a start, end, and barrier node, followed by pressing the clear all button.	The nodes that have been manipulated must change their colour back to white to indicate that they have been reset to their None state.	0:31	Pass
11	Set 2: 6	Accepted	Test to make sure that the methods relating to placing a start node on the graph function as expected and have no error within.	Within the Pathfinding state, set a start node, followed by clicking on a different node on the graph.	The original node to have been set to the start state must change its colour back to white to indicate it being reset, meanwhile the new node to have been chosen must change its colour to the start node colour to indicate it having changed state.	0:39	Pass

12	Set 2: 7	Accepted	Test to make sure that the methods relating to placing a end node on the graph function as expected and have no error within.	Within the Pathfinding state, set an end node, followed by clicking on a different node on the graph.	The original node to have been set to the end state must change its colour back to white to indicate it being reset, meanwhile the new node to have been chosen must change its colour to the end node colour to indicate it having changed state.	0:43	Pass
13	Set 2: 8	Accepted	Test to make sure that the methods relating to placing a barrier node on the graph function as expected and have no error within.	Within the Pathfinding state, set a barrier node, followed by clicking on a different node on the graph.	Both nodes to have been clicked by the user should have changed their colour to black, indicating that they have changed	0:48	Pass
14	Set 2: 4	Accepted	Test to ensure that the Boolean_Button functions as expected and still works within the Pathfinding state class correctly.	Within the Pathfinding state, click on the Pathfinding Button.	The Pathfinding Button must change its visual appearance from the 'D' image to the 'A*' image, which indicates that the Pathfinding type has changed.	0:53	Pass
15	Set 2: 12	Accepted	Test to ensure that the Boolean_Button functions as expected and still works within the Pathfinding state class correctly.	Within the Pathfinding state, click on the Heuristic Button.	The Heuristic Button must change its visual appearance from the 'M' image to the 'E' image, which indicates that the Heuristic type has changed.	0:55	Pass
16	Set 2: • 2	Accepted	Test to ensure that the pathfinding	Within the Pathfinding state, place the sufficient	• I expect the algorithm to run and display the	0:58	Pass

	• 3		algorithm itself works as expected within the Graph and Pathfinding state classes.	nodes for a simulation to occur, followed by pressing the 'Enter' key whilst in Dijkstra pathfinding mode.	correct shape of the visualisation on the graph. This is an outwards 'wave' of the nodes shifting state. • Last path's statistics must be displayed in a new window.		
17	Set 2: • 2 • 3 • 4	Accepted	Test to ensure that the pathfinding algorithm itself works as expected within the Graph and Pathfinding state classes – as well as the input from the Manhattan distance heuristic. Test that the path statistics window functions as correct.	Within the Pathfinding state, place the sufficient nodes for a simulation to occur, followed by clicking the Pathfinding button to switch to A* pathfinding - whilst in Manhattan heuristic mode - and then pressing the 'Enter' key.	• I expect the algorithm to run and display the correct shape of the visualisation on the graph. This will be in a directed 'wave' that forms toward the end node. • Last path's statistics must be displayed in a new window.	1:13	Pass
18	Set 2: • 2 • 3 • 4 • 12	Accepted	Test to ensure that the pathfinding algorithm itself works as expected within the Graph and Pathfinding state classes – as well as	Within the Pathfinding state, place the sufficient nodes for a simulation to occur, followed by clicking the Pathfinding button to switch to A* pathfinding and the Heuristic button	• I expect the algorithm to run and display the correct shape of the visualisation on the graph. This will be in directed 'lines' that	1:23	Pass

			the input from the Euclidean distance heuristic. Test that the path statistics window functions as correct.	to switch to Euclidean pathfinding and then pressing the 'Enter' key.	form towards the end node. • Last path's statistics must be displayed in a new window.		
19	Set 2: 9	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, whilst there are no nodes on the graph, click on the 'Clear Barriers' button.	A 'No current barriers on the graph' error must appear to the user in a new window.	1:34	Pass
20	Set 2: 9	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, set a start and end node, followed by pressing the 'Clear Barriers' button.	A 'No current barriers on the graph' error must appear to the user in a new window.	1:38	Pass
21	Set 2: 10	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, whilst there are no nodes on the graph, click on the 'Clear All' button.	A 'No non-empty nodes on the graph' error must appear to the user in a new window.	1:43	Pass
22	Set 2: 11	Erroneous	Testing to ensure that there are no methods	Within the Pathfinding state, whilst there are no	There should not be any changes to the screen.	1:45	Pass

			that run on a false input.	nodes on the graph, press on the 'Enter' key.	Nothing should occur from the input.		
23	Set 2: 11	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, set a start node, followed by pressing the 'Enter' key.	An 'insufficient nodes' error must appear to the user in a new window.	1:50	Pass
24	Set 2: 11	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, set an end node, followed by pressing the 'Enter' key.	An 'insufficient nodes' error must appear to the user in a new window.	1:55	Pass
25	Set 2: 11	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, set a barrier node, followed by pressing the 'Enter' key.	An 'insufficient nodes' error must appear to the user in a new window.	1:59	Pass
26	Set 2: 11	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, place the sufficient nodes for a simulation to occur but create barriers to make a path being found impossible. Finally, press the 'Enter' key.	A 'No path' error must appear to the user in a new window.	2:03	Pass

27	Set 2: 11	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, place the sufficient nodes for a simulation to occur, and press the 'Enter' key. After the path is found, press the start node button.	A 'graph must first be cleared' error must appear to the user in a new window.	2:12	Pass
28	Set 2: 11	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, place the sufficient nodes for a simulation to occur, and press the 'Enter' key. After the path is found, press the end node button.	A 'graph must first be cleared' error must appear to the user in a new window.	2:14	Pass
29	Set 2: 11	Erroneous	Testing to ensure that the Error_message and validation checking system I have in place functions as expected.	Within the Pathfinding state, place the sufficient nodes for a simulation to occur, and press the 'Enter' key. After the path is found, press the barrier node button.	A 'graph must first be cleared' error must appear to the user in a new window.	2:16	Pass
30	Set 2: 11	Accepted	Testing to ensure that the state of the software after the path is found works as it should – meaning graph should be cleared before any more actions occur.	Within the Pathfinding state, place the sufficient nodes for a simulation to occur, and press the 'Enter' key. After the path is found, press 'Enter' again.	On the second press of the 'Enter' key, nothing should happen.	2:19	Fail

31	Set 2: 1	Accepted	Testing to ensure that the dimensions are correctly passed into the Pathfinding state class.	Enter the '10x10' dimension sized graph in the pathfinding state.	The pathfinding state is initialised with a 10 by 10 sized graph.	2:28	Pass
32	Set 2: 1	Accepted	Testing to ensure that the dimensions are correctly passed into the Pathfinding state class.	Enter the '25x25' dimension sized graph in the pathfinding state.	The pathfinding state is initialised with a 25 by 25 sized graph.	2:31	Pass
33	Set 2: 1	Accepted	Testing to ensure that the dimensions are correctly passed into the Pathfinding state class.	Enter the '50x50' dimension sized graph in the pathfinding state.	The pathfinding state is initialised with a 50 by 50 sized graph.	2:33	Pass
34	Set 2: 1	Accepted	Testing to ensure that the dimensions are correctly passed into the Pathfinding state class.	Enter the '100x100' dimension sized graph in the pathfinding state.	The pathfinding state is initialised with a 100 by 100 sized graph.	2:35	Pass
35	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '10x10' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (10, 10) then run the Dijkstra Pathfinding algorithm.	The algorithm should function as expected throughout its entire running.	2:38	Pass
36	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the	Enter the '25x25' dimension sized graph in the pathfinding state and	The algorithm should function as expected throughout its entire running.	2:43	Pass

			provided boundary data on each dimension sized graph.	place a start node at (0,0) and an end node at (25, 25) then run the Dijkstra Pathfinding algorithm.			
37	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '50x50' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (50, 50) then run the Dijkstra Pathfinding algorithm.	The algorithm should function as expected throughout its entire running.	2:50	Pass
38	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '100x100' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (100,100) then run the Dijkstra Pathfinding algorithm.	The algorithm should function as expected throughout its entire running.	2:54	Pass
39	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '10x10' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (10, 10) then run the A* Pathfinding algorithm with the Manhattan heuristic.	The algorithm should function as expected throughout its entire running.	3:01	Pass
40	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the	Enter the '25x25' dimension sized graph in the pathfinding state and	The algorithm should function as expected throughout its entire running.	3:07	Pass

			provided boundary data on each dimension sized graph.	place a start node at (0,0) and an end node at (25, 25) then run the A* Pathfinding algorithm with the Manhattan heuristic.			
41	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '50x50' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (50, 50) then run the A* Pathfinding algorithm with the Manhattan heuristic.	The algorithm should function as expected throughout its entire running.	3:14	Pass
42	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '100x100' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (100,100) then run the A* Pathfinding algorithm with the Manhattan heuristic.	The algorithm should function as expected throughout its entire running.	3:49	Pass
43	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '10x10' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (10, 10) then run the A* Pathfinding algorithm with the Euclidean heuristic.	The algorithm should function as expected throughout its entire running.	4:08	Pass

44	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '25x25' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (25, 25) then run the A* Pathfinding algorithm with the Euclidean heuristic.	The algorithm should function as expected throughout its entire running.	4:44	Pass
45	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '50x50' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (50, 50) then run the A* Pathfinding algorithm with the Euclidean heuristic.	The algorithm should function as expected throughout its entire running.	11:50	Pass
46	Set 2: 2	Boundary	Testing whether the algorithm works as intended with the provided boundary data on each dimension sized graph.	Enter the '100x100' dimension sized graph in the pathfinding state and place a start node at (0,0) and an end node at (100, 100) then run the A* Pathfinding algorithm with the Euclidean heuristic.	The algorithm should function as expected throughout its entire running.	13:47	Pass
47	Set 2: 2	Accepted	Testing to make sure that the algorithms consistently visualise the same form despite the differences in graph	Within the Pathfinding State, place a start node on (0, 0) and an end node on (10, 10), following this, run the Dijkstra pathfinding algorithm with	The algorithm visualisation on the graph should be the same on every single dimension size.	21:27	Pass

			size and visualisation attempt.	this format on every dimension size.			
48	Set 2: 2	Accepted	Testing to make sure that the algorithms consistently visualise the same form despite the differences in graph size and visualisation attempt.	Within the Pathfinding State, place a start node on (0, 0) and an end node on (10, 10), following this, run the A* pathfinding algorithm (Manhattan Heuristic) with this format on every dimension size.	The algorithm visualisation on the graph should be the same on every single dimension size.	23:43	Pass
49	Set 2: 2	Accepted	Testing to make sure that the algorithms consistently visualise the same form despite the differences in graph size and visualisation attempt.	Within the Pathfinding State, place a start node on (0, 0) and an end node on (10, 10), following this, run the A* pathfinding algorithm (Euclidean Heuristic) with this format on every dimension size.	The algorithm visualisation on the graph should be the same on every single dimension size.	25:18	Pass

Amending Failed Test Case

Failed Test Case

Test no.	Objective no.	Test Type	Purpose of test*	Test conditions / Input Data	Expected output	Evidence	Pass / Fail
30	Set 2: 11	Accepted	Testing to ensure that the state of the software after the path is found works as it should – meaning graph should be cleared before any more actions occur.	Within the Pathfinding state, place the sufficient nodes for a simulation to occur, and press the 'Enter' key. After the path is found, press 'Enter' again.	On the second press of the 'Enter' key, nothing should happen.	2:19	Fail

Failure Identification

Analysing the relevant code, we can identify that within the 'outer loop' of the pathfinding state, which occurs in the Pathfinding_State_File.py file, within the Pathfinding_State object, the conditional statement related to the initiation of the running_algorithm method only checks whether the relevant nodes for the algorithm are currently placed on the graph – meaning that there is no validation seeing if any nodes are currently in the 'Path' state. This is a check that I had coded within the 'inner loop' in the Graph object but not within the actual Pathfinding state. Therefore, the solution to this failure would be implementing a method of validating whether any nodes on the graph were in the 'Path' state, and if so, not running the section of the function which is responsible for the 'Enter' button conditional statement. This can be done using the addition of a new local variable 'Error' which will be used instantiated as False upon every iteration of the while loop in the main pathfinding loop. A nested loop which iterates through the graph to check for 'Path' state nodes will set this variable to True if they are detected, otherwise the variable will be left as False and the section of code responsible for the running_algorithm method will be left accessible.

Relevant Code Failure

Within the Pathfinding_State object:

```
    def pathfinding_loop(self):
        try:
            while True:
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        sys.exit()

                    self.draw_start()

                    self.draw_end()

                    self.draw_barrier()

                    if self.Button_Back_Object.draw():
                        del self

                    elif self.Button_Barrier_Node_Clear_Object.draw():
                        self.graph.clear_barriers()

                    elif self.Button_Clear_All_Object.draw():
                        self.graph.clear_graph()

                    elif self.Button_Pathfinding_Type_Swap.draw():
                        self.Button_Pathfinding_Type_Swap.update()

                    elif self.Button_Heuristic.draw():
                        self.Button_Heuristic.update()
```

```
if event.type == pygame.KEYDOWN:

    if (
        event.key == pygame.K_RETURN
        and self.graph.start
        and self.graph.end
    ):
        self.graph.running_algorithm(
            self.Button_Pathfinding.get_boolean_condition(),
            self.Button_Heuristic.get_boolean_condition(),
        )
    elif (
        event.key == pygame.K_RETURN
        and self.graph.start
        and not self.graph.end
    ):
        Insufficient_Nodes_Error = ErrorMessage(
            "insufficient_nodes_error"
        )
        Insufficient_Nodes_Error.show_error_message()
    elif (
        event.key == pygame.K_RETURN
        and self.graph.end
        and not self.graph.start
    ):
        Insufficient_Nodes_Error = ErrorMessage(
            "insufficient_nodes_error"
        )
        Insufficient_Nodes_Error.show_error_message()

    elif (
```

```
                        event.key == pygame.K_RETURN
                        and not self.graph.start
                        and not self.graph.end
                    ):
                        Insufficient_Nodes_Error = ErrorMessage(
                            "insufficient_nodes_error"
                        )
                        Insufficient_Nodes_Error.show_error_message()

                pygame.display.update()

        except UnboundLocalError:
            pass
```

Improved Code Implementation

The changes required to fix the test failure is highlighted using comments.

```
def pathfinding_loop(self):
        try:
            while True:

                # Improvement
                Error = False

                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        sys.exit()

                    self.draw_start()
```

```
self.draw_end()

self.draw_barrier()

if self.Button_Back_Object.draw():
    del self

elif self.Button_Barrier_Clear_Object.draw():
    self.graph.clear_barriers()

elif self.Button_Clear_All_Object.draw():
    self.graph.clear_graph()

elif self.Button_Pathfinding.draw():
    self.Button_Pathfinding.update()

elif self.Button_Heuristic.draw():
    self.Button_Heuristic.update()

# Improvement
for row in self.graph.grid:
    for node in row:
        if node.state.is_path():
            Error = True

# Improvement
if event.type == pygame.KEYDOWN and not Error:
    if (
        event.key == pygame.K_RETURN
        and self.graph.start
```

```
        and self.graph.end
    ):
        self.graph.running_algorithm(
            self.Button_Pathfinding.get_boolean_condition(),
            self.Button_Heuristic.get_boolean_condition(),
        )
    elif (
        event.key == pygame.K_RETURN
        and self.graph.start
        and not self.graph.end
    ):
        Insufficient_Nodes_Error = ErrorMessage(
            "insufficient_nodes_error"
        )
        Insufficient_Nodes_Error.show_error_message()
    elif (
        event.key == pygame.K_RETURN
        and self.graph.end
        and not self.graph.start
    ):
        Insufficient_Nodes_Error = ErrorMessage(
            "insufficient_nodes_error"
        )
        Insufficient_Nodes_Error.show_error_message()

    elif (
        event.key == pygame.K_RETURN
        and not self.graph.start
        and not self.graph.end
    ):
        Insufficient_Nodes_Error = ErrorMessage(
```

```
                        "insufficient_nodes_error"
                    )
                    Insufficient_Nodes_Error.show_error_message()

                pygame.display.update()

            except UnboundLocalError:
                pass
```

Re-Attempting Test Case

Test Case Code-Change for Evidence

This is an added code line to the pathfinding_loop method within the Pathfinding_State class where the rest of the relevant code is located. This line will be used to prove that the Enter key has in fact been pressed a second time. The video evidence will be linked in its respective column below.

```
if event.type == pygame.KEYDOWN and not Error:

        if event.key == pygame.K_RETURN:
            print('Enter Key Pressed')
```

Test Case

Test no.	Objective no.	Test Type	Purpose of test*	Test conditions / Input Data	Expected output	Evidence	Pass / Fail
30	Set 2: 11	Accepted	Testing to ensure that the state of the software after the	Within the Pathfinding state, place the sufficient nodes for a simulation to	On the second press of the 'Enter' key, nothing should happen.	Test Case Evidence	Pass

			path is found works as it should – meaning graph should be cleared before any more actions occur.	occur, and press the 'Enter' key. After the path is found, press 'Enter' again.			

User Acceptance Testing

User Navigation Survey

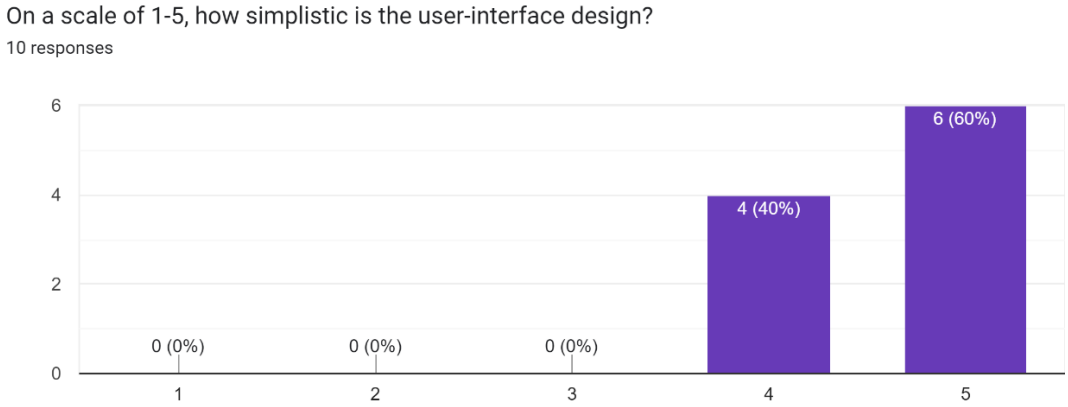
Survey Questions and Analysis

After allowing my clients – the A-Level Computer Science class, Further Mathematics Class, and respective teachers – to trial the software, I proposed this set of questions to test whether I was meeting my software objectives.

Test Cases

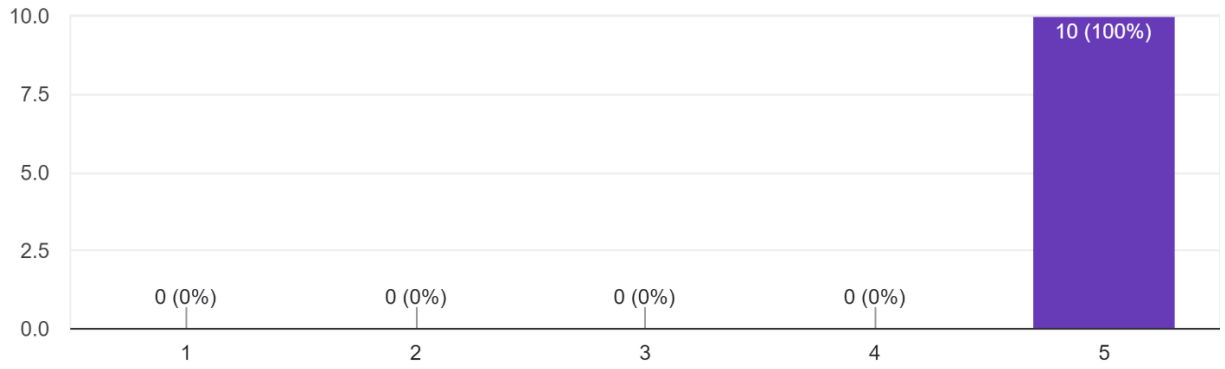
Test no.	Objective no.	Test Question
1	Set 1: 1.C	On a scale of 1-5, how simplistic is the user-interface design?
2	Set 1: 2.A	On a scale of 1-5, how clear are the instructions on how to operate the software?
3	Set 2: 1.B	On a scale of 1-5, how simplistic is the background of the Pathfinding state?
4	Set 2: 1.A	On a scale of 1-5, are the buttons clearly labelled?
5	Set 2: 4.B Set 2: 12.B	On a scale of 1-5, is it easy to identify the Pathfinding and Heuristic swap buttons?
6	Set 2: 1.C Set 2: 2.A	On a scale of 1-5, is each node-type individually identifiable?
7	Set 2: 3.B	On a scale of 1-5, are the Pathfinding statistics clearly labelled?

Test Case Analysis



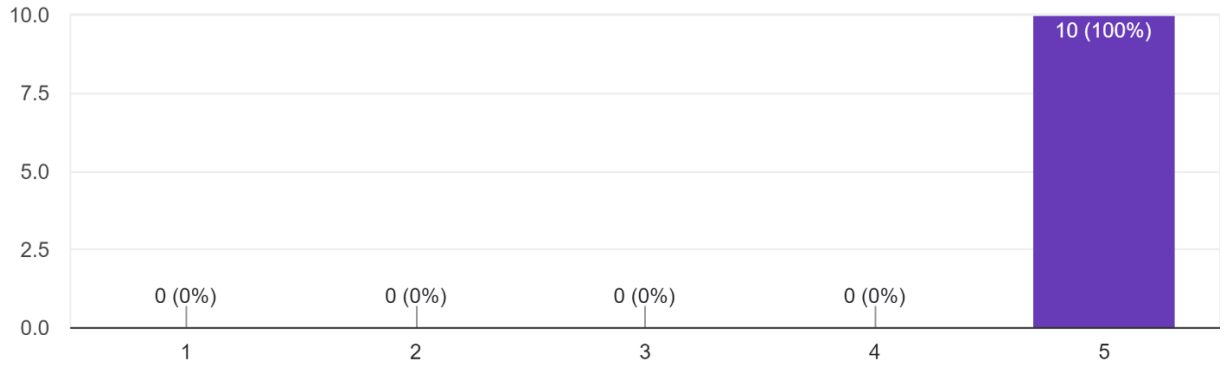
On a scale of 1-5, how clear are the instructions on how to operate the software?

10 responses



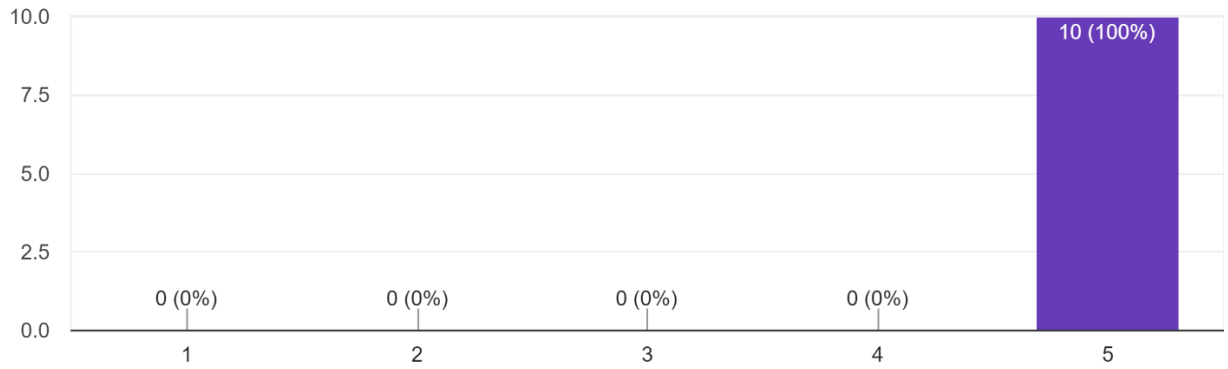
On a scale of 1-5, how simplistic is the background of the Pathfinding state?

10 responses



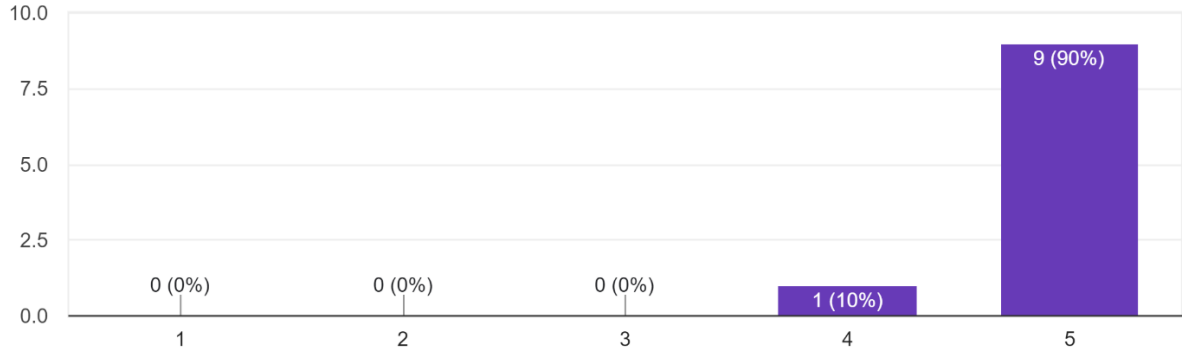
On a scale of 1-5, are the buttons clearly labelled?

10 responses



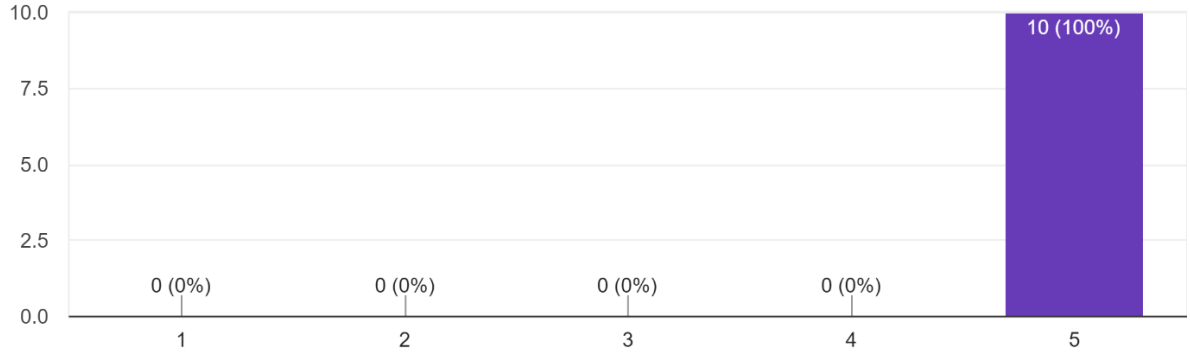
On a scale of 1-5, is it easy to identify the Pathfinding and Heuristic swap buttons?

10 responses



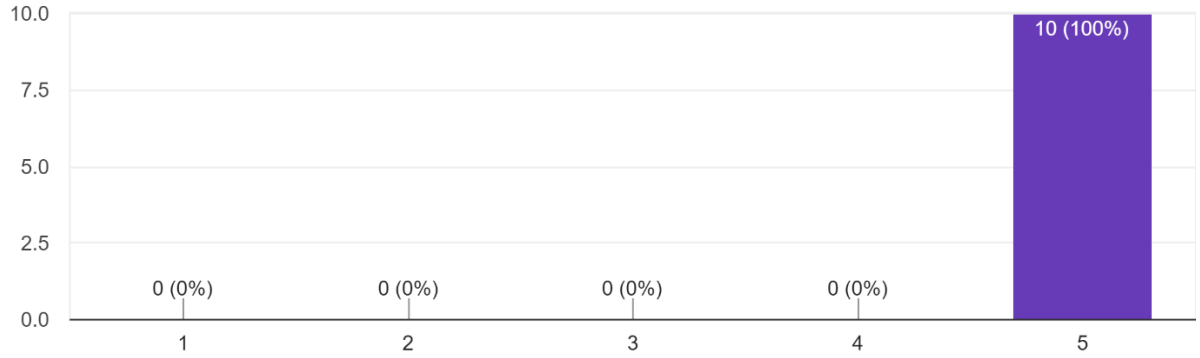
On a scale of 1-5, is each node-type individually identifiable?

10 responses



On a scale of 1-5, are the Pathfinding statistics clearly labelled?

10 responses



Evaluation

Extent of Meeting Software Objectives

Set 1: Menu and Help State Requirements

Set 1 - Software Objective 1: The Program must contain a main menu state.

The success of this software objective is evidenced by System Test cases 1 and 2. Additionally in the technical implementation I have provided the code responsible for the displaying of a main menu state. The user-interface elements for this state have also been provided in the Design section.

Sub-Objective A – This sub objective has been achieved as evidenced by the code in the Menu_State_File.py file. It is in this code that the Menu_State class exists, which is composed of a UI_Button object responsible for taking the user to the Pathfinding state. Additionally, I have provided the UI designs for this button as well as the button's code itself, which does in fact provide the ability to transition state at the click of the user's mouse.

Sub-Objective B – This sub objective has been achieved as evidenced by the code in the Menu_State_File.py file. It is in this code that the Menu_State class exists, which is composed of a UI_Button object responsible for taking the user to the Help state. Additionally, I have provided the UI designs for this button as well as the button's code itself, which does in fact provide the ability to transition state at the click of the user's mouse.

Sub-Objective C – This sub objective is evidenced to a largely positive extent by the User Acceptance Test – Test Case 1, which returned data with very positive results – 60% of the users believed that the objective was fulfilled to a perfect standard meanwhile the rest of the users responded that they believed it was fulfilled to 80%.

Sub-Objective D – This sub objective has been met as evidenced by the pass result in Test Case 2, which fully reflects the goal of the objective.

Set 1 - Software Objective 2: The Program must contain a 'help' state.

The success of this software objective is evidenced by Test Case 4. Additionally in the technical implementation I have provided the code responsible for the displaying of a Help state. The user-interface elements for this state have also been provided in the Design section.

Sub-Objective A – To evidence that annotated instructions have been provided, the interface elements section in Design has an image of the background of the Help state, which displays said instructions. Proving this may be done through Test Case 2 in the User

Acceptance Test as the users gave back a 100% success on how clear the annotated instructions were in the Help state.

Sub-Objective B – This is evidenced by the code in the Help_State_File.py file, within the Help_State class. This class is composed of the UI_Button class, which has code also documented within the Implementation section. Additionally, the visual element of the button is displayed in the Interface Elements design section.

Sub-Objective C – This is evidenced fully by UAT Test Case 4 the test completely proved this objective's completion to a 100% standard.

Set 2: Visualisation State Requirements

Set 2 - Software Objective 1: There must be a simplistic user interface.

The User Acceptance Test largely covers the basis of the evidence proving the success of this software objective. Test Cases 1, 2, 3, 4, and 5 and their success from the analysis of the feedback from the users.

Sub-Objective A – Test Case 4 explains that the users believed that the clarity of the Button completely fulfilled the requirements – the feedback being a 5/5 with a 100% success rate from all the users.

Sub-Objective B – This is completely evidenced by Test Case 3 as the users overwhelmingly agreed with the statement that the background is simplistic. This is evidenced by the 100% average response from 10 out of 10 surveyed test users.

Sub-Objective C – Once again, this is completely evidenced by Test Case 6, as the users completely agreed with the statement that the nodes are individually identifiable. The visual element of the graph is also evident within the User Interface Elements in the Design section, which displays the graph size.

Sub-Objective D – This is evidenced by System Test Cases 31, 32, 33, 34, which show how all the cases where the dimension sizes are chosen, they are passed into the Pathfinding state successfully. The code within the User_Interface_Objects.py – specifically the Graph_Dimensions_Input object – further depicts the implementation of the dimension size input.

Sub-Objective E - This is evidenced by the code in the Pathfinding_State_File.py file, within the Pathfinding_State class. This class is composed of the UI_Button class, which has code also documented within the Implementation section. Additionally, the visual element of the button is displayed in the Interface Elements design section. The self-destruct which is initiated is also within loop method of the Pathfinding_State object.

Sub-Objective F – This is completely evidenced by System Test Case 3, which successfully showed how the user can leave the software during this state.

Set 2 - Software Objective 2: The pathfinding algorithm must be visualised clearly on the graph.

The visualisation of the pathfinding algorithm clearly is evidenced in a multitude of ways through both the User Acceptance Testing and System Testing. Additionally, the Visual Elements in Design and Technical Implementation further add reinforcement behind how this software objective has been met to the full extent.

Sub-Objective A – The success of the identifiability of each node on the graph is evidenced by Test Case 6 within the User Acceptance Testing, as every member to take the survey gave the opinion that they believed it was achieved to the full extent. The colour switching is

evidenced within the implementation of the Node and State class in the Pathfinding_Objects.py file.

Sub-Objective B – The achievement of this objective is proved through a combination of the User Interface Elements section in Design, as well as the Node and State classes within the Pathfinding_Objects.py file implementation. The reconstruct_path method within the Dijkstra pathfinding class which is inherited to the A* Pathfinding class within the implementation showcases how the path is retraced. This is utilised by the running_algorithm method in the Graph object to implement it.

Set 2 - Software Objective 3: Information regarding the last performed pathfinding algorithm must be displayed.

This software objective is mainly evidenced through the Output_Last_Path_Statistic implementation code as well as User Acceptance Testing Test Case 16, 17, and 18. The aforementioned test cases display the statistics being shown on the screen clearly – further justified by the User-Acceptance Test Case 7.

Sub-Objective A – This success of this objective is completely evidenced by the initialisation method in the Dijkstra_Pathfinding class – this is also passed into the AStar_Pathfinding class through inheritance. Within this method, the statistic attributes are instantiated with a value of 0 – indicating that this is their value until Pathfinding algorithm is run.

Sub-Objective B – Test Case 7 from the User Acceptance Testing section completely covers the scope of this objective. Out of every user asked, I had a 100% response rate of the objective being completely fulfilled – as evidenced by the '5' responses on a scale of 1-5.

Sub-Objective C – The success of this implementation is evidenced by the Output_Last_Path_Statistic object code within the User_Interface_Objects.py file. This includes a line of code with an f string implementation that specifically rounds the float number to 3 decimal places. The 'time_elapsed' is also a priority class decorated attribute of the Pathfinding objects which are passed into the Graph data structure, and finally encapsulated within the Pathfinding_State class.

Sub-Objective D - The success of this implementation is evidenced by the Output_Last_Path_Statistic object code within the User_Interface_Objects.py file. This includes a line of code with an f string implementation. The 'path_length' is also a priority class decorated attribute of the Pathfinding objects which are passed into the Graph data structure, and finally encapsulated within the Pathfinding_State class.

Sub-Objective E - The success of this implementation is evidenced by the Output_Last_Path_Statistic object code within the User_Interface_Objects.py file. This includes a line of code with an f string implementation. The 'nodes_accessed' is also a priority class decorated attribute of the Pathfinding objects which are passed into the Graph data structure, and finally encapsulated within the Pathfinding_State class.

Set 2 - Software Objective 4: The user must be able to switch between Dijkstra and A* Pathfinding modes.

This software objective is mainly evidenced through the Boolean_Button, and Graph Data Structure objects in the implementation. Additionally, it is proved in the User Acceptance Testing Test Case 5, as well as the System Test Cases 14 and 17.

Sub-Objective A – This is evidenced by both the Boolean_Button implementation which the Graph data structure is composed of, as well as being completely proved by System Test Cases 14 and 17.

Sub-Objective B – This sub-objective is completely evidenced by the User-Acceptance Test Case 5, which showed that there was a 90% response rate of 5/5 on a scale of how strongly this objective was met. Meanwhile the other 10% provided a response of 4/5, which still leads to an overwhelmingly high margin by which this object has been met.

Sub-Objective C – This is evidenced by the code in the Boolean_Button object – within the 'update' method. Additionally, it is completely proved through the System Test Cases 14 and 17, which show it functioning.

Set 2 - Software Objective 5: The user must be able to switch between Manhattan and Euclidean heuristic modes.

This software objective is mainly evidenced through the Boolean_Button, and Graph Data Structure objects in the implementation. Additionally, it is proved in the User Acceptance Testing Test Case 5, as well as the System Test Cases 15 and 18.

Sub-Objective A – This is evidenced by both the Boolean_Button implementation which the Graph data structure is composed of, as well as being completely proved by System Test Cases 15 and 18.

Sub-Objective B – This sub-objective is completely evidenced by the User-Acceptance Test Case 5, which showed that there was a 90% response rate of 5/5 on a scale of how strongly this objective was met. Meanwhile the other 10% provided a response of 4/5, which still leads to an overwhelmingly high margin by which this object has been met.

Sub-Objective C – This is evidenced by the code in the Boolean_Button object – within the 'update' method. Additionally, it is completely proved through the System Test Cases 15 and 18, which show it functioning.

Set 2 - Software Objective 6: The user must be able to select a node on the graph to be the starting position.

This software objective is mainly evidenced through the State and Node objects within the Pathfinding_Objects.py file, as well as the UI_Button object. Additionally, the methods in the Pathfinding_State class show how the Start Node button implementation occurs. Throughout the Testing phase, it is proved by System Testing Test Cases 6 and 11.

Sub-Objective i – In addition to the code implementation that I explained in the overview, this objective is completely evidenced by System Testing Test Case 6. This displays that the placing of the start node system functions.

Sub-Objective ii - This objective is achieved and proved by both System Case Test Cases 6 and 11. Additionally, the code in the implementation of the Pathfinding_State object shows the creation of the button each iteration of the while loop instance – showing that it can be done at every point in the program's pathfinding state's.

Sub-Objective iii – This objective is completely evidenced by System Testing Test Case 11. It shows the entirety of the process related to this objective.

Set 2 - Software Objective 7: The user must be able to select a node on the graph to be the ending position.

This software objective is mainly evidenced through the State and Node objects within the Pathfinding_Objects.py file, as well as the UI_Button object. Additionally, the methods in the Pathfinding_State class show how the End Node button implementation occurs. Throughout the Testing phase, it is proved by System Testing Test Cases 7 and 12.

Sub-Objective i – In addition to the code implementation that I explained in the overview, this objective is completely evidenced by System Testing Test Case 7. This displays that the placing of the end node system functions.

Sub-Objective ii - This objective is achieved and proved by both System Case Test Cases 7 and 12. Additionally, the code in the implementation of the Pathfinding_State object shows the creation of the button each iteration of the while loop instance – showing that it can be done at every point in the program's pathfinding state's.

Sub-Objective iii – This objective is completely evidenced by System Testing Test Case 12. It shows the entirety of the process related to this objective.

Set 2 - Software Objective 8: The user must be able to create barrier nodes on the graph. (Nodes which the shortest route cannot pass through)

This software objective is mainly evidenced through the State and Node objects within the Pathfinding_Objects.py file, as well as the UI_Button object. Additionally, the methods in the Pathfinding_State class show how the Barrier button implementation occurs. Throughout the Testing phase, it is proved by System Testing Test Cases 8 and 13.

Sub-Objective i – In addition to the code implementation that I explained in the overview, this objective is completely evidenced by System Testing Test Case 8. This displays that the placing of the barrier node system functions.

Sub-Objective ii - This objective is achieved and proved by both System Case Test Cases 8 and 13. Additionally, the code in the implementation of the Pathfinding_State object shows the creation of the button each iteration of the while loop instance – showing that it can be done at every point in the program's pathfinding state's.

Set 2 - Software Objective 9: The user must be able to clear the barriers that have been placed on the graph.

This software objective is mainly evidenced through the clear_barriers method within the Graph object within the Pathfinding_Objects.py file, as well as the UI_Button object. Additionally, the methods in the Pathfinding_State class show how the Barrier Clear button implementation occurs. Throughout the Testing phase, it is proved by System Testing Test Cases 9, 19, and 20.

Sub-Objective i – This objective is fully evidenced by the System Testing Test Case 9, which shows the barrier clearing process functioning. Additionally, the implementation of the Graph data structure has the clear_barrier method further proves this.

Sub-Objective ii – This objective is evidenced by both the System Testing Test Case 9 as well as the Pathfinding_State loop, which had the implementation including the Clear Barriers button being instantiated every iteration during the while loop within the pathfinding_algorithm method.

Sub-Objective iii – This objective is fully proved using the System Testing Test Cases 19 and 20. These test cases show how the error message appears when the method is instantiated at a point without barriers on the graph.

Set 2 - Software Objective 10: The user must be able to reset the graph to its original state.

This software objective is mainly evidenced through the clear_all method within the Graph object within the Pathfinding_Objects.py file, as well as the UI_Button object. Additionally, the methods in the Pathfinding_State class show how the Clear All button implementation occurs. Throughout the Testing phase, it is proved by System Testing Test Cases 10 and 21.

Sub-Objective i – This objective is fully evidenced by the System Testing Test Case 10, which shows how the process functions without any error. It can also be seen within the implementation – the Graph object containing the clear_all method, which is then passed into the Pathfinding_State class.

Sub-Objective ii - This objective is achieved and proved by both System Case Test Case 10. Additionally, the code in the implementation of the Pathfinding_State object shows the creation of the button each iteration of the while loop instance – showing that it can be done at every point in the program's pathfinding state's.

Sub-Objective iii – This is fully evidenced by System Testing Test Case 21 as it shows how the error message system functions when an Erroneous input is entered – that being when no non-default nodes existed on the graph.

Set 2 - Software Objective 11: The user must be able to run the pathfinding algorithm.

This software objective is mainly evidenced through the running_algorithm method within the Graph data structure object. This objective is supported by a large set of System Test Cases, including Test Cases 16, 17, 18, 22, 23, 24, 25, 26, 27, 28, 29, 30.

Sub-Objective i – This objective is fully evidenced by the System Test Cases of 22-30. These Test Cases account for every single scenario in which the conditions for the Pathfinding to occur are not met – or if an error occurs during the Pathfinding. This mainly includes there not being sufficient nodes on the graph prior to pressing 'Enter', there not being a possible path between the nodes, and pressing another button after finding a path – without having cleared the graph first.

Sub-Objective ii – This objective is fully proven through the Revised Test Case 30. This was an initially failed test, but I since amended the issue, and the new Test Case fully evidences the success of meeting this objective.

Sub-Objective iii - The code in the implementation of the Pathfinding_State object shows the conditional statement checking for the 'Enter' key being pressed each iteration of the while loop instance – showing that it can be done at every point in the program's pathfinding state's.

Software Effectiveness Overview

Efficacy of Solution

I believe that due to the amount of effort put into the analysis of the project, I was able to formulate a set of objectives that successfully encompass the characteristics of the client's desired product. Throughout the communication done with the client – Interview Analysis, User-Acceptance Testing, and the Third-Party feedback, I can state that I successfully met every objective that was specified by the client, as well as surpassing said objectives by adding features to the program which were not requested by the user – such as allowing the user to choose graph dimension size, ability to visualise the A* pathfinding algorithm, as well as choosing between the Euclidean and Manhattan heuristics. This is reflected by the closing client feedback which I received from the faculty:

"Saeed developed the software using a clear and detailed process, and we are quite pleased with what he produced. He kept in touch with us and his classmates frequently during the development to get input on each stage the project was in. Through the inquiries and surveys, he not only made sure that all of our needs were being met, but he also went above and beyond by adding his own features, like the option to visualize the A-star pathfinding algorithm and the help screen that explains the software to the user."

However, despite the satisfaction of the clients and fulfilment of the objectives, in my opinion, there are two parts to my implementation that hold it back from being a complete success.

Firstly, I regret implementing the Graph data structure with a Matrix-based solution. This had originally been my decision due to the simplicity of the implementation within Python, but after analysing the final product, it is determinable that there were some substantial drawbacks to the choice. Primarily, the time complexity of the solution is impacted very negatively, as despite the optimisation of the pathfinding algorithms and supporting data structures, the overarching time complexity of the graph - $O(N^2)$ – means that those optimisations are not fully recognised. This is evidenced through System Test Cases 38, 42, and 46. Despite the test cases having succeeded, it is evident that the run-time for the boundary tests lends to the ineffectiveness of the time complexity on the solution – despite all the optimisation that has been done. The run-time does partly lend itself to the nature of pathfinding algorithms and processes within, however it is factual that the test cases could have most definitely been more efficient with a non-Matrix based implementation of the Graph data structure.

Additionally, I am a firm believer in user-comfortability and quality of life within software that is meant to be used by many people. Therefore, I believe that I could have potentially improved my solution by improving the accessibility of the project. The reason behind why I believe my implementation was weak in this area is that the solution is based on a colour system. However, unlike many systems which use such designs, there is no option for the users to vary the colours to their liking – for whatever reason it is required. This lack of flexibility in the colour scheme could potentially exclude users who have colour vision

deficiencies or disabilities. In addition, the absence of options to customize the interface may lead to a suboptimal user experience, which can decrease the effectiveness and adoption of the software.

In conclusion, I believe that the solution has been a successful implementation of what was desired by the client, and even surpassed the expectations. The solution's scalability is not problematic for the future of the project, as within the analysis, I have already considered how to accommodate the addition of new users to the utility of the software. Through the thorough testing conducted, it is unlikely for there to be any bugs or errors discovered within the implementation as the user-base increases.

Conclusionary Client Feedback

Client Objective Feedback Analysis

Objective	Feedback	Analysis
Set 1 - Software Objective 1: The Program must contain a main menu state.	"We are very pleased with how Saeed implemented the main menu. The main menu provides a clear and intuitive way for users to navigate the software and select the appropriate options for their needs. The menu design is visually appealing and user-friendly, making it easy for users to get started with the software and explore its features. Saeed's implementation of the main menu demonstrates his excellent understanding of user experience."	A large part of the nature of the success of this objective is not only based on the actual implementation of the code, but additionally dependent on the opinions of the client. As can be seen from the feedback, the clients have expressed their satisfaction with the implementation of the main menu state. This evidences the successful meeting of the objective.
Set 1 - Software Objective 2: The Program must contain a 'help' state.	"Saeed has provided a clear and concise guide to using the software, with instructions and annotations that are easy to understand and follow. The inclusion of a picture of the pathfinding display in the help state is an excellent feature that helps users visualize the process and understand how the algorithm works. Saeed's attention to detail and dedication to user experience are evident in the high-quality implementation of the help state, which enhances the overall usability and effectiveness of the software."	It can be said that just as with the main menu state, where the success of the objective being met depends on the user's opinion, the help state also relies on the client's satisfaction with how clearly annotated the instructions are. When analysing the feedback, it is clear that the clients were satisfied with the method utilised to implement the help state, proving that the software objective has met its success criteria.

Set 2 - Software Objective 1: There must be a simplistic user interface.	"The user interface is easy to navigate and understand, with intuitive controls and clear visual indicators that help users visualize the algorithm's progress. The simplistic user interface makes the software accessible to a wide range of users, regardless of their knowledge in pathfinding."	Unlike the previous two objectives whose success was partly based on their implementation, this objective's success is wholly evidenced through the client's satisfaction with it. The feedback implies that there are no qualms with the objective's statement, which indicates that the objective has successfully passed.
Set 2 - Software Objective 2: The pathfinding algorithm must be visualised clearly on the graph.	"The algorithm is presented in a clear and easy-to-understand manner, with a well-designed graph that displays the progress of the algorithm in real-time. Saeed's implementation of the algorithm visualization is a testament to his technical expertise and his understanding of user experience, which allows users to easily follow the progress of the algorithm and understand how it works. The clear visualization of the algorithm on the graph enhances the overall user experience, making the software more engaging and effective."	This software objective had a two-part success criterion – that being the actual implementation of the pathfinding algorithm visualisation, as well as the user's opinion on whether this was clear or not. Implementation-wise, the success of the objective is supported by a substantial amount of testing evidence as well as the actual code. As for the user's opinion on clarity, we can see from the feedback that there were no negative comments on its operation, indicating the software objective's complete success.
Set 2 - Software Objective 3: Information regarding the last performed pathfinding algorithm must be displayed.	"Saeed's method of outputting the last path's statistics was not originally one of the features that we had expected in this implementation. However, seeing it within the implementation, it seems an integral part of the project which displays clear and concise information related to the algorithm."	This was another software objective with a two-part success criterion based on a combination of the implementation of the task, as we well as the clarity within (though this is not definitively mentioned within the title of the objective, it is a sub-point). Implementation-wise, the functionality has been

		shown to function throughout the test-cases (16, 17, 18) which run the algorithm – as the path statistics are consistently outputted to the user. Additionally, analysing the feedback from the user, it states that the information displayed is 'clear and concise' which achieves the second part of the success criterion – additionally evidenced through user-acceptance test case 7. Thus, this objective has also been successful.
Set 2 - Software Objective 4/5: The user must be able to switch between Dijkstra and A* Pathfinding modes as well as between Manhattan and Euclidean heuristic modes.	"Although the A* pathfinding mode and its heuristics were not a part of the requested features, they add a lot of functionality and further assist in the education of the students on heuristics. There is much evidence on the thorough design of these functions - all the way up to the careful labelling of the buttons - and successful completion of this objective."	This objective set is once again based on the two-part success criterion. In this case, the implementation of the objective is clearly evidenced through a multitude of testing and code display. The clarity aspect of both objectives, however, can be proved through the clients' conclusionary feedback, as well as the User-Acceptance test. This feedback mentions the labelling of the buttons, which achieves the second part of the objective's success criterion – therefore making this successfully met.
Set 2 - Software Objective 6, 7, 8: The user must be able to select a node on the graph to be the starting position, ending position, as well as a barrier.	"Saeed has very clearly implemented and met this objective. The tests that he has submitted as well as during his user testing phase displayed that the code works as intended and without error. The displaying of the visual changes on the graph was	These objectives follow the trend of being able to be proved through the two-step success criterion. The implementation step has once again been repeatedly proved to be successful through a combination of much testing and code. As for the user's opinion on the

	clear to see and did not contain flaws."	implementation, we can see from the feedback which stated that the node display was 'clear to see' that the second requirement has been met, thus making this objective a success.
Set 2 - Software Objective 9/10: The user must be able to clear the barriers that have been placed on the graph and reset the entire graph in general.	"The methods of clearing the graph that Saeed has implemented function as he had intended, which he has rigorously demonstrated to us through the combination of his testing video and the user testing. It is evident that he has met the requirements on this objective, as both the error validations have been thoroughly tested – as well as the true functionality."	This set of objectives relies mainly on the actual implementation of itself. This was tested in great depth throughout the Testing phase of the project, and further reinforced through the feedback given by the clients, which stated that the 'error validations have been thoroughly tested'. This indicates that the objective was a success as mentioned by the clients.
Set 2 - Software Objective 11: The user must be able to run the pathfinding algorithm.	"Saeed formulated an impressive method of implementing the running of the algorithms as he thoroughly demonstrated through the code that he submitted. It was further assuring with the way he conducted the amending of the errors he discovered in this system."	This is another implementation-based software objective and has been proven throughout the multitude of testing cases that have been run on the pathfinding algorithms – including the 'revised-case' which was mentioned in the clients' feedback. As the feedback does not disagree with the success of the implementation, this reflects the success of the objective.

Potential Solution Improvements

Technical Enhancements

Graph Infrastructure Improvement

There is currently a large drawback to the type of Graph Infrastructure I have implemented. Due to the simplicity of programming it in Python, I implemented the graph data structure as a matrix – this is because using multi-dimensional lists, a matrix graph infrastructure is very easy to develop in Python. However, this sort of implementation introduced multiple drawbacks to the design of the graph:

1. Space complexity: The matrix implementation requires more space than the adjacent list implementation, especially for sparse graphs, where there are fewer edges than nodes. This means that memory usage can become a concern, especially for large graphs.

2. Time complexity: While the matrix implementation can be faster for checking if an edge exists between two nodes, it can be slower than the adjacent list implementation for traversing the graph and finding the shortest path. This is because the matrix implementation requires iterating over all the edges to find the shortest path.

3. Limited scalability: As the number of nodes and edges in the graph grows, the matrix implementation becomes less scalable. This is because the matrix requires a fixed amount of memory, which can quickly become too large for larger graphs.

The improvement to the infrastructure that may be made would be adapting the system into an adjacent list graph infrastructure. This is a graph infrastructure in which each node is represented by a list, and the edges that connect these nodes are stored as elements within these lists. For each node in the graph, there is a corresponding list that contains information about the edges that connect it to other nodes in the graph. Each element in this list represents an edge, and it contains information such as the destination node of the edge and the weight (if any) associated with the edge. The advantages of this method of infrastructure are:

1. An adjacent list implementation uses less memory than a matrix implementation, especially for sparse graphs where there are fewer edges than nodes. This is because an adjacent list only stores information about the edges that actually exist in the graph, whereas a matrix stores information about all possible edges. Therefore, an adjacent list implementation is more space-efficient for pathfinding algorithm visualiser software that works with large graphs.

2. An adjacent list implementation is typically faster than a matrix implementation for traversing the graph and finding the shortest path between two nodes. This is because an adjacent list stores the edges of a node in a list, making it easy to access and traverse. In contrast, a matrix requires iterating over all edges to find the shortest path, which can be computationally expensive for large graphs. Therefore, an adjacent list implementation is more time-efficient for pathfinding algorithm visualiser software that needs to calculate paths in real-time.

There is another large by-product advantage of the decreased time complexity of the infrastructure implementation as well. This is because the time complexity optimisation of the pathfinding algorithms I made will have a greater effect – as the overarching time complexity of the graph will not be overpowering the time complexity of the algorithms.

Priority Queue Structure Improvement

Despite the optimisations I have made to the Priority Queue, it has not reached its full potential – as I have had to make adjustments to the Binary Heap structure to account for the A* algorithm using it – as it was not suitable prior due to it not originally account for variable reassignment. An adjustment set that I could make would be to create two overarching Priority Queue structures – one being the Binary Heap structure for the Dijkstra algorithm, with the other structure being for the A* algorithm. However, the new structure would have to be optimised for the A* algorithm. Some suitable structures that could be used for the A* algorithm include:

1. Fibonacci Heap, which is a more advanced tree-based data structure that can also be used to implement a priority queue efficiently. It has very good time complexities specifically for operations that involve merging or decreasing the priority of an element.

2. Pairing Heap, which is a data structure that can be used to implement the Priority Queue for the A* algorithm. It has a faster average time complexity for deleting an element, which is also required in the A* algorithm.

3. Binomial Heap, which is another data structure that can be used to implement the Priority Queue for the A* algorithm. It has a faster worst-case time complexity for most operations, including decreasing the key value of an element and deleting an element.

Advantages of each structure over Binary Heap structure:

Fibonacci

1. In A*, we need to update the priority of nodes multiple times during the search process. The decrease-key operation in a Fibonacci Heap has an amortized time complexity of $O(1)$, which is faster than the $O(\log N)$ time complexity of a Binary Heap. This means that updating the priority of a node in a Fibonacci Heap is faster, which in turn speeds up the A* algorithm.

2. The merge operation in a Fibonacci Heap has a time complexity of $O(1)$, while the same operation in a Binary Heap has a time complexity of O(n). This means that merging two Fibonacci Heaps is much faster than merging two Binary Heaps.

3. Although the Fibonacci Heap has a higher worst-case time complexity than a Binary Heap $O(\log N)$ vs $O(1)$, the worst-case scenario rarely occurs in practice. In most cases, the Fibonacci Heap outperforms the Binary Heap due to its faster decrease-key and merge operations.

Pairing Heap

Pairing heap's decrease-key operation takes $O(\log N)$ average time complexity, whereas binary heap's takes O(log n) worst-case time. Since A* uses a lot of decrease-key operations, the pairing heap can be faster than binary heap in practice.

Binomial Heap

1. The time complexity of operations like insertion, deletion, and decrease key is better in a binomial heap than in a binary heap. Specifically, insertion and decrease key operations in binomial heap are O(log n) amortized, whereas in binary heap, they are O(log n) worst-case. This is because binomial heap uses a linked list of trees, which allows for faster merge and decrease key operations.

2. Binomial heap has a better space complexity than binary heap. In a binary heap, the tree is always complete, meaning that all levels except possibly the last one is filled, while in a binomial heap, the number of nodes is not constrained to any specific pattern. This allows binomial heap to use less space for the same number of nodes.

User Experience Enhancements

Colour Implementation Improvement

A quality-of-life change that may be made to the software may be allowing the user to choose the colour scheme on the graph. This would have a doubled-purpose of not only allowing the user to choose how they want the states to be displayed – it would simultaneously explain to the user what each colour is representing, which potentially may enlighten them on the states of the nodes, if they had not already been aware of how the algorithm worked. Some of the reasons behind why this change could be important are:

1. Different users have different preferences for colours. Allowing the user to choose their own colours can improve their overall experience with the software, making it more tailored to their needs.

2. Some users may have visual impairments or colour blindness that affect their ability to distinguish certain colours. By allowing the user to choose the colours, the software can become more accessible to a wider range of users.

3. Depending on the background colour or the lighting conditions, certain colours may not be as clear or visible as others. Allowing the user to choose the colours can ensure that the nodes are clearly visible and easy to distinguish, making the software more effective.