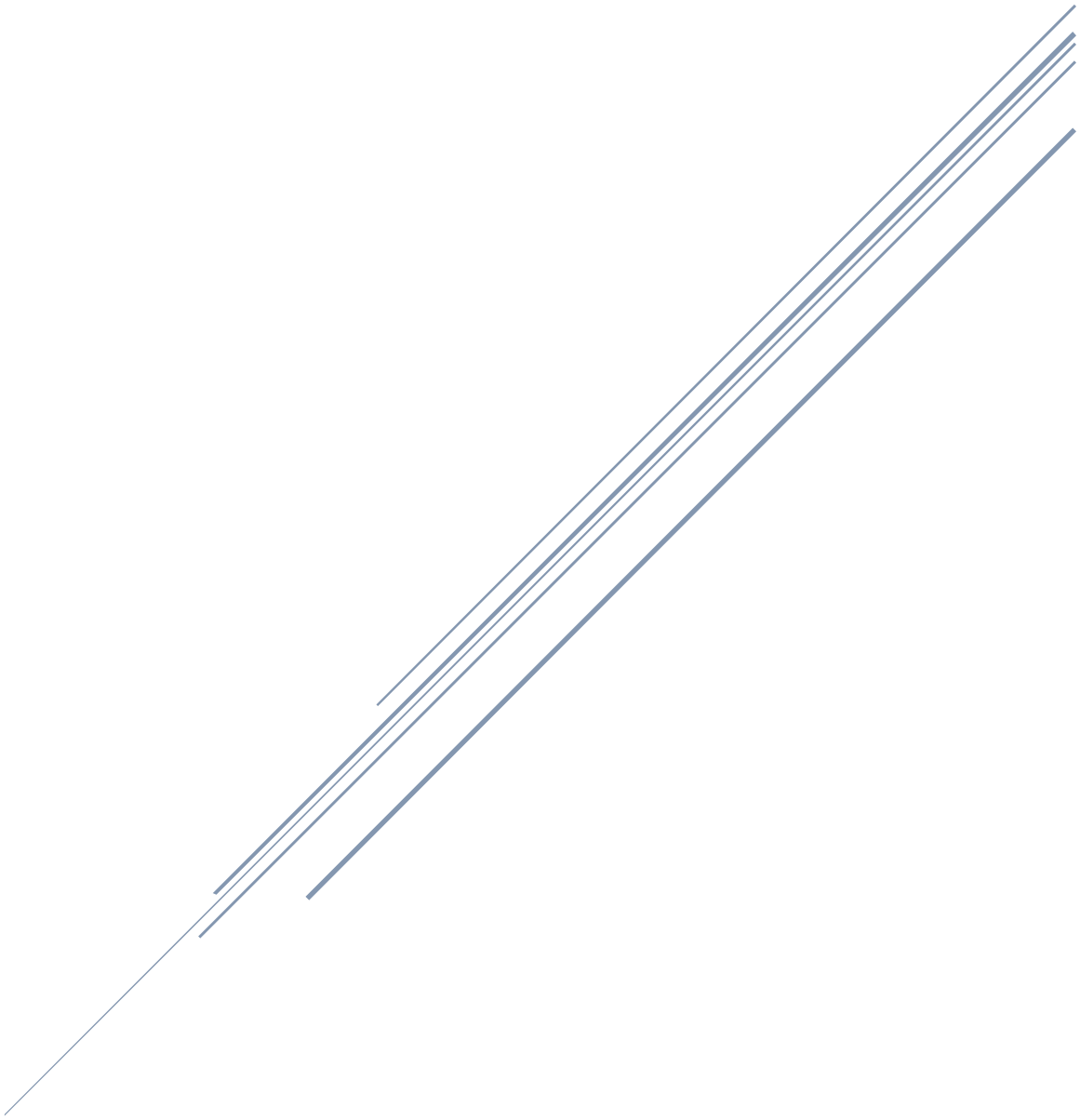
A-LEVEL COMPUTER SCIENCE PROJECT

Pathfinding Algorithm Visualiser (PathPlanner)

AQA A-Level Computer Science



Harris Academy Crystal Palace
Eno Gerguri

Candidate number: 9467 Centre number: 14390

Table of Contents

Analysis

1 Problem identification

1.1 Description of the organisation

SatellitePathX holds a significant position in the GPS location sector, well-known for its groundbreaking techniques and advanced technology. The company's relentless pursuit of perfection persists as it recognises the urgent requirement for a pathfinding solution based on satellite imagery to revolutionise the effectiveness and precision of navigation.

SatellitePathX have its headquarters set up in London and sells a subscription to access their navigation tools monthly on their web application which is free to download on the Apple Store and Google Play Store. This young company was started by a man named Managold Yaakov (CEO) two years ago and has taken off becoming one of the fastest-growing businesses of all time.

1.2 Description of the problem

Up until this point, SatellitePathX were purchasing rights to existing paths that had already been mapped out by other companies. Now they want to take it all in-house, to save costs on renting the maps out and reliability on their servers working when they make requests for access to information using their API. Over the last two years, SatellitePathX has procured many satellite images of different roads and requires as such that these images be used for its own pathfinding software.

My project aims to address the specific issue of calculating the most efficient path between two locations within a road network, leveraging the capabilities of satellite imagery as our primary guidance tool. This is not to say it is in the scope of the project to derive the satellite map ourselves, but that the employees I will be helping will be basing the maps they create off real satellite images.

At this critical juncture, this business is in desperate need of an innovative solution, one that can effectively leverage the immense capabilities of satellite imagery. My objective is straightforward: to convert these captures from satellites into a dynamic tool capable of efficiently mapping out the most efficient path between any given destinations. The dynamic tool will be able to map out between a start and an end point with walls that the user placed, but the user is the one who based on the satellite image, place the walls and the start and target points to be visualised.

The underlying drive behind initiating this endeavour is pragmatic and vital. With a vast collection of satellite images at our disposal, SatellitePathX is now poised to uncover its authentic value via its employees' visualisations. The objective at hand is to construct a software application capable of effectively posting to a database the shortest path found on a given map.

Computational Methods

These problems can be solved by making use of computational methods. For example, abstraction can be used during the design of the interface, by not including unnecessary information. Many applications out there are very feature-intensive and bulky, which can slow down performance and make it more complex for a company to use the software. I can also use abstraction when interviewing my client for information, as they may provide unnecessary details and I have to remove that and see what they are really looking for.

Furthermore, decomposition can be used to solve the creation of a map. For example, by decomposing the problem, I can understand that a map consists of areas that are ok to traverse and areas that are blocked because it is pavement, greenery, or other reasons why someone could not drive through the area. These can be represented as blocks on a map showing the boundaries our pathfinding algorithm is allowed to take. There is a starting point and a target point at which we want to end up. Then we must save this to a database for use by the company in its new internal API which will be created in the future by their development team.

Data collection allows me to take in information from the client. This could be in the form of questionnaires and interviews. Data collection allows me to take in information from the client. This could be in the form of questionnaires and interviews. I could question my client on what the solution to their problem should include and be able to do, ensuring the final solution is correct. This can be used together with data analysis, to make sense of the data I have collected, this could be achieved by making use of graphs and charts of the use of a particular feature and its satisfaction rating.

2 Stakeholders

2.1 Stakeholders who will have an interest in the solution

The main stakeholder of this solution is my client, Managold Yaakov. This is the person who has asked to create a solution to a problem they identified within their company. This person will make the most use of the solution, therefore must be suitable for this person. Since this stakeholder has limited experience in computer systems and using software, the software must be as user-friendly as possible, which means that I can make use of a desktop application. A desktop application can have graphical features and they will be familiar with opening an application on their computer and working the programme, which can make things very simple for someone who is not a tech expert and wants little set-up time on his side. The solution will require the user to input a map, so it is easy if it is a visual screen where the user can simply click where the start and end point would be as well as the walls and visualise the algorithm working.

Another stakeholder of the solution is the development team of SatellitePathX. They will be creating their internal API based on the database I output for them and will be using my programme to help them visualise how the pathfinding algorithm works. This stakeholder will act as a secondary stakeholder, who will code an API based on my project and work that I submit and explain to them. Since this group of people are a lot more experienced in computer work and design, they will likely be able to work off of the database based on a simple explanation that I give them of the code and the database created. Too many help dialogues can make it difficult for them to use the programme and lose the original goal of sleekness and efficiency, as opposed to the chunky and “fat” solutions of competitors.

The last stakeholder would be the users of SatellitePathX’s pathfinding software. Switching servers from rented maps to internally created ones may result in a better-case scenario for some, whilst diminishing the customer experience for others based on the strengths and weaknesses of the internal mappings created. I will have to be careful to ensure that I maintain a consistent pathfinding algorithm for a universal customer experience, that will hopefully be improved after the switch. My testing software will have a couple of algorithms with the ability to easily program more as needed for testing and then from that I can give my recommendations for which algorithm to use in each situation the GPS product may need. I will also have to advise the company to put their software into maintenance mode at hours that the least amount of people will be using it, so the least amount of people are disturbed by the switch from their rented mappings to their proprietary internal database.

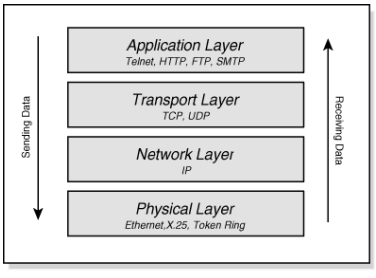
3 Research the problem

3.1 Analysing the problem

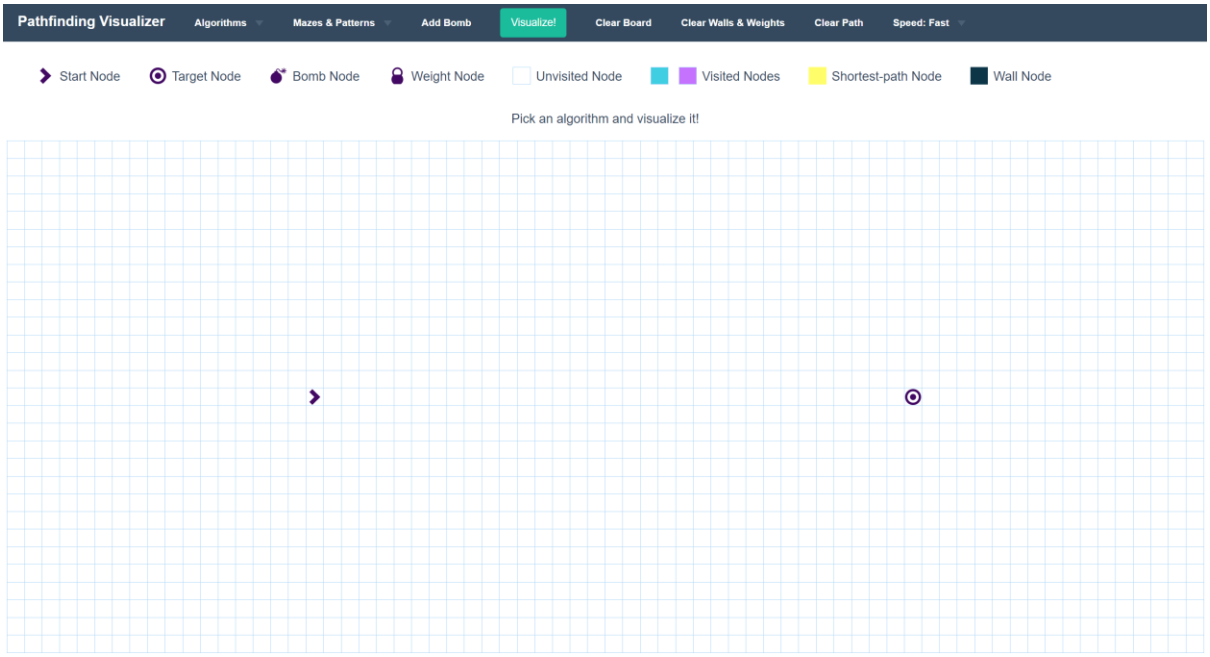
The problem which I am creating a solution to must satisfy the end user of the system. For example, my client wanted a new database to be created with stored solutions to maps as a prototype that may be used for future development by his internal software development team, based on the principles I laid out for them. To do this, I could make use of data from their software team based on the usage of their current model and explain to them my model so they can make a swift change at the best possible time for their company to not receive complaints about their servers being down temporarily.

Google's Firebase is an example of a service that offers real-time databases. It is made for web-based solutions but has third-party wrappers for desktop programming languages such as C# or Python. Real-time databases allow each instance of the solution to be able to create, read, update and delete records in real-time, so changes made by one instance of the solution will be seen by another. Initially, Firebase can be used at no additional cost, however, to expand storage capacity and speeds you must pay a fee to upgrade to a higher package. This would be necessary if the solution is going to experience high traffic loads. This is likely to happen as many people take many trips every day where pathfinding is necessary for them, so this cost must be factored in. Compared to renting out another company's maps though, in the long-term should see a profit emerge from SatellitePathX owning their maps. Since my project is only being asked as a prototype for my primary stakeholder to investigate whether or not it is the right move for the company at the time, I will simply be using a folder with the data stored and an SQLite3 database to index the data correctly. This will be not too far off how a serverless solution such as Google's Firebase will give, but instead, my solution will run on the local host, to allow for a cheaper prototype that will still perform its designed functionality.

Typically, an internet connection would be required. This cost will also need to be factored in, but considering the company is two years old and one of the fastest growing, already with headquarters in London, this does not seem like it will be an issue. On the possibility that there is a cable that is cut or a blackout, their services may be temporarily unavailable. To mitigate this risk, they should make use of all their infrastructure to store copies of the database as well as different versions of it, so that even if there is an issue at one site, another site will still be in operation serving most of their customers, keeping damage retrained to a very local area. Users' pathfinding habits can be looked at to see that most people stay in a particular city and town, and SatellitePathX could also potentially allow customers to download a limited number of maps onto their account to also be used offline in the case of no access to WiFi. The company's hardware like its router and internet service provider should be high-end to maximise its customer satisfaction as lag will be reduced. Though not required for my prototype, it is the advice I give for their company's direction.



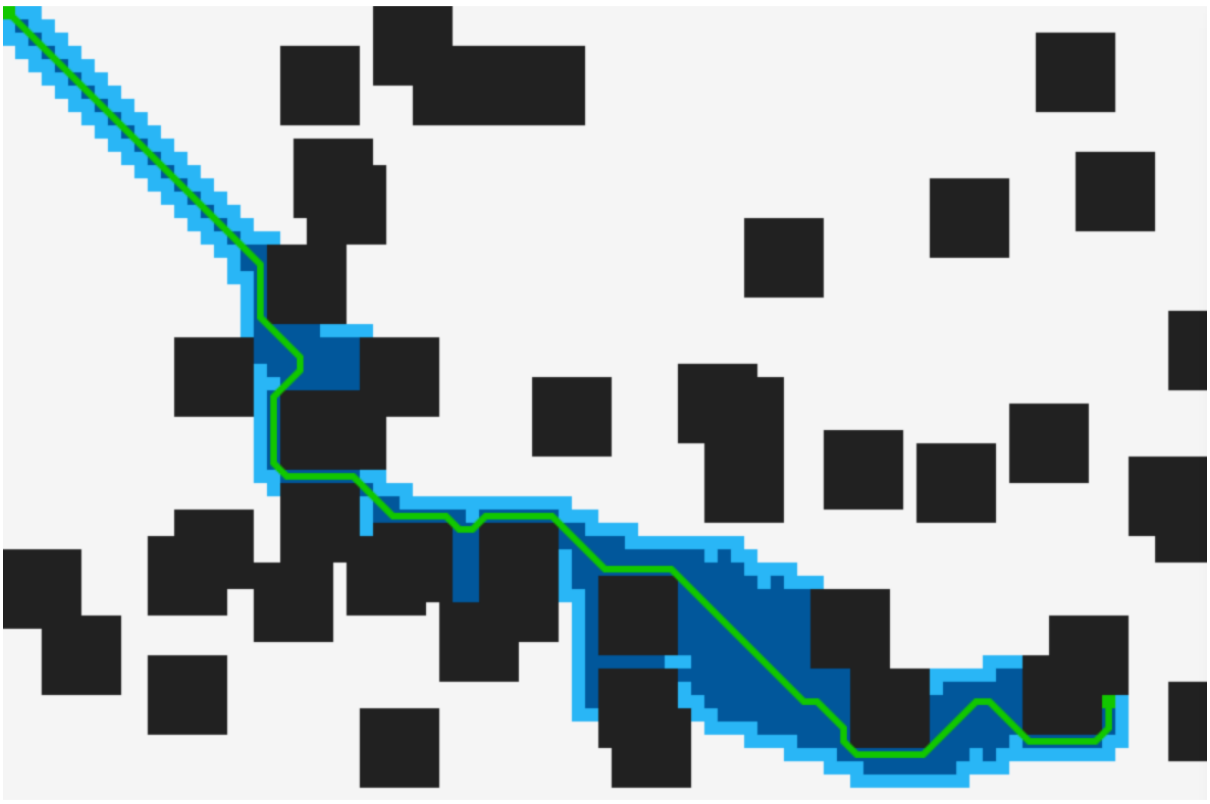
3.2 Existing solutions



(Mihailescu, 2016)

Clement Mihailescu has created quite the solution to the issue at hand. It is a free online pathfinding visualiser under his name, where the user can enter, build a map of their choice and pick a plethora of customising options and algorithms to visualise. However, it is this exact complexity that makes it so unusable in our scenario, where we need a simple usable application.

From his example, I can implement the grid system as seen above to have the user easily pick a start point and an endpoint and visualise the search. The advantage of that is that when visualising the data, it is easily represented by a 2D list, and therefore is easier to traverse and apply the search algorithm. On the other hand, each square will only be able to represent so much space from the image. For example, we can map one square to be the equivalent of a 1x1 square metre on the image, or 10x10 square metre. The problem with this is, the smaller area the square represents the more squares there will be and the longer the algorithm will take to run. Although the difference could be only a few seconds, if they were to use my application to make their maps independently then it could amount to a lot of time given hundreds of thousands of images or even millions. Therefore, I need to inform the manager of the powerful servers they will need to run all of these computations.



(Tyriar, 2012)

This solution from Tyriar looks promising and is well-made. I could make use of this minimalist design to make my application run faster as well as allow less technologically able-minded people to use this application. The only disadvantage would be that the visualisation is not necessarily the best looking, and the manager has asked that they visualise how the path is found. I could implement the aesthetic of Mihailescu's solution, whilst maintaining the simplicity of Tyriar's solution.

The black zones could easily represent housing, roadblocks, rivers and other obstructions that must be avoided to find a path.

The visualisation of the pathfinding and then a final highlighted path will be perfect for this application.

3.3 Advantages and disadvantages of research

The research I have done on existing solutions has allowed me to gain a better understanding of what is required of a pathfinding visualiser such as keeping the design simple, yet aesthetic. Represented by a 2D list and the issues that may arise from the number of squares used and their proportion to the images that are being mapped onto. The user interface and the backend of the visualisation of the pathfinding algorithm have been determined and shown to be the best solution for the client.

However, my research has not shown me the part of the system of mapping satellite images to the square grid, as this would be the role of the software team if they approve of my application and decide to go forward with it.

3.4 Forms of data

3.4.1 Database based

There is a database for the Satellites as such:

SatelliteData	
SatelliteName	string
SatelliteCordinates	integer
created_at	timestamp
images	list[SatelliteImage]

It stores the satellites of the company as well as the satellite coordinates that are updated in intervals where its movement is observable. It was the time it was correctly set up in orbit and the list of all the images it has ever taken.

There is a database for the Satellite Image as well:

SatelliteImage	
created_at	timestamp
bitmap	image

This stores the bitmap of the image of the GPS as well as when the photo was taken.

This is what their software team will use if they wish to proceed with my model after they test my application.

It will not be relevant for my project code specifically but it is important to have this on my periphery to understand that the dev team will want a database system, thus I am considering all angles and possible points for my stakeholders and taking their needs into account.

4 Data collection

At this stage, I am going to get as much information from the client as I can regarding what the client wants from the solution. This will ensure that I produce a solution that does exactly what the client wants it to do. This will also allow for strong communication between myself and the client, meaning less time will be wasted in the future and the chances for mistakes to be made are reduced.

4.1 Questionnaire

A questionnaire can help narrow the focus of the client to the options that I can personally glean information from. This does not allow expansion on his part and can inhibit any extra important details which is why I will later balance this with an interview where he can expand and I can use abstraction to see what is important in what he says and what is irrelevant for me to complete my project.

The following questionnaire was given to my client and staff members to complete.

1. How important is it for the pathfinding visualiser to have a user-friendly interface?

- ○ Not important
- ○ Somewhat important
- ◉ Very important

2. What specific features would you like to see in the pathfinding visualiser?

- ◉ Real-time visualization
- ○ Step-by-step visualization
- ○ Customizable grid and obstacles
- ○ Ability to visualize different algorithms
- ○ Ability to save and load visualizations
- ○ Performance metrics

3. What is the primary use case for this pathfinding visualiser?

- ○ Game development
- ○ Problem-solving
- ◉ Project planning
- ○ Educational purposes

4. How frequently will you be using the pathfinding visualiser?

- ◉ Frequently (daily)
- ○ Regularly (few times a week)
- ○ Occasionally (few times a month)
- ○ Rarely (few times a year)

5. What level of customization would you like in the pathfinding visualiser?

- ○ Extensive, highly customizable
- ○ Moderate, some customization options
- ◉ Minimal, only basic settings

6. Are there any specific integrations or compatibility requirements for the pathfinding visualiser? (Select all that apply)

- ◉ Localhost, independent machine
- ○ Integration with existing platforms or systems
- ○ Mobile app
- ○ Web-based, accessible from any device

7. What kind of support or assistance would you expect to receive during the implementation of the pathfinding visualiser?

- ○ I do not expect any support
- ○ Customization and development services
- ◉ Training sessions or workshops
- ○ Technical support and troubleshooting assistance
- ○ Comprehensive documentation and user guides

8. Is there any additional information or specific requirements you would like to share regarding the pathfinding visualiser solution?

Please find different algorithms and tell us which ones should be used when for our team to be able to implement them for our customers who want more refined products. Our team looks forward to using your prototype for testing and to visualise how the pathfinding occurs to get results. Thank you.

From the answers my client has given me for this questionnaire, I can see that a local application showing real-time visualisation of different pathfinding algorithms is what he requires. This will likely be the two common ones, Dijkstra's algorithm and A* search algorithm so that we can make observations and recommendations for the business based on their performance. I also know they want a minimalist design, with a sleek user interface for use in their future project planning daily.

For these reasons, I know I can use Python, Pygame and Tkinter as my choice of framework to allow for a local host application that works on all devices because it allows for real-time visualisation of Dijkstra's algorithm and A* search. I also know that the Pygame framework can allow for a simple minimalistic design. Perhaps one downside may be the speed of the visualisation, but the backend algorithm can still run very quickly for their satellite image purposes once they decide if they want to use it.

4.2 Interview

4.2.1 Face-to-face interview with client summary

An interview is where my client can expand on details unlike before and I will have to be able to use abstraction to get what is important in what he says and remove the unnecessary details that he includes and are irrelevant to the success of this project.

The following is a transcript of the interview I had with my client.

Me: The first question I would like to ask, is why have you decided to make a pathfinding visualiser rather than immediately solve the problem of making your maps?

Yaakov: We wanted to test whether it is the move the company wants to go through with and commit fully to. As you know, we are a young and fast-growing company, major shifts like this can significantly affect our share prices, so we are very volatile. Being able to visualise the pathfinding process via your local application will allow our team to make the right decision about where we wish to go in the future.

Me: Could you please explain how your satellite system works, and if it will be necessary to access it for this project?

Yaakov: Yes, we have access to a list of satellites which is in our SQL database alongside a table for the list of photos for each satellite. You will not need to access this database for this project, however, we

do want to see a SQL database to store a list of screenshots of visualised paths for our personal use after.

Me: Are there any issues with the current GPS you use?

Yaakov: Our services are relatively good quality compared to our competitors, however renting maps is very expensive and introduces a dependency on our business that could shut off our services at any point. We are dedicated to our customers and wish them to have the quickest pathfinding and user experience possible and making our maps internal could allow for more flexibility as we control the technology in-house. Our share prices could rise as well if we significantly reduce the cost of renting maps and provide the best customer service on the market.

Me: Does your team have their servers, or do you use cloud services such as Amazon's AWS or Microsoft Azure?

Yaakov: We are a young company so have not yet invested in private servers, so we use cloud computing for our software team and all the company's data and business computer users. We are currently using Microsoft Azure and plan to continue with them as they are the best option value-wise.

Me: Do you have much computer experience, how confident are you easily navigating a computer?

Yaakov: I am a businessman, not really a computer scientist. I can perform simple tasks on the computer and navigate Microsoft Office 365 relatively easily. With a minimalist design and simple button presses, this should not be an issue for me.

Me: Is it important that you can access this application from your mobile phone?

Yaakov: It is very unlikely that our team will be running the app on our phones but on the office computers. Perhaps one of my employees may open it up on their phone once or twice, but I do not think we will seriously use this application on mobile devices.

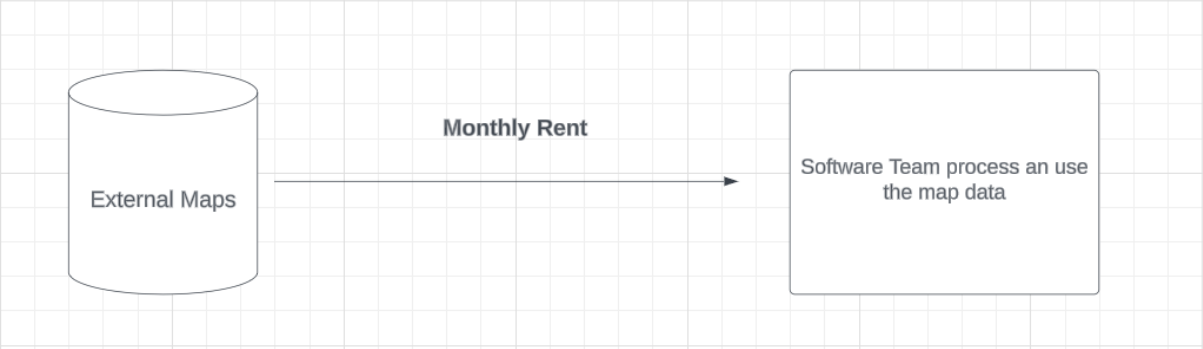
Me: OK, thank you for your ideas and time today.

End of interview.

5 Diagrams

5.1 The current system

Here is a summary of the processes undertaken in the current system:



5.2 Capabilities and Limitations of the current system

Capabilities

The current system allows a specialised company to deal with the maps and therefore have high-quality maps and customer service. They have dedicated and specialised teams that ensure that their services are consistent and competitive. This guarantees high quality and meticulous accuracy.

It can provide efficient mapping for SatellitePathX's customers so that they get quick services that meet their daily travel needs. The system facilitates the prompt generation of routes for SatellitePathX's customers.

Limitations

The dependence on the external company is expensive as the monthly payment is subject to rises in pricing, which can decrease the shareholder price of the company as the renting price is so high and vital to the operations of the company.

There are also strategic and operational risks. For example, servers could go down or there could be legal issues regarding the external company which could lead to a sudden shutdown of the mapping services and thus a shutting down of SatellltePathX.

6 The proposed solution

6.1 Solution requirements

6.1.1 Hardware requirements

- A Keyboard
  - This is a basic input device used to input characters into the computer. It is essential for both the users of the system and me, to use the computer.
- A Mouse
  - This is another basic input device used to allow the user to move a mouse pointer around the screen. This is necessary for the basic operation of the computer.

Required computer specification

The minimum hardware requirements that a computer needs to run this software is the minimum hardware requirements to be able to run Microsoft's Windows 10 64-bit (Microsoft, 2015), however, for an optimal experience, the user should have a computer whose specifications are in line with the recommended hardware specification.

Minimum hardware specification

Processor – any processor with a clock speed of 1GHz or more
RAM – at least 2 GB
Hard disk space – at least 21GB
Graphics – Direct X 9 or later graphics card.
Display – a minimum of 1920x1080.

Recommended hardware specification

Processor – any processor with a clock speed of 2GHz or more
RAM – at least 2 GB
Hard disk space – 30GB or more
Graphics – Direct X 9 or later graphics card.
Display – 1920x1080 or higher.

6.1.2 Software requirements

- A desktop operating system – Windows 10/8/7 OS/MacOS/Linux
  - Need a machine to run the local application on that can perform basic tasks such as handling I/O devices and setting up localhost
- Python 3.12 or above
  - Need this to run pygame and tkinter the packages that my program is built off of

6.1.3 Solution requirements/success criteria

Functional requirements

- User Log In System
    - As the CEO he must track his employees' work so having a user login for everyone using the application will ensure the boss knows who is who
    - The team is a team, working together. They will be able to see who is doing what visualised paths so they can comment on each other's work and hold each other accountable.
    - Stakeholders and the board of directors can use the info on who is doing what work to determine promotions.
- Map Building
    - The solution must allow an easy way of building a map on the web application.
    - The mouse will be used to click and create the walls on the map.
    - A left click on the mouse will be used to place walls or the start and the end node
    - A right click on the mouse will be used to remove walls or the start and end nodes to be placed once again to the user's liking
- Pathfinding Functionality
    - The solution must calculate the most efficient path using Dijkstra's algorithm and A* search algorithm (depending on if they press "d" or "a" for each algorithm respectively).
    - The identified path must be screenshotted and saved to an SQL database for future reference upon user request (for example by pressing the "s" key on their keyboard).
- Minimalist User Interface
    - The solution should feature a user-friendly interface to accommodate users with limited technical expertise.
    - Both Yaakov and the SatellitePathX development team should be able to navigate the application seamlessly.
    - User input for mapping, including start and end points, should be intuitive and straightforward.
- Cross-Device Accessibility
    - The solution must be accessible from various devices, including desktops and laptops, but not mobile phones as my client said that is not necessary.
    - Users should experience consistent functionality when accessing the application from different computers.
- Real-Time Visualisation
    - Real-time visualisation of the pathfinding algorithm must be a feature of the solution.

    - Users should be able to observe the algorithm's progression as it calculates the optimal path.
- Tagging System
    - The software team using the software must be able to tag their saved screenshots to save them into different categories that they need (e.g. "London", "Coffee Shop").
    - The software team will have to be able to make their tags.
- Comment System
    - The team will be able to make comments on their visualised paths and then see the comments made by team members.
    - To find the photo they wish to comment on, the user must be able to browse the saved screenshots and see a small thumbnail of the visualised path before confirming their selection to speed up commenting and avoid mistakenly choosing the wrong screenshot, which would slow them down and frustrate the dev team.

Non-functional requirements

- Performance
    - The solution must load within a reasonable timeframe, aiming for under 20 seconds before a timeout. It should be within 3 seconds though maximum realistically barring any extenuating circumstances.

User requirements

The solution may need a double-check since processing massive amounts of data can be very difficult, so users might need to ensure the routes created by the program match up with the roads in the real world. The algorithm could have issues with such huge datasets, so we just want to confirm that it's accurately tracing paths across road networks.

This user checking would occur for the first few runs of my program by their software team to ensure that it is correctly working.

This means that the prototype I build needs to be made using OOP so that they can extract the necessary functionality to work large scale without the visualisation. For now, I can focus on creating OOP code for the prototype visualisation.

Limitations of Solution

This solution will be visualised using Pygame, which means that with extremely large maps, this solution may be slow to visualise. Theoretically, my client could turn the visualisation off if they just want the final shortest path without the visualisation and then show the visualisation only at the very end so that every frame is not being refreshed. This would be a task for their software team, for now, my solution will be efficient for the average map.

Another limitation. Would be the fact that I am not converting a satellite image to find the walls but this is outside the scope of this solution as my client wants a minimalist prototype so this again would be something the client's software team can adapt for themselves.

6.2 Summary of the solution to be created

The solution I am going to create is a pathfinding visualiser local application for my client's software team for their testing needs. It will be capable of creating a map and subsequently visualising Dijkstra's pathfinding algorithm and A* search algorithm with a minimalist design. There will be a secure login system to identify comments made by different teammates and a system to tag saved visualised paths for easier identification.

Due to the visual needs and the fact that my client wanted a local application, I will be using the Pygame and Tkinter frameworks for the creation of the project. My client has the hardware to be able to run my application and will not struggle to use it as his team is familiar with the basic navigation of computer software. This will take away the need to install anything and can be visited by anyone in their office on a machine.

The dev team management and user system will allow for monitoring by the CEO and greater collaboration efforts as everyone works together and is held accountable towards their company's goal.

The minimalist design is the main Unique Selling Point of the visualiser as a complaint from my client during my research was solutions that were too complex or did not have enough visualisation. My solution will solve both issues by its simple, yet effective visualisation.

Design

1 System objectives

Although I have already defined solution requirements above, by defining system objectives I can expand on the solution requirements to deduce what type of interface the system should have based on the solution requirements. Each design created will follow the applicable functional requirements listed above. Some of the objectives outlined here can also be classed as primary objectives; required features that must be added to the system.

1.1 User interface

- The background colour will be plain white.
- Must have gridlines to represent the squares
- No navigation will be present as it is clutter.
- An exit and minimise button will be displayed on the top right-hand side of the interface, so is not necessary for me to implement it as it is a pygame and tkinter default.
- The interface should be of suitable dimensions to view on a screen with a resolution of 1920x1080 pixels (800x800 px would be valid).
    - This is not required of the mobile-based application as my client said they will not be using their mobile devices.

1.2 Input

- The user will be able to press the visualise button on their keyboard (key `d` for Dijkstra and key `a` for A* algorithm which will begin the visualisation from the start point until it finds the target point).
- The user will also be able to press the button `s` to save a screenshot of the map and upload it to the database.
- The user will be able to create a wall by left-clicking on the map.
- The user will be able to remove a wall by right-clicking on the map.
- The user will be able to move the start and finish on the map by the pick and place feature.
- The user will be able to comment by pressing the button `c` (The user will be able to browse the screenshots taken by pressing the “Browse…” button in the comment system window).
- The user can empty the grid by pressing the `e` button.

1.3 Processing

- A database will be created for initial use.
- Items are added to this database in real time.
    - The database will be able to store the screenshot and the `DateTime` it was taken.
    - The user will have it by default ordered from newest to oldest or oldest to newest depending on the part of the project they are accessing and any filtering they are doing via the applications GUI options.

1.4 Output

- A real-time visualisation of Dijkstra’s or A* search algorithm on the given map
- Saved screenshot upon user request.
- Saved screenshot comments upon user request.

- Saved tags upon user request.

1.5 Internal Objectives

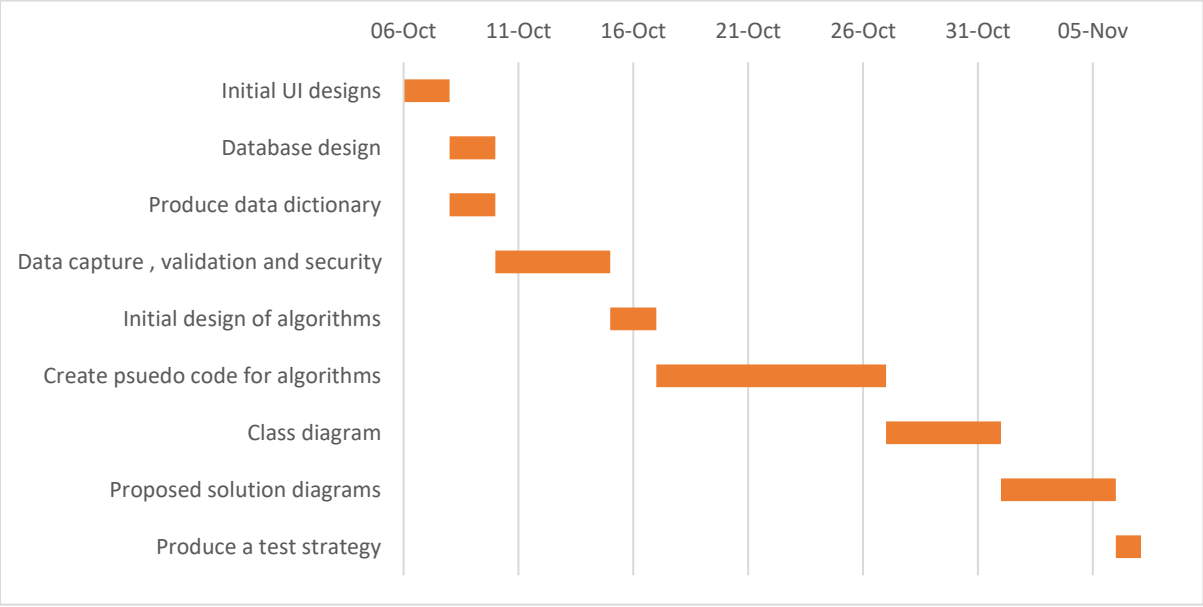
These objectives are objectives that I need to achieve to develop the solution.

- Learn SQL
    - Structured Query Language (SQL) is the language that is used to manipulate and access databases. It is therefore essential that I am comfortable with the language to create an effective solution.
- Learn Pygame
    - I will be creating my game using Pygame so it is important for me to learn this package and use it effectively to meet the client's objectives.
- Learn Tkinter
    - I will be using Tkinter for the auxiliary windows in my program that are not games. For example, the login system will be made using Tkinter as well as the comment system etc.

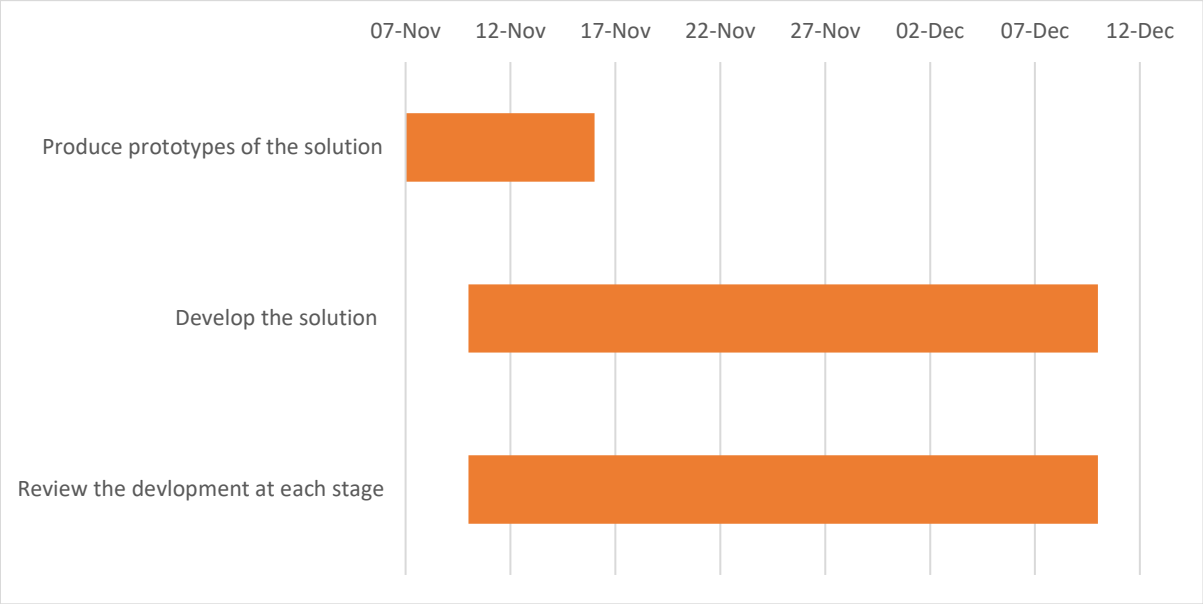
2 Scheduling

To produce an effective solution, I have made the following Gantt chart to illustrate my plan of how I will develop the solution.

2.1 Design Plan



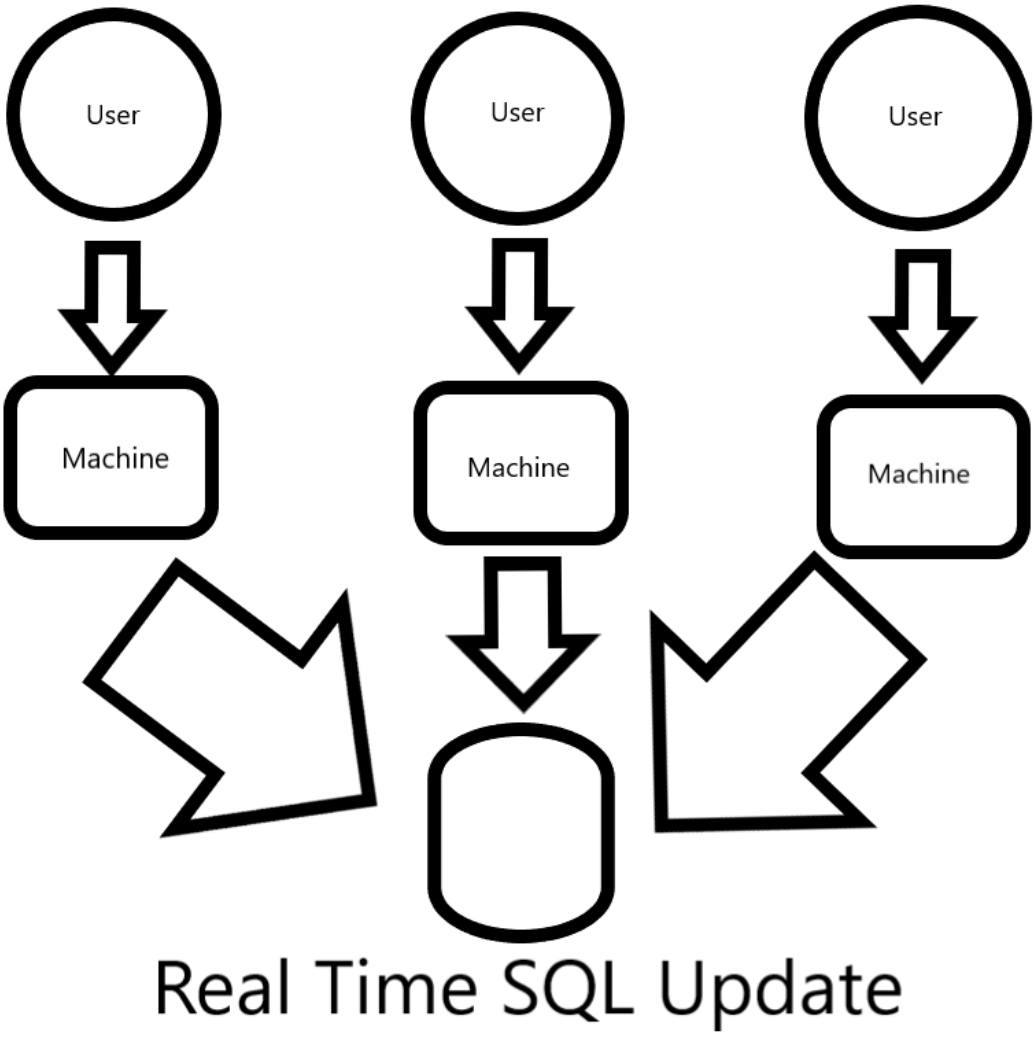
2.2 Development plan



3 New System

3.1 Use-Case diagram of new system features

The purpose of this use case diagram is to show how each user can utilise the system.



3.1.1 Use-Case descriptions

Log In

Summary: The user will log in as their account so their activity can be tracked.

Precondition: The user must enter their username and password before they can log in and their username and password must be checked in the database.

Actors: User operating the PC

Description:

Best case

1- The user enters their username and password.
2- Their username and password exist, and they are smoothly logged in and are aware that they are logged in.

Worst case

1- The user does not enter their username or password correctly.
2- The user is made aware by a message dialogue that their details are incorrect.
3- They must then re-enter their details to try again.

Create User

Summary: The user will create their account so they can access the programme and be tracked.

Precondition: The user must enter their email address, username and password and then re-type their password. The requirements for each field will be discussed later.

Actors: User operating the PC

Description:

Best case

1- The user enters their email address, username and password twice correctly.
2- Their user does not already exist and the fields they entered meet the requirements so they are let known that their account has been created by a message box dialog.

Worst case

1- The user does not enter their email address, username, password or password retype correctly.
2- The user is made aware by a message dialogue that their details are incorrect or do not meet the field requirements.
3- They must then re-enter their details to try again.

Place Start and End Node

Summary: The user will click on the grid to place the start node first and then to place the end node.

Precondition: Users must know where they want to click to place the start and end nodes correctly.

Actors: User operating the PC

Description:

Best case

1- The user configures the map to their liking.
2- The user presses the correct square they want for the start and end nodes.

Worst case

1- The user does not configure the map correctly.
2- They must pick the start and end nodes up to replace their node in the grid.

Place Walls

Summary: The user will click on the grid to place walls which will act like barriers during the visualisation.

Precondition: Users must know where they want to click to place the walls.

Actors: User operating the PC

Description:

Best case

1- The user configures the map to their liking.
2- The user presses the correct square they want for the walls.

Worst case

1- The user does not configure the map correctly.
2- They must pick the walls up to replace their node in the grid.

Pick-Up Start and End Node

Summary: The user will right-click on the grid to pick up the start node or the end node to make the node they previously occupied empty.

Precondition: The user must identify the node of the start or end and then hover their mouse over it.

Actors: User operating the PC

Description:

Best case

1- The user right-clicks on the start or end node and the node is emptied.

Worst case

1- The user right-clicks on the wrong node and potentially sets an occupied node to empty.
2- The user must fix that node, which they emptied as well as try again to right-click the node they did want to empty.

Pick-Up Walls

Summary: The user will right-click on the grid to pick up a wall to make the node they previously occupied empty.

Precondition: The user must identify the wall node they want to remove and then hover their mouse over it.

Actors: User operating the PC

Description:

Best case

1- The user right-clicks on the wall and the node is emptied.

Worst case

1- The user right-clicks on the wrong node and potentially sets an occupied node to empty.
2- The user must fix that node, which they emptied as well as try again to right-click the wall node they did want to empty.

Visualise Path

Summary: Visualise the path in the map from the start to the target point.

Precondition: The user must have placed start and end points on the map.

Actors: User operating the PC

Description:

Best case

1- The user configures the map to their liking.
2- The user presses the “a” or “d” button on their keyboard and the path is visualised using A* search algorithm or Dijkstra’s algorithm respectively.

Worst case

1- The user does not configure the map correctly.
2- The user presses “Visualise” and is unsatisfied with the result because of their mistake.
3- The user must now empty the grid and try again.
4- Or the user does not place both the start and the end and so the visualisation does not run because it cannot run without a start and a target.

Empty Grid

Summary: Empty the grid of the start points and the end points

Precondition: No precondition, the user can press the “e” button at any time.

Actors: User operating the PC

Description:

Best case

1- The user presses the “e” button, and the grid clears every node.

Worst case

1- The user presses the button “e” and an expected glitch or error occurs in the program, which means that the user has to restart the program.

Save Path

Summary: Takes a screenshot of the map with the visualised path and updates the SQL database with the new screenshot saved once a tag is selected in the tag system.

Precondition: The user can save the screen at any point except while the path is being visualised.

Actors: User operating the PC

Description:

Best case

1- The user presses the "s" button.
2- The user sees the tag system menu show up.

Worst case

1- The user presses the "Save path" button.
2- The user tag system screen glitches or has some sort of unexpected error, where they may have to restart the program.

Select Tag System

Summary: Allows the user to select a tag and update the SQL database with the new screenshot saved.

Precondition: The user must select a tag they want to associate their screenshot with, if not chosen the first one returned in the SQL query will be selected and if the user has not created a tag then it is not possible to save as the user will run into an error dialogue.

Actors: User operating the PC

Description:

Best case

1- The user selects the tag they want and sees the tag successfully selected dialog.
2- The screenshot is uploaded to the SQL database.

Worst case

1- The user does not select any tags and has not created any tags.
2- An error dialog shows up preventing the user from saving the screenshot as no tag is selected.
3- The user must create a tag and return to the select tag window to use it.

Create Tag

Summary: Allows the user to create a tag, which will be stored on the SQL database.

Precondition: The user must enter a tag name that does not already exist in the database. If they enter a tag that already exists in the database, an error message box dialogue will appear.

Actors: User operating the PC

Description:

Best case

1- The user enters a tag name that does not already exist and press the “create tag” button.
2- The tag is uploaded to the database and a message dialog saying successfully created tag appears.

Worst case

1- The user enters a tag name that already exists and presses the “create tag” button.
2- The user sees an error message dialogue telling them that the tag already exists.
3- The user must try again to create a tag with a different name.

Comment System

Summary: Allows the user to see comments made on a selected photo and then make a comment on that photo.

Precondition: The user must select the photo they wish to comment on and also type out a comment.

Actors: User operating the PC

Description:

Best case

1- The user enters the photo ID they want to comment on.
2- The photo is retrieved successfully as well as the previous comments made, and they make a comment on it and press the “comment” button.
3- The comment is successfully uploaded to the SQL database and the user sees a message box dialog saying that the comment has been successfully uploaded.

Worst case

1- The user enters a photo ID that does not exist in the database and sees an error dialogue to enter an ID that exists.
2- The user enters a comment, but the comment is not uploaded successfully due to a random error or glitch in the program causing the user to restart the application.

Browse Screenshots

Summary: Allows the user to browse screenshots and filter them by time taken, tag and which user took the screenshot. Small thumbnails will let the user know what that specific screenshot looks like to remind them.

Precondition: No precondition. The user can browse the screenshots and filter them to their heart's content.

Actors: User operating the PC

Description:

Best case

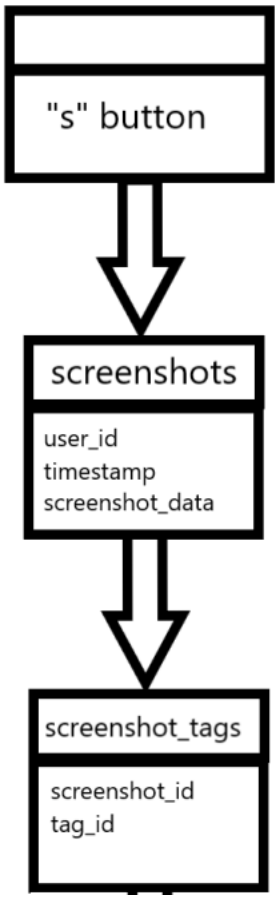
1- The user filters as they wish and search as they wish and then select their photo and press the "select" button to load it into the comment system (more on this later).

Worst case

1- The user decides they do not want to browse photos and close the window. This will not affect the comment system window as it will simply return as `None` photo taken.

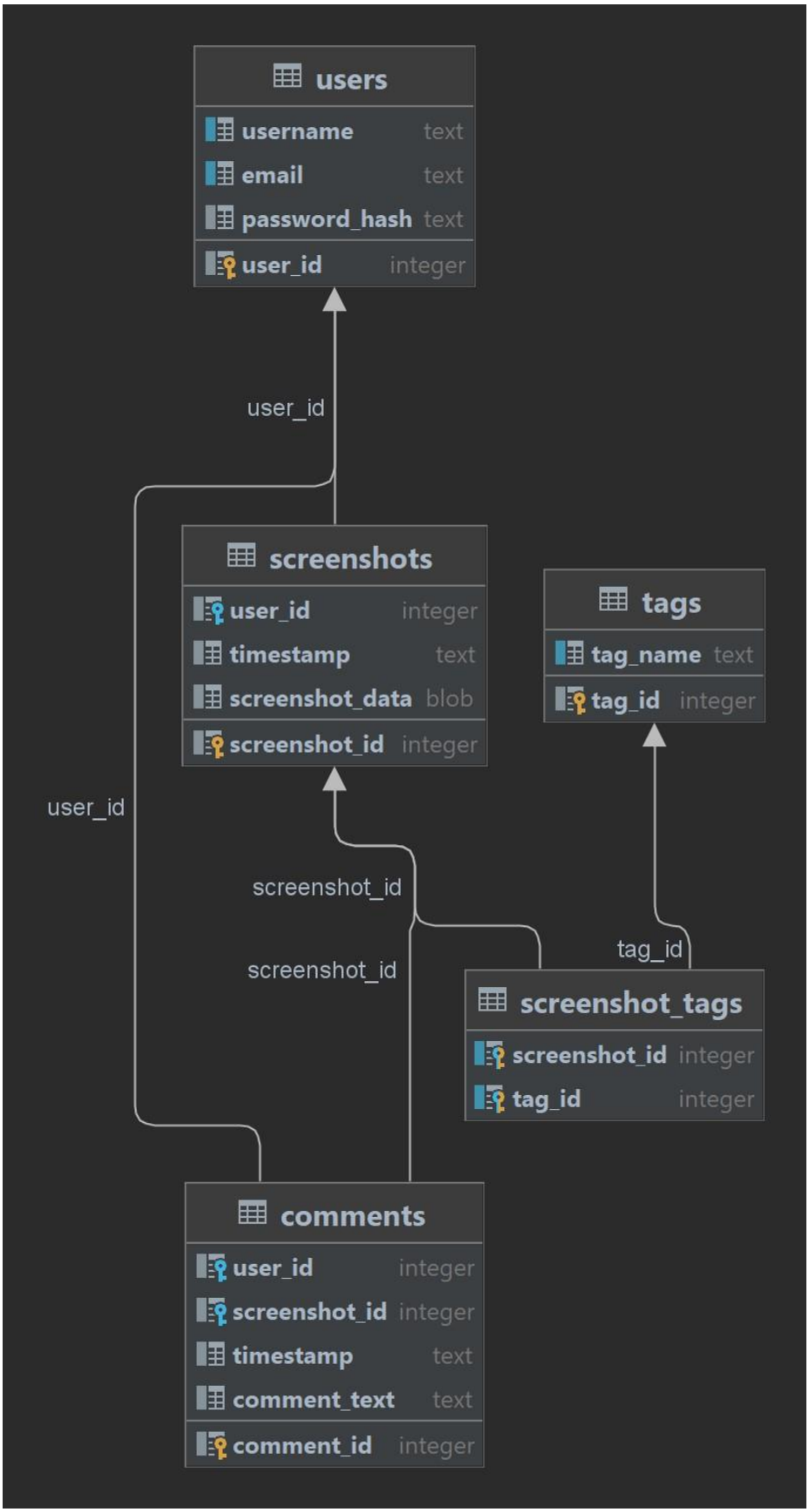
3.2 Input Output Diagrams

The following Data Flow Diagram (DFD) is an extension of the system objectives outlined in the previous section. It will represent the different processes and data involved in the system objectives.



3.3 Inventory Database ERD

All the data stored in the solution be held in a real-time database. The following Entity Relationship diagram (ERD) summarises the structure of the database to be used with the solution.


users
username text
email text
password_hash text
user_id integer
user_id
screenshots
user_id integer
timestamp text
screenshot_data blob
screenshot_id integer
tags
tag_name text
tag_id integer
user_id
screenshot_id
screenshot_id
tag_id
screenshot_tags
screenshot_id integer
tag_id integer
comments
user_id integer
screenshot_id integer
timestamp text
comment_text text
comment_id integer

3.4 Data Dictionary

A data dictionary defines the structure of the database itself. It records what data is stored, and the name, description, and characteristics of each data element.

Field Name	Field Size	Data Type	Data Format	Description	Example
user_id	MAX	INTEGER		Unique identifier for each user. This stays unique even if two users have the same username.	123
username	MAX	TEXT		The username of the user that they input upon registration. For the purposes of this company, this will be simply the employee's name.	Eno_Gerguri
email	MAX	TEXT		User's email address. If necessary, the program can be adapted in the future to use this. Right now, it serves as a way to add a unique user and also stop one person from flooding the database with multiple accounts.	Eno_Gerguri@example.com
password_hash	MAX	TEXT		Hashed password which was encrypted using SHA256. This is used for user authentication and the hash allows for an extra layer of security in case someone hostile steals the database, they still will not be able to click the passwords.	828a1ea07a5741209d93f45dd7ed3644f85db6a875985d1c3208b2e91bb8e0e8
screenshot_id	MAX	INTEGER		Unique identifier for each screenshot saved to the database. This allows for each screenshot to stay unique even if	456

				multiple screenshots were taken at the same time.	
timestamp	19	TEXT	YYYY/MM/DD HH:MM:SS	Date and Time the screenshot was taken. This will be useful for browsing the screenshots later as the team may want to see what they did that Monday, or the previous week.	2024/02/19 14:30:01
screenshot_data	MAX	BLOB		Binary data representing the screenshot image. This is where the actual image is stored for use by the company at a later date.	[BINARY HERE]
tag_id	MAX	INTEGER		Unique identifier for each tag.	789
tag_name	MAX	TEXT		Name of the tag, which will be associated with screenshots. This is how the company can categorise the screenshots.	dijkstra_visualisation
comment_id	MAX	INTEGER		Unique identifier for each comment. Some people may coincidentally leave the exact same message, so this identifier allows to properly query even if there are duplicates.	101
timestamp	19	TEXT	YYYY/MM/DD HH:MM:SS	Date and Time the comment was made. This is useful when someone decides to comment on a screenshot, they can see who else made comments and when. This may be tracked by the CEO to monitor	2024/02/19 15:45:00

				the productivity of its employees.	
comment_text	MAX	TEXT		Text of the comment. This will be shown to others when they go to comment on the same screenshot. This allows the team to get a feel for what everyone else is saying about a certain visualisation.	Great visualisation!

You can see that there are two different values called “timestamp”, the way these will be identified is by the table they are a part of, as is commonly done in SQL databases. For example, `screenshots.timestamp` or `comments.timestamp`. This easily differentiates both variables to be easily manipulated in SQL commands for easy readability and the use of the same name is not significant or bad practice between different SQL tables. Later on, in the code though, we will need unique descriptive variable names.

3.5 Database normalisation

Database normalisation is the optimisation of a database to make it organised in such a way that there is no data duplication, data is consistent throughout the database, the structure of each table is flexible enough to enter as many or as few items needed and it should enable a user to make complex queries relating data from different tables.

3.5.1 Zero Normal Form (0NF)

This describes the state database is in, in its normalised form. The database is most likely to run into errors and is least efficient and effective at this point.

My database is well-made so it should not run into errors, which will be confirmed by testing.

3.5.1 Zero Normal Form (0NF)

This describes the state database is in, in its normalised form. The database is most likely to run into errors and is least efficient and effective at this point.

3.5.2 First Normal Form (1NF)

In this form, all the tables in the database contain no repeating attributes or groups of attributes. Each data item cannot be broken down any further, each row must contain a primary key and the field must have a unique name. An example of how the database used in the solution could be normalised to the First Normal Form is by deleting repeated data is making sure each row contains a primary key.

My database has no repeating attributes for each table.

3.5.3 Second Normal Form (2NF)

For a database to be in the Second Normal Form, it must first be in the First Normal Form. It also must ensure that non-key attributes depend on every part of the primary key. This means that any database in the First Normal Form with simple primary keys (not compound primary keys) will automatically be in the Second Normal Form. Normalising to this form ensures no redundant data is being stored.

No redundant data is stored in my database.

3.5.4 Third Normal Form (3NF)

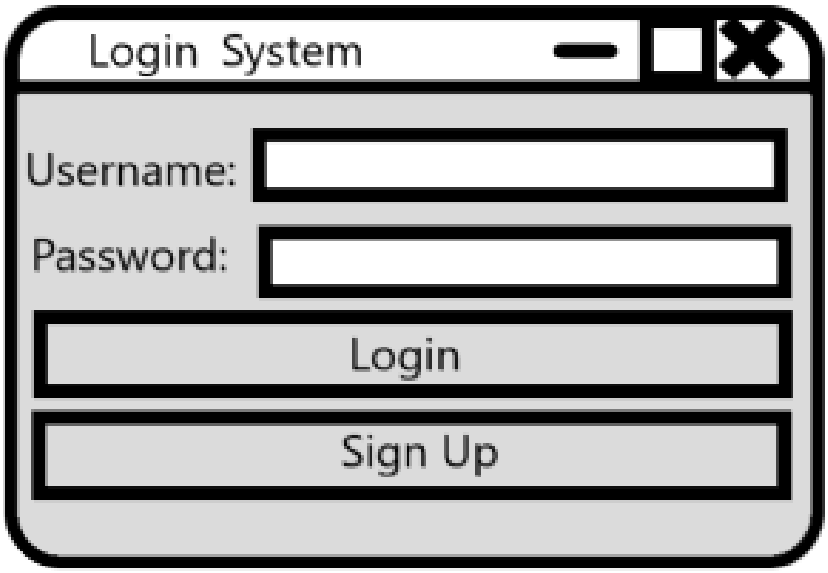
To normalise to the Third Normal Form, the database must already be in the Second Normal Form. In this form, there are no non-key attributes that depend on another non-key attribute. Similarly, to 2NF, 3NF tries to remove another source of redundant data. It does this by making sure the value of one attribute cannot be worked out by looking at another attribute. My database does not suffer from this issue as none of my attributes are interdependent and work together.

In my database, no non-key attributes depend on one another.

4 User Interface

In this section, I will be designing the user interface for the different sections of the system.

4.1 Login Window



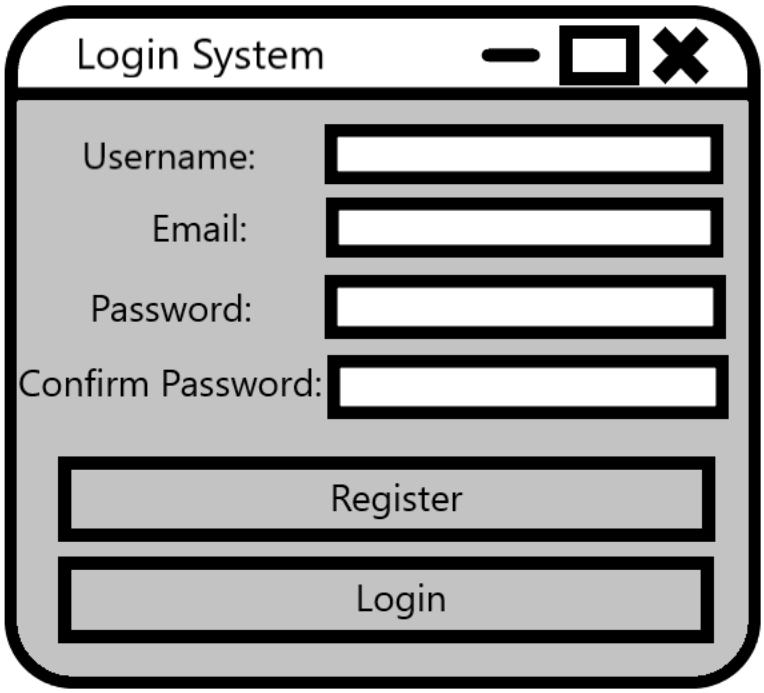
This is the login window.

There are two entry widgets one for the username and one for the password. The entry for the password would be hidden with an asterisk *.

We can see two buttons at the bottom of the screen:

1. **“Login”** – This button will simply log the user in by querying the database. If it is not a correct username or password an error message dialog will appear.
2. **“Sign Up”** – This button will switch the window to the sign-up frame.

4.2 Sign-Up Window



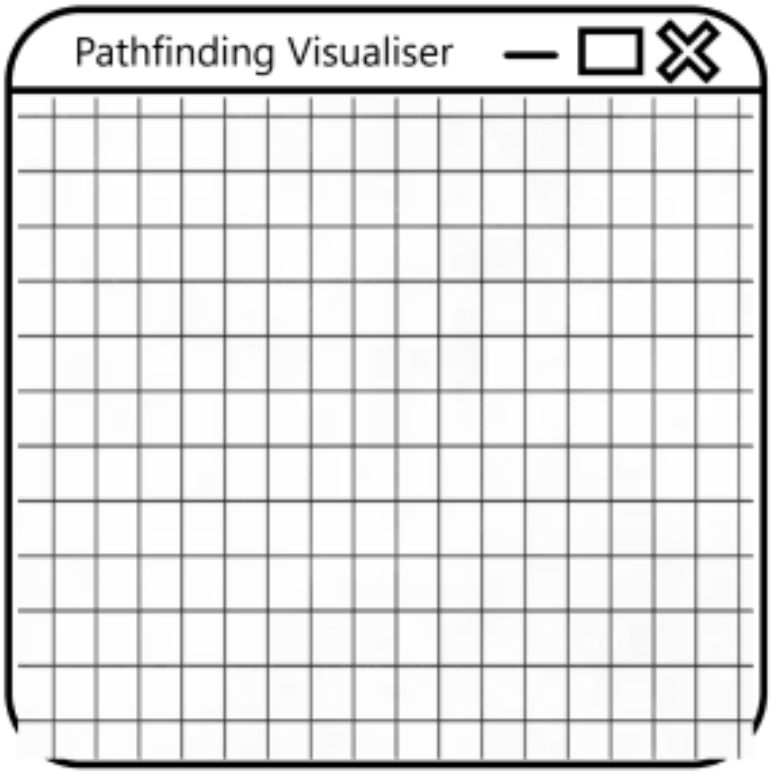
This is the sign-up window.

Here are four entry widgets. The username is likely to be the employee's name. The email address has a special regex check which is common to ensure that its input is in the form of an email address. The password is confidential and hidden with asterisks *. The password and confirm password entry widgets must be the same for the login system to be approved and work.

There are two buttons at the bottom of the screen:

1. **"Register"** – This will check if the user does not already exist with the same username or email address. Will also check whether the password or retype password are the same.
2. **"Login"** – This will load the log-in frame.

4.3 Pathfinding Visualiser



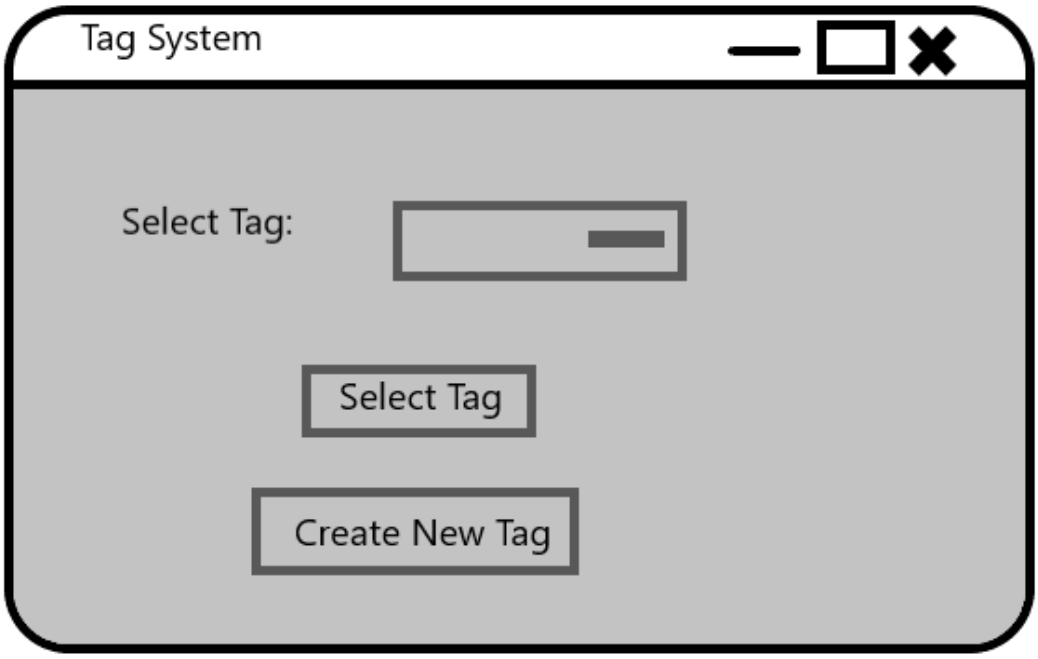
This is the pathfinding visualiser grid. It appears after logging in.

It is an empty grid where the visualisation will take place. The user can place their start and end nodes here by pressing down on a grid. Then walls will be placed.

Being on this screen allows access to the keyboard button commands discussed earlier in inputs.

When visualising the squares will be different colours based on their state. By default, the start node is orange, the target node is turquoise and the final visualised path is purple.

4.4 Select Tag Window

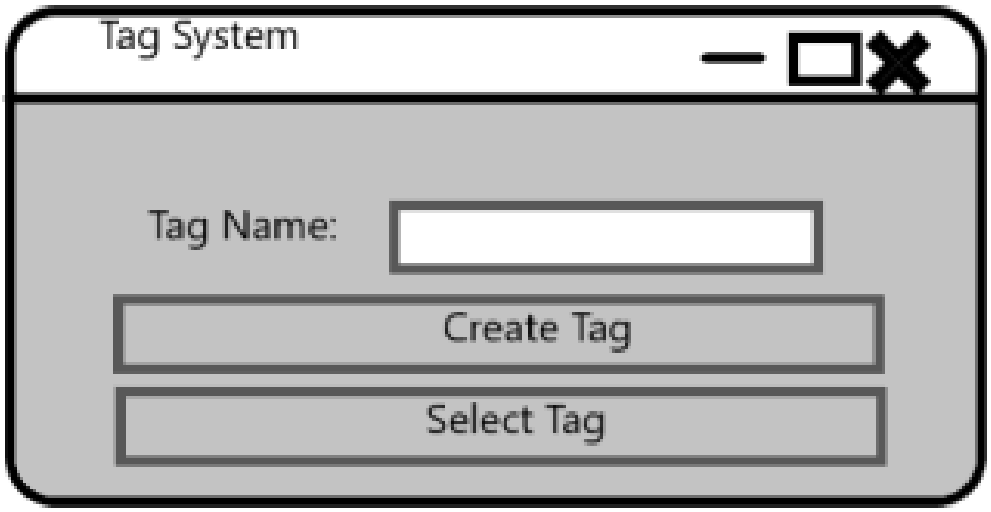


This is the select tag window. It appears after pressing “s” on the grid window and is the beginning of the screenshot procedure.

It has a “Select Tag:” label, which points to an option menu which has already queried the database for the tags that exist. By default, it selects the first tag in the database, but if no tags exist it will default to be empty.
The user can confirm their decision by pressing “Select Tag”, which will upload the screenshot to the database with the selected tag.

If no tag exists or the user wants to create a new tag, it can be done so by pressing the “Create New Tag” button, which will switch to the Create Tag frame.

4.5 Create Tag Window



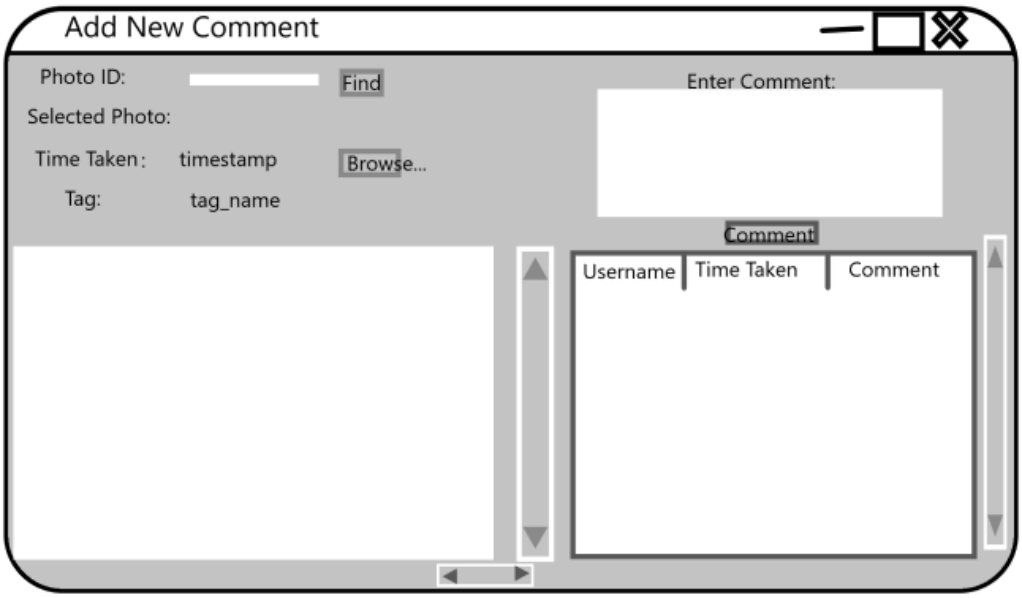
This is the create tag frame. It appears when pressing the “Create New Tag” button on the select tag frame.

The “Tag Name:” label indicates to the user to type the name of the tag they wish to create in the adjacent entry widget.

Pressing “Create Tag” will query the database to confirm the new tag and then set you back to the select tag frame for the user to continue in their tag selection process.

Pressing “Select Tag” will put the user back in the select tag frame.

4.6 Add New Comment System



Here is a rudimentary drawing of how the add new comment window will look.

This can be accessed by pressing the “c” key or after selecting a tag for a screenshot the user will be able to optionally add a comment also. In the case that the user is in the screenshotting action, the photo they took a screenshot will automatically appear and populate the widgets with relevant information.

Upper Left

Starting from the top left we see a “Photo ID:” label which adjacent to it is an entry widget where the user can enter a screenshot ID to find the photo they wish to comment on. Upon pressing the “Find” button the database will be queried for the information of the screenshot with that ID.

Beneath that, we see a “Selected Photo: “ label. This points to the screenshot ID which will be displayed right next to it once a certain screenshot is browsed by the user.

Below that we see the “Time Taken:” label. This indicates to the right the timestamp of the screenshot, so that would be when the screenshot they are currently selecting was taken.

To the right of that is the “Browse…” button. This window will allow the user to search and browse the screenshots in the database with ease. Once selected it will return them to this window with the newly selected photo.

Underneath that, is the “Tag:” label. This indicates the tag group that the screenshot is from and that will be displayed right next to it where it says “tag_name” as a placeholder.

Bottom Left

An empty, white canvas is shown. This is where the screenshot will be displayed fully once selected so the user can view the visualised path whilst they make a comment on it, to be used for reference.

There are two scrollbars, for the program is designed to have variable row sizes, so the ability to scroll the different images is important to the flexibility and robustness of this program. The vertical bar will scroll the image up and down and if needed there is a horizontal bar to scroll left to right.

Upper Right

In the upper right, we see where the user will be making their comment.

The label is “Enter Comment:” indicating to the user this is where they should type their thoughts. The textbox beneath takes in the user’s input and has been made large unlike most entry widgets as it may be taking in a lot of content. Then to upload the comment to the database the user presses the “Comment” button.

Bottom Right

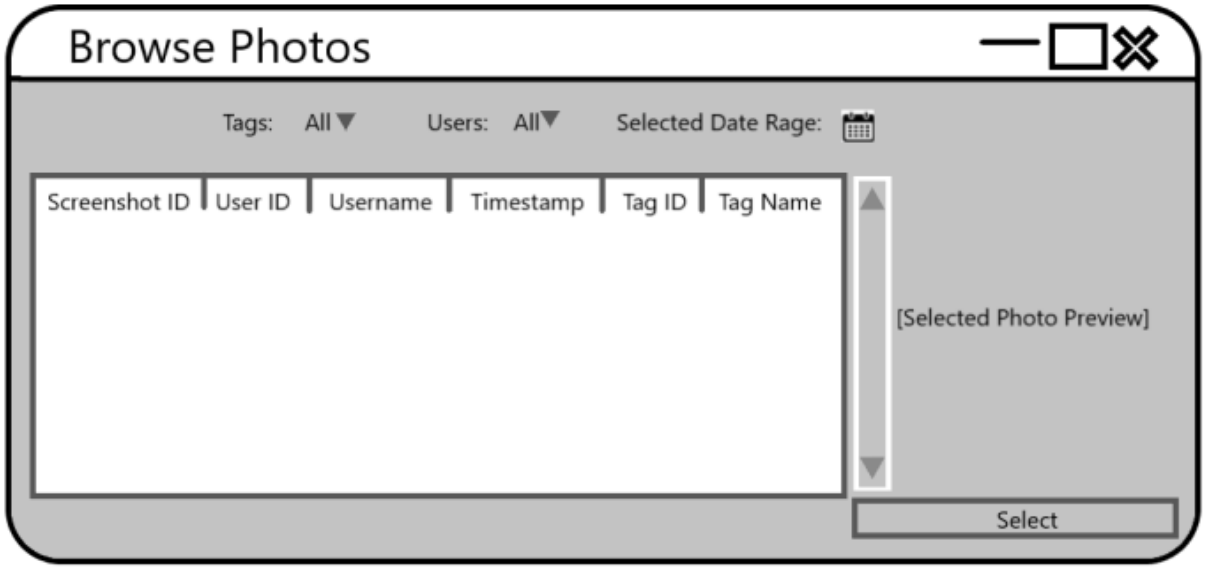
In the bottom right we see the history of the comments of the selected screenshot.

When one user is commenting on a screenshot, they will want to see the comments made by other people so that they do not repeat themselves, or gain more insight on how the team is collaborating around a specific screenshot.

It allows the user to see who made the comment via the “Username” column. It allows the user to see the time the comment was made via the “Time Taken” column. Then of course the full comment can be seen via the largest column “Comment”.

Since there may be many comments/discussions on a certain screenshot of a visualised path, there is a vertical scroll bar to be able to scroll through all of the comments that are registered in the database on that specific screenshot.

4.7 Browse Photos Window



This is the browse photos window, which appears after pressing the "Browse…" button in the "Add New Comment" window.

At the top, we can see a range of filtering options. We see the labels describing the option and then a dropdown menu with all the options which have been queried from the database to fill. It defaults to "All" so all the screenshots will be displayed for browsing initially. The date range will open a new window to select between two dates and then return to this window to show the two dates in a label next to it.

Screenshots in the tree view will be refreshed upon every change to the options menu or date selected automatically.

Filtering options:

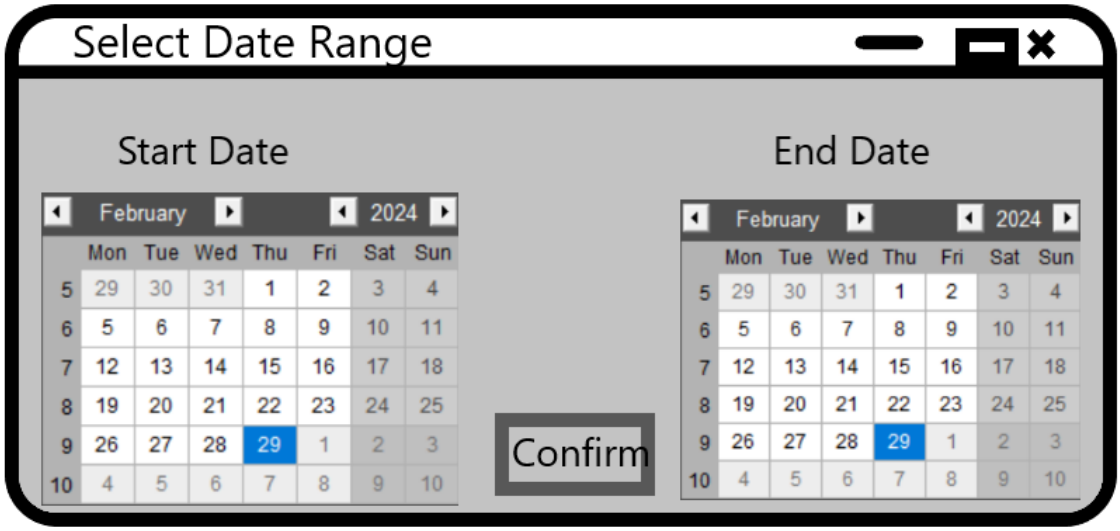
1. **"Tags"** – This option allows the user to filter the options by the tag group they were saved with. This would be useful for someone in the team only looking to comment on a certain type of screenshot which is identified by the tag. It is shown as follows: "tag_id: tag_name" so that the user can do it by either tag id or tag name.
2. **"Users"** – This option allows the user to filter options by the user who saved the screenshot initially. This is useful for collaboration between members of the team or for the boss tracking work done by a specific employee. It is shown as follows: "user_id: username" so that the user can identify by user id or username.
3. **"Date Range"** – This option allows the user to filter options by the date the screenshot was saved. This could be useful if the team wants to review their screenshots from the day before or the week before so can also be used as a measure of productivity for the team as a whole by the boss. Once a date is selected by clicking on the calendar icon a label will appear showing the date range that the user has selected.

Each entry shows the: screenshot_id, user_id, username, timestamp, tag_id, and tag_name of the screenshot so that the user can make an informed judgement on which one to select and comment on.

Whenever an entry is selected, the right side where it is labelled “[Selected Photo Preview]” shows a small thumbnail of the image, so the user can roughly see what the visualised path is like before they confirm their selection. The image is scaled to 200x200 pixels so it is not meant to be as detailed as the canvas dedicated to it in the “Add New Comment” window but it is meant to give a small idea to the person of how it looks as extra detail so that they can make the correct selection for the screenshot they wish to comment on.

The ”Select” button at the bottom right of the screen confirms the user’s selection of their currently selected screenshot in the tree view. Subsequently, it will be loaded into the “Add New Comment” window, ready for the user to make a comment.

4.8 Date Range Selector Window



A range can be selected here and its parent will be edited to filter valid screenshots to only the date range specified here.

There will also be generic info message dialogues and error message dialogues, but these all look very similar with different text in the middle stating the specific info or error message, thus I will not include them here but in the testing phase they will be displayed as they are simply the default message box windows from the Tkinter package.

5 Data Capture

Data is inputted throughout the program at different points.

5.1 Validation

Validation is an automatic computer check to ensure that the data entered is sensible and reasonable. However, it does not check the accuracy of data. Validation will be used wherever possible in the solution to ensure that data is as correct as possible.

5.1.1 Login System Validation

When logging into the program some necessary checks need to occur to ensure proper authentication for users.

We cannot have a user with no name or no password so when logging in we must validate to make sure the user entered data into the entry widgets.

Once that basic validation is complete, a table lookup check is performed to check whether that username and that hashed password match with what has been created in the database. If the username or password does not match what is in the database it means that the user does not exist and so an error pops up and the validation has now been performed successfully.

When creating a user, there are many more validation checks which must occur to meet security requirements.

Firstly, the username is validated to ensure that the user has typed out something for their username, in this case, it is likely to be the employee's name.

Secondly, the email address goes through a regex validation to ensure that it follows the form of an email address.

Thirdly, the password must go through many validations to ensure that system security is maintained in this program. We have the first validation to check if it is over seven characters as those passwords are typically the hardest to crack. Then we have a check to ensure there is at least one digit and one special character minimum so that it makes the password less conventional and harder to crack by algorithms devised by hackers.

Fourthly, we must confirm the password retype is the same as the initial password entered so that the user knows for sure what they typed as their password and minimise the chance of a typo in the password the user created.

5.1.2 Pathfinding Visualiser Grid Validation

When visualising on the grid, Pygame must perform detections for when an input event occurs. I will use validation checks to ensure that the input is valid on the grid.

The validation check would be upon a click of the mouse, to get the position it was clicked at and if it was clicked on the grid, which would mean that the set node function can be set correctly and if not

then no node state is changed as it would be handled by Pygame as a window move or a closing of the window which I handle to quit Pygame if that happens.

Another validation check would be upon pressing “a” for the A* search algorithm and “d” for Dijkstra’s algorithm. For the algorithm to be visualised we must validate whether the start node and the end node have both been placed. If not, then we cannot go through with the visualisation.

When placing walls by clicking on the left mouse button, we must perform another validation on the wall not being placed on the start node or end node so as to avoid glitches in the program.

When placing the end node using the left mouse button, another validation must be performed so that it is not being placed on the start node.

5.1.3 Tag System Validation

The tag system must ensure that a tag is selected from the tags that exist within the database. When selecting and creating a new tag, many validations must take place to ensure the data input is valid for the program.

One validation that occurs on the select tag frame would be ensuring that a tag has been selected and the options menu is not empty, as we cannot allow the user to save a screenshot using an empty tag to meet the requirements of the organisation my client requires. If a tag is not selected it means no tags exist and an error will occur stating that the user has not selected a tag.

In the create new tag frame, there will be a first validation that the user typed out a tag name as the user cannot create an empty tag. The second validation is a table lookup to check whether or not there is another tag with the same name. This avoids duplicate data in the database.

5.1.3 Comment System Validation

Validation in the photo ID entry widget occurs when typing in the widget. There is a validation check to ensure that the entered character is an integer. This means the only characters that can be typed are integers.

Upon pressing the “Find” button, a table-lookup validation is made to check the screenshot ID is in the database.

Another validation is performed when writing a comment, there must be a check to ensure that text was entered so that an empty comment is not saved onto the database.

5.2 CRUD permissions

The solution will have permissions for CRUD (Create, read, update, and delete) operations. The following user groups will be created, which will determine the permissions each user will have. Users will be placed into the group depending on their role in the business.

A summary of user groups is shown below in order of most power to least. User groups share the permissions of user groups below them.

5.2.1 Admin Group

This user group will have full CRUD access on both tables in the database as well as direct access to the raw database. No users of the system should be in this user group, its purpose is for debugging only.

This would be accessing the database with SQLBrowser or similar database viewing software which allows you to make internal changes and admin controls.

This will be given to the board of directors and senior engineers at the company.

5.2.2 Staff Group

This group will have read-only access to the screenshots of the paths.

This is available to the staff of the company to create presentations to investors and other similar presentation tasks.

The permissions are explained in further detail in the following table.

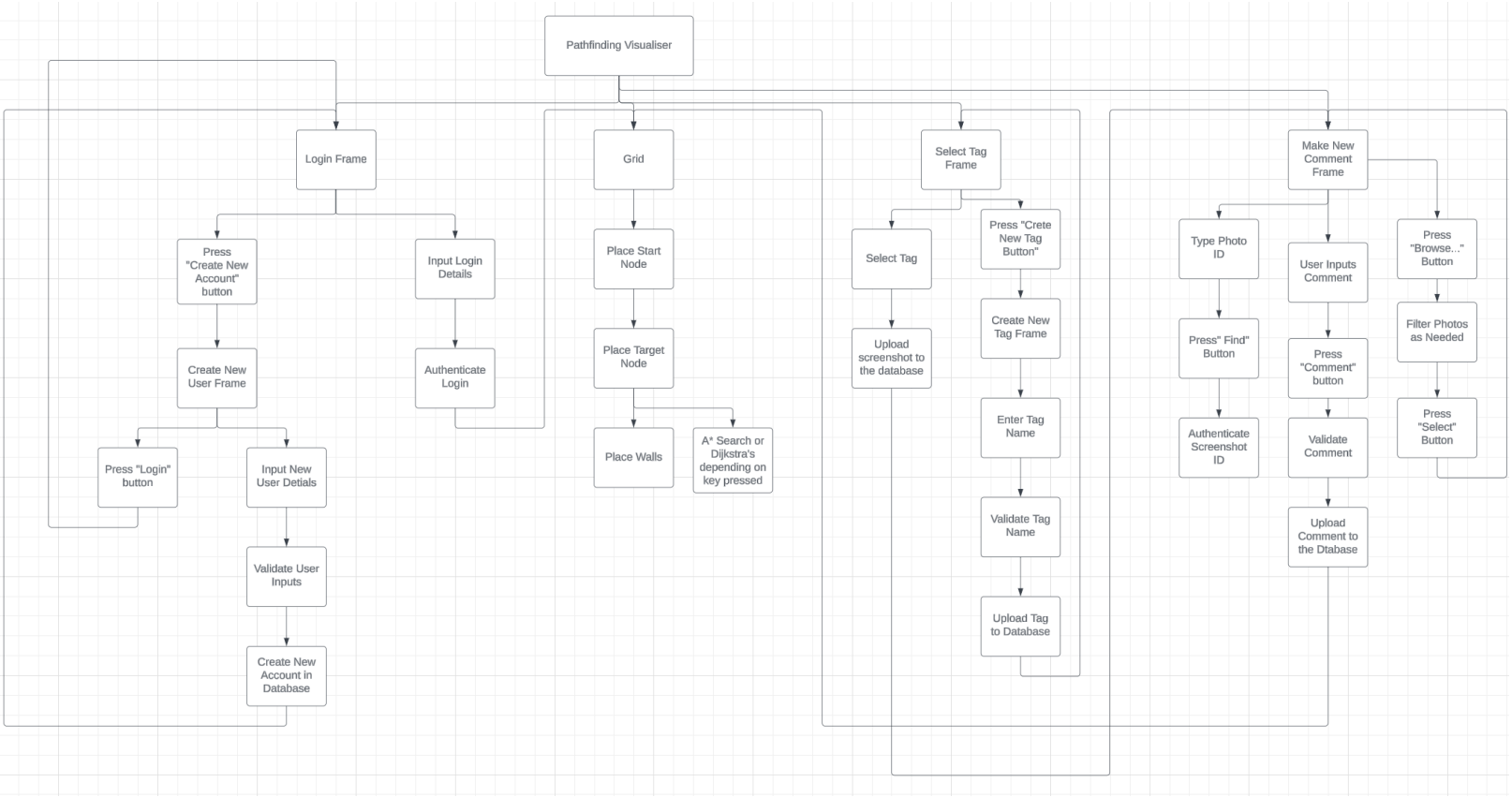
Permission group	**Permissions**
Admin	Are authorised to use external tools and software for debugging and the purposes of the company. Users below this group may not use external tools to add to the database or delete or make any modifications whatsoever.
Staff group	This permission group is the most basic group the solution has to offer. It still allows full functionality within the program, but no external tools are permitted to be used on the database. It Is aimed towards staff members who are preparing for presentations to stakeholders of the company and wish to show them some paths.

6 Algorithms

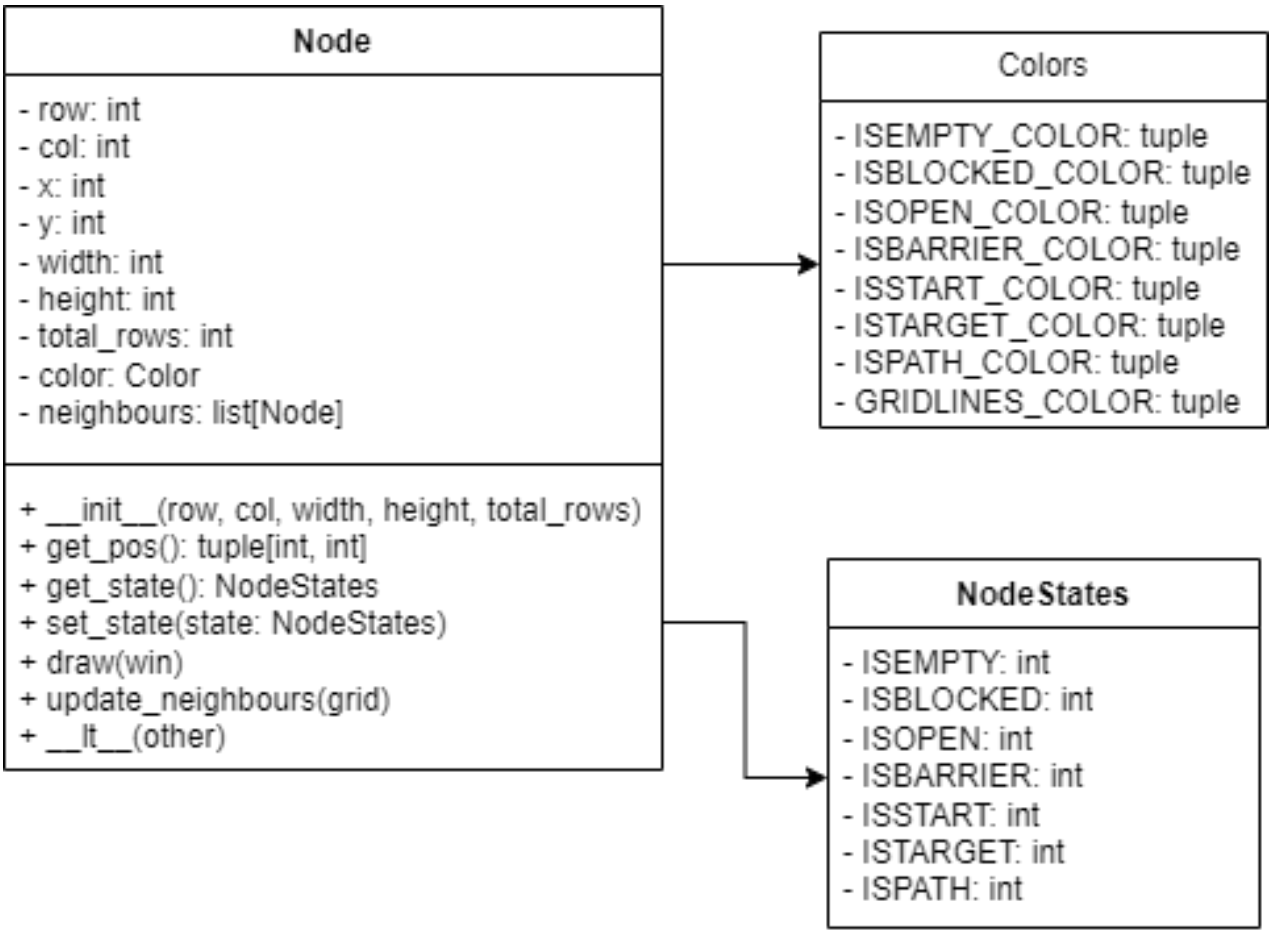
The purpose of designing algorithms is to make it easier for the programmer to begin programming the system.

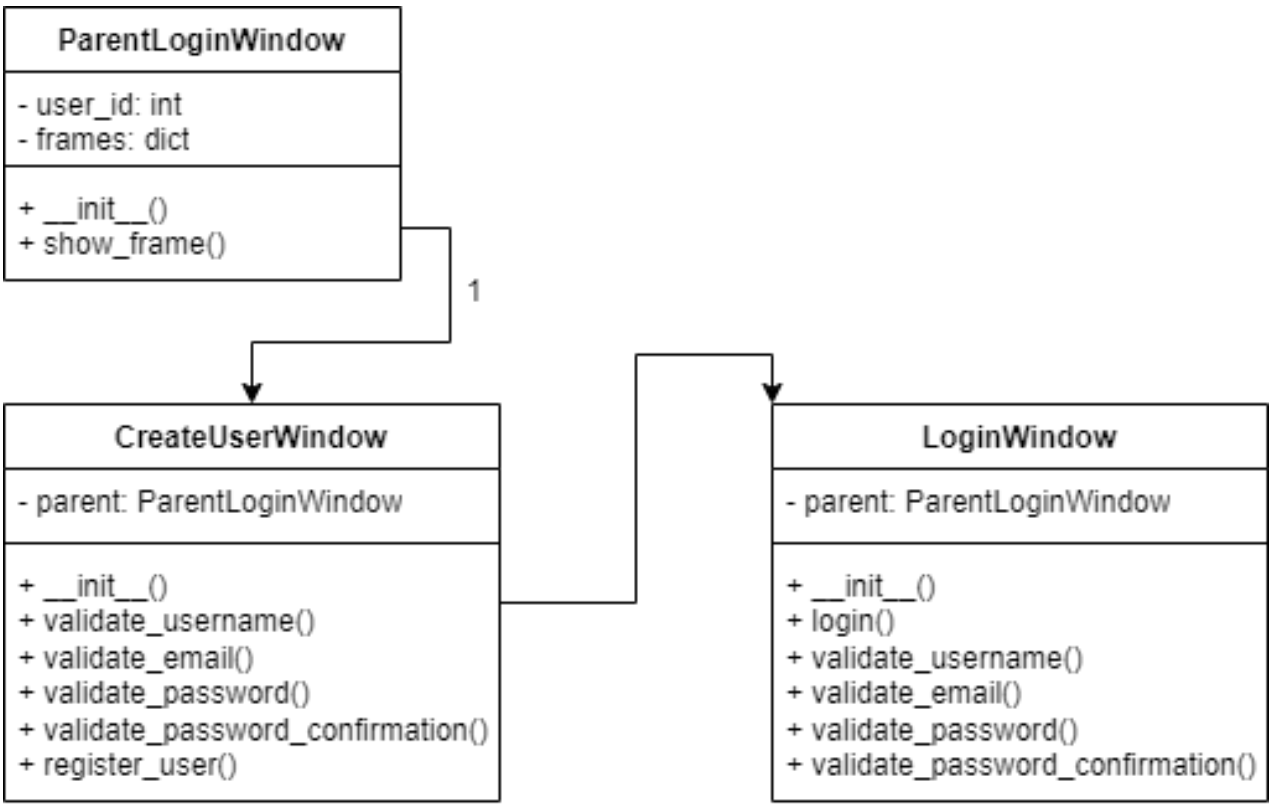
6.1 Structure Diagram

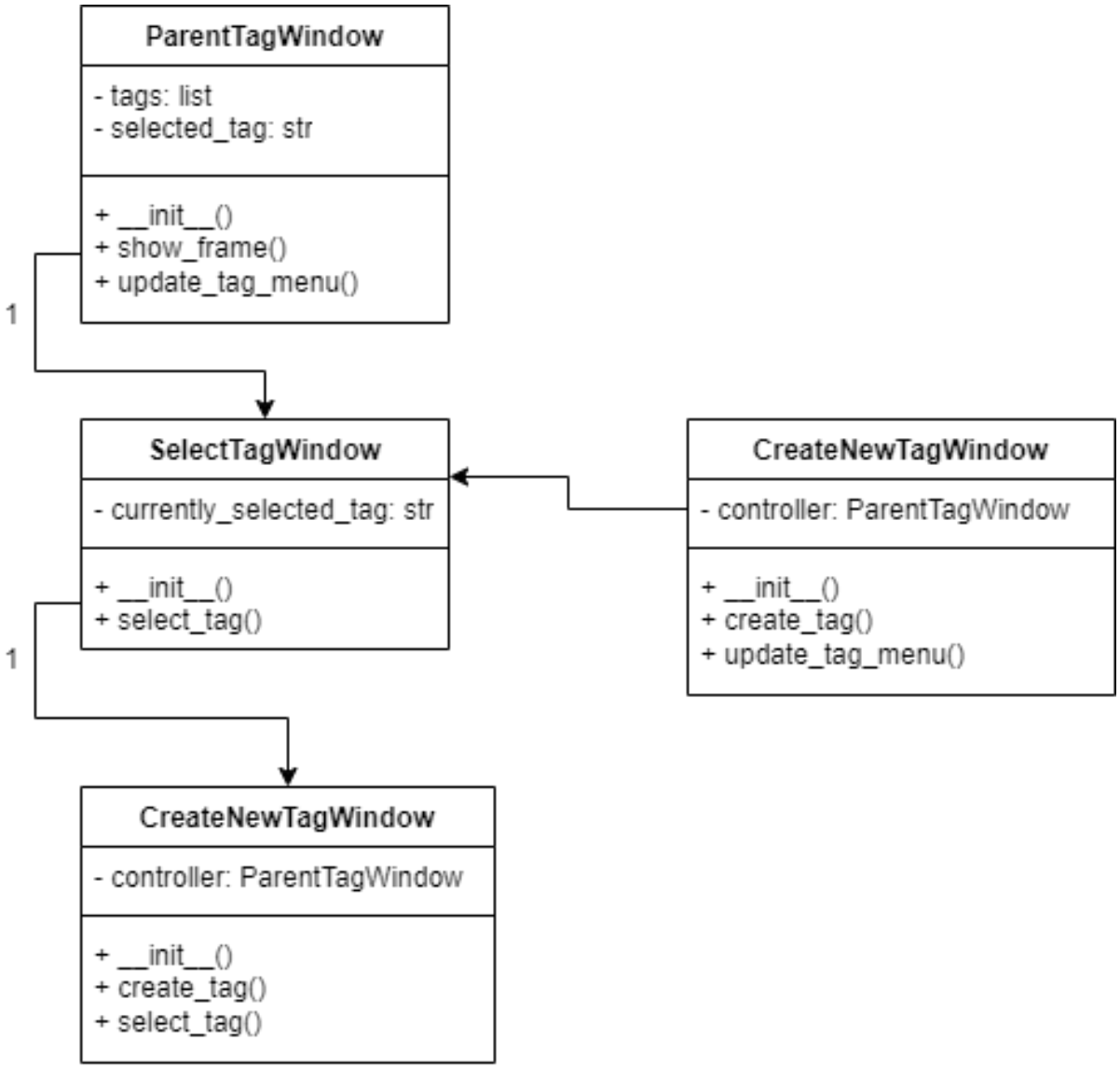
I have created the following structure chart to show all the sections of the system.

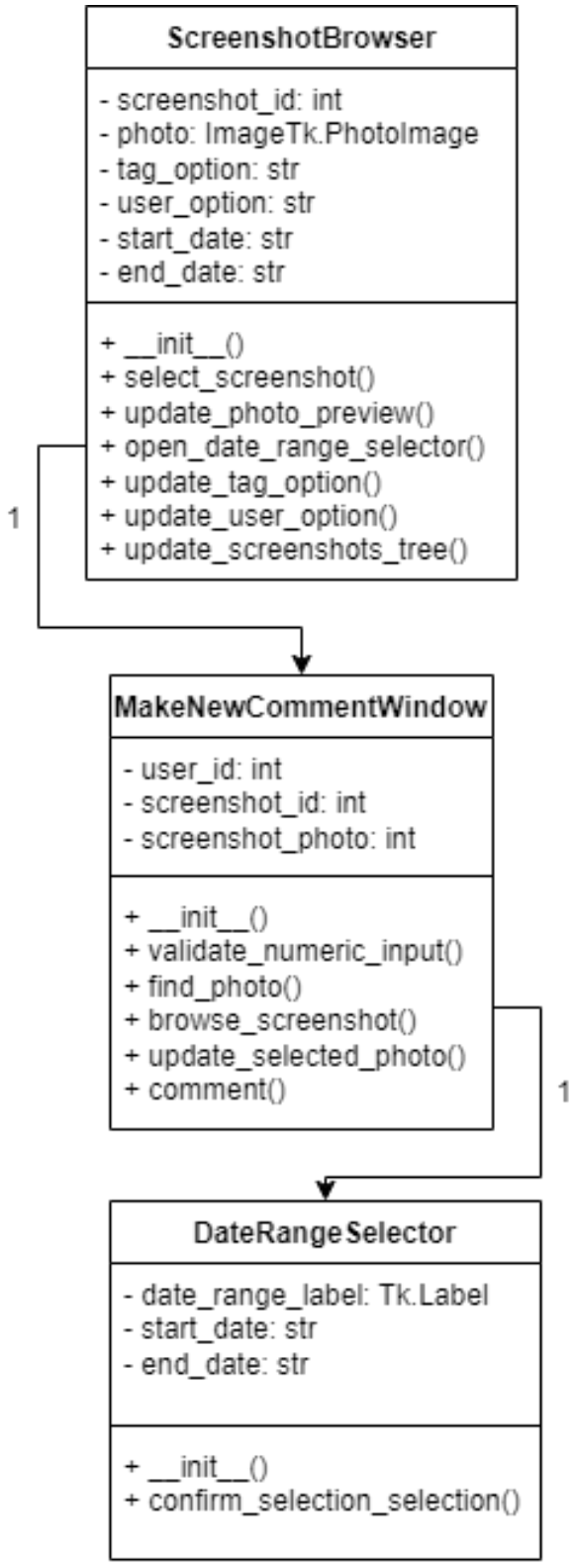


6.2 Class Diagrams



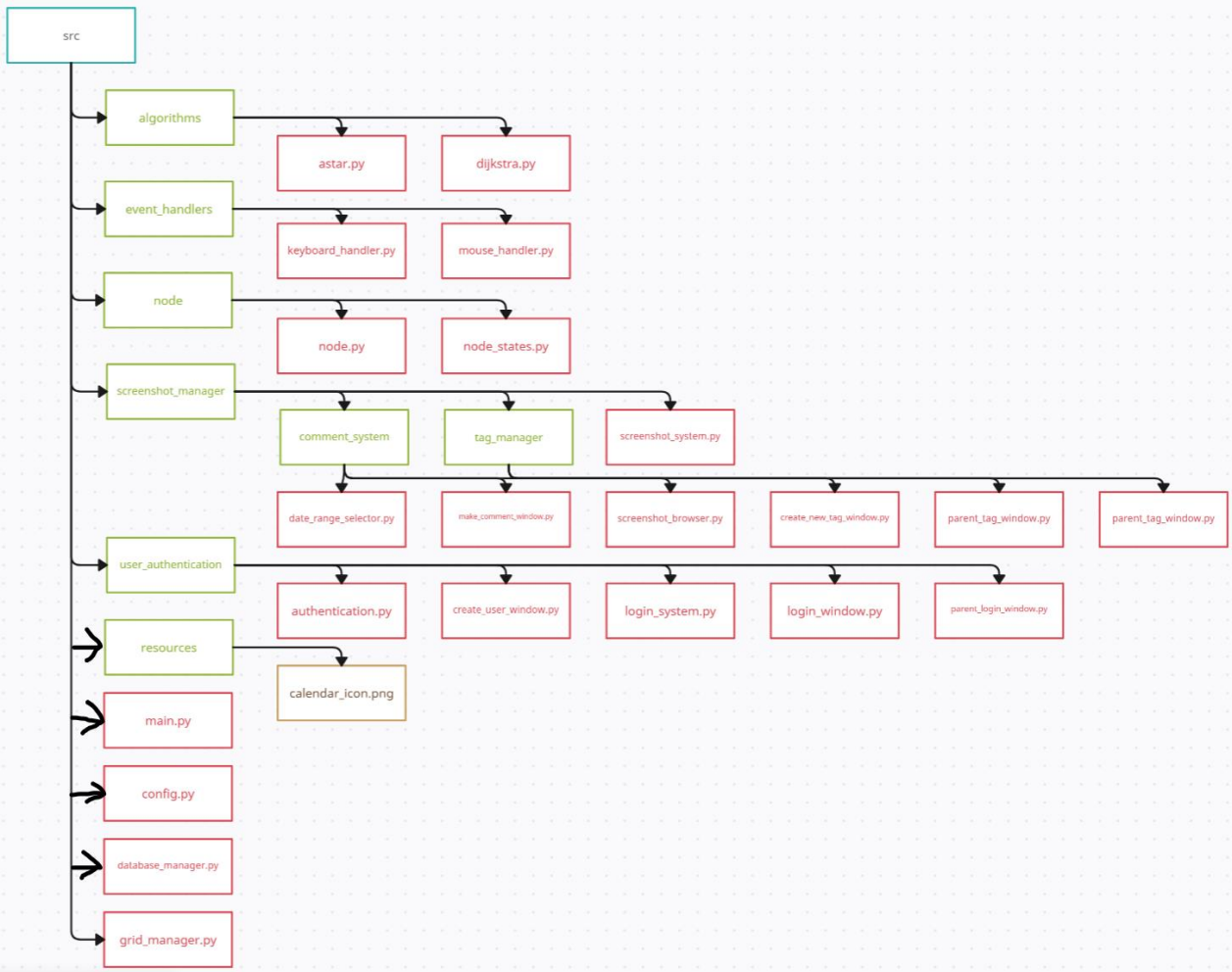

ParentLoginWindow
- user_id: int
- frames: dict
+ __init__()
+ show_frame()
1
CreateUserWindow
- parent: ParentLoginWindow
+ __init__()
+ validate_username()
+ validate_email()
+ validate_password()
+ validate_password_confirmation()
+ register_user()
LoginWindow
- parent: ParentLoginWindow
+ __init__()
+ login()
+ validate_username()
+ validate_email()
+ validate_password()
+ validate_password_confirmation()


ParentTagWindow
- tags: list
- selected_tag: str
+ __init__()
+ show_frame()
+ update_tag_menu()
1
SelectTagWindow
- currently_selected_tag: str
+ __init__()
+ select_tag()
1
CreateNewTagWindow
- controller: ParentTagWindow
+ __init__()
+ create_tag()
+ update_tag_menu()
CreateNewTagWindow
- controller: ParentTagWindow
+ __init__()
+ create_tag()
+ select_tag()



The class diagrams above show the functions of the classes in my program and how they relate to each other.

6.3 Package File Tree Diagram



6.4 Dijkstra's Algorithm

Dijkstra's algorithm can be of great use to my client's needs. Stakeholders, specifically users utilising my client's GPS may find themselves standing in the middle of London. There are many different roads leading to different destinations and the stakeholder wishes to find the local café.

Each road has a certain length, and probably a level of traffic that may make it slower to cross. This would be called the weight of the path to the node. A higher weight means higher traffic or road length. Dijkstra's algorithm helps find the shortest path that the user can take. Our grid is simplified and is not weighted, so every square is considered equidistant with a weight of 1. In real life, this assumption would mean that the squares on the grid each represent the same distance. For a satellite image, this is not terrible because each square may represent 10 metres squared as an example. This is the assumption that we are working with.

The structure we create of the shortest path is a priority queue, which can be visualised as an actual queue in real life. Each node will have a distance and we will be ordering them by that distance to find the shortest path. The shortest distance is always at the front of the queue and after every iteration, this is updated to ensure that it stays true. This allows the algorithm to prioritise the paths that are more likely to produce the shortest path. This process repeats until all the nodes have been visited (no solution) or the target node is found, and with it, the shortest distance.

Dijkstra's Algorithm Process:

1. Mark the start node with a current distance of 0 because that is the distance to itself.
2. Mark the rest of the grid with infinity. Firstly, this would indicate we have not explored the option. Secondly, the option might not even be reachable.
3. Find the neighbours (the four nearest nodes, up, down, left, and right) of the node in our case as they are the next possible steps.
4. Select the neighbour with the smallest current distance as the new current node.
5. For each neighbour, add the weight to the current distance to test which one has the smallest distance and if it is smaller than the distance of that node to the start, update the current distance.
6. Repeat steps 3-5 until the target node is found.

Limitations

As previously mentioned, this algorithm works especially well when the distance between nodes is weighted. The weight could be traffic, speed limits, or the distance of the road. For example, when looking at how to get from London to Manchester, the edge with the lowest weight will likely be the motorway and we can ignore a conglomerate of roads in the surrounding vicinity.

There is no heuristic, so there may be many unnecessarily explored nodes that are not in the direction of the end node. In our example, this would be looking at the motorways going roughly in the right direction.

6.5 A* Search

This algorithm is efficient and effective when we give the start and target node in a map, to find the shortest distance between them. It is an extension of Dijkstra's algorithm that takes the limitation of exploring nodes not in the general direction of the end node and alleviates it by utilising a heuristic function, which guides the search to more efficiently find the target node.

Consider someone who wants to find the shortest path to a monument on the other side of the city that they are standing in. The A* algorithm will find the route by considering both the actual distance travelled and the heuristic estimating the remaining distance to the target node. This way the algorithm can tell if it is getting closer to further away from the end node and decide which path is worth it to explore first.

At every update, the A* search algorithm picks a node according to its 'f' value which is the sum of 'g' and 'h'. It picks the node having the lowest 'f' as this is deemed the shortest route and then processes it. In my case, it turns red to indicate it has been explored.

$$f(n) = g(n) + h(n)$$

1. **n** – n is the previous node on the path. It begins with the start node and progressively explores outwards.
2. **g(n)** – g(n) is the cost of the path from the start node to n.
3. **h(n)** – h(n) is a heuristic that estimates the cost of the cheapest path from n to the target node.

A* Algorithm Process:

1. Start by making the start node cost 0 as we are already there.
2. Assign an estimated cost to reach the goal for each node using the heuristic function.
3. Add the start node to the priority queue.
4. Pop the node with the lowest total cost from the priority queue.
5. If the popped node is the target node, the shortest path has been found.
6. Otherwise, expand the popped node by considering its neighbours (nodes up, down, left and to the right of the current node) and evaluate its cost.
7. Update the total cost of each neighbour by adding the actual cost and the heuristic estimate.
8. Add unexplored neighbours to the priority queue, which is sorted shortest distance first to the largest distance last.
9. Repeat steps 4-8 util the target node is reached, or the priority queue is empty and thus there is no solution.

This solution avoids exploring a lot of irrelevant nodes and often is faster than Dijkstra's algorithm due to exploring the more likely direction to find a path first.

Limitations

The efficiency heavily depends on the heuristic function. This is not easy to define. In this project, I will be using the Manhattan distance, which is the diagonal length from the current node to the target node. This is a good starting point, that is industry standard and anything more complex can be made by the dev team.

In very large maps, the heuristic function may not be as efficient and could lead to many unnecessary explored nodes.

Development (Technical Solution)

main.py

```
import pygame

from config import WIDTH, HEIGHT, ROWS
from database_manager import create_database
from event_handlers.keyboard_handler import handle_keyboard_events
from event_handlers.mouse_handler import handle_mouse_events
from grid_manager import *
from user_authentication.login_system import handle_login


def visualiser(visualiser_window, WIDTH, user_id):
    grid = make_grid(ROWS, WIDTH)

    start = None
    end = None

    run = True
    started = False
    while run:
        draw(visualiser_window, grid, ROWS, WIDTH)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            if started:  # User cannot change grid during visualisation
                continue

            start, end = handle_mouse_events(grid, start, end)
            grid, start, end = handle_keyboard_events(
                    visualiser_window, user_id, event, grid, start, end
            )

    pygame.quit()


if __name__ == "__main__":
    create_database()
    user_id = handle_login()

    if user_id:  # In case the user does not log in but just closes the
window
        visualiser_window = pygame.display.set_mode((WIDTH, HEIGHT))
        pygame.display.set_caption("Pathfinding Visualiser")
        visualiser(visualiser_window, WIDTH, user_id)
```

Here we can see that upon running the program, we create the database as we will see later on, only if the database or table does not exist so that the rest of the code can run without SQL errors.

We then log the user in so that we can open the visualiser.

database_manager.py

```
import datetime
import os
import sqlite3
from tkinter import messagebox

import pygame

from config import SCREENSHOT_DIR, DATABASE_FILE


def create_database():
    conn = sqlite3.connect(DATABASE_FILE)
    c = conn.cursor()

    # Create users Table
    c.execute("""CREATE TABLE IF NOT EXISTS users (
                        user_id INTEGER PRIMARY KEY,
                        username TEXT UNIQUE,
                        email TEXT UNIQUE,
                        password_hash TEXT
                    )""")

    # Create screenshots Table
    c.execute("""CREATE TABLE IF NOT EXISTS screenshots (
                        screenshot_id INTEGER PRIMARY KEY,
                        user_id INTEGER,
                        timestamp TEXT,
                        screenshot_data BLOB,
                        FOREIGN KEY (user_id) REFERENCES users(user_id)
                    )""")

    # Create tags Table
    c.execute("""CREATE TABLE IF NOT EXISTS tags (
                        tag_id INTEGER PRIMARY KEY,
                        tag_name TEXT UNIQUE
                    )""")

    # Create screenshot_tags Junction Table
    c.execute("""CREATE TABLE IF NOT EXISTS screenshot_tags (
                        screenshot_id INTEGER,
                        tag_id INTEGER,
                        PRIMARY KEY (screenshot_id, tag_id),
                        FOREIGN KEY (screenshot_id) REFERENCES
screenshots(screenshot_id),
                        FOREIGN KEY (tag_id) REFERENCES tags(tag_id)
                    )""")

    # Create comments Table
    c.execute("""CREATE TABLE IF NOT EXISTS comments (
                        comment_id INTEGER PRIMARY KEY,
                        user_id INTEGER,
                        screenshot_id INTEGER,
                        timestamp TEXT,
```

```
                        comment_text TEXT,
                        FOREIGN KEY (user_id) REFERENCES users(user_id),
                        FOREIGN KEY (screenshot_id) REFERENCES
screenshots(screenshot_id)
                    )""")

    conn.commit()

    c.close()
    conn.close()

def save_screenshot(user_id, tag):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    screenshot_path = os.path.join(SCREENSHOT_DIR, f"{timestamp}.png")

    if not os.path.exists(SCREENSHOT_DIR):
        os.makedirs(SCREENSHOT_DIR)

    pygame.image.save(pygame.display.get_surface(), screenshot_path)

    # Save the screenshot to the database
    conn = sqlite3.connect(DATABASE_FILE)
    c = conn.cursor()

    with open(screenshot_path, 'rb') as f:
        screenshot_blob = f.read()

    c.execute(
        "INSERT INTO screenshots (user_id, timestamp, screenshot_data)
VALUES (?, ?, ?)",
        (user_id, timestamp, sqlite3.Binary(screenshot_blob))
    )
    screenshot_id = c.lastrowid
    conn.commit()

    c.execute("SELECT tag_id FROM tags WHERE tag_name = ?", (tag,))
    tag_id_row = c.fetchone()
    tag_id = tag_id_row[0]

    c.execute(
            "INSERT INTO screenshot_tags (screenshot_id, tag_id) VALUES
(?, ?)",
            (screenshot_id, tag_id)
    )
    conn.commit()

    c.close()
    conn.close()

    return screenshot_id


def add_tag(tag_name):
    conn = sqlite3.connect(DATABASE_FILE)
```

```
    c = conn.cursor()

    try:
        c.execute("SELECT tag_name FROM tags WHERE tag_name=?",
(tag_name,))
        existing_tag = c.fetchone()

        if existing_tag:
            messagebox.showerror("Error", "Tag already exists")
            return False

        c.execute("INSERT INTO tags (tag_name) VALUES (?)", (tag_name,))
        conn.commit()
    except sqlite3.Error as error:
        messagebox.showerror("Error", "Tag creation failed")
        return False
    finally:
        c.close()
        conn.close()

    return True


def upload_comment(comment_text, user_id, screenshot_id):
    try:
        conn = sqlite3.connect(DATABASE_FILE)
        c = conn.cursor()

        current_timestamp = datetime.datetime.now().strftime(
                '%Y-%m-%d %H:%M:%S'
        )

        c.execute("""
            INSERT INTO comments (user_id, screenshot_id, timestamp,
comment_text)
            VALUES (?, ?, ?, ?)
        """, (user_id, screenshot_id, current_timestamp, comment_text))

        conn.commit()
    except sqlite3.Error as e:
        messagebox.showerror("Error", "Error while uploading comment")
    finally:
        c.close()
        conn.close()


def find_photo_by_id(photo_id):
    try:
        conn = sqlite3.connect(DATABASE_FILE)
        c = conn.cursor()

        # Retrieve photo details, time taken, associated tag name, and
comments with username based on photo_id
        c.execute("""
```

```
            SELECT screenshots.timestamp, tags.tag_name,
screenshots.screenshot_data, users.username, comments.timestamp,
comments.comment_text
            FROM screenshots
            JOIN screenshot_tags ON screenshots.screenshot_id =
screenshot_tags.screenshot_id
            JOIN tags ON screenshot_tags.tag_id = tags.tag_id
            LEFT JOIN comments ON screenshots.screenshot_id =
comments.screenshot_id
            LEFT JOIN users ON comments.user_id = users.user_id
            WHERE screenshots.screenshot_id = ?

        """, (photo_id,))
        results = c.fetchall()

        if results:
            time_taken, tag_name, screenshot_blob = results[0][:3]
            comments = [
                (
                    username,
                    comment_timestamp,
                    comment_text
                ) for _, _, _, username, comment_timestamp, comment_text
in results if comment_text
            ]
            return screenshot_blob, time_taken, tag_name, comments
        else:
            # Handle the case where the photo with the given ID doesn't
exist
            return None, None, None, None
    except sqlite3.Error as e:
        messagebox.showerror("Error", "Error while retrieving photo
details")
        print(e)
        return None, None, None, None
    finally:
        c.close()
        conn.close()


def get_valid_screenshots(tag_option, user_option, start_date, end_date):
    conn = sqlite3.connect(DATABASE_FILE)
    c = conn.cursor()

    query = "SELECT screenshots.screenshot_id, screenshots.user_id,
users.username, screenshots.timestamp, tags.tag_id, tags.tag_name " \
            "FROM screenshots " \
            "INNER JOIN users ON screenshots.user_id = users.user_id " \
            "INNER JOIN screenshot_tags ON screenshot_tags.screenshot_id =
screenshots.screenshot_id " \
            "INNER JOIN tags ON tags.tag_id = screenshot_tags.tag_id " \
            "WHERE 1=1"

    params = []
```

```
    if tag_option != "All":
        tag_id, tag_name = tag_option.split(": ")
        query += " AND tags.tag_id = ?"
        params.append(tag_id)

    if user_option != "All":
        user_id, username = user_option.split(": ")
        query += " AND users.user_id = ?"
        params.append(user_id)

    if start_date and end_date:
        start_date_formatted = start_date + "_00-00-00"
        end_date_formatted = end_date + "_23-59-59"
        query += " AND screenshots.timestamp BETWEEN ? AND ?"
        params.extend([start_date_formatted, end_date_formatted])

    c.execute(query, params)
    results = c.fetchall()

    c.close()
    conn.close()

    return results


def get_tags():
    conn = sqlite3.connect(DATABASE_FILE)
    c = conn.cursor()
    c.execute("SELECT tag_id, tag_name FROM tags")
    tags = c.fetchall()
    conn.close()
    return tags


def get_users():
    conn = sqlite3.connect(DATABASE_FILE)
    c = conn.cursor()
    c.execute("SELECT user_id, username FROM users")
    usernames = c.fetchall()
    conn.close()
    return usernames
```

In the `create_database` function at the top, we use “CREATE TABLE IF NOT EXISTS” to ensure that if there is already a database I do not accidentally create duplicate data or erase existing data.

The “save_screenshot” function gets the current timestamp so that the time the screenshot was taken can be uploaded to the database. It saves the screenshot into a file using pygame’s save display function so that it can be read into a database and the user may want a folder of the images locally even though it is also being saved onto the database. First, it uploads the screenshot into the `screenshots` table and then it queries the `tags` database for the ID of the given tag so that it can finally upload the tag ID and screenshot ID into the `screenshot_tags` database for a composite primary key that is both foreign also. It then returns the screenshot ID it just saved, this is because as will be explained later it will be needed

for the comment system if the user wants to optionally make a comment also on the screenshot that was just saved.

The `add_tag` function uses try and except causes to catch errors as the arguments given to it come from user input which might be invalid, so we must validate it here using table-lookup. If the tag already exists we must raise an error to the user as our requirements defined earlier dictate. Two tags with the same name would be duplicated, redundant data so to keep the database normalised to the third normal form we must ensure that we validate it here. If the tag does not exist already, it is uploaded to the `tags` database. Upon it being successfully uploaded I return True.

The `upload_comment` function is simple in that it queries the database to add the comment the user made. The parameter has already been validated where this function is used so that the comment text is not empty and thus redundant so there was no need to add it. If there is some sort of connection error then an error is raised to the user so that they know that their comment was not uploaded and the admin can debug the issue using third-party software they have permission to use for debugging purposes.

The `find_photo_by_id` function plays a pivotal role in the comment system as we will see later. It selects all the relevant information by combining multiple tables such as the screenshot tags and tags and comments and users to correctly query all the data that is required. This is simple since the database is normalised, a benefit is SQL queries that are not confusing or producing jumbled data. This data is returned by the function.

The `get_valid_screenshots` function allows a simple filtering mechanism from the database file based on its parameters to return relevant information for every screenshot that meets the requirements. We can see a common query condition “WHERE 1=1” which is typically used to easily append to the selection criteria if we need to for example, `query += “ AND tags.tag_id = ?”`. It returns the results of the query, which would be the screenshots that meet the requirements.

The `get_tags` function simply fetches all the tags in the database and returns them. This is later used in the select tag window in its option menu.

The `get_users` function simply fetches all the users in the database and returns them. This is used in the browse window later.

config.py

```
DATABASE_FILE = 'screenshot_database.db'
SCREENSHOT_DIR = 'screenshots'

WIDTH = 800
HEIGHT = 800

ROWS = 50


class Colors:
    ISEMPTY_COLOR = (255, 255, 255)  # White
    ISBLOCKED_COLOR = (255, 0, 0)  # Red
    ISOPEN_COLOR = (0, 255, 0)  # Green
    ISBARRIER_COLOR = (0, 0, 0)  # Black
    ISSTART_COLOR = (255, 165, 0)  # Orange
    ISTARGET_COLOR = (64, 224, 208)  # Turquoise
    ISPATH_COLOR = (128, 0, 128)  # Purple
    GRIDLINES_COLOR = (128, 128, 128)  # Grey
```

This configuration file will allow the software team to put in their requirements as well as upload the solution live on their Microsoft Azure services.

The database file is where the company can hook up their Azure solution if they decide to go through with it. For the sake of this project, we will just be saving it to the same directory as the source root “src”.

The width and the height refer to the pygame window and this can be configured for the shapes and sizes of satellite images they will be modelling.

The rows refer to the pygame grid and this can be configured for the shapes and sizes of satellite images they will be modelling.

The colours can be changed to suit their needs but I have commented what is the default.

login_system.py

```
from user_authentication.parent_login_window import ParentLoginWindow


def handle_login():
    parent_login_window = ParentLoginWindow()
    parent_login_window.mainloop()
    return parent_login_window.user_id
```

One function which returns the user ID of the person who just logged in.

parent_login_window.py

```
import tkinter as tk

from user_authentication.create_user_window import CreateUserWindow
from user_authentication.login_window import LoginWindow


class ParentLoginWindow(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)
        self.title("Login System")

        self.user_id = None

        container = tk.Frame(self)
        container.pack()
        container.grid_rowconfigure(0, weight=1)
        container.grid_rowconfigure(0, weight=1)

        self.frames = {}

        for F in (LoginWindow, CreateUserWindow):
            frame = F(container, self)
            self.frames[F] = frame
            frame.grid(row=0, column=0, sticky="nsew")

        self.show_frame("LoginWindow")

    def show_frame(self, window_name):
        frame_names = {
            "LoginWindow": LoginWindow,
            "CreateUserWindow": CreateUserWindow
        }
        window = frame_names[window_name]
        frame = self.frames[window]
        frame.tkraise()
```

Stores the two different frames that will be switched between in this one window.

Stores also the user ID for them to be accessed by both its children. We see a clear case of the use of inheritance here and a structured OOP form being used.

login_window.py

```
import tkinter as tk
from tkinter import messagebox

from user_authentication.authentication import verify_login


class LoginWindow(tk.Frame):
    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)
        self.controller = controller

        # Username label and entry
        self.username_label = tk.Label(self, text="Username:")
        self.username_label.grid(row=0, column=0, padx=10, pady=5,
sticky="e")
        self.username_entry = tk.Entry(self, width=30)
        self.username_entry.grid(row=0, column=1, padx=10, pady=5)

        # Password label and entry
        self.password_label = tk.Label(self, text="Password:")
        self.password_label.grid(row=1, column=0, padx=10, pady=5,
sticky="e")
        self.password_entry = tk.Entry(self, show="*", width=30)
        self.password_entry.grid(row=1, column=1, padx=10, pady=5)

        # Login button
        self.login_button = tk.Button(
                self, text="Login", command=self.login, width=10
        )
        self.login_button.grid(
                row=2, column=0, columnspan=2, padx=10, pady=5,
sticky="ew"
        )

        # Switch to create new user button
        self.signup_button = tk.Button(
                self,
                text="Sign Up",
                command=lambda: controller.show_frame("CreateUserWindow"),
                width=10
        )
        self.signup_button.grid(
                row=3, column=0, columnspan=2, padx=10, pady=5,
sticky="ew"
        )

    def login(self):
        username = self.username_entry.get()
        password = self.password_entry.get()

        if not username or not password:
            messagebox.showerror(
                    "Error",
```

```
                    "Please enter both username and password"
            )
            return

        user_id = verify_login(username, password)
        if user_id:
            messagebox.showinfo("Success", "Login successful")
            self.controller.user_id = user_id
            self.controller.destroy()
        else:
            messagebox.showerror(
                    "Error", "Invalid username or password"
            )
```

On initialising the frame, the widgets are created.

We can see how there is a function call to switch to the “Create New User” frame upon pressing “Create New User” just as the structure diagram described.

The `login` function validates that the user has typed in a username and password and then calls `verify_login` validating the username and password via a table lookup. We can also see the default tkinter message box being used to show visual error messages for the user indicating a failure and then giving the specific reason for the error.

create_user_window.py

```
import re
import tkinter as tk
from tkinter import messagebox

from user_authentication.authentication import create_user


class CreateUserWindow(tk.Frame):
    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)
        self.controller = controller

        # Username label and entry
        self.username_label = tk.Label(self, text="Username:")
        self.username_label.grid(row=0, column=0, padx=10, pady=5,
sticky="e")
        self.username_entry = tk.Entry(self)
        self.username_entry.grid(row=0, column=1, padx=10, pady=5)

        # Email label and entry
        self.email_label = tk.Label(self, text="Email:")
        self.email_label.grid(row=1, column=0, padx=10, pady=5,
sticky="e")
        self.email_entry = tk.Entry(self)
        self.email_entry.grid(row=1, column=1, padx=10, pady=5)

        # Password label and entry
        self.password_label = tk.Label(self, text="Password:")
        self.password_label.grid(row=2, column=0, padx=10, pady=5,
sticky="e")
        self.password_entry = tk.Entry(self, show="*")
        self.password_entry.grid(row=2, column=1, padx=10, pady=5)

        # Confirm password label and entry
        self.confirm_password_label = tk.Label(self, text="Confirm
Password:")
        self.confirm_password_label.grid(row=3, column=0, padx=10, pady=5,
                                         sticky="e")
        self.confirm_password_entry = tk.Entry(self, show="*")
        self.confirm_password_entry.grid(row=3, column=1, padx=10, pady=5)

        # Register button
        self.register_button = tk.Button(self, text="Register",
                                         command=self.register_user)
        self.register_button.grid(row=4, column=0, columnspan=2, padx=10,
                                  pady=5, sticky="ew")

        # Switch to login button
        self.login_button = tk.Button(
                self,
                text="Login",
                command=lambda: self.controller.show_frame("LoginWindow")
        )
```

```
        self.login_button.grid(row=5, column=0, columnspan=2, padx=10,
pady=5,
                               sticky="ew")

    def validate_username(self, username):
        # Check if username is not empty
        if not username:
            messagebox.showerror("Error", "Please enter a username")
            return False
        # Additional checks can be added here, such as length requirements
        return True

    @staticmethod
    def validate_email(email):
        # Regular expression for validating email format
        email_regex = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
        # Check if email matches the pattern
        if not re.match(email_regex, email):
            messagebox.showerror("Error", "Please enter a valid email
address")
            return False
        return True

    @staticmethod
    def validate_password(password):
        # Check if password is not empty
        if not password:
            messagebox.showerror("Error", "Please enter a password")
            return False
        # Check if password meets basic requirements
        if len(password) < 8:
            messagebox.showerror(
                    "Error", "Password must be at least 8 characters long"
            )
            return False
        if not any(char.isdigit() for char in password):
            messagebox.showerror(
                    "Error", "Password must contain at least one digit"
            )
            return False
        if not any(not char.isalnum() for char in password):
            messagebox.showerror(
                    "Error",
                    "Password must contain at least one special character"
            )
            return False
        return True

    @staticmethod
    def validate_password_confirmation(password, confirm_password):
        # Check if passwords match
        if password != confirm_password:
            messagebox.showerror("Error", "Passwords do not match")
            return False
```

```
        return True

    def register_user(self):
        username = self.username_entry.get()
        email = self.email_entry.get()
        password = self.password_entry.get()
        confirm_password = self.confirm_password_entry.get()

        # Validate username, email, and password
        if (
                not self.validate_username(username) or
                not self.validate_email(email) or
                not self.validate_password(password) or
                not self.validate_password_confirmation(
                        password,
                        confirm_password
                )
        ):
            return

        # Perform user registration
        created_new_user = create_user(username, email, password)
        if created_new_user:
            messagebox.showinfo("Success", "User created")
            self.controller.show_frame("LoginWindow")
        else:
            messagebox.showerror("Error", "Username or Email already
taken.")
```

Again we see widgets created when initialised.

I validate the username by ensuring that the user typed something.

I validate the email address using a common regex check that is widely known to follow the structure of an email address.

I validate the password by checking if it has been typed. Another validation is performed with whether it is greater than seven characters for security purposes as those are harder to crack. It will also validate that there is a number and a special character minimum in the password.

The password confirmation is also validated to ensure it is the same typed password so that the user does not make an accidental mistake in typing their password.

Once all the widgets have been validated, the `create_user` function uploads it to the database.

authentication.py

```
import sqlite3
import hashlib

from config import DATABASE_FILE


def create_user(username, email, password):
    conn = sqlite3.connect(DATABASE_FILE)
    c = conn.cursor()

    c.execute(
        "SELECT * FROM users WHERE username = ? OR email = ?",
        (username, email)
    )
    existing_user = c.fetchone()
    if existing_user:
        c.close()
        conn.close()
        return False

    hashed_password = hash_password(password)

    c.execute(
        "INSERT INTO users (username, email, password_hash) VALUES (?, ?,
?)",
        (username, email, hashed_password)
    )
    conn.commit()
    conn.close()
    return True

def verify_login(username, password):
    conn = sqlite3.connect(DATABASE_FILE)
    c = conn.cursor()

    c.execute(
        "SELECT user_id, password_hash FROM users WHERE username = ?",
        (username,)
    )
    user_data = c.fetchone()
    conn.close()

    if user_data:
        user_id, hashed_password = user_data
        if hashed_password == hash_password(password):
            return user_id

def hash_password(password):
    # Hash the password using SHA-256
    return hashlib.sha256(password.encode('utf-8')).hexdigest()
```

We are using SHA256 as it is an unbreakable encryption, thus the users will be safe from being hacked.

grid_manager.py

```
import pygame

from node.node import Node
from config import Colors


def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            node = Node(i, j, gap, gap, rows)
            grid[i].append(node)

    return grid


def draw_grid(win, rows, width):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(
                win, Colors.GRIDLINES_COLOR, (0, i * gap), (width, i *
gap)
        )
        for j in range(rows):
            pygame.draw.line(
                    win, Colors.GRIDLINES_COLOR, (j * gap, 0), (j * gap,
width)
            )


def draw(win, grid, rows, width):
    win.fill(Colors.ISEMPTY_COLOR)

    for row in grid:
        for node in row:
            node.draw(win)

    draw_grid(win, rows, width)
    pygame.display.update()
```

I create the grid based on the configuration of the number of rows and width of the window.

node.py

```
import pygame

from src.node.node_states import NodeStates
from src.config import Colors


class Node:
    def __init__(self, row, col, width, height, total_rows):
        self.row = row
        self.col = col
        self.x = row * width
        self.y = col * height
        self.width = width
        self.height = height
        self.total_rows = total_rows
        self.color = Colors.ISEMPTY_COLOR
        self.neighbors = []
        self.total_rows = total_rows

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

    def get_state(self):
        match self.color:
            case Colors.ISEMPTY_COLOR:
                return NodeStates.ISEMPTY
            case Colors.ISBLOCKED_COLOR:
                return NodeStates.ISBLOCKED
            case Colors.ISOPEN_COLOR:
                return NodeStates.ISOPEN
            case Colors.ISBARRIER_COLOR:
                return NodeStates.ISBARRIER
            case Colors.ISSTART_COLOR:
                return NodeStates.ISSTART
            case Colors.ISTARGET_COLOR:
                return NodeStates.ISTARGET
            case Colors.ISPATH_COLOR:
                return NodeStates.ISPATH
            case _:
                raise ValueError(f"Invalid color: {self.color}")

    def set_state(self, state):
        match state:
            case NodeStates.ISEMPTY:
                self.color = Colors.ISEMPTY_COLOR
            case NodeStates.ISBLOCKED:
                self.color = Colors.ISBLOCKED_COLOR
            case NodeStates.ISOPEN:
                self.color = Colors.ISOPEN_COLOR
            case NodeStates.ISBARRIER:
                self.color = Colors.ISBARRIER_COLOR
            case NodeStates.ISSTART:
                self.color = Colors.ISSTART_COLOR
```

```
            case NodeStates.ISTARGET:
                self.color = Colors.ISTARGET_COLOR
            case NodeStates.ISPATH:
                self.color = Colors.ISPATH_COLOR
            case _:
                raise ValueError(f"Invalid state: {state}")

    def draw(self, win):
        pygame.draw.rect(
                win, self.color, (self.x, self.y, self.width, self.height)
        )

    def update_neighbors(self, grid):
        self.neighbors = []
        if self.row < self.total_rows - 1 \
                and grid[self.row + 1][self.col].get_state() \
                != NodeStates.ISBARRIER:  # DOWN
            self.neighbors.append(grid[self.row + 1][self.col])

        if self.row > 0 \
                and grid[self.row - 1][self.col].get_state() \
                != NodeStates.ISBARRIER:  # UP
            self.neighbors.append(grid[self.row - 1][self.col])

        if self.col < self.total_rows - 1 \
                and grid[self.row][self.col + 1].get_state() \
                != NodeStates.ISBARRIER:  # RIGHT
            self.neighbors.append(grid[self.row][self.col + 1])

        if self.col > 0 \
                and grid[self.row][self.col - 1].get_state() \
                != NodeStates.ISBARRIER:  # LEFT
            self.neighbors.append(grid[self.row][self.col - 1])

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

Use of a match statement to get the state and to set the state of the node. This affects the colour of the node.

The `update_neighbors` function returns a node's neighbours, which will be used in the pathfinding algorithm. Its name is in American to follow the programming standard that code is done in American English.

node_states.py

```
from enum import IntEnum, auto


class NodeStates(IntEnum):
    ISEMPTY = auto()
    ISBLOCKED = auto()
    ISOPEN = auto()
    ISBARRIER = auto()
    ISSTART = auto()
    ISTARGET = auto()
    ISPATH = auto()
```

Use of an Enum for the match statements in `node.py`.

Increases readability of the code for other developers if they decide to go in and tinker with the program.

mouse_handler.py

```
import pygame

from config import ROWS, WIDTH
from node.node_states import NodeStates


def handle_mouse_events(grid, start, end):
    pos = pygame.mouse.get_pos()
    row, col = get_clicked_pos(pos, ROWS, WIDTH)
    node = grid[row][col]

    mouse_events = pygame.mouse.get_pressed()
    if mouse_events[0]:
        if not start and node != end:
            start = node
            start.set_state(NodeStates.ISSTART)
        elif not end and node != start:
            end = node
            end.set_state(NodeStates.ISTARGET)
        elif node != start and node != end:
            node.set_state(NodeStates.ISBARRIER)
    elif mouse_events[2]:
        node.set_state(NodeStates.ISEMPTY)
        if node == start:
            start = None
        elif node == end:
            end = None

    return start, end


def get_clicked_pos(pos, rows, width):
    gap = width // rows
    y, x = pos

    row = y // gap
    col = x // gap

    return row, col
```

We can see that the start node and then the end node will be placed. After this, walls can be placed.

There is a validation check to ensure that a wall is not on a start node or end node. There is also a check so that the end node is not placed on the start node.

keyboard_handler.py

```
from tkinter import messagebox

import pygame

from algorithms.astar import astar
from algorithms.dijkstra import dijkstra
from database_manager import get_tags, save_screenshot
from grid_manager import draw, make_grid
from config import ROWS, WIDTH
from screenshot_manager.comment_system.make_comment_window import
MakeNewCommentWindow
from screenshot_manager.screenshot_system import handle_select_tag


def handle_keyboard_events(
        visualiser_window,
        user_id,
        event,
        grid,
        start,
        end
):
    if event.type != pygame.KEYDOWN:
        return grid, start, end

    match event.key:
        case pygame.K_a:  # Perform astar algorithm
            if start and end:
                for row in grid:
                    for node in row:
                        node.update_neighbors(grid)
                astar(
                        lambda: draw(visualiser_window, grid, ROWS,
WIDTH),
                        grid,
                        start,
                        end
                )

        case pygame.K_d:  # Perform Dijkstra's algorithm
            if start and end:
                for row in grid:
                    for node in row:
                        node.update_neighbors(grid)
                dijkstra(
                        lambda: draw(visualiser_window, grid, ROWS,
WIDTH),
                        grid,
                        start,
                        end
                )
```

```
        case pygame.K_e:  # Empty the grid
            start = None
            end = None
            grid = make_grid(ROWS, WIDTH)

        case pygame.K_s:  # Screenshot the grid
            tags = [tag[1] for tag in get_tags()]
            tag = handle_select_tag(tags)
            if tag:
                screenshot_id = save_screenshot(user_id, tag)
                MakeNewCommentWindow(user_id, screenshot_id).mainloop()
            else:
                messagebox.showerror(
                        "Screenshot Cancelled",
                        "Screenshot was not saved as no tag was provided"
                )

        case pygame.K_c:  # Make a comment on a screenshot the user took
            MakeNewCommentWindow(user_id).mainloop()

    return grid, start, end
```

Here are the keys that have functionality:

1. **"a"** – Hitting the "a" key will visualise the A* search algorithm. The user will not be able to edit the map whilst the shortest path is being found.
2. **"c"** – Hitting the "c" key will open the add new comment system, allowing the user to search for a screenshot if they wish and make a comment on it. Perfect for someone who does not want to visualise something but goes straight into commenting.
3. **"d"** – Hitting the "d" key will visualise Dijkstra's algorithm. The user will not be able to edit the map whilst the shortest path is being found.
4. **"e"** – Hitting the "e" key will empty the grid. It resets every square as if the program had just opened. This allows the user to visualise multiple shortest paths in succession.
5. **"s"** – Hitting the "s" key will open the tag select system as it stands for "screenshot"/"save", so this is the first step in uploading their visualised path/map.

astar.py

```
from queue import PriorityQueue

import pygame

from node.node_states import NodeStates


def heuristic(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)


def reconstruct_path(path_history, current, draw):
    while current in path_history:
        current = path_history[current]
        current.set_state(NodeStates.ISPATH)
        draw()


def astar(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    path_history = {}
    g_score = {node: float("inf") for row in grid for node in row}
    g_score[start] = 0
    f_score = {node: float("inf") for row in grid for node in row}
    f_score[start] = heuristic(start.get_pos(), end.get_pos())

    open_set_hash = {start}

    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(path_history, end, draw)
            start.set_state(NodeStates.ISSTART)
            end.set_state(NodeStates.ISTARGET)
            return True

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1

            if temp_g_score < g_score[neighbor]:
                path_history[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + heuristic(
```

```
                        neighbor.get_pos(), end.get_pos()
                    )
                    if neighbor not in open_set_hash:
                        count += 1
                        open_set.put((f_score[neighbor], count, neighbor))
                        open_set_hash.add(neighbor)
                        neighbor.set_state(NodeStates.ISOPEN)

            draw()

            if current != start:
                current.set_state(NodeStates.ISBLOCKED)
```

dijkstra.py

```
import pygame
from queue import PriorityQueue

from node.node_states import NodeStates


def dijkstra(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    path_history = {}
    g_score = {node: float("inf") for row in grid for node in row}
    g_score[start] = 0

    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        current = open_set.get()[2]

        if current == end:
            reconstruct_path(path_history, end, draw)
            start.set_state(NodeStates.ISSTART)
            end.set_state(NodeStates.ISTARGET)
            return True

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1

            if temp_g_score < g_score[neighbor]:
                path_history[neighbor] = current
                g_score[neighbor] = temp_g_score
                if neighbor not in [item[2] for item in open_set.queue]:
                    count += 1
                    open_set.put((g_score[neighbor], count, neighbor))
                    neighbor.set_state(NodeStates.ISOPEN)

        draw()

        if current != start:
            current.set_state(NodeStates.ISBLOCKED)

    return False

def reconstruct_path(path_history, current, draw):
    while current in path_history:
        current = path_history[current]
        current.set_state(NodeStates.ISPATH)
        draw()
```

screenshot_system.py

```
from screenshot_manager.tag_manager.parent_tag_window import
ParentTagWindow


def handle_select_tag(tags):
    parent_tag_window = ParentTagWindow(tags)
    parent_tag_window.mainloop()
    return parent_tag_window.selected_tag
```

One function which returns the tag selected by the user.

parent_tag_window.py

```
import tkinter as tk

from screenshot_manager.tag_manager.create_new_tag_window import
CreateNewTagWindow
from screenshot_manager.tag_manager.select_tag_window import
SelectTagWindow


class ParentTagWindow(tk.Tk):
    def __init__(self, tags):
        tk.Tk.__init__(self)
        self.title("Tag System")
        self.tags = tags

        self.selected_tag = None

        container = tk.Frame(self)
        container.pack()
        container.grid_rowconfigure(0, weight=1)
        container.grid_rowconfigure(0, weight=1)

        self.frames = {}

        for F in (SelectTagWindow, CreateNewTagWindow):
            frame = F(container, self)
            self.frames[F] = frame
            frame.grid(row=0, column=0, sticky="nsew")

        self.show_frame("SelectTagWindow")

    def show_frame(self, window_name):
        frame_names = {
            "SelectTagWindow": SelectTagWindow,
            "CreateNewTagWindow": CreateNewTagWindow
        }
        window = frame_names[window_name]
        frame = self.frames[window]
        frame.tkraise()

    def update_tag_menu(self):
        tag_menu = self.frames[SelectTagWindow].tag_dropdown["menu"]
```

```
        new_tag = self.tags[-1]
        tag_menu.add_command(
                label=new_tag,
                command=tk._setit(

self.frames[SelectTagWindow].currently_selected_tag,
                        new_tag
                )
        )
```

Stores the two different frames that will be switched between in this one window.

Stores also the selected tag for it to be accessed by its child and after it is closed to still be accessible to get the selected tag that we are looking for. We see a clear case of the use of inheritance here and a structured OOP form being used.

select_tag_window.py

```
import tkinter as tk
from tkinter import messagebox


class SelectTagWindow(tk.Frame):
    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)
        self.controller = controller

        self.currently_selected_tag = tk.StringVar()
        self.currently_selected_tag.set(
                self.controller.tags[0] if self.controller.tags else ""
        )

        label = tk.Label(self, text="Select Tag:")
        label.grid(row=0, column=0, padx=10, pady=5)

        self.tag_dropdown = tk.OptionMenu(
                self,
                self.currently_selected_tag,
                self.currently_selected_tag.get(),
                *self.controller.tags[1:]
        )
        self.tag_dropdown.grid(row=0, column=1, padx=10, pady=5)

        select_button = tk.Button(self, text="Select Tag",
                                  command=self.select_tag)
        select_button.grid(row=1, column=0, columnspan=2, padx=10, pady=5)

        create_new_tag_button = tk.Button(
                self,
                text="Create New Tag",
                command=lambda: self.controller.show_frame(
                        "CreateNewTagWindow"
                )
        )
        create_new_tag_button.grid(
                row=2, column=0, columnspan=2, padx=10, pady=5
        )

    def select_tag(self):
        tag_name = self.currently_selected_tag.get()
        if tag_name:
            messagebox.showinfo("Success", "Tag selected")
            self.controller.selected_tag = tag_name
            self.controller.destroy()
        else:
            messagebox.showerror("Error", "Tag name cannot be empty")
```

Validates if a tag was selected, if not then shows an error for the user and allows them to try and select one again.

create_tag_window.py

```
import tkinter as tk
from tkinter import messagebox

from database_manager import add_tag


class CreateNewTagWindow(tk.Frame):
    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)
        self.controller = controller

        # Tag name label and entry
        self.tag_name_label = tk.Label(self, text="Tag Name:")
        self.tag_name_label.grid(row=0, column=0, padx=10, pady=5,
sticky="e")
        self.tag_name_entry = tk.Entry(self)
        self.tag_name_entry.grid(row=0, column=1, padx=10, pady=5)

        # Create tag button
        self.create_tag_button = tk.Button(self, text="Create Tag",
                                           command=self.create_tag)
        self.create_tag_button.grid(row=1, column=0, columnspan=2,
padx=10,
                                    pady=5, sticky="ew")

        # Switch to select tag button
        self.select_tag_button = tk.Button(
                self,
                text="Select Tag",
                command=lambda:
self.controller.show_frame("SelectTagWindow"))
        self.select_tag_button.grid(row=2, column=0, columnspan=2,
padx=10,
                                    pady=5, sticky="ew")

    def create_tag(self):
        tag_name = self.tag_name_entry.get()

        if not tag_name:
            messagebox.showerror("Error", "Tag name cannot be empty")
            return

        created_new_tag = add_tag(tag_name)
        if created_new_tag:
            self.controller.tags.append(tag_name)
            self.controller.update_tag_menu()
            messagebox.showinfo("Success", "Tag created")
            self.controller.show_frame("SelectTagWindow")
```

The `create_tag` function validates that the user typed a tag name. Then it performs a table lookup validation to ensure that the tag does not already exist. Then, if it is duplicate data, I show an error to the user that the tag already exists and thus the database remains normalised.

make_comment_window.py

```
import io
from PIL import Image, ImageTk
import tkinter as tk
from tkinter import ttk, messagebox

from database_manager import find_photo_by_id, upload_comment
from screenshot_manager.comment_system.screenshot_browser import \
    ScreenshotBrowser


class MakeNewCommentWindow(tk.Tk):
    def __init__(self, user_id, screenshot_id=None):
        super().__init__()
        self.title("Add New Comment")

        self.user_id = user_id
        self.screenshot_id = screenshot_id
        self.screenshot_photo = None

        # Upper Left Section
        upper_left_frame = tk.Frame(self)
        upper_left_frame.grid(row=0, column=0, padx=10, pady=0,
sticky="nsew")
        self.grid_rowconfigure(0,
                               weight=1)
        self.grid_columnconfigure(0,
                                  weight=1)

        photo_id_label = tk.Label(upper_left_frame, text="Photo ID:")
        photo_id_label.grid(row=0, column=0, padx=5, pady=5)
        vcmd = (self.register(self.validate_numeric_input), '%P')
        self.photo_id_entry = tk.Entry(
                upper_left_frame, validate="key", validatecommand=vcmd
        )
        self.photo_id_entry.grid(row=0, column=1, padx=5, pady=5)
        find_button = tk.Button(
                upper_left_frame, text="Find", command=self.find_photo
        )
        find_button.grid(row=0, column=2, padx=5, pady=5)

        selected_photo_label = tk.Label(
                upper_left_frame, text="Selected Photo:"
        )
        selected_photo_label.grid(row=1, column=0, padx=5, pady=5)
        self.selected_photo_label = tk.Label(upper_left_frame, text="")
        self.selected_photo_label.grid(row=1, column=1, padx=5, pady=5)

        time_taken_label = tk.Label(upper_left_frame, text="Time Taken:")
        time_taken_label.grid(row=2, column=0, padx=5, pady=5)
        self.time_taken_label = tk.Label(upper_left_frame,
text="timestamp")
        self.time_taken_label.grid(row=2, column=1, padx=5, pady=5)
        browse_button = tk.Button(
```

```
                upper_left_frame,
                text="Browse...",
                command=self.browse_screenshot
        )
        browse_button.grid(row=2, column=2, padx=5, pady=5)

        tag_label = tk.Label(upper_left_frame, text="Tag:")
        tag_label.grid(row=3, column=0, padx=5, pady=5)
        self.tag_label = tk.Label(upper_left_frame, text="tag_name")
        self.tag_label.grid(row=3, column=1, padx=5, pady=5)

        # Lower Left Section
        lower_left_frame = tk.Frame(self)
        lower_left_frame.grid(row=1, column=0, padx=10, pady=0,
sticky="nsew")
        self.grid_rowconfigure(1, weight=2)

        # Create a canvas widget to contain the image and add scrollbars
        self.canvas = tk.Canvas(lower_left_frame, width=600, height=500,
                                bg="white")
        self.canvas.pack(side="left", fill="both", expand=True)

        self.scrollbar_x = tk.Scrollbar(lower_left_frame,
orient="horizontal",
                                        command=self.canvas.xview)
        self.scrollbar_x.pack(side="bottom", fill="x")
        self.scrollbar_y = tk.Scrollbar(lower_left_frame,
orient="vertical",
                                        command=self.canvas.yview)
        self.scrollbar_y.pack(side="right", fill="y")

        self.canvas.configure(xscrollcommand=self.scrollbar_x.set,
                              yscrollcommand=self.scrollbar_y.set)

        self.canvas.update_idletasks()
        self.canvas.config(scrollregion=self.canvas.bbox("all"))

        # Upper Right Section
        upper_right_frame = tk.Frame(self)
        upper_right_frame.grid(
                row=0, column=1, padx=10, pady=10, sticky="nsew"
        )
        self.grid_columnconfigure(1, weight=1)

        comment_entry_label = tk.Label(
                upper_right_frame, text="Enter Comment:"
        )
        comment_entry_label.pack()
        self.comment_entry = tk.Text(upper_right_frame, height=10,
width=50)
        self.comment_entry.pack()
        comment_button = tk.Button(
                upper_right_frame, text="Comment", command=self.comment
        )
```

```
        comment_button.pack()

        # Lower Right Section
        lower_right_frame = tk.Frame(self)
        lower_right_frame.grid(
                row=1, column=1, padx=10, pady=10, sticky="nsew"
        )

        self.comments_table = ttk.Treeview(lower_right_frame, columns=(
            "Username", "Time Taken", "Comment"), show="headings")
        self.comments_table.heading("Username", text="Username")
        self.comments_table.heading("Time Taken", text="Time Taken")
        self.comments_table.heading("Comment", text="Comment")
        self.comments_table.column("Username", width=65)
        self.comments_table.column("Time Taken", width=125)
        self.comments_table.column("Comment", width=400)
        self.comments_table.pack(side="left", fill="both", expand=True)

        scrollbar_y = ttk.Scrollbar(lower_right_frame, orient="vertical",
                                    command=self.comments_table.yview)
        scrollbar_y.pack(side="right", fill="y")
        self.comments_table.configure(yscrollcommand=scrollbar_y.set)

        self.resizable(False, False)

        if self.screenshot_id:
            self.find_photo(self.screenshot_id)

    @staticmethod
    def validate_numeric_input(new_value):
        if new_value.isdigit() or new_value == "":
            return True
        else:
            return False

    def find_photo(self, photo_id=None):
        if not photo_id:
            photo_id = int(self.photo_id_entry.get())

        photo, time_taken, tag, comments = find_photo_by_id(photo_id)
        if photo:
            self.screenshot_id = photo_id
            self.update_selected_photo(photo, time_taken, tag, comments)
        else:
            messagebox.showerror("Error", "Photo not found")


    def browse_screenshot(self):
        screenshot_browser = ScreenshotBrowser(self)
        screenshot_browser.mainloop()

    def update_selected_photo(self, photo, time_taken, tag, comments):
        screenshot_photo = Image.open(io.BytesIO(photo))
        self.screenshot_photo = ImageTk.PhotoImage(screenshot_photo)
```

```
        self.canvas.delete("image")

        self.canvas.create_image(0, 0, anchor="nw",
                                 image=self.screenshot_photo,
tags="image")
        self.canvas.config(scrollregion=self.canvas.bbox("all"))

        self.photo_id_entry.delete(0, tk.END)
        self.selected_photo_label.config(text=self.screenshot_id)
        self.time_taken_label.config(text=time_taken)
        self.tag_label.config(text=tag)

        self.comments_table.delete(*self.comments_table.get_children())
        for comment in comments:
            self.comments_table.insert("", "end", values=comment)

    def comment(self):
        comment_text = self.comment_entry.get("1.0", tk.END).strip()
        if comment_text:
            upload_comment(comment_text, self.user_id, self.screenshot_id)
            messagebox.showinfo("Success", "Comment uploaded
successfully")
            self.destroy()
        else:
            messagebox.showerror("Error", "Please enter a comment")
```

When this tkinter window is initialised it stores the user ID of the user who is currently commenting so that later it can be identified in the database who is commenting.

The widgets follow the upper left, lower left, upper right, and lower right frame set-up as described in the documented design.

It stores the screenshot_id also which by default is None as it will be selected by the user typically. If this window is opened up after a tag is selected, then the newly saved screenshot ID will automatically be selected and this is why this option is there.

The text entry into the photo ID widget is validated via `validate_numeric_input` to only permit integers to be inputted. Any other type of character simply will not be typed.

There is another validation after the user presses the “Find” button to check that the user has inputted a number at all. Subsequently, there is a table lookup performed to validate that the photo ID the user gave exists; an error message dialogue will appear if this is the case.

The photo will be updated upon any screenshot ID selection method or change in the tree view so the user can see the preview of the visualised image.

A comment will be validated to ensure that something has been typed out. If not, we do not want to hold, unnecessary, empty comments in the database because it is redundant.

screenhot_browser.py

```
import io
import tkinter as tk
from tkinter import ttk

from PIL import ImageTk, Image

from database_manager import (
    find_photo_by_id, get_tags, get_users,
    get_valid_screenshots
)
from screenshot_manager.comment_system.date_range_selector import \
    DateRangeSelector


class ScreenshotBrowser(tk.Toplevel):
    def __init__(self, comment_system_window):
        super().__init__(comment_system_window)
        self.title("Browse Photos")
        self.comment_system_window = comment_system_window

        self.screenshot_id = None
        self.photo = None

        self.tag_option = "All"
        self.user_option = "All"
        self.start_date = None
        self.end_date = None

        # Tags OptionMenu
        tags_label = tk.Label(self, text="Tags:")
        tags_label.grid(row=0, column=0, padx=5, pady=5, sticky="e")
        self.tags_var = tk.StringVar()
        self.tags_optionmenu = ttk.OptionMenu(
                self,
                self.tags_var,
                "All",
                *["All"] + [f"{tag[0]}: {tag[1]}" for tag in get_tags()],
                command=self.update_tag_option
        )
        self.tags_optionmenu.grid(row=0, column=1, padx=5, pady=5,
sticky="w")

        # Users OptionMenu
        users_label = tk.Label(self, text="Users:")
        users_label.grid(row=0, column=2, padx=5, pady=5, sticky="e")
        self.users_var = tk.StringVar()
        self.users_optionmenu = ttk.OptionMenu(
                self,
                self.users_var,
                "All",
                *["All"] + [
                    f"{user[0]}: {user[1]}" for user in get_users()
                ],
```

```
                command=self.update_user_option
        )
        self.users_optionmenu.grid(row=0, column=3, padx=5, pady=5,
sticky="w")

        # Date Range Selector
        self.date_range_label = tk.Label(self, text="Selected Date Range:
")
        self.date_range_label.grid(row=0, column=4, padx=5, pady=5,
sticky="e")
        self.calendar_icon = ImageTk.PhotoImage(
                Image.open(r"resources\calendar_icon.jpg").resize((20,
20))
        )
        self.calendar_button = tk.Button(
                self,
                image=self.calendar_icon,
                command=self.open_date_range_selector
        )
        self.calendar_button.grid(row=0, column=5, padx=5, pady=5,
sticky="w")

        # Treeview
        self.screenshots_tree = ttk.Treeview(
                self,
                columns=(
                    "User ID", "Username", "Timestamp", "Tag ID", "Tag
Name"
                )
        )
        self.screenshots_tree.grid(
                row=1, column=0, columnspan=6, padx=5, pady=5,
sticky="nsew"
        )
        self.screenshots_tree.column("#0", width=90)
        self.screenshots_tree.heading("#0", text="Screenshot ID")
        self.screenshots_tree.column("User ID", width=75)
        self.screenshots_tree.heading("User ID", text="User ID")
        self.screenshots_tree.column("Username", width=75)
        self.screenshots_tree.heading("Username", text="Username")
        self.screenshots_tree.column("Timestamp", width=125)
        self.screenshots_tree.heading("Timestamp", text="Timestamp")
        self.screenshots_tree.column("Tag ID", width=50)
        self.screenshots_tree.heading("Tag ID", text="Tag ID")
        self.screenshots_tree.column("Tag Name", width=75)
        self.screenshots_tree.heading("Tag Name", text="Tag Name")
        self.screenshots_tree.bind("<<TreeviewSelect>>",
                                   self.update_photo_preview)

        # Configure vertical scrollbar
        self.tree_scroll_y = tk.Scrollbar(self, orient="vertical",

command=self.screenshots_tree.yview)
        self.tree_scroll_y.grid(row=1, column=6, padx=5, pady=5,
```

```
sticky="ns")

self.screenshots_tree.configure(yscrollcommand=self.tree_scroll_y.set)

        # Selected Photo Preview
        self.photo_preview_label = tk.Label(
                self, text="[Selected Photo Preview]"
        )
        self.photo_preview_label.grid(
                row=1, column=7, padx=5, pady=5, sticky="nsew"
        )

        # Select Button
        select_button = tk.Button(
                self, text="Select", command=self.select_screenshot
        )
        select_button.grid(row=2, column=7, padx=5, pady=5, sticky="nsew")

        # Configure grid weights
        self.grid_rowconfigure(1, weight=1)
        self.grid_columnconfigure(0, weight=1)
        self.grid_columnconfigure(6, weight=0)

        self.update_screenshots_tree()

    def select_screenshot(self):
        selected_item_id = self.screenshots_tree.selection()[0] \
            if self.screenshots_tree.selection() else None
        if selected_item_id:
            selected_item = self.screenshots_tree.item(selected_item_id)
            self.screenshot_id = selected_item["text"]
            self.comment_system_window.find_photo(self.screenshot_id)
            self.destroy()

    def update_photo_preview(self, event):
        selected_item = self.screenshots_tree.selection()[0] \
            if self.screenshots_tree.selection() else None
        if selected_item:
            self.screenshot_id = self.screenshots_tree.item(
                    selected_item, "text"
            )
            photo_blob = find_photo_by_id(self.screenshot_id)[0]
            photo = Image.open(io.BytesIO(photo_blob))
            photo.thumbnail((200, 200))
            self.photo = ImageTk.PhotoImage(photo)
            self.photo_preview_label.config(image=self.photo)
        else:
            self.photo_preview_label.config(image="")

    def open_date_range_selector(self):
        date_range_selector = DateRangeSelector(self.date_range_label)
        self.wait_window(date_range_selector)
        if date_range_selector.start_date and
date_range_selector.end_date:
```

```
            self.start_date = date_range_selector.start_date
            self.end_date = date_range_selector.end_date
            self.update_screenshots_tree()

    def update_tag_option(self, value):
        self.tag_option = self.tags_var.get()
        self.update_screenshots_tree()

    def update_user_option(self, value):
        self.user_option = self.users_var.get()
        self.update_screenshots_tree()

    def update_screenshots_tree(self):
        valid_screenshots = get_valid_screenshots(
                self.tag_option,
                self.user_option,
                self.start_date,
                self.end_date
        )

self.screenshots_tree.delete(*self.screenshots_tree.get_children())
        for screenshot in valid_screenshots:
            self.screenshots_tree.insert(
                    "",
                    "end",
                    text=screenshot[0],
                    values=(
                        screenshot[1],
                        screenshot[2],
                        screenshot[3],
                        screenshot[4],
                        screenshot[5]
                    )
            )
```

Stores the default filtering settings “All” and “None” for dates, so by default will show all available screenshots.

The `select_screenshot` function validates that there is a selected option in the treeview. Then if so, it finds the photo and edits its parent’s widgets to reflect the change and destroys itself as its function is complete.

The photo preview function updates the thumbnail image on the right side of the treeview whenever a new one is selected. Since we already evaluated what is in the treeview exists, we can be confident that the screenshot_id we use to find the photo data exists.

We can see iteration being used at the end in the for loop going through the valid screenshots that we queried from the database and inserting them one by one into the treeview.

date_range_selector.py

```
import tkinter as tk
from tkcalendar import Calendar


class DateRangeSelector(tk.Toplevel):
    def __init__(self, date_range_label):
        super().__init__(date_range_label)
        self.title("Select Date Range")

        self.date_range_label = date_range_label

        # Left Frame for Start Date
        self.left_frame = tk.Frame(self)
        self.left_frame.pack(side="left", padx=10, pady=10)

        start_label = tk.Label(self.left_frame, text="Start Date")
        start_label.pack(side="top")

        self.start_calendar = Calendar(
                self.left_frame, selectmode="day", date_pattern="yyyy-mm-
dd"
        )
        self.start_calendar.pack(side="bottom")

        # Right Frame for End Date
        self.right_frame = tk.Frame(self)
        self.right_frame.pack(side="right", padx=10, pady=10)

        end_label = tk.Label(self.right_frame, text="End Date")
        end_label.pack(side="top")

        self.end_calendar = Calendar(
                self.right_frame, selectmode="day", date_pattern="yyyy-mm-
dd"
        )
        self.end_calendar.pack(side="bottom")

        # Bottom Frame for Confirm Button
        self.bottom_frame = tk.Frame(self)
        self.bottom_frame.pack(side="bottom", pady=10)

        self.confirm_button = tk.Button(
                self.bottom_frame,
                text="Confirm",
                command=self.confirm_selection
        )
        self.confirm_button.pack()

        self.start_date = None
        self.end_date = None

    def confirm_selection(self):
        self.start_date = self.start_calendar.get_date()
```

```
        self.end_date = self.end_calendar.get_date()
        self.date_range_label.config(
            text=f"Selected Date Range: {self.start_date} to
{self.end_date}")
        self.destroy()
```

Two calendar icons as in the documented design with a select button in the middle. Allows the user to select the start date and the end date to filter their search results of the screenshots taken.

This is useful when the team wants to see what they did on specific days and for the boss to evaluate the performance of the employees.

Testing

At this stage, I will get my client to test the features of my solution to check if it is intuitive and easy to use as required, simplicity is key.

All I have told him is:

“

Grid Keys:

A – perform astar

C – make a new comment

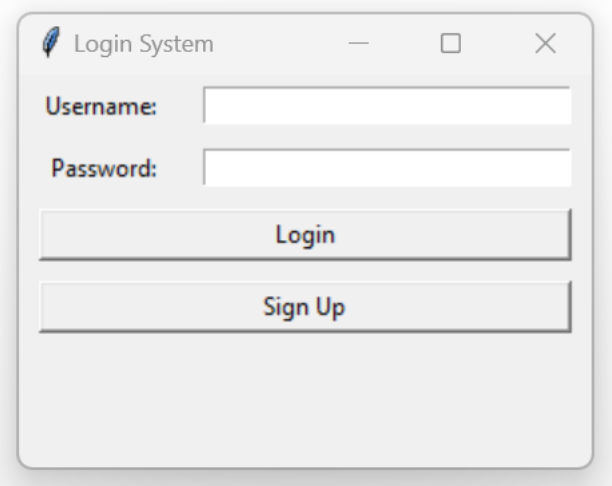
D – perform Dijkstra

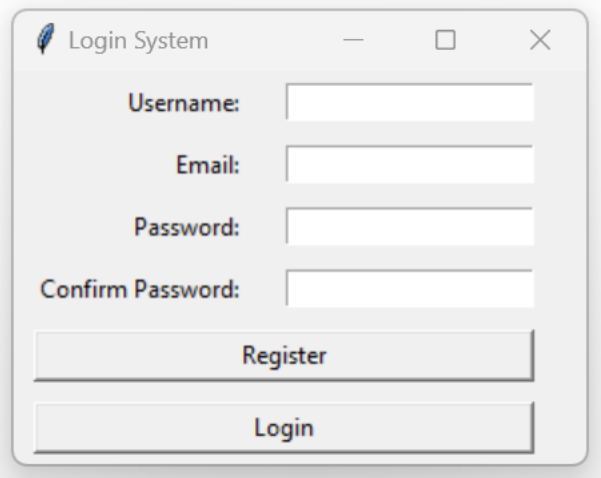
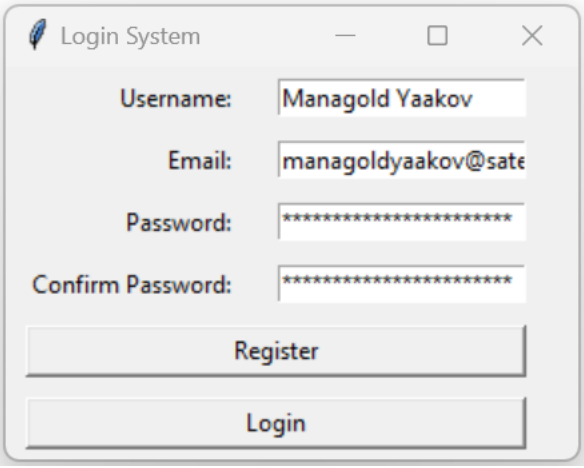
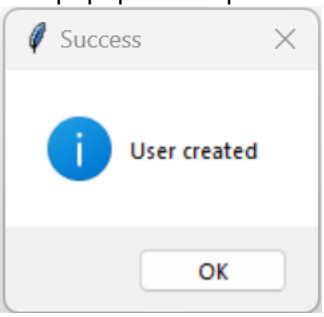
E – empty the grid

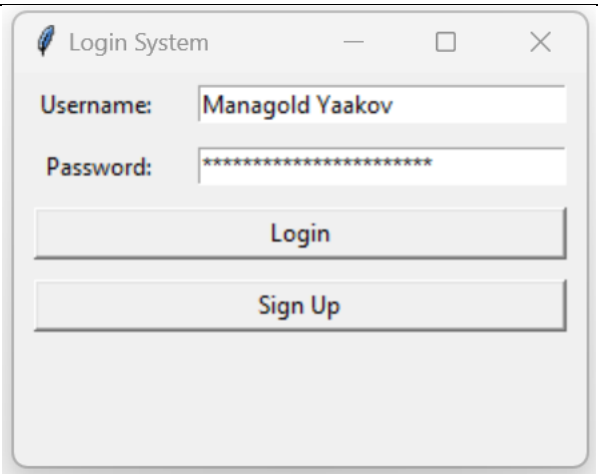
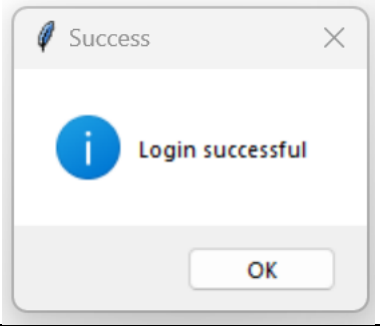
S – save the visualised path

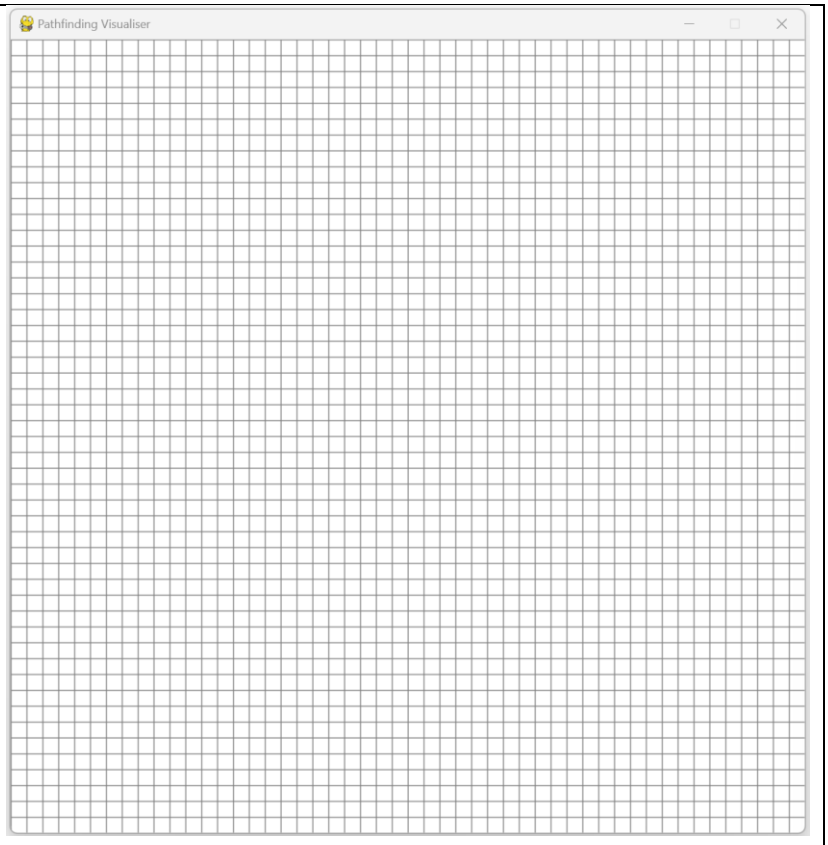
“

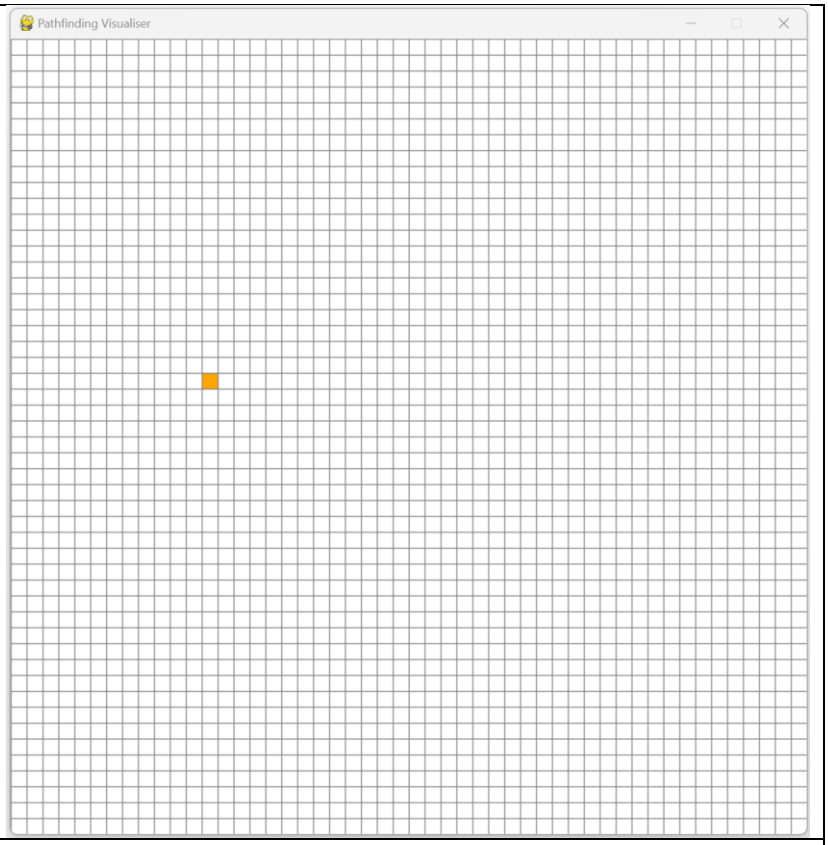
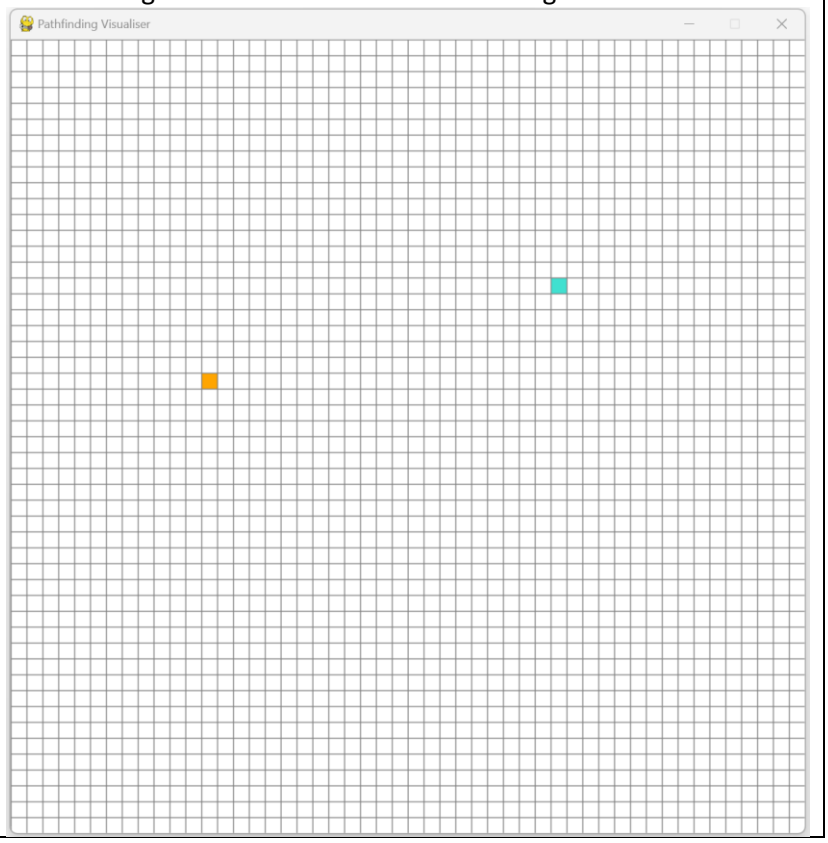
1 End User Testing

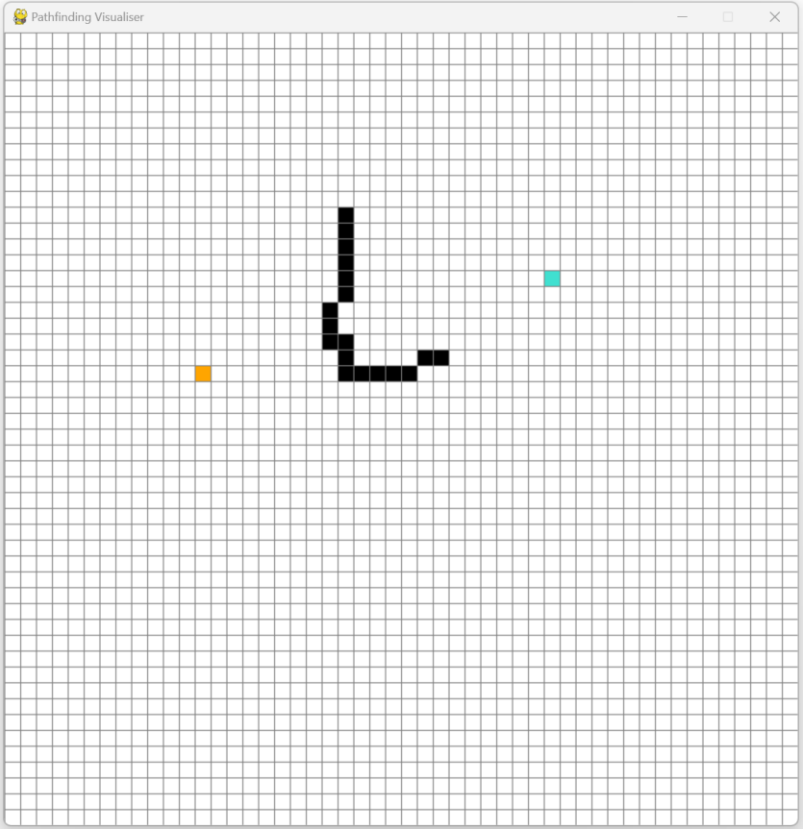
No.	Task	Test Data	Evidence
1	Successfully sign up	Username: Managold Yaakov Email: managoldyaakov@satellitepathx.com Password: my_hidden_password* 2024	The first thing that comes up is the login page. I have not been told I have a special account set up so I will make one.  I have noticed that in the directory where I ran this file, there is a new file: `screenshot_database.db`  I now press the “Sign Up” button. I am switched to the following screen:

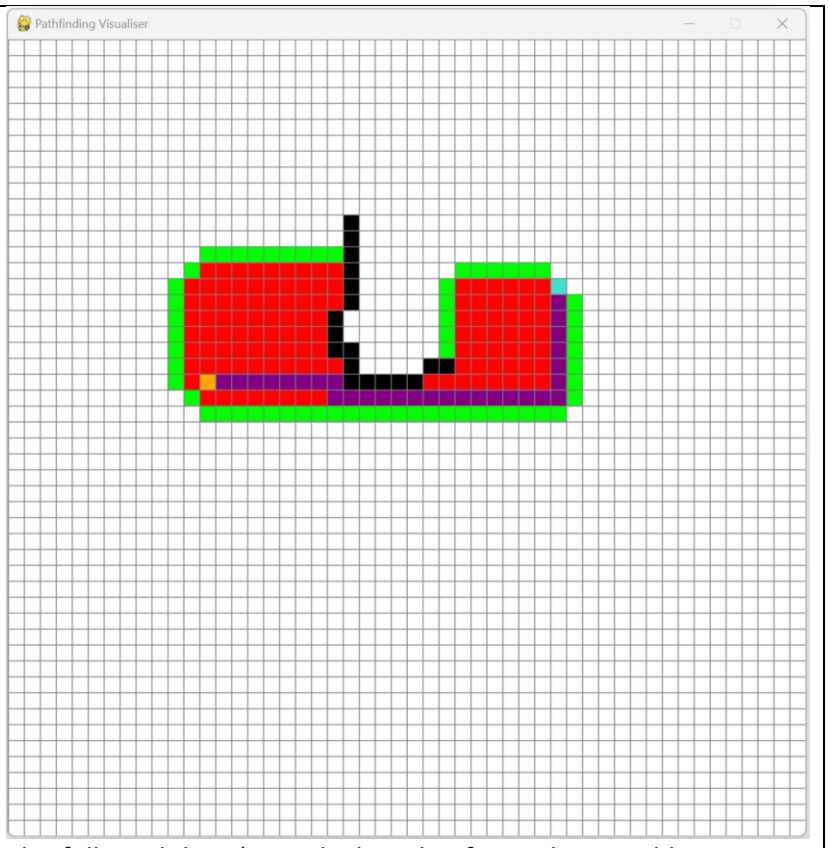
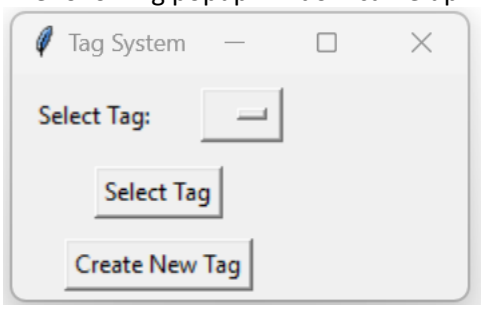
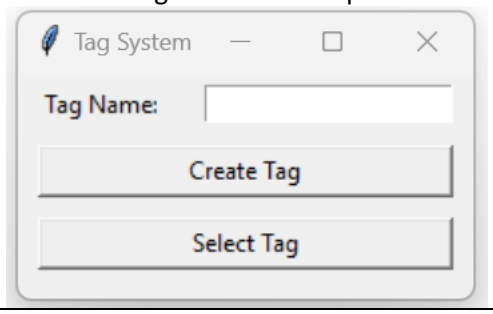
			 I will enter my details here I see:  My password is hidden which is expected. I will now press the "Register" button to complete my registration is my assumption. This popup came up:  I have been signed up for sure now. I will press "OK" to dismiss the popup window.
2	Log in to the program	Username: Managold Yaakov Password: my_hidden_password* 2024	The window redirected to the original login screen ready for me to log in with the new user I just created.

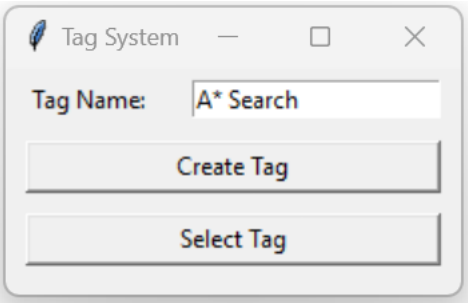
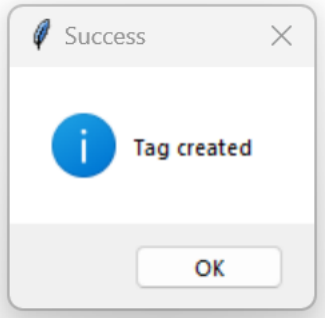
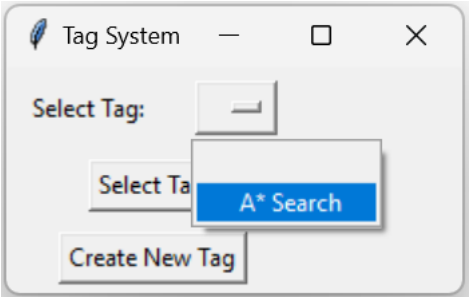
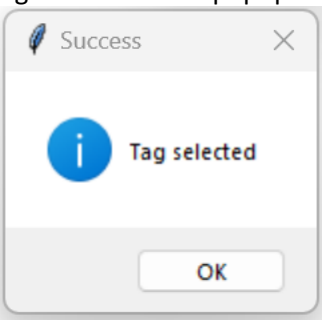
			 I will now press the “Login” button as that seems typical of user login systems. A popup appeared letting me know my login was successful: 
3	Place the start node on the grid	Mr Yaakov will press on his own desired node to make it the start.	After logging in this grid appeared, this must be the visualisation grid:

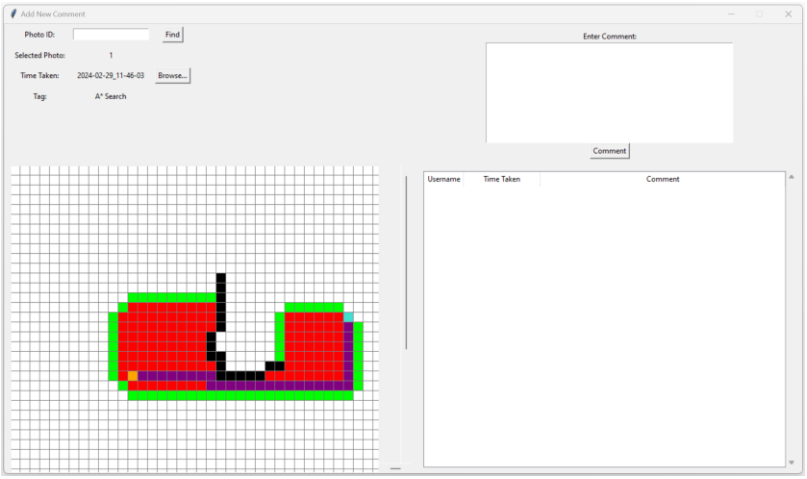
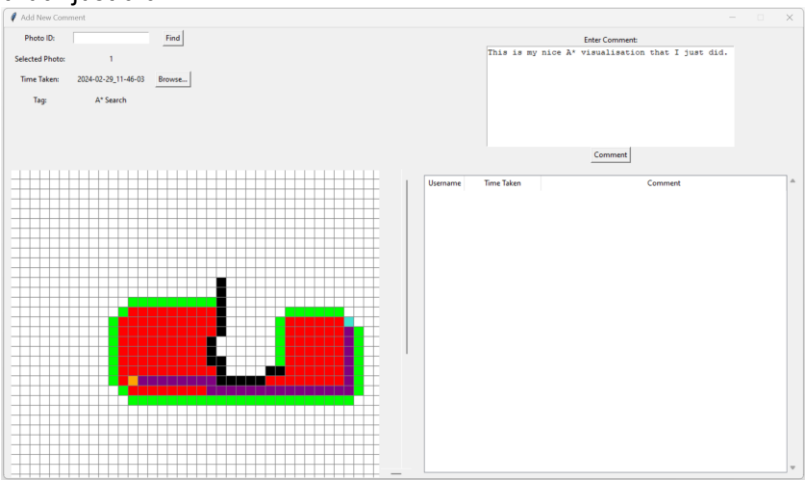
			 I will click on a node to make the starting position. The node became orange, this must be my start node:

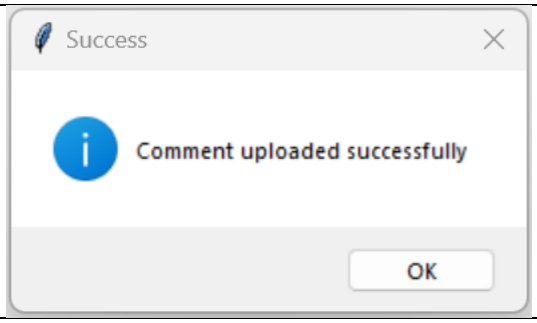
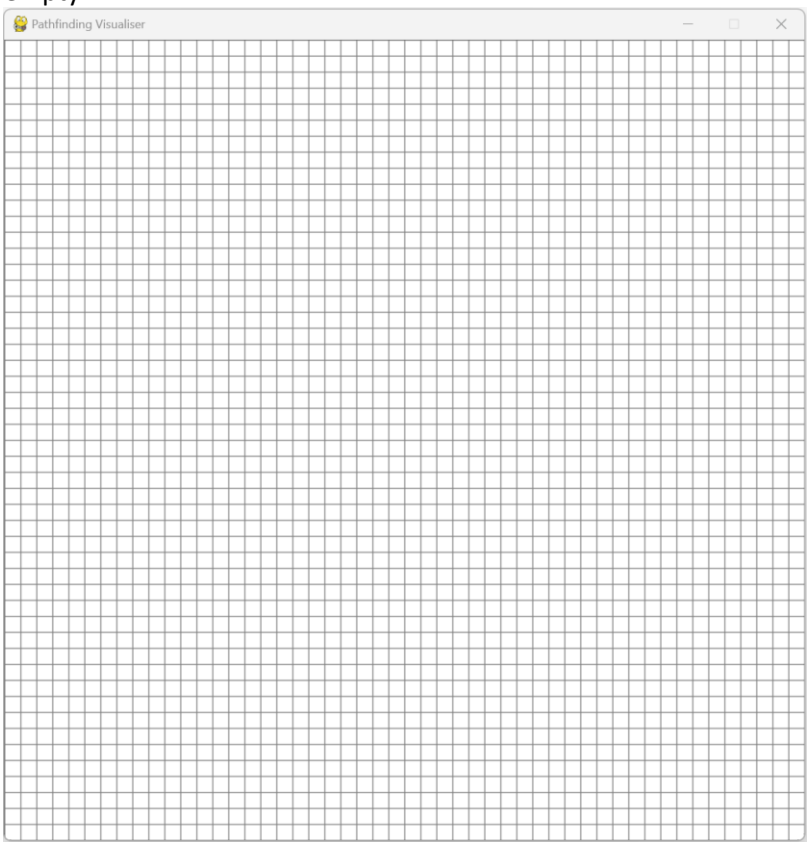
			
4	Place the target node on the grid	Mr Yaakov will press on his own desired node to make it the target.	I will now again somewhere to add the ending node: 

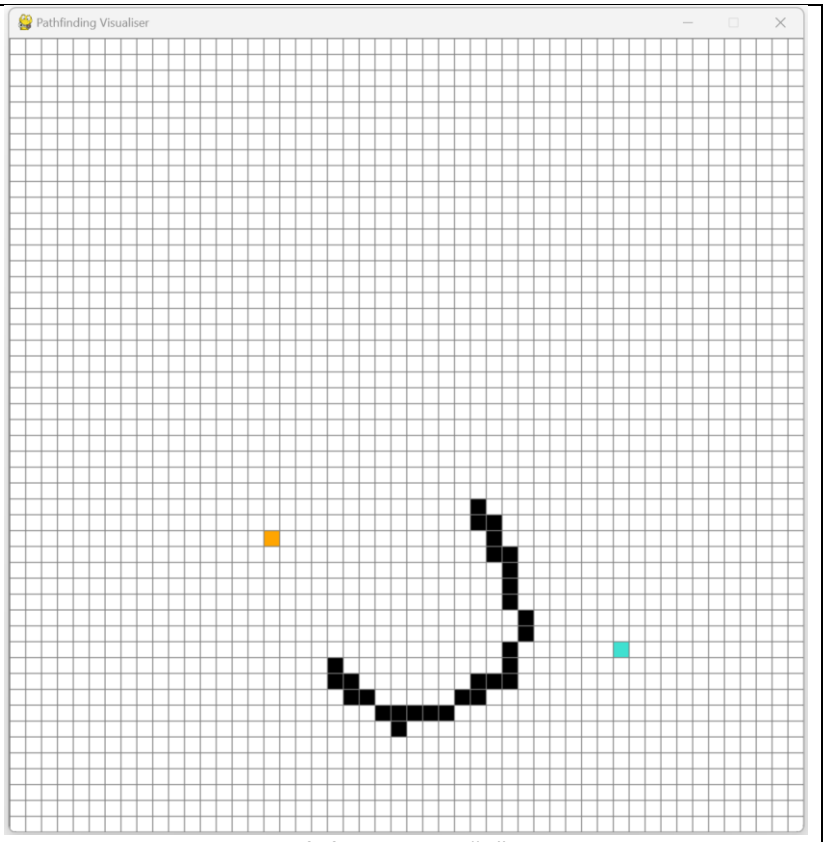
			It appears in the colour turquoise.
5	Place walls on the grid	Mr Yaakov will press on the empty squares he wishes to make walls.	I will now drag and drop in between the two as a simple wall the algorithm must go around:  That felt smooth.
6	Visualise the A* search algorithm	The grid is set up, and no more test data is needed, only an input of the button “a”.	I can see from this list Eno gave me, that by pressing the “a” key on my keyboard that the A* algorithm will be visualised, so I will now press it. I saw the visualisation occur and now the shortest path is displayed in purple!

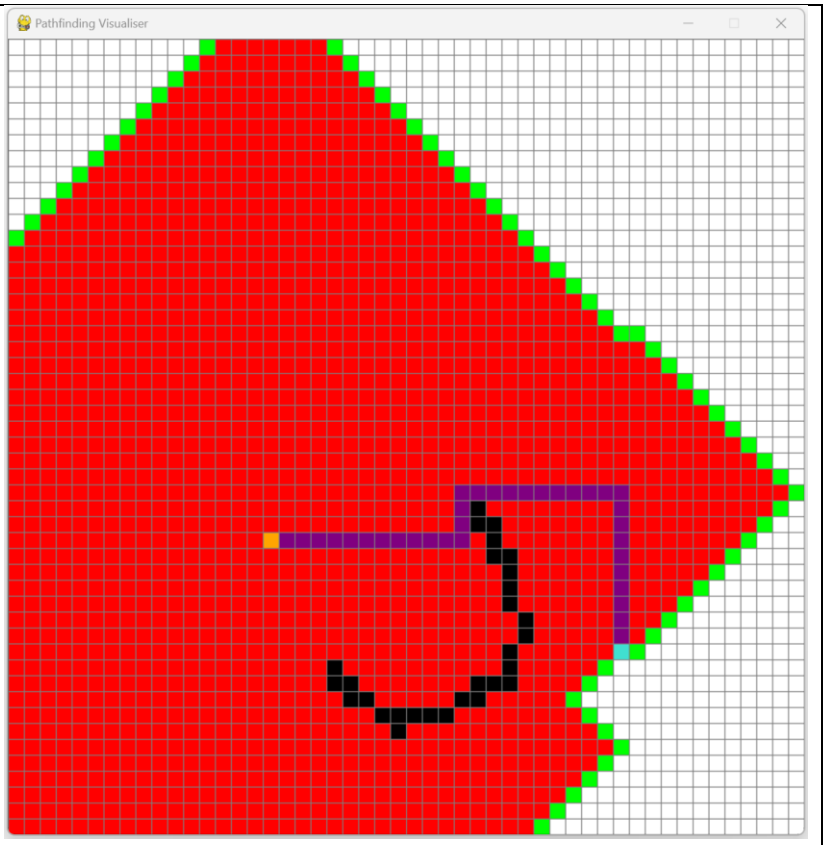
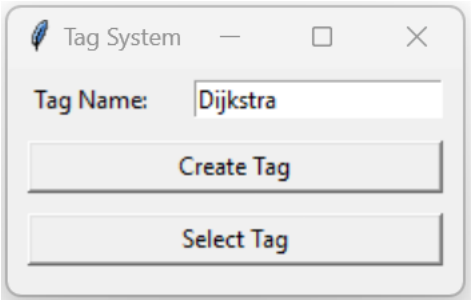
			 This followed the A* search algorithm from what I could see.
7	Create New Tag	Tag Name: A* Search	I now want to save my screenshot so I will press the “s” key as that is what Eno’s instructions state. The following popup window came up:  I do not have a tag selected, nor have I made one, so I will press the “Create New Tag” button. The following screen came up: 

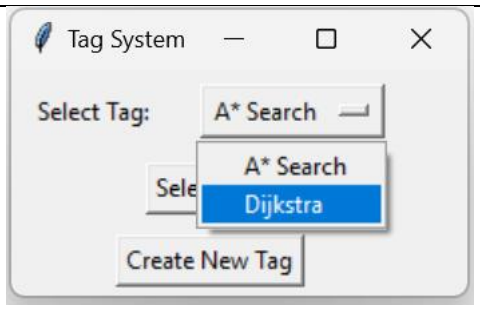
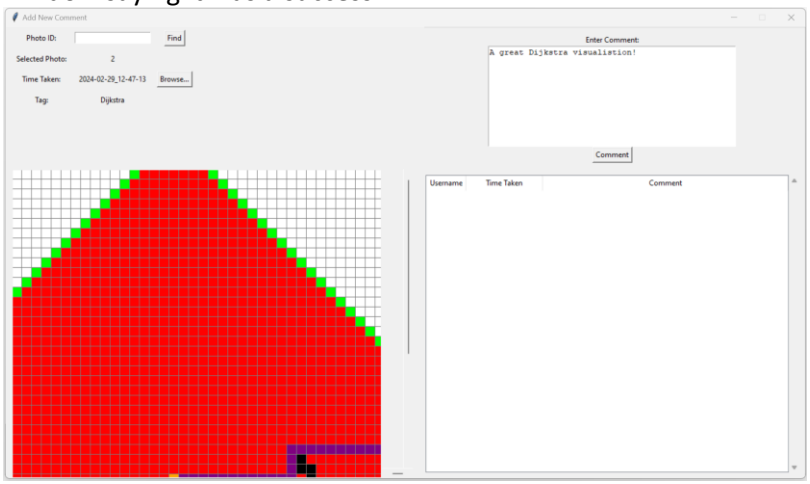
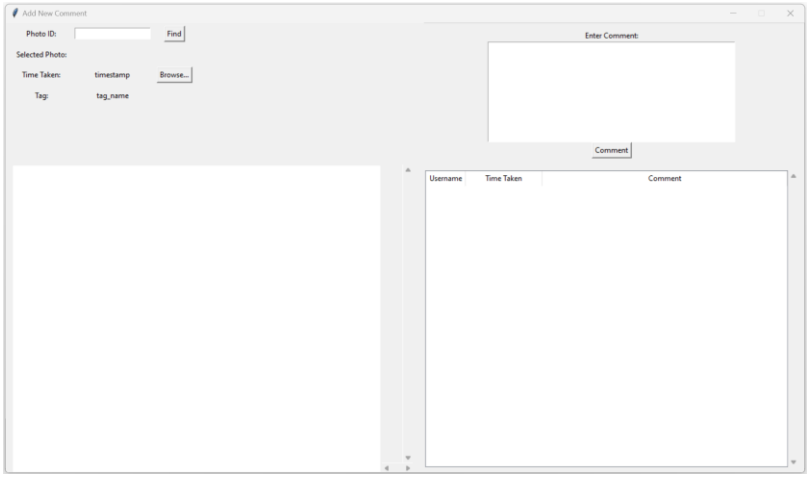
			For the sake of this input, I will simply differentiate this by the algorithm I used. So, for this screenshot, I will make the tag name "A* Search":  I will now press the "Create Tag" button. This popup window came up to tell me the tag has been created:  I will now press "OK".
8	Select Tag	Mr Yaakov will use the tag he just created to select it for this photo saving.	It redirected me to the original select tag screen after I just created that tag. I will select the tag I just created:  I will now press the "Select Tag" button. I got this window popup telling me I selected the tag successfully:  I will now press the "OK" button.

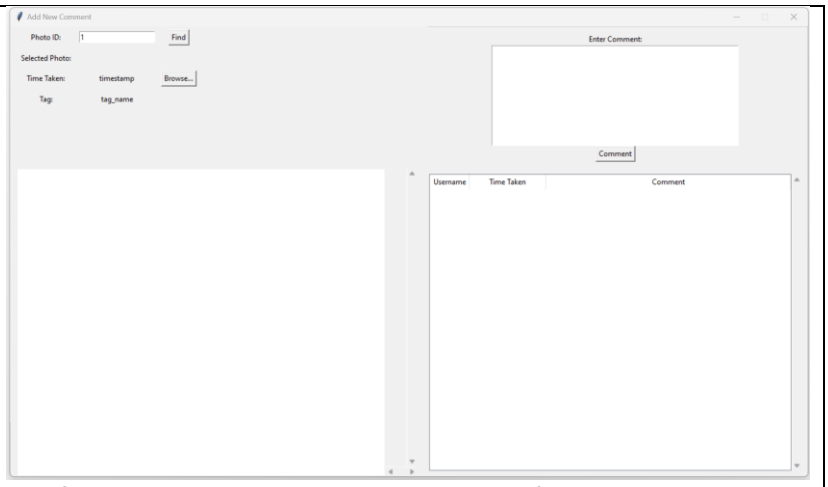
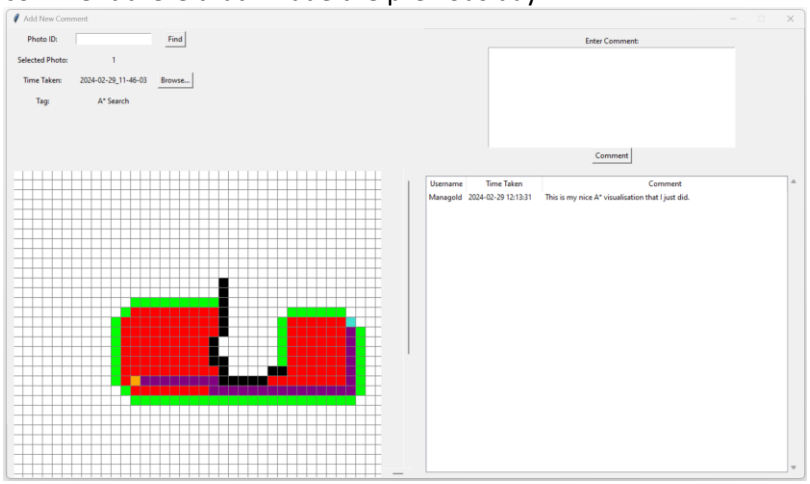
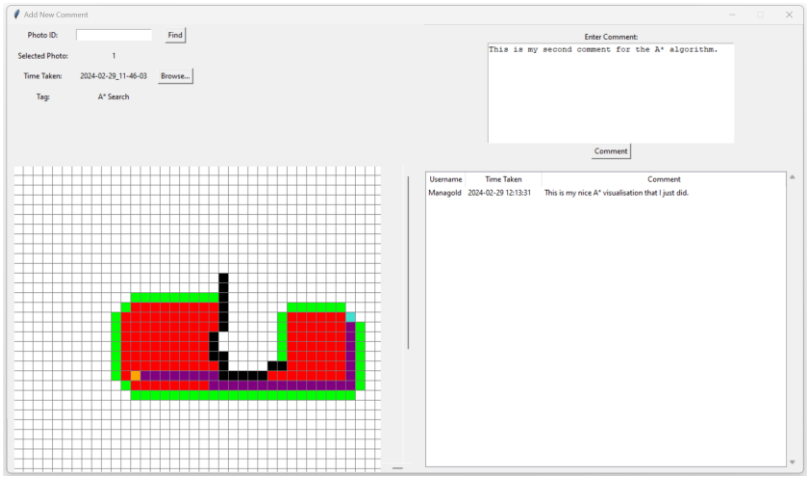
9	Create Comment	Comment: This is my nice A* visualisation that I just did.	Upon selecting my tag, the following comment system window opened. I can see the visualisation bottom right and a prompt to enter a comment top right. I can see some information top left which is the photo ID which is 1 as it is my first screenshot. I can see the time and day I took this screenshot which is just a moment ago. I can also see the tag I selected for it. I'm not quite sure what the bottom left is, I'll assume it is the comments for the photo and since my photo is new, it has no comments. My screenshot has been saved and now I can optionally add a comment:  I will add in my generic comment "This is my nice A* visualisation that I just did.":  I will now press the "Comment" button to confirm my comment. A popup window showed up to tell me my comment was uploaded successfully:

			
10	Empty the grid	There is no test data, only the keyboard input of “e”.	Now I made my comment, I am back to my visualised path grid. I wish to empty it, so I follow Eno’s note he left for me and I press the “e” key on my keyboard to empty my grid. My grid now is the default grid like I first opened and is completely empty:  Worked well.
11	Visualise Dijkstra’s Algorithm	Mr Yaakov will create a map for the visualisation and then hit the "d" key.	I will now remake a map in preparation to use Dijkstra’s algorithm:

			 According to what Eno left for me, the “d” key will visualise Dijkstra’s algorithm so I will now press it. It was distinctly unique from the A* visualisation, here is the visualised path:

			 A great visualisation!
12	Save a visualised path	Tag Name: Dijkstra Comment: A great Dijkstra visualisation! We have already saved a screenshot but this test focuses on the screenshot aspect rather than individuating the processes of the tag system and comment system.	I press “s” on my keyboard to save and see the familiar tag system window, I noticed that the previous tag I made was selected by default in the option menu. I then press the “Create Tag” button and type out the name of the new tag I wish to create “Dijkstra”:  I press “Create Tag” and I see the familiar popup confirming my tag selection. I am redirected to the select tag screen and I select the Dijkstra I just made and see the familiar success popup window:

			 I am greeted with the familiar comment system window and type my comment out: “A great Dijkstra visualisation!”. I press the select button and see the familiar confirmation popup window saying it was a success:  I can be confident now that this screenshot was also saved.
13	Make new comment from the grid screen	Photo ID: 1 Comment: This is my second comment for the A* algorithm.	I am on the main grid screen now and I will follow Eno’s instruction by pressing c to open the comment system:  This same comment system comes up but looks different from before since I have not selected a photo yet. I know that the first photo I saved was the A* search photo and that is the one I wish to edit. I typed in `1` for the photo id and pressed “Find”:

			 The following screen came up, but unlike before there was the comment there that I made the previous day:  I will now type out my comment and press “Comment” to upload it:  The “comment uploaded successfully” popup dialogue came up and I pressed “OK”.

2 Robustness

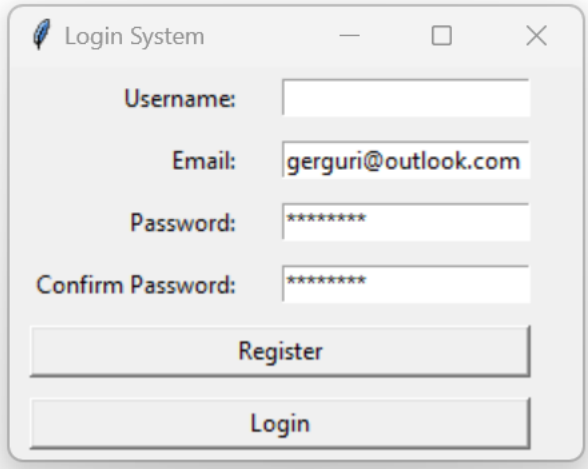
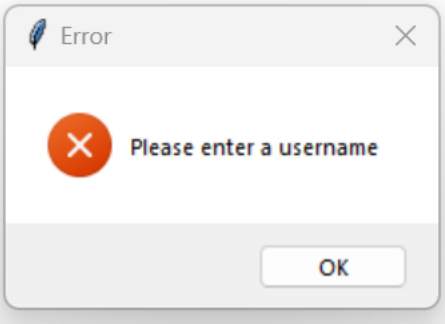
This end user almost fully black box testing proves to me the robustness of my system. Even though my client said he does not have great computer skills he could still navigate and perform the tasks he wanted to test for himself with the little brief teaching which he wanted according to the questionnaire I gave him. That simple instruction set means anyone with it can process the essential features of my application. Even an intern or someone new can easily pick this up and use it efficiently. It shows that my application has a consistent and navigable user interface which allows effective operation. It is simplistic enough for a simple solution as the client wanted but it is also complex enough to meet the client's needs and wants.

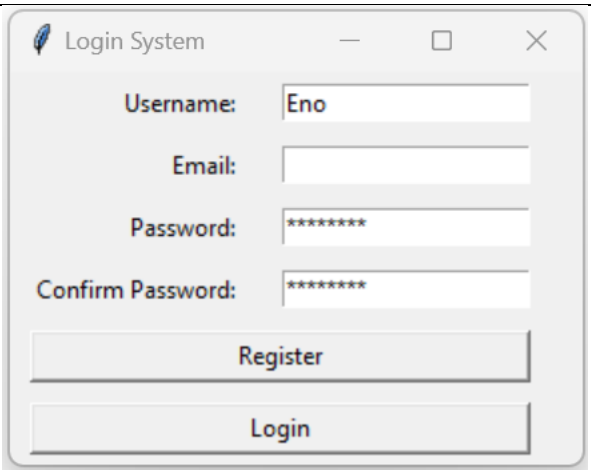
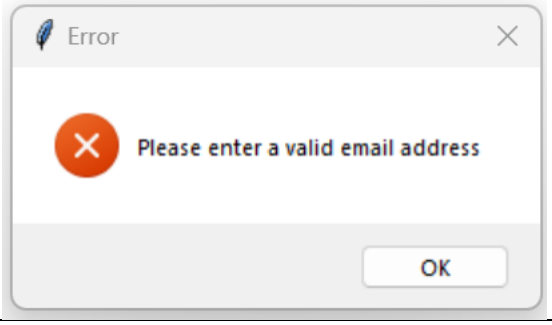
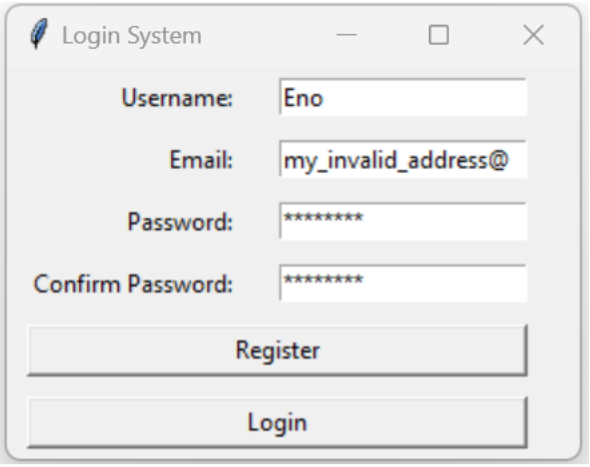
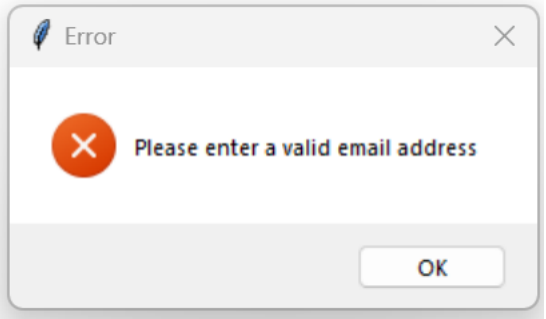
3 Error Testing

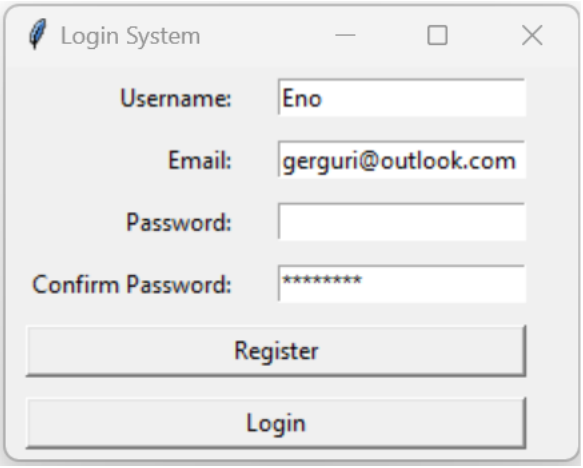
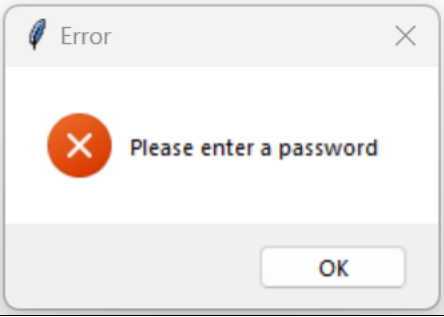
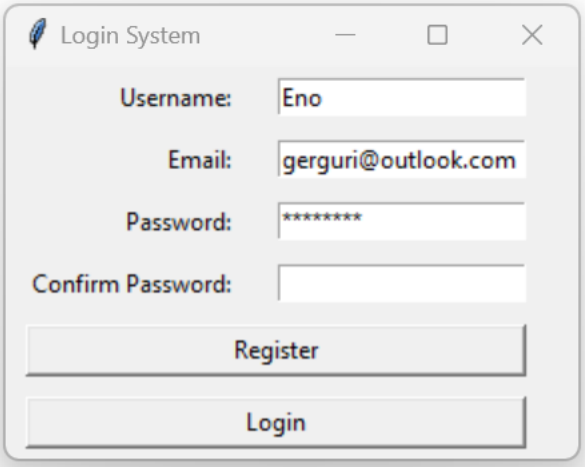
My client sent me back his database file for me to complete testing since he was satisfied.

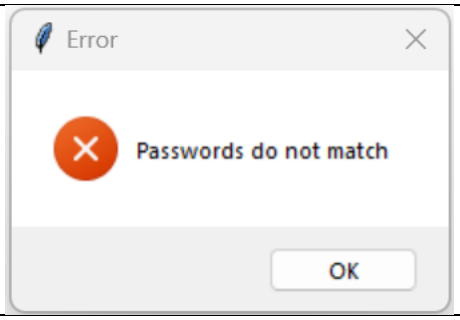
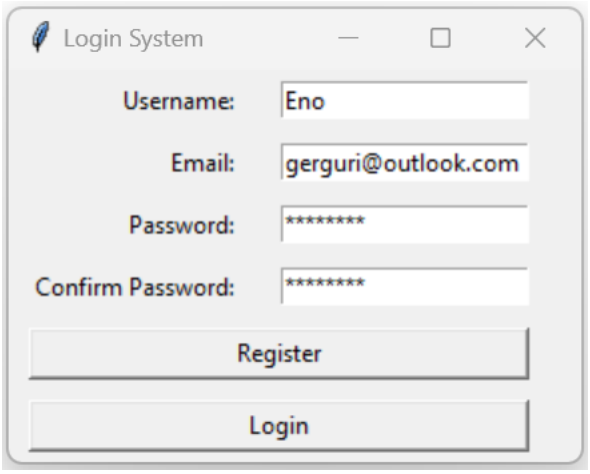
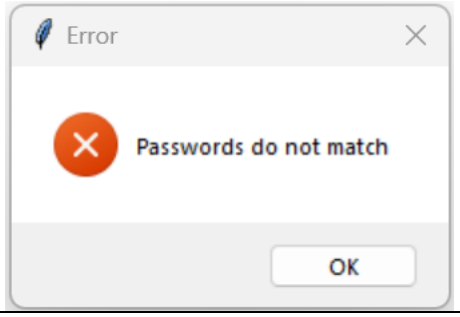
I will now test all the possible error checks and validations to ensure that the program does not glitch or break upon a user's mistake in input.

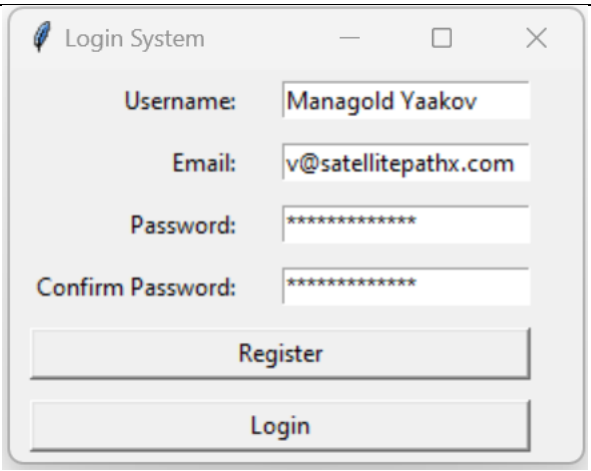
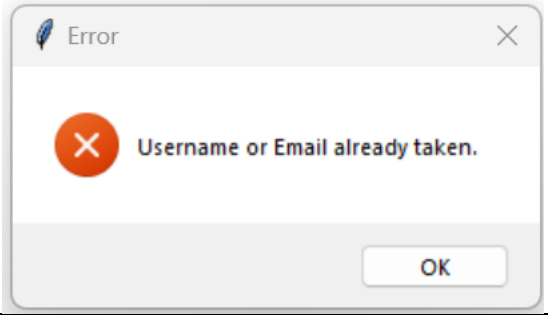
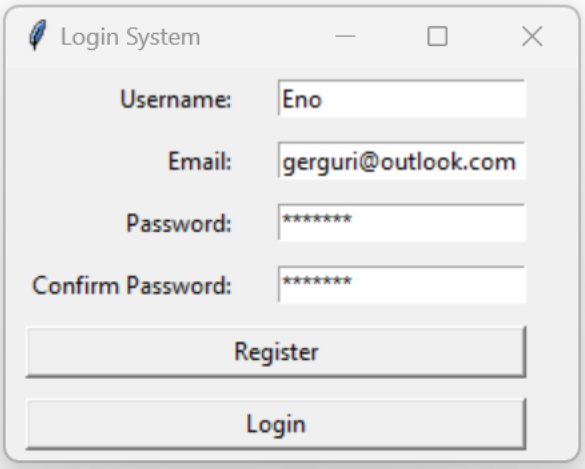
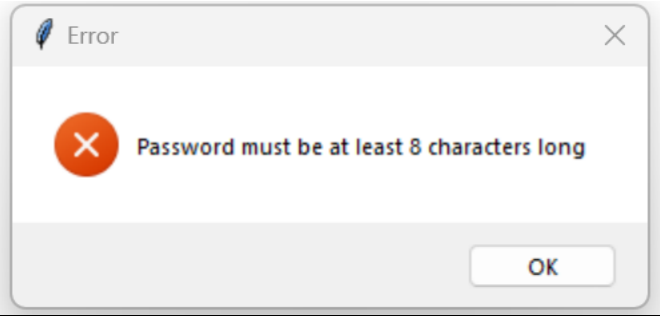
These will be the erroneous types of tests:

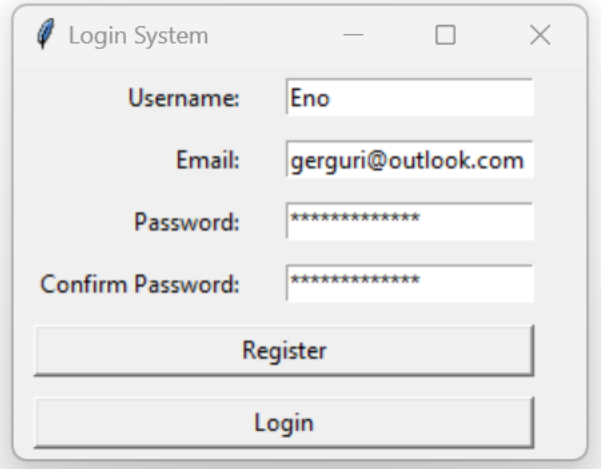
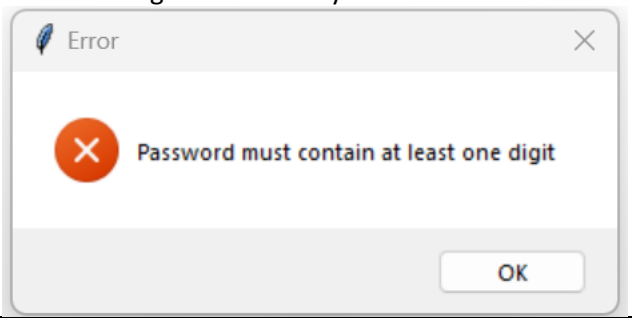
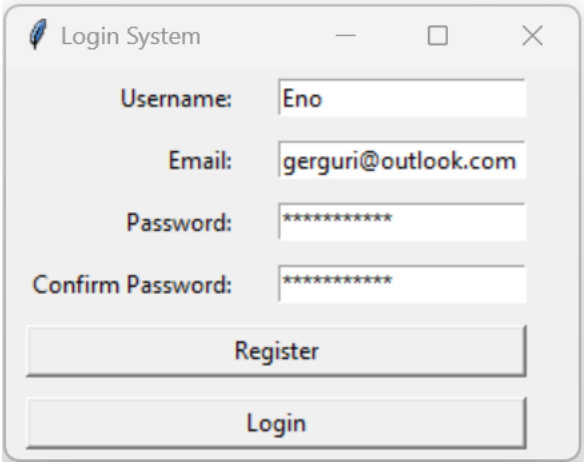
No.	Test	Test Data	Result
1	Empty Create User Username	Username: Email Address: enogerguri@outlook.com Password: 23rTY*ds Retype Password: 23rTY*ds	When the app opens, I click “Sign Up”. Then I type in the test data and press “Register”:  The following error correctly shows up: 
2	Empty Create User Email	Username: Eno Email Address: Password: 23rTY*ds Retype Password: 23rTY*ds	When the app opens, I click “Sign Up”. Then I type in the test data and press “Register”:

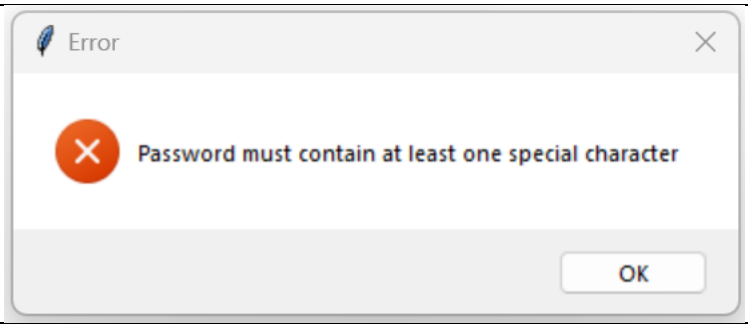
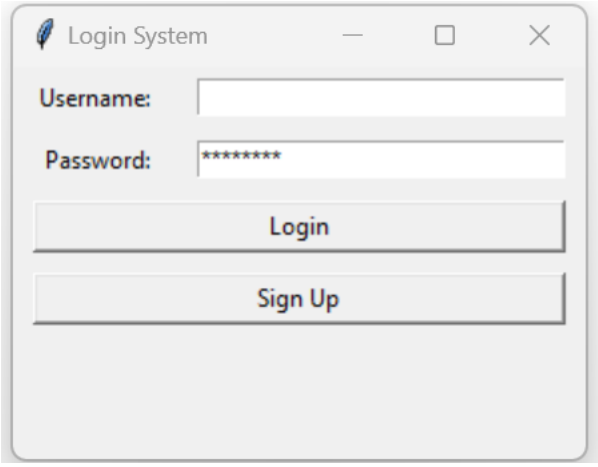
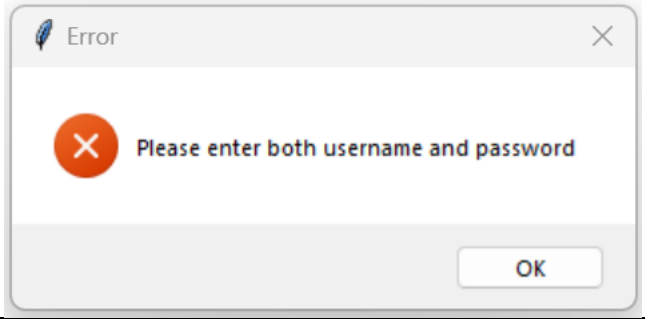
			 The following error correctly shows up: 
3	Create User Invalid Email Address	Username: Eno Email Address: my_invalid_address@ Password: 23rTY*ds Retype Password: 23rTY*ds	When the app opens, I click “Sign Up”. Then I type the test data and press “Register”:  The following error correctly shows up: 

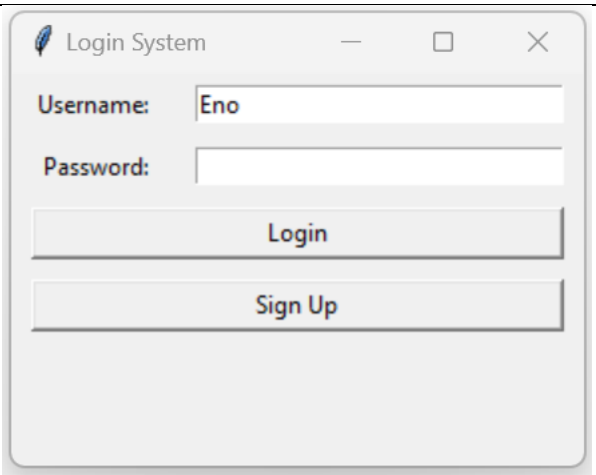
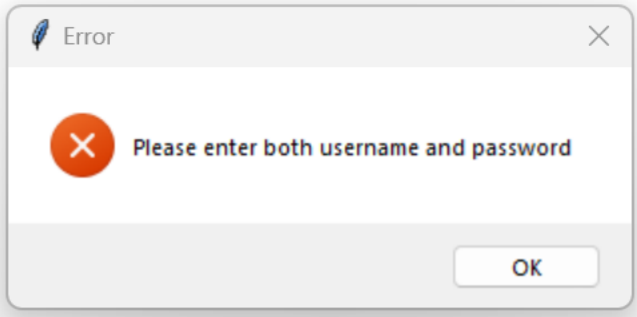
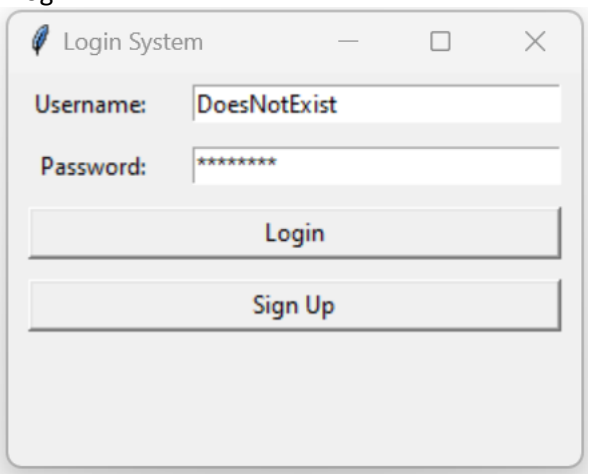
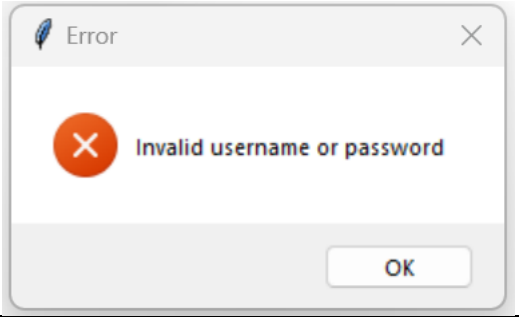
4	Empty Create User Password	Username: Eno Email Address: enogerguri@outlook.com Password: Retype Password: 23rTY*ds	When I open the app, I press “Sign Up”. Then I type the test data and press “Register”:  The following error correctly shows up: 
5	Empty Create User Retype Password	Username: Eno Email Address: enogerguri@outlook.com Password: 23rTY*ds Retype Password:	When I open the app, I press “Sign Up”. Then I type the test data and press “Register”:  The following error correctly shows up:

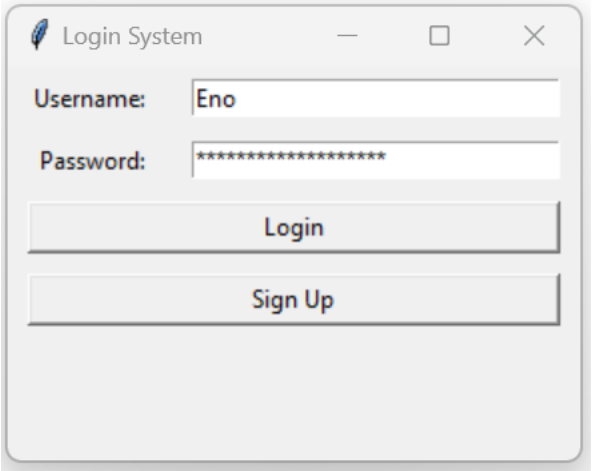
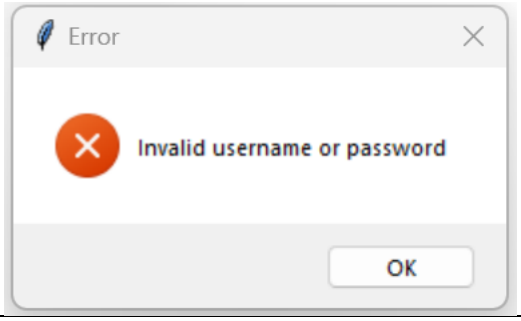
			
6	Create a User Password and retype Password mismatch	Username: Eno Email Address: enogerguri@outlook.com Password: 23rTY*ds Retype Password: mismatch	When I open the app, I press “Sign Up”. Then I type the test data and press “Register”:  The following error correctly shows up: 
7	Create User Already Existing Username or Email	Username: Managold Yaakov Email Address: managoldyaakov@satellitepathx.com Password: my_password1* Retype Password: my_password1*	When I open the app, I press “Sign Up”. Then I type the test data and press “Register”:

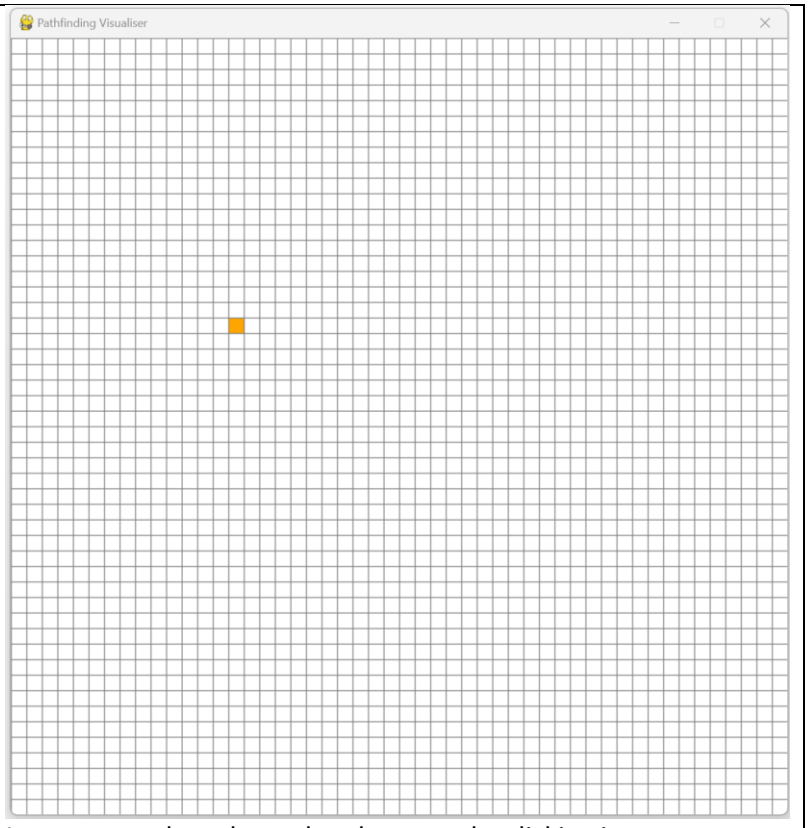
			 The following error correctly shows: 
8	Create a User password of less than 8 characters	Username: Eno Email Address: enogerguri@outlook.com Password: short1* Retype Password: short1*	When I open the app, I press “Sign Up”. Then I type the test data and press “Register”:  The following error correctly shows: 

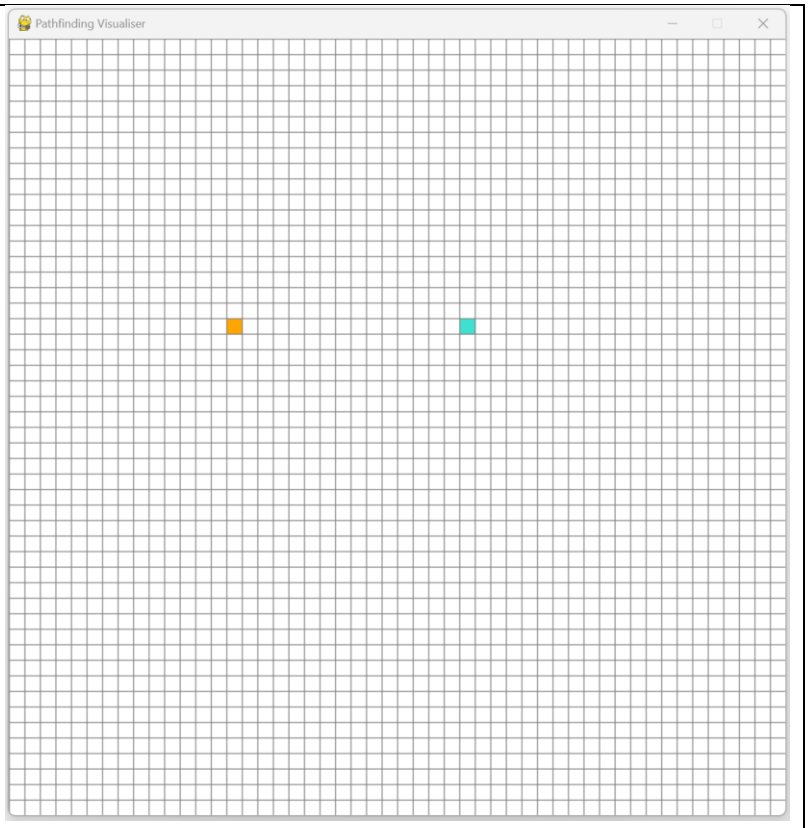
9	Create a User password with no integer characters	Username: Eno Email Address: enogerguri@outlook.com Password: good@password Retype Password: good@password	When I open the app, I press “Sign Up”. Then I type the test data and press “Register”:  The following error correctly shows: 
10	Create a User password with no special characters	Username: Eno Email Address: enogerguri@outlook.com Password: password123 Retype Password: password123	When I open the app, I press “Sign Up”. Then I type the test data and press “Register”:  The following error correctly comes up:

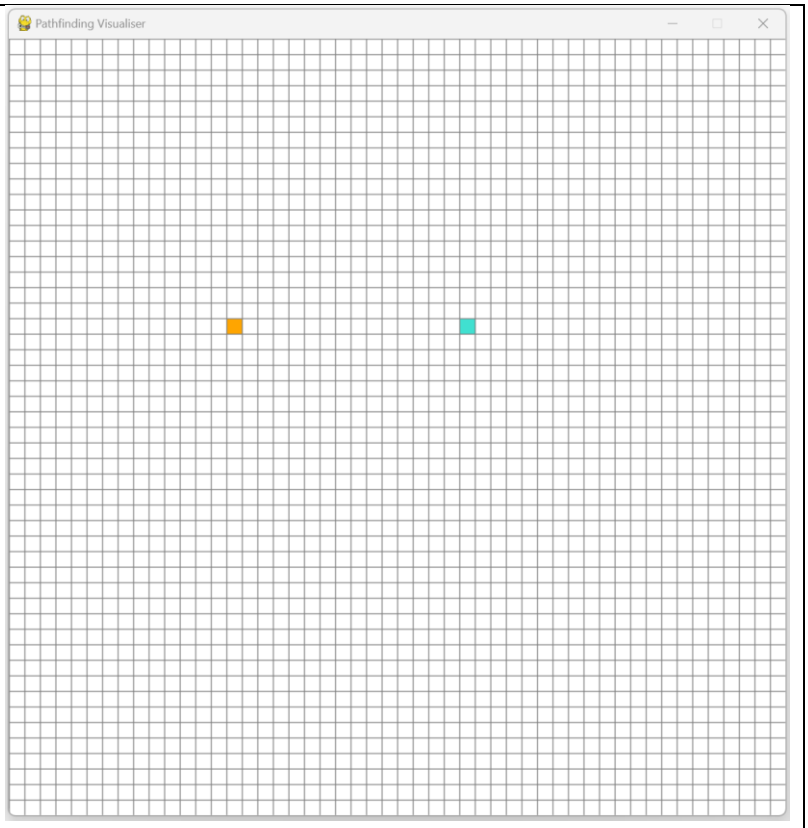
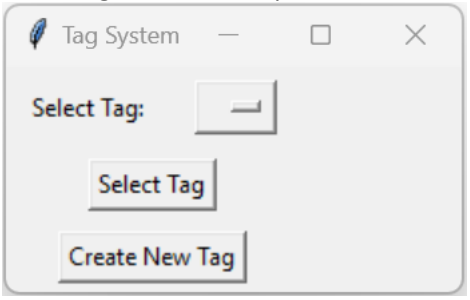
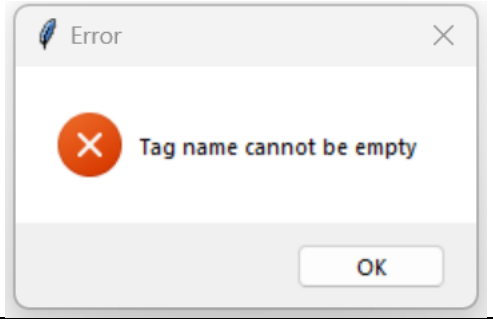
			
11	Login Empty Username	Username: Password: 23rTY*ds	When the app opens, I immediately enter the test data and press “Login”:  The following error correctly comes up: 
12	Login Empty Password	Username: Eno Password:	When the app opens, I immediately enter the test data and press “Login”:

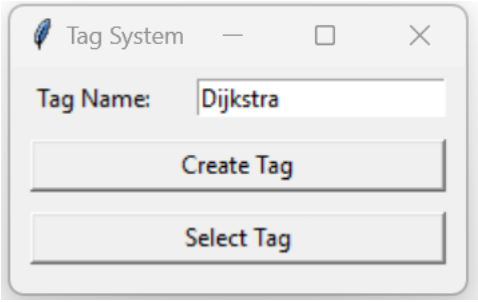
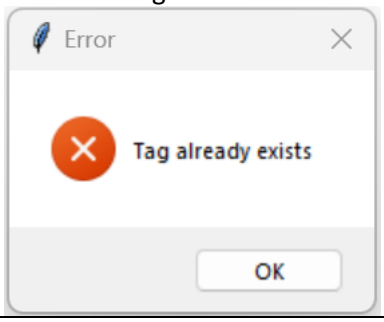
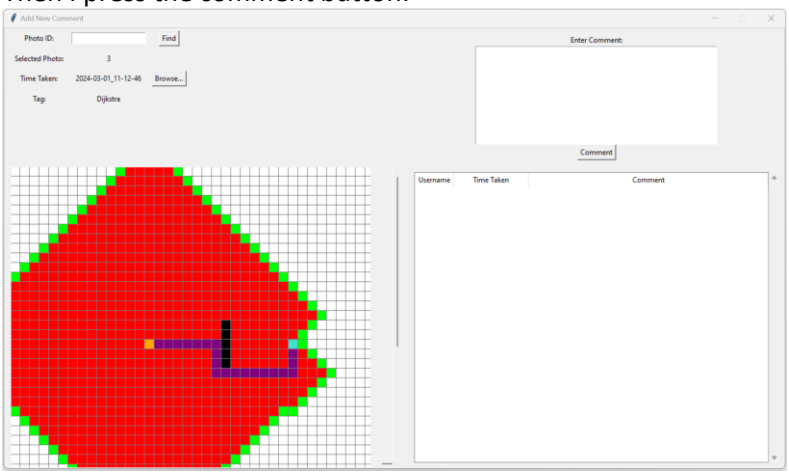
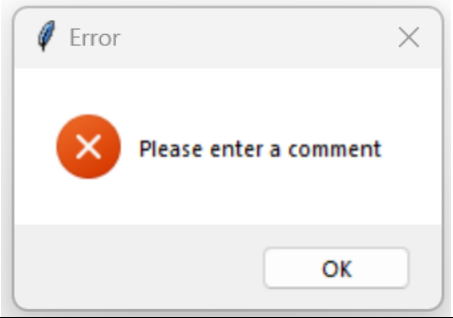
			 The following error correctly comes up: 
13	Login Incorrect Username	Username: DoesNotExist Password: 23rTY*ds	When the app opens, I immediately enter the test data and press “Login”:  The following error correctly comes up: 

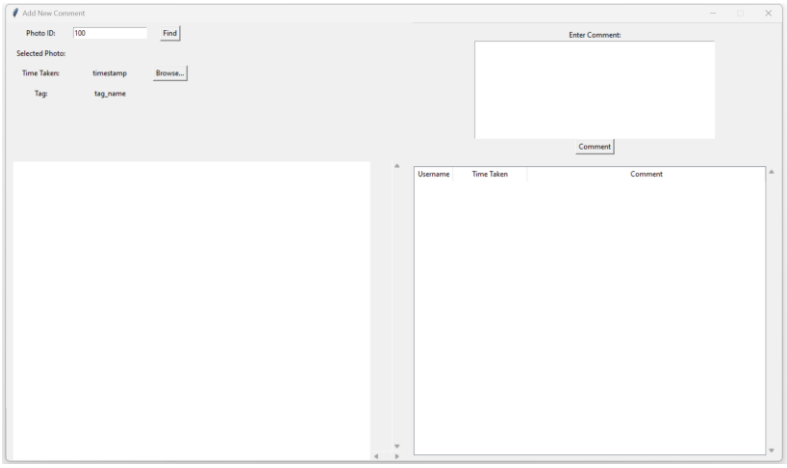
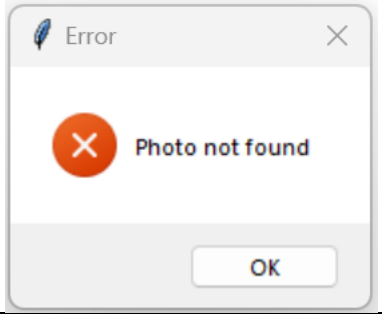
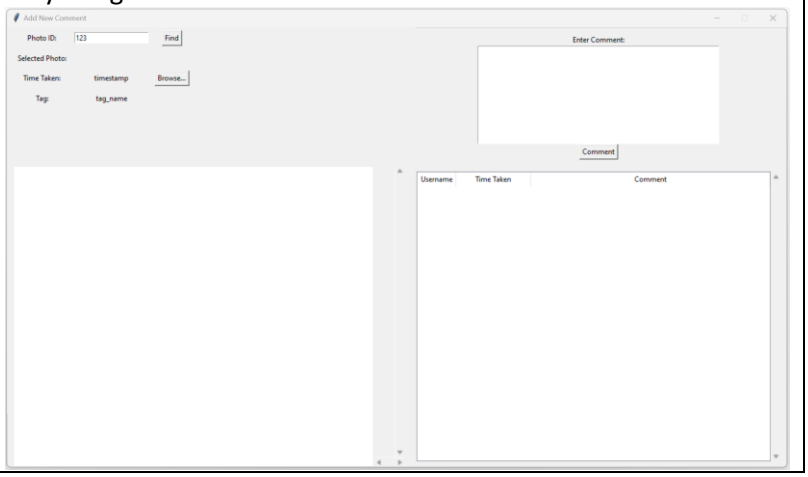
14	Login Incorrect Password	Username: Eno Password: IncorrectPassowrd1*	When the app opens, I immediately enter the test data and press “Login”:  The following error correctly comes up: 
15	Place the end node on top of the start node	I will place the start node on the grid wherever is most suitable.	I log in and place the start node:

			 I now try to place the end node on top by clicking it. The map did not change, this was a success.
16	Place wall on top of start node		I log in, and place the start and end nodes:

			 I press the start node to try and place a wall. The map stays the same, this test was a success.
17	Place wall on top of end node		I log in and place a start node and an end node:

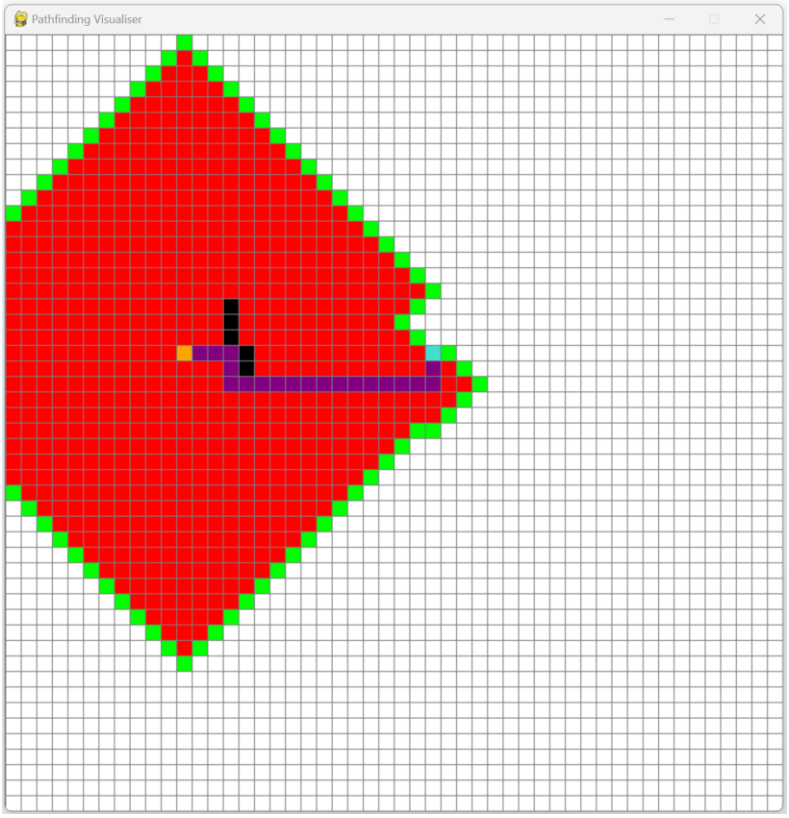
			 I try and press the end node. The map does not change. The test is a success.
18	Select Empty Tag		For this test, I had to use an empty tag database because otherwise, you cannot select empty tags as it will default to the first tag made. First I log in and then I press “s”:  I then press “Select Tag” and get the appropriate error: 

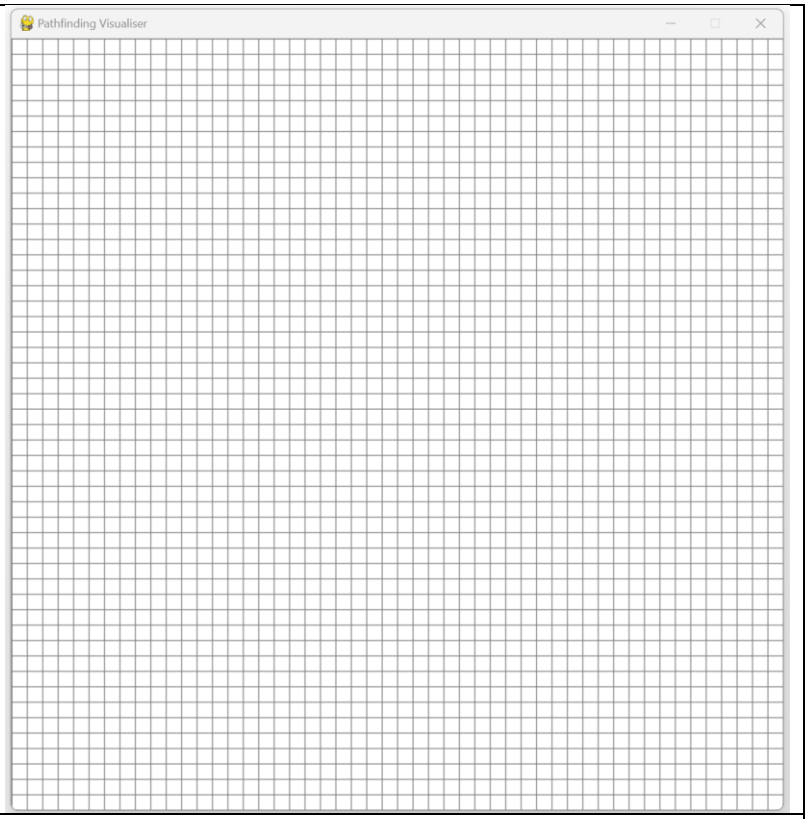
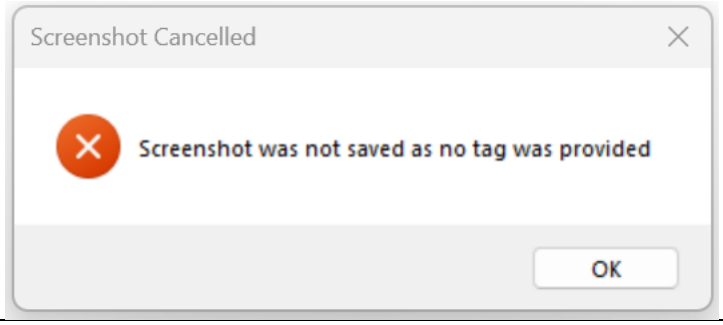
19	Create a Tag That Already Exists	Tag Name: Dijkstra	I log in and press “s” to bring up the select tag window. Then I press “Create New Tag” to go to the tag creation frame. I now type the test data:  I now press “Create Tag”. The following correct error message pops up: 
20	Do not enter a comment but still press “Comment”		I log in to the program, press “c”, browse a screenshot and select it. Then I press the comment button:  The appropriate error message comes up: 

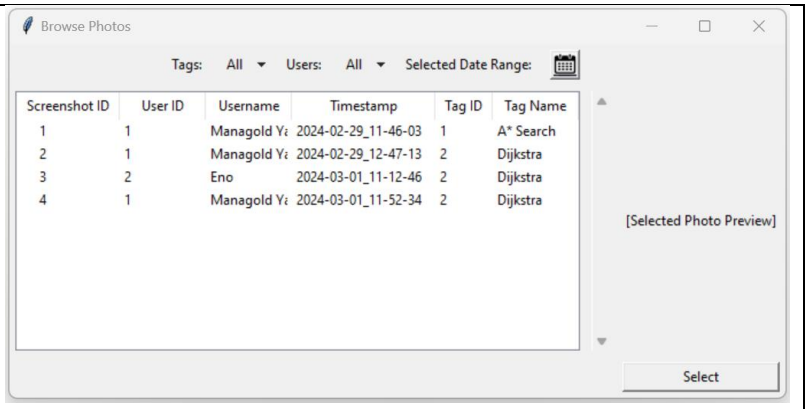
21	Try to find a photo ID that does not exist	100	I log in to the program and press “c”. I then enter the test data:  I press “Find” and the appropriate error message appears: 
22	Try entering characters other than integers into the photo ID entry widget	Photo ID: hello123@*	I log in and press “c”. I type in the test data. The test is successful as it is only left with 123 as they are the only integers: 

4 Feature Testing

The rest of this testing is dedicated to showing the features of the program that have not been covered by the user testing or the error tests.

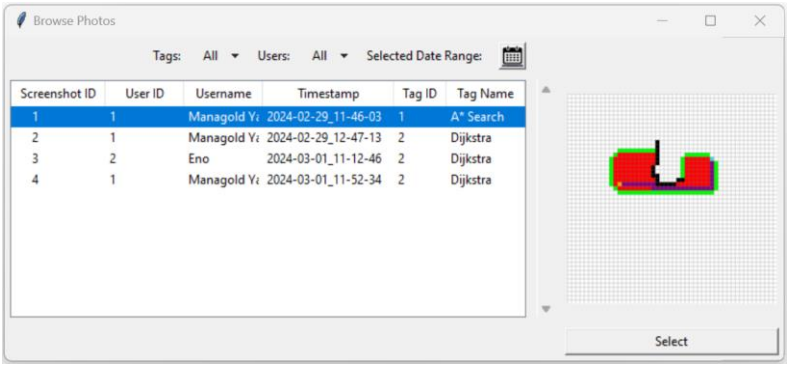
No.	Test	Test Data	Result
1	Empty Grid	None	Firstly, I log in and then I visualise a basic path:  Press “e” as the instruction key set dictates. The map has become empty as before laying out the map:

			
2	Cancel Screenshot	None	I log in lay out a map and visualise it. Then I press “s” to start the save screenshot. I then press the X button and the following popup appropriately appears: 
3	Browse Screenshots	None	I log in and press “c” to go and comment on a photo. It shows up and I press “Browse…” and the following window shows up:

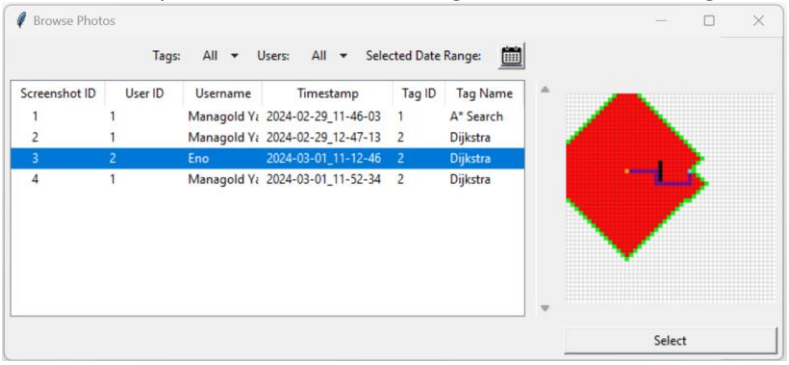


It shows me all the screenshots taken and saved on this database.

I will now click a couple to see what happens:

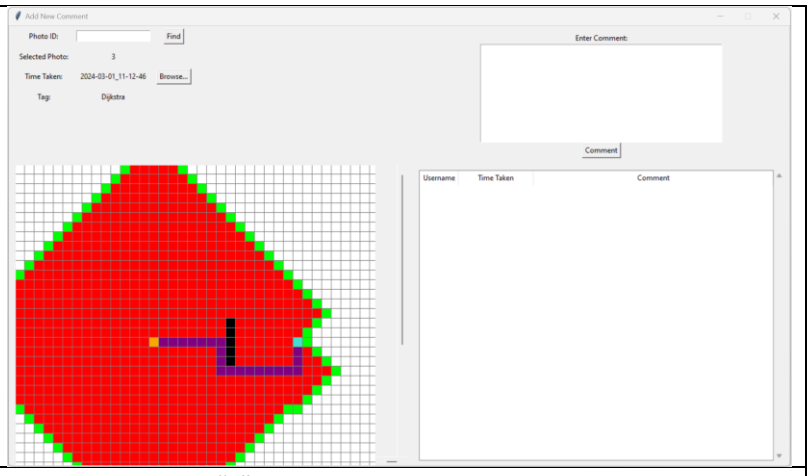
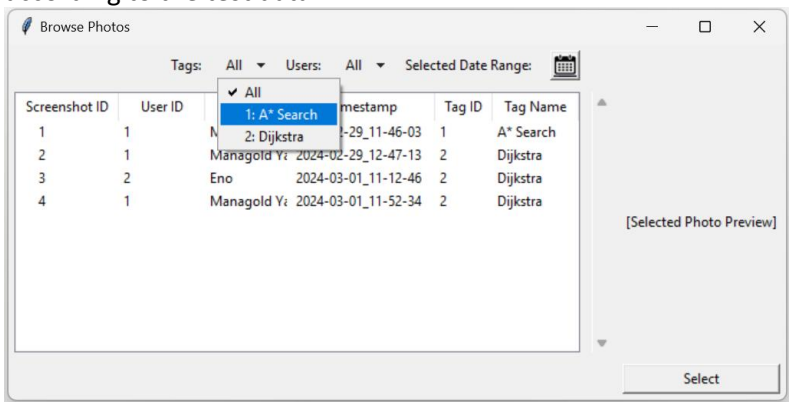
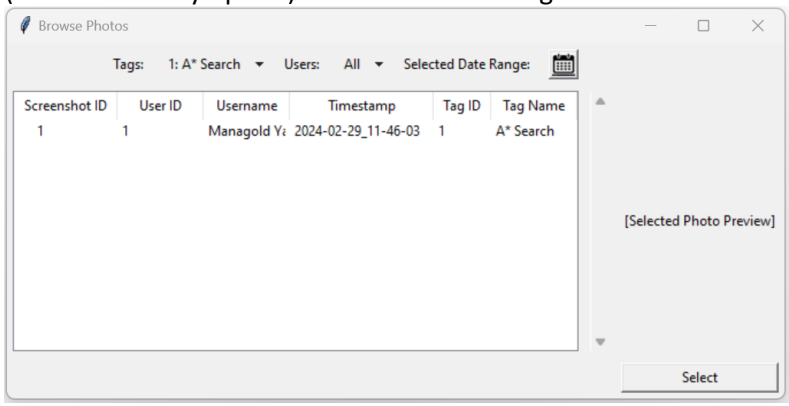


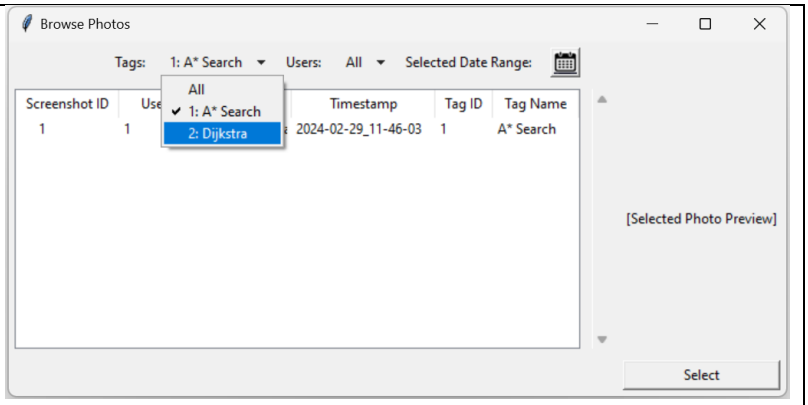
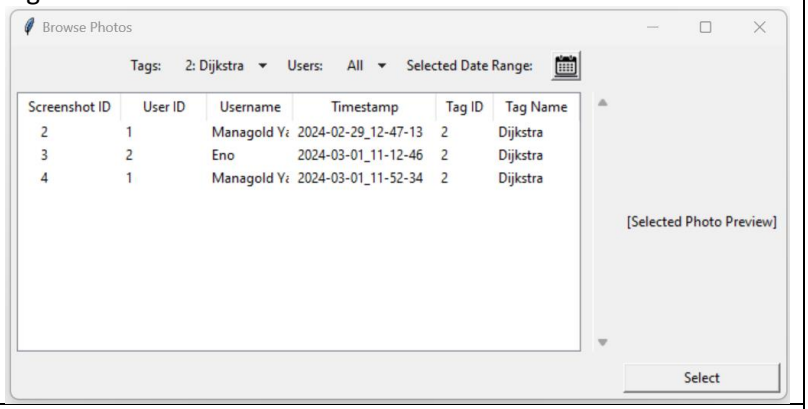
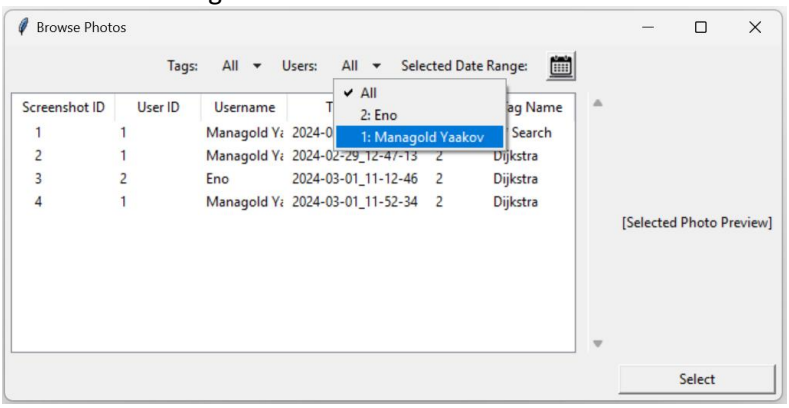
I can see the preview of what the image looks like on the right!

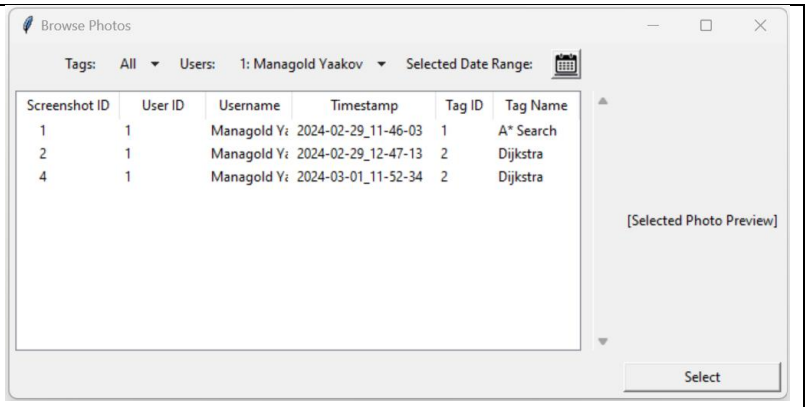
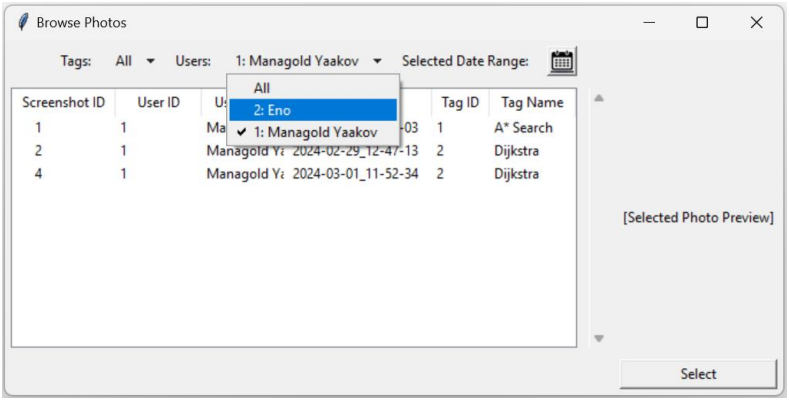
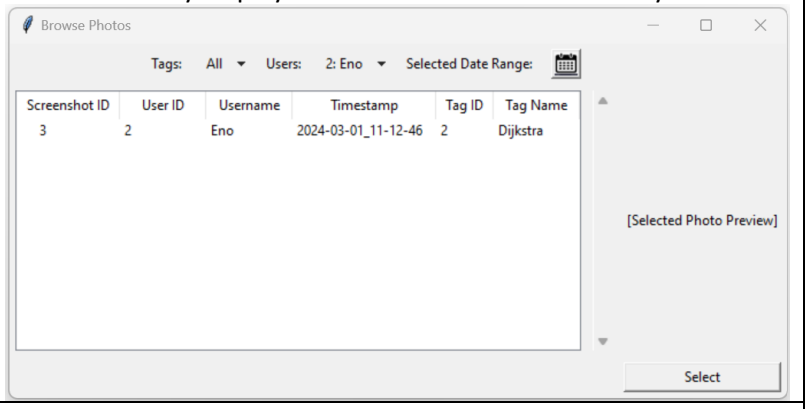


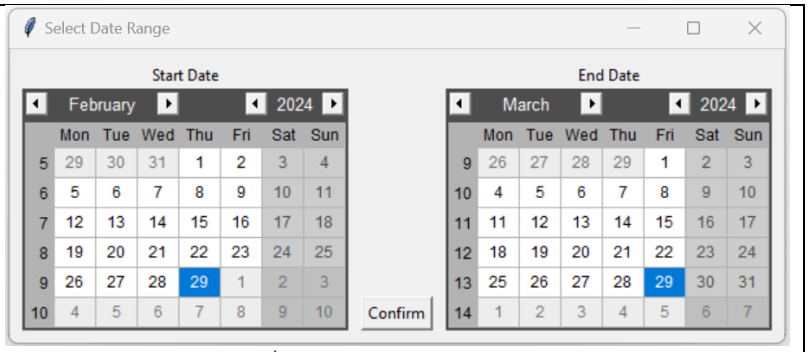
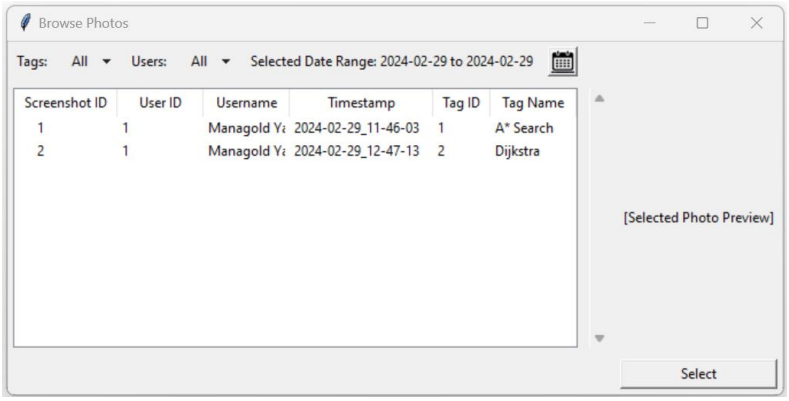
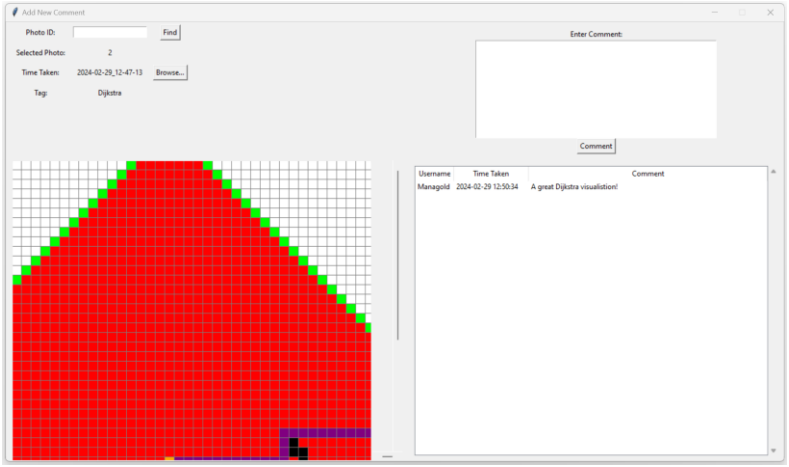
I will now press the “Select” button.

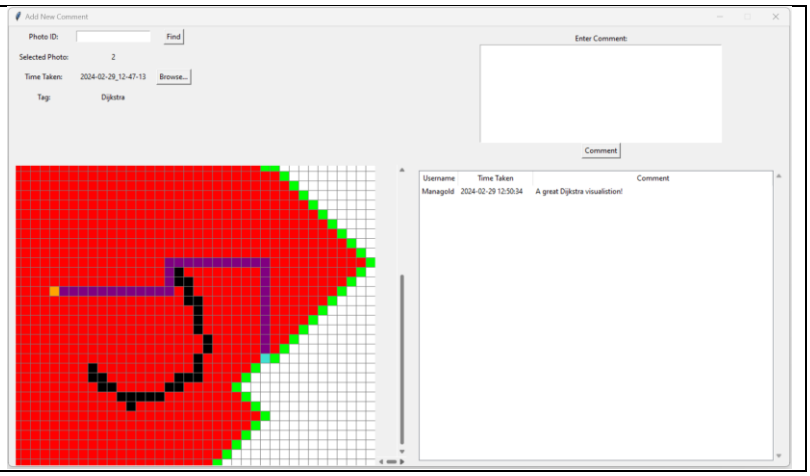
The comment system window is now selected with that photo:

			
4	Browse Filter by Tag	1: A* Search 2: Dijkstra	I log in and press “c” to go and comment on a photo. It shows up and I press “Browse…”. Then I press the tags dropdown and select “1. A* Search” according to the test data:  Instantly, the treeview was refreshed and now I see the options (in this case only option) with the selected tag:  Now I will select the second test data option:

			 Now it correctly displays the saved screenshots with the Dijkstra algorithm: 
5	Browse Filter by User	1: Managold Yaakov 2: Eno	I log in and press “c” to go and comment on a photo. It shows up and I press “Browse…”. Then I press the user’s dropdown and select “1. Managold Yaakov” according to the test data:  Instantly, the treeview was refreshed and now I see the options with the selected user:

			 Now I will select the second test data option:  Now it correctly displays the saved screenshot taken by Eno: 
6	Browse Filter by Date Range	29/02/2024 01/03/2024	I log in and press “c” to go and comment on a photo. It shows up and I press “Browse…”. I then press the little calendar icon and the following screen appears:

			 I selected it for the 29th of February (The picture shows a different selection above) and pressed "Confirm".  The browse window now shows another label next to the selected date range that shows the date range I selected and the treeview has been updated with just the screenshots in the selected time frame.
7	Scroll on the photo canvas view	None	I log in and press "c" to comment and then type 2 into the photo ID widget and press "Find":  I will now test the horizontal scroll and the vertical scroll:

			 Add New Comment Photo ID: Find Selected Photo: 2 Time Taken: 2024-02-29_12-47-13 Browse... Tag: Dijkstra Enter Comment: Comment Username Time Taken Comment Managold 2024-02-29 12:50:34 A great Dijkstra visualistion!

Evaluation

1 Objectives Evaluation

I will start the evaluation of the software by going through the success criteria I previously defined in my Analysis. This is the best way of understanding how successful my pathfinding visualiser is and will be able to give me valuable feedback on what to improve on if this project is revisited by the company to go on and develop it further. Additionally, I will go over my development mistakes and strengths to better provide my skills to future clients.

No.	Criteria	Evaluation
1	User Login System As the CEO he must track his employees' work so having a user login for everyone using the application will ensure the boss knows who is who. The team is a team, working together. They will be able to see who is doing what visualised paths so they can comment on each other's work and hold each other accountable. Stakeholders and the board of directors can use the info on who is doing what work to determine promotions.	I have met this success criterion through my Tkinter login system window. I have a login frame and a sign-up frame that performs validation checks for a smooth experience that meets the security requirements. The database stores the user associated with any screenshot and comment, which allows the CEO to track the productivity of his workers. The fact that the database stores this also allows for collaboration when the dev team is commenting on a screenshot, and they see the names of all their colleagues who have commented. It would now be up to the CEO in the implementation to use this data for determining promotions. My solution fully meets this criterion.
2	Map Building The solution must allow an easy way of building a map on the web application. The mouse will be used to click and create the walls on the map. A left click on the mouse will be used to place walls or the start and the end node. A right click on the mouse will be used to remove walls or the start and end nodes to be placed once again to the user's liking.	I have met this success criterion through my pygame grid. It has a very simple structure of first placing the start node, then the end node and then any walls the user wishes to create to best resemble the map from a satellite image. It handles mouse inputs for the left click which places the start, end, or wall nodes. It also handles the right click which lifts the start, end, or wall nodes. The user can easily drag and drop which significantly speeds up map building compared to having to singularly click every empty node that the user wants to fill. My solution has met the needs of the client and the dev team can easily map build.
3	Pathfinding Functionality The solution must calculate the most efficient path using Dijkstra's algorithm and A* search algorithm (depending on if they press "d" or "a" for each algorithm respectively). The identified path must be screenshotted and saved to an SQL database for future reference upon user request (for example by pressing the "s" key on their keyboard).	I have met this criterion through the implementation of Dijkstra's algorithm and the A* search algorithm as described in my documented design. Every step is visualised for the user immediately upon button press "d" or "a" respectively. Once the final shortest path is found, a purple line is formed that represents the shortest path from the start node to the target node. If the user wants to, at any point they can press "s" and save the grid currently displayed. So long as they select a tag the screenshot will immediately be saved and uploaded to the database.

		I feel like I have fully met this objective also as all the functionality is there, tested and working as intended with good feedback from my client.
4	Minimalist User Interface The solution should feature a user-friendly interface to accommodate users with limited technical expertise. Both Yaakov and the SatellitePathX development team should be able to navigate the application seamlessly. User input for mapping, including start and end points, should be intuitive and straightforward.	I have met this objective via Tkinter's grid system and packing methods. This has allowed me to make simplistic easy-to-follow windows, proved by the user-end testing I performed where with minimal teaching, my client naturally picked up everything they had to do. My grid also has no clutter, only showing the grid and no other buttons that might distract the user. This could be considered a downside also though considering that every employee needs the short note of teaching to start using the program. In the future I could create a help menu accessed by pressing "h" which will explain that the first press will place the start node, the next will place the end node and the rest are walls. It would also explain the different button functions such as "a" for A* search, "c" for comment, "d" for Dijkstra's algorithm visualisation, "e" for empty and "s" for the save screenshot function. This will simplify the instructions significantly to only one button and can avoid a lot of confusion within the workplace. Since the note is short and simple, I will say I mostly fulfilled this objective but if this project were revisited that help menu would alleviate a lot of problems in the dev team. "Just press 'h' for the help menu" versus currently "press 'a' for A* and then press 's' for save and then press…" The explanation is much shorter in the would-be improved version but verbose in my current solution.
5	Cross-Device Accessibility The solution must be accessible from various devices, including desktops and laptops, but not mobile phones as my client said that is not necessary. Users should experience consistent functionality when accessing the application from different computers.	I have met this criterion as my program is in Python and thus all devices that can run Python and the packages, I installed will have consistent functionality, whether their OS is MacOS, Windows or Linux. It is all possible and consistent. It could be argued that I could increase this universality by using a Python program to convert the Python to an executable (.exe) file to simply be double-clicked and run without the Python requirements. This would increase the start time by a few seconds but increase the potential devices that could run this without having to install Python. Due to this fact, I think I have mostly met this criterion and if this project were revisited, I could interview my client again for more details and discussion on if an executable function would be more appropriate.
6	Real-Time Visualisation Real-time visualisation of the pathfinding algorithm must be a feature of the solution. Users should be able to observe the algorithm's progression as it calculates the optimal path.	I have met this objective completely, as the pathfinding algorithm visualises every single step of the way. In real-time my program certainly does this and is backed up in my testing. One thing that could be added here is the use of weights for roads with lots of traffic, or longer distance.

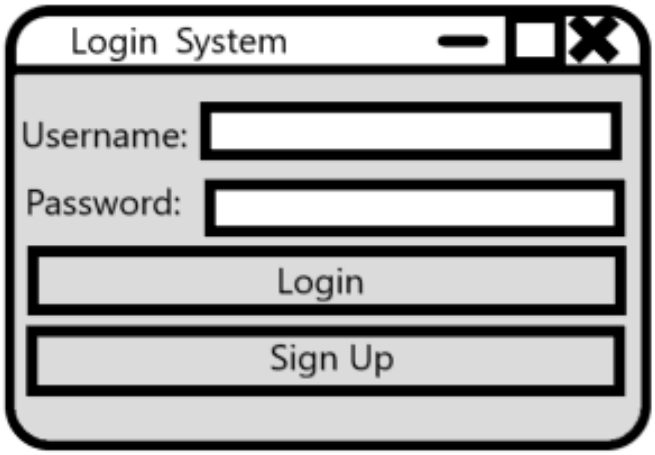
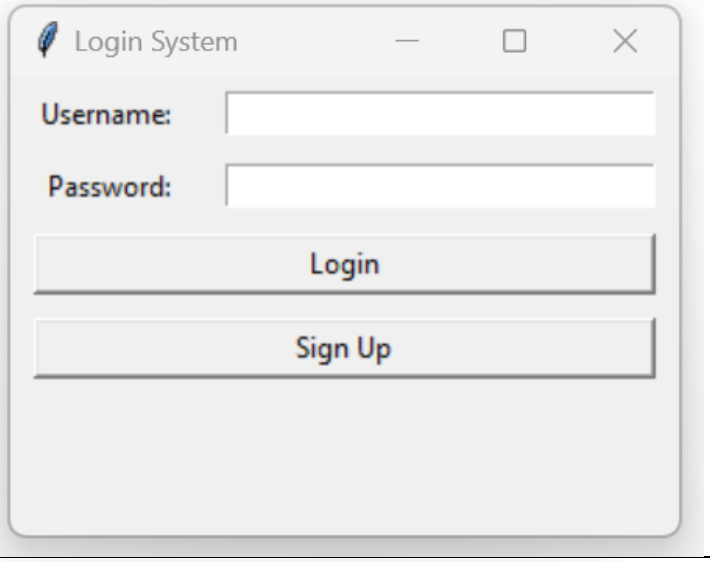
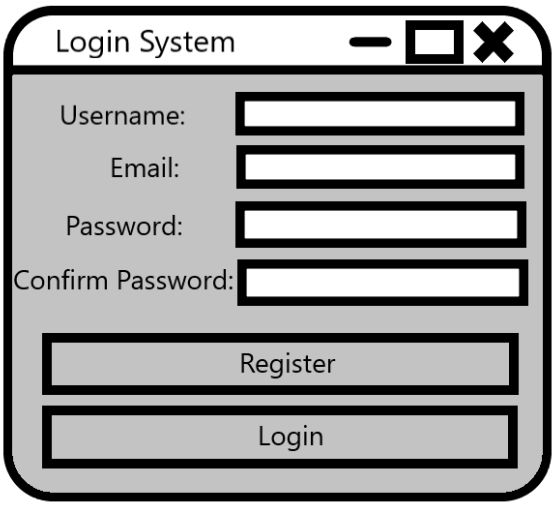
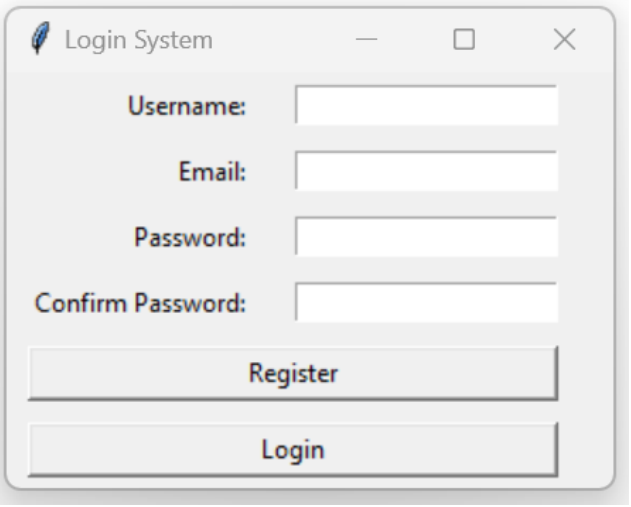
		Another thing would be the use of a better heuristic than just the Manhattan distance, which is good, but not great. This could be worked on in future implementations. I have met this objective according to industry standards.
7	Tagging System The software team using the software must be able to tag their saved screenshots to save them into different categories that they need (e.g., “London”, “Coffee Shop”). The software team will have to be able to make their tags.	I have met this criterion with the use of my tag system. My tag system contains two frames, a select tag frame and a create new tag frame. The select tag frame forces a tag to be selected for every visualised path which will ensure organisation in the database and each screenshot is taken with purpose. The tag system also works in the browse screenshots window where the user can filter by selected tag. This shows that the tag feature is fully functional, as shown in the testing phase also, thus showing how I have met this objective.
8	Comment System The team will be able to make comments on their visualised paths and then see the comments made by team members. To find the photo they wish to comment on, the user must be able to browse the saved screenshots and see a small thumbnail of the visualised path before confirming their selection to speed up commenting and avoid mistakenly choosing the wrong screenshot, which would slow them down and frustrate the dev team.	I have met this objective with the use of my comment system. The comment system window allows the user to browse screenshots to select the one they are looking for that meets their requirements. A thumbnail is shown on the right before they confirm their selection to avoid mistakes when selecting a photo. It can also be found by the screenshot ID if necessary for example in a team setting where they are working on a specific screenshot. The comment system window then shows a tree view of all the comments with the user who made it and the time they made it. This meets the requirement as it allows for collaboration and easy discussion. Overall, I have met this objective sufficiently.

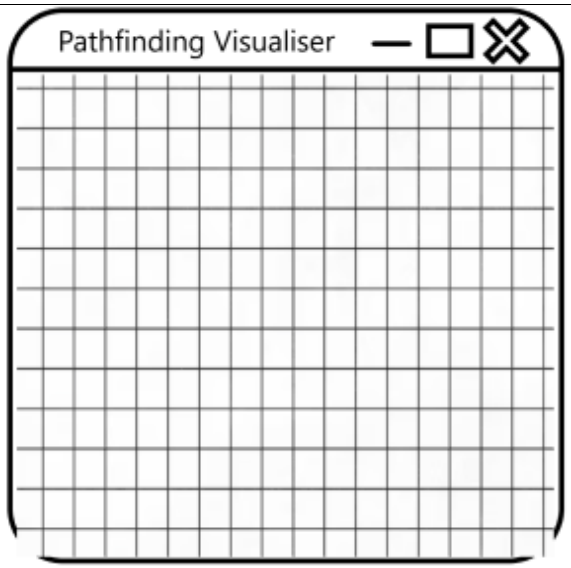
2 Summary of tasks for further development

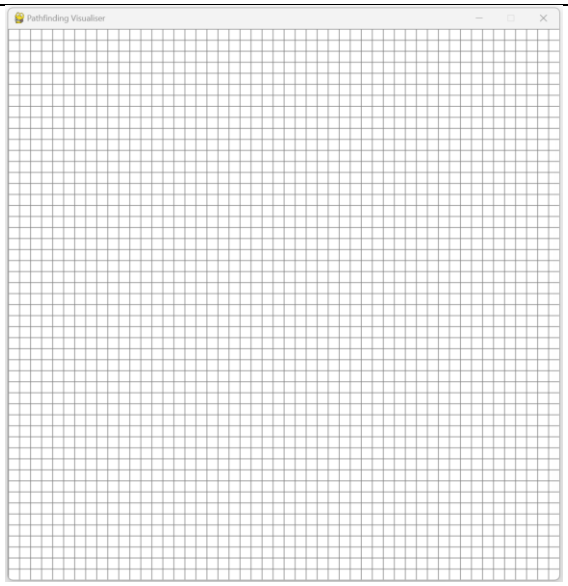
Based on the evaluation of the success criteria in the future I will focus on developing:

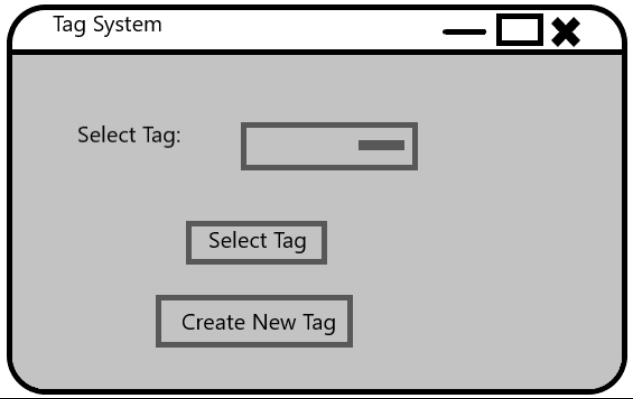
- A help menu would be accessible by pressing “h” when logged in and on the grid. The menu would include the other keyboard instructions and give a simple one-line explanation of how the map creation system works.
- The creation of an executable file so the need for Python to be installed on the computer is removed and thus each user can begin using the program with minimal setup.
- The implementation of weights between the nodes to indicate high traffic, or that one road is longer than the other.
- The implementation of more complex heuristic functions to test the best ones for use within the company.

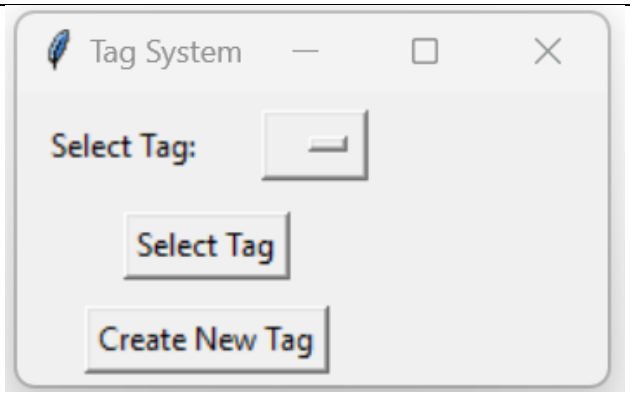
3 Comparing my Design to my Implementation

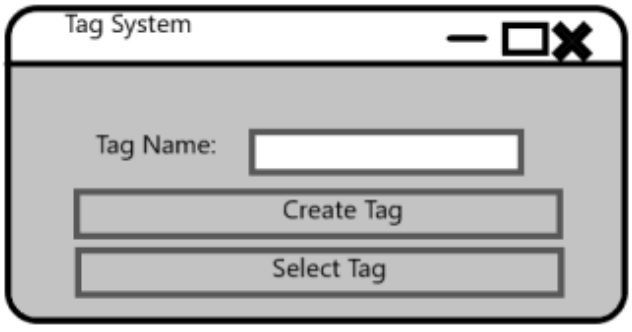
Design	Implementation
	
	

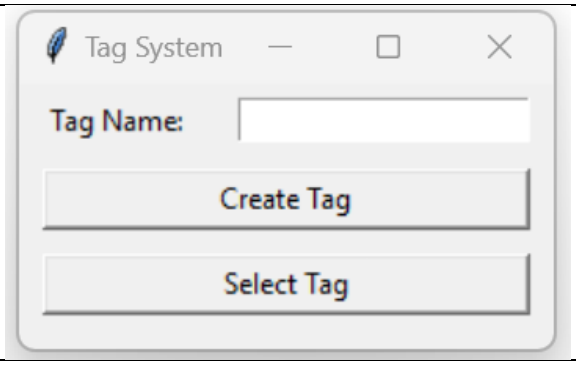

Pathfinding Visualiser

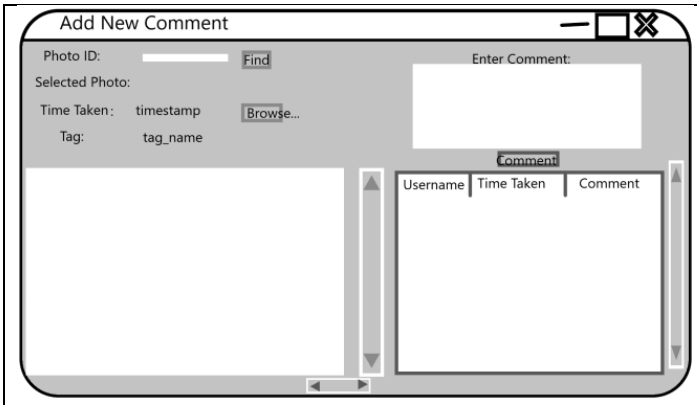

Pathfinding Visualiser

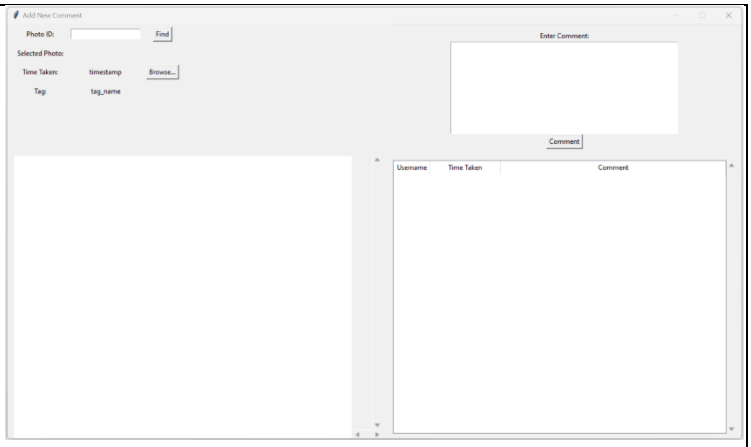

Tag System
Select Tag:
Select Tag
Create New Tag

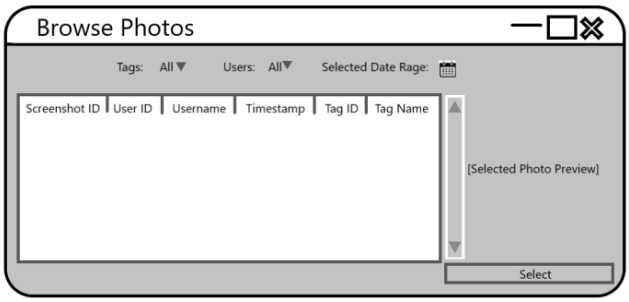

Tag System
Select Tag:
Select Tag
Create New Tag

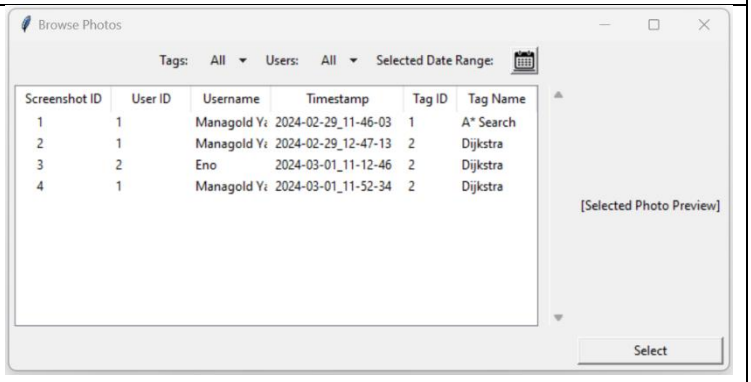

Tag System
Tag Name:
Create Tag
Select Tag


Tag System
Tag Name:
Create Tag
Select Tag

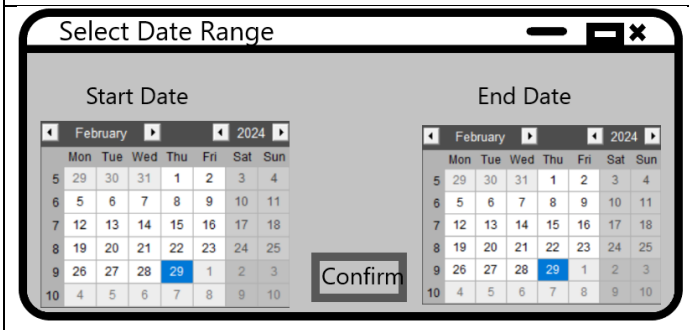


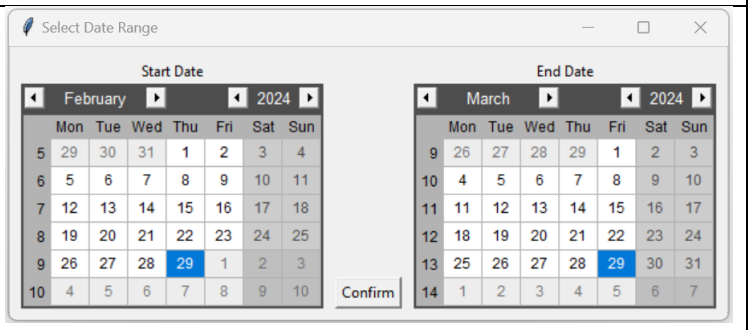






This screenshot has some data entries in it.





It is evident that all my results are almost identical to my designs showing how reasonable and achievable they were in the first place.

There are minimal differences between the two and thus I have successfully achieved the desired look which is both simple and effective.

4 Maintenance of the Solution

I have built my solution from the latest versions of Python and all packages. Despite this, newer versions of the packages are constantly being released and there is a risk of some of the functions being deprecated. New patches often improve security or change functionality slightly to improve performance. I have used core functions which are the staple of the packages in my code, so this is unlikely to be a risk for this solution. There is no potential for security vulnerabilities except for in the `sqlite3` package. The dev team can keep the rest of the packages in their current versions. This will decrease the risk of a sudden break in the code or glitches.

If the team does decide to upgrade their environment versions, the documentation in the new updates would have to be rigorously studied to ensure breaking changes are not pushed to the repository. All changes the dev team makes must be documented and logged with testing at every change so that any breaking changes can be easily detected and fixed if necessary. If they try to update too quickly and unnecessarily it could result in the program breaking and a lot of stress on the CEO and dev team, completely halting their operations until the bug is fixed. Any 'quick fix' might be a poor short-term solution and cause a lot of long-term problems to accumulate.

Adaptive maintenance involves modifying the system to accommodate changes in user requirements, industry standards or technological advancements. When a breaking change is identified I will adapt my software and documentation to fix it. I will update my code by pulling out the latest changes from my remote repository (local computer storage) with the desired changes and committing them using a tool called "git". Just before this, I will use a command called "git add" to prepare the changes for a commitment. Once committed locally I will push my changes to my remote repository using "GitHub" (I already have an account set up). I will use the command called "git push" which uploads the committed changes to my GitHub account. Each commit will have a description commit message to easily backtrack to previous versions of my bug tracker just in case an error in the latest version arrives. The repository will obviously be private for confidential reasons within SatellitePathX. This will ensure that PathPlanner (this solution) Is never offline for too long.

My code is 'self-commenting code'. This will help with maintenance as it is so simple, with descriptive names and coherent logic that can be followed through, even by novice programmers. This way it is less susceptible to bugs and errors being made. Even if an error is made, it would be easy to spot and correct due to how logical my code is. It utilises a lot of classes and modules that are separated logically to make any function easy to find and follow. Over time, following these conventions will make it easier to manage the code if it gets into the thousands of lines of code (roughly 1500 right now). New developers, even novice ones, will have an easy time understanding and contributing to the solution if they seek to revisit this project, I have created for them.

Uploading the code to a GitHub repository would also provide extra security against a security breach and the files being deleted or erased from physical damage to components. GitHub's servers are extremely reliable and will keep the codebase safe for future use also.

5 Conclusion

At the end of this project, after the end-user testing, my client and I had a final meeting in which we went over the entire project in a summary and discussed the potential additions that could be made to the program, which I discussed a little earlier. He gave me feedback also. He said that he was happy with my work ethic and solution as I stayed on schedule and fulfilled the objectives I laid out in the beginning, delivering on all my promises. He thanked me for the readable code because even the CEO not being tech-savvy could make sense of the code and what it was doing. He said his team would be confident developing it from there and using it to make business decisions.

This meeting is also where I gave my advice to the business about their next steps. I told him, Mangold Yaakov, that the A* search algorithm is best used when the start point and the target point are known on the map. This is when the user is going from one point to a known destination. This is because it is optimised this way to use the minimum resources to find the shortest path in this scenario. I then let him know that he should use Dijkstra's algorithm when the user of the GPS wants to find nearby shops or destinations such as cafés, restaurants or parks. This is because Dijkstra's algorithm searches equally on all sides until the target is found and so if the end point is not necessarily known, Dijkstra's algorithm would be most effective. He was pleased with this guidance and told me that he would let his team know in his next meeting, so they can get to work on testing my advice. He said that his team would work immediately on getting the codebase into a repository on GitHub for the employees to fork and download to complete their tasks. This is where our meeting ended.

In terms of my overall thoughts, I think that the codebase was very strong and very polished with a well-designed file structure. I used the best practices across the industry and wrote code that would be considered 'self-commenting code' which will easily be built upon my client's software team.

Overall, I have completed what I set out to do at a high standard and gave direction to my client's company that will solve their issue of renting maps and push their business to the next level of propriety GPS location software.

Bibliography

Microsoft. (2015). *Windows 10*. Retrieved November 2, 2016, from https://www.microsoft.com/en-gb/windows/windows-10-specifications

Mihailescu, C. (2016). from https://github.com/clementmihailescu/Pathfinding-Visualizer

Tyriar. (2012). from https://github.com/Tyriar/pathfinding-visualiser