StockBot

A-Level Computer Science Project

Praveet Minash Khambhaita

Teacher: Mr J Bell

The Nobel School - Centre Number: 17721

Candidate Number: 5117

This project is submitted for:

H446 - OCR A-Level Computer Science (NEA)

Class of 2025

Declaration

I have read and understood the Notice to Candidate. I certify that the work submitted for this assessment is my own. I have clearly referenced any sources and any AI tools used in the work. I understand that false declaration is a form of malpractice.

Praveet Minash Khambhaita
Class of 2025

I declare that all stakeholder information is true and their own.

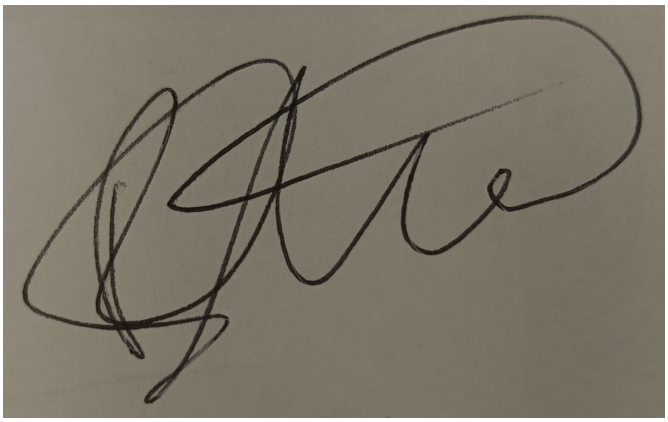


Fig. 1 Signed Mr Ryan Slater

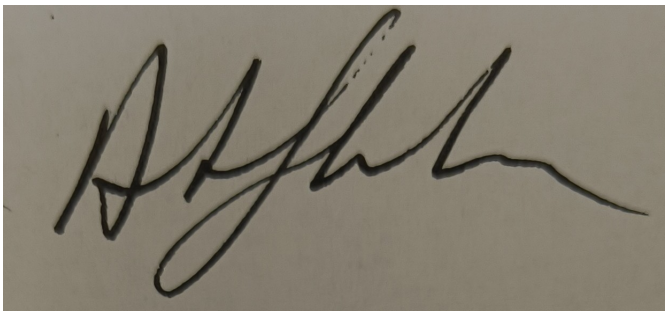


Fig. 2 Signed Mr Alex Johnson

Preface

This project represents many months of dedication, learning, and passion for creating technology that solves real-world problems. As the son of a hard-working warehouse employee at Currys, I've witnessed first-hand the challenges faced by retail workers who navigate large stockrooms daily. Their efficiency directly impacts customer satisfaction, yet the tools available to them often fall short of modern technological capabilities.

StockBot was born from these observations: a solution designed to optimise the simple but critically important task of item collection in warehouse environments. The inefficiencies I observed during visits to my father's workplace provided inspiration for the solution developed here.

This application has been proudly developed using open-source tools on Fedora, the leading edge of secure, accessible and transparent technology. From the Python language to the SQLite database system, every component of StockBot is built upon FOSS and using FOSS tools where possible.

This project demonstrates that even complex computational challenges can be addressed with relatively simple, elegant solutions when we take the time to thoroughly understand the problem space. The pathfinding algorithms employed here may not be revolutionary, but their application to the specific constraints of retail warehousing creates a tool that can meaningfully improve workplace efficiency and reduce physical strain for warehouse workers.

I hope that StockBot serves as both a practical solution and an example of how computer science students from all walks of life can develop applications that address real-world challenges. Thank you for taking the time to read my project, and I hope you find it interesting & insightful.

— Praveet Minash Khambhaita
@pmkhambhaita

Table of contents

Section 1

Analysing the problem

1.1 Defining the problem itself

Warehouses often employ several workers for menial labour jobs such as the retrieval of thousands of items over the space of an average day. This business model is impractical due to the costs of hiring warehouse workers and managers for such simple tasks as noting stock levels and collecting items. My project aims to optimise warehouse operations through a **hypothetical** robot - the StockBot. It will retrieve multiple stock items at a time, finding the shortest path required to collect all items and reach its destination. The program will utilise a path-finding algorithm to trace the most optimal path to pick up items and drop off at a fixed location, while cross-referencing a database to ensure items are in stock. I envision this to be applied in a warehouse or alternative bulk storage facility, such as that of an e-commerce business. My solution reduces business costs and improves efficiency, removing a large portion of human operations and replacing them with autonomous workers.

Fig. 1.1 - A sample warehouse environment. [7]

1.2 Current systems model

The current process model for a warehouse worker typically involves receiving orders, compiling product lists, collecting items from designated locations, and placing them in shipping areas.

While this manual process is feasible for smaller operations, it becomes increasingly inefficient and error-prone as the scale of the warehouse expands. Human workers are susceptible to errors, such as picking incorrect items or placing them in the wrong locations. Manual processes can also be time-consuming, especially for large orders or complex warehouses. Additionally, repetitive tasks like collecting and placing products can lead to fatigue and reduced productivity. These factors contribute to higher operational costs and potential delays in order fulfilment.

The limitations of human workers can be addressed through automation; robots can perform these tasks with greater accuracy and efficiency, reducing the risk of errors and improving overall productivity. Unlike humans, robots can operate continuously without breaks or rest, enabling 24/7 operations.

While the initial investment in robotic technology may be higher, the long-term cost savings from reduced labour and improved efficiency can make it a worthwhile investment. Furthermore, robots can enhance workplace safety by handling hazardous materials or performing tasks in dangerous environments. This reduces the risk of injuries to human workers and improves overall safety standards within the warehouse.

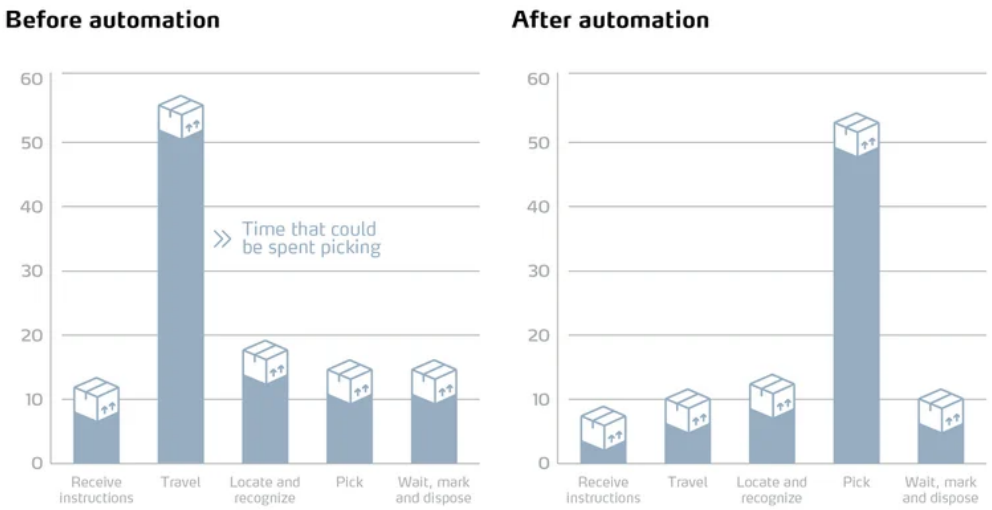


Fig. 1.2 - Automation benefits [8]

As you can see in Figure 1.3, the current model of a warehouse worker seems less complex than a computational approach, however the focus is not on the size or number of elements, but rather the time taken to complete each task. While a computational approach may have more tasks, it can complete them in a much shorter time, leading to better efficiency.

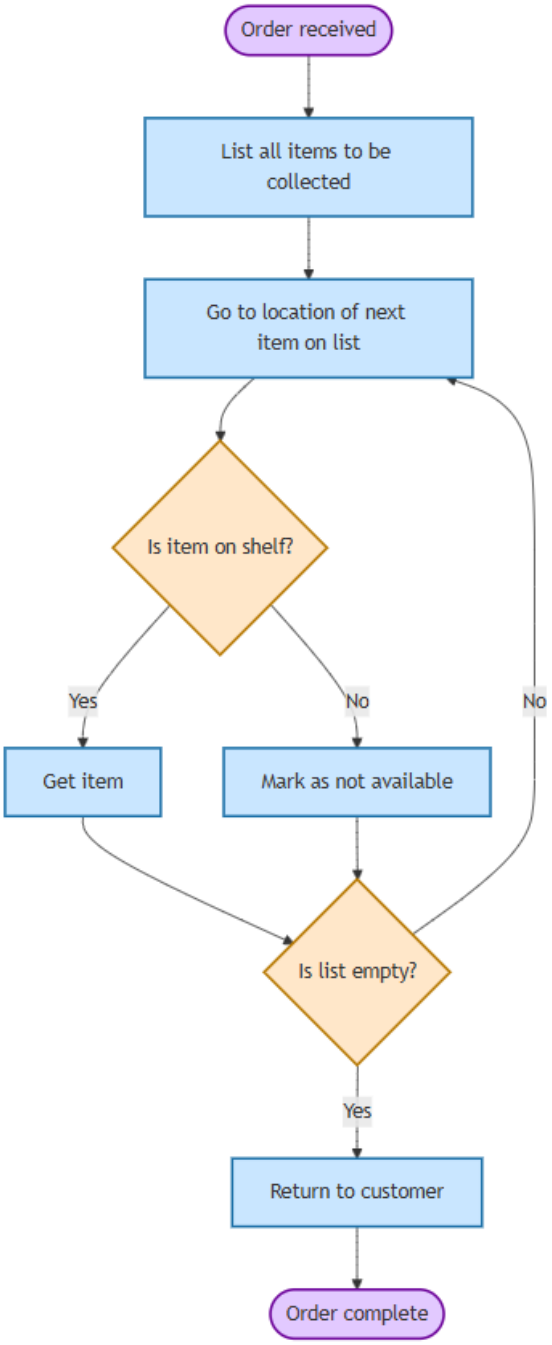


Fig. 1.3 - The current process of a warehouse worker

1.3 Computational methods

This problem is well suited for a computer as there is a valid and usable solution using computational methods. StockBot can perform operations much more efficiently than a human, as these are menial tasks requiring minimal thinking power but rather a continual set of instructions to perform certain tasks **and those tasks only**, a framework which computers excel at. Below I have listed justifications of a computational approach to this problem.

1.3.1 Thinking Abstractly

Since this is not an industrial project, I will need to decide what aspects of the solution I can remove to make my program more appropriate, keeping only the features that are necessary. Initially, these are the abstractions I proposed, which I will consider with my stakeholders:

- Restrict warehouses to 2 dimensions: 3D warehouses will be difficult to represent or visualise, and adds unnecessary complexity as items are often grouped vertically, meaning from a 2D perspective they are all located at that position, regardless of height.
- Grid layout only: most warehouses are grids for optimal storage capacity. I felt that the extremely small minority without such a layout would not benefit from the solution, hence a focus on rectangular formats only.
- Simple visualisations: Too complex GUIs can lead to unnecessary complexity and cognitive overload, therefore I will be focussing on creating an interface with only the functions I implement (while still maintaining usability), with little else in terms of design.

1.3.2 Thinking Ahead

I plan to use Python to develop my solution, due to its versatility and easy-to-understand syntax as a high-level language. When developing my program, I aim to follow an object-oriented approach, as it allows for simplified debugging, excellent organisation and reducing duplicate code snippets (the method can just be called again.) See below for a table of technologies I aim to use.

Tech	**Summary**	**Justification**
Python	High-level programming language	Versatile, easy-to-understand language
Object-Oriented Programming	Programming paradigm focusing on objects	Simplifies debugging, enables code reuse and improves organisation
SQLite	Lightweight SQL database	Integrates well with Python, easy storage solution
Tkinter	Python GUI library	Creates a user-friendly, easy to understand interface for my program

1.3.3 Thinking Procedurally

I plan to break my solution down into 3 main components for the final solution - this is most likely how my file structure will be split:

- Shortest Path Algorithm - this is the central part of the program, the back-end which makes the program useful.
- User interaction - this will be all sub-components relating to how the user interacts with the program, from the GUI to help menus and so on.
- Database operations - this will be for managing the warehouse in which the program is operating in.

1.3.4 Thinking Logically

This focuses on the more logical aspect of my program, i.e. ensuring not only my approach is logical but also features of my program are understandable and each feature has unique characteristics: this prevents duplication and unnecessary code being written.

1.3.5 Thinking Concurrently

This will be useful for determining what operations or sub-programs should be running at any given time, and how I can optimise my program for faster usage times.

1.4 Stakeholders in a solution

1.4.1 Stakeholder definition

My end user for such a product could hypothetically be any business, small or large, that uses warehouses or other organised storage systems. For example, an e-commerce business could benefit from a robotic worker to efficiently pack multiple orders.

In my case, my father will be the primary stakeholder - he works for Currys, where I have seen first-hand the day-to-day warehouse operations as well as possible improvements that could be made to the current model. My secondary stakeholders are 2 teachers, Mr Ryan Slater and Mr Alex Johnson, both of whom have taken a vested interest in this project.
To evaluate which features they would require, I will conduct some interviews and questionnaires to build the best possible product and implement the most relevant features.

1.4.2 Stakeholder use of solution

My father will be able to use the stock robot to:

- **Automate repetitive tasks:** The robot will take over tasks like retrieving stock and keeping track of inventory, freeing up employees to focus on more complex tasks.
- **Improve efficiency:** The robot will be able to work faster and more accurately than humans, leading to increased productivity and faster management, meaning that new stock can be ordered much more quickly.
- **Reduce errors:** The robot will be able to scan stock codes and verify items, reducing the risk of retrieving the incorrect item.
- **Improve safety:** The robot will be able to navigate the warehouse in a safer manner compared to a human, and could eliminate the risk of injury from items falling from shelves.

1.5 Investigation

1.5.1 Primary Stakeholder Data

As a worker of Currys, my father is often required to retrieve many items over the course of his work-day, and he often takes up to 20 mins to find an item. To address this problem, I intend to collect information through my father over the course of a work week to address the most critical issues he faces, and consult my secondary stakeholders for confirmation and extra ideas.

Data collection

From my father, I will be collecting the time taken to process a customer's request for an item. I will take 5 samples out of his day across 5 days, totalling 25 interactions. I feel this is an appropriate sample to judge current operations in his workplace.

Categories:

1. Warehouse: Time taken to reach the warehouse from a random point in the store. While my project may not have a significant impact on this time as it will be operating elsewhere, I felt it necessary to gain a full picture from start to end.

2. Locate: Time taken to locate the item that the customer is requesting. This is what StockBot will have a significant impact on in terms of optimising business operations and efficiency.

3. Return: Time taken to return the item requested to the customer. StockBot should have medium impact on this, as workers can simply wait by the doors of the warehouse rather than having to walk in and out.

Gathered data

Day	#	Warehouse	Locate	Return	Total
Mon	1	2:14	7:36	2:05	11:55
	2	3:25	5:42	2:38	11:45
	3	2:52	10:15	2:33	15:40
	4	3:12	7:18	1:47	12:17
	5	2:38	8:12	2:25	13:15
Tue	1	4:03	6:22	3:10	13:35
	2	2:45	5:38	2:12	10:35
	3	3:18	9:45	3:07	16:10
	4	2:59	6:53	2:43	12:35
	5	3:42	7:15	2:21	13:18
Wed	1	3:27	5:13	3:05	11:45
	2	2:49	11:33	2:38	17:00
	3	3:05	8:07	1:55	13:07
	4	3:37	6:48	2:15	12:40
	5	2:42	7:19	2:32	12:33
Thu	1	3:52	6:45	2:57	13:34
	2	2:38	9:12	1:58	13:48
	3	3:23	7:47	2:33	13:43
	4	4:05	8:23	3:12	15:40
	5	2:51	5:19	2:07	10:17
Fri	1	3:32	7:55	2:22	13:49
	2	3:48	6:38	2:41	13:07
	3	2:57	5:28	2:13	10:38
	4	3:15	8:45	2:52	14:52
	5	3:08	7:12	2:37	12:57
Average		**3:11**	**7:31**	**2:31**	**13:13**

Questionnaire: Primary Stakeholder

Purpose

This questionnaire is designed to gather detailed insights from my father. The goal is to understand the challenges he faces in his daily tasks and how the proposed solution can address these issues effectively.

Section 1: Current Challenges

1. **How much time do you typically spend searching for an item in the warehouse?**
    - Less than 5 minutes
    - **5–10 minutes**
    - 10–20 minutes
    - More than 20 minutes
2. **What are the most common difficulties you face when searching for items manually?** (Select all that apply)
    - **Items are misplaced or not where they should be**
    - Difficulty navigating the warehouse layout
    - Lack of real-time stock information
    - **Fatigue from walking long distances**
    - Other (please specify): **Large warehouse means long search times**
3. **How often do you encounter situations where an item is out of stock or unavailable?**
    - **Rarely**
    - Occasionally
    - Frequently
4. **How do you currently track or verify stock levels before retrieving an item?**
    - Manual counting
    - Checking a paper-based inventory list
    - **Using a digital system (if applicable, please describe): Proprietary software**
    - No formal process

5. **Is the tracking method outlined above accurate?**
    - Rarely
    - Occasionally
    - **Most of the time**
    - All of the time
6. **On average, how many items do you retrieve in a single trip to the warehouse?**
    - **1–5 items**
    - 6–10 items
    - More than 10 items
7. **Have you explored the possibility of an automated workforce in the warehouse?**
    - **Yes**
    - No
    - Unsure
8. **If yes, why was it not implemented?**
    - **Cost**
    - **Lack of justifiable benefits**
    - Poor usability
    - Bad user experience
    - Prefer not to disclose
9. **What is a necessity to even consider this solution?**
    - **Faster collections**
    - Good usability
    - Scalability

Section 2: Potential Improvements

1. **If a system could automatically check stock availability before you search for an item, how useful would you find this feature?**
    - **Very useful**
    - Somewhat useful
    - Not useful

2. **Would obstacle avoidance have a large impact on your warehouse?**
    - Yes, significantly
    - **Yes, but only slightly**
    - No, it wouldn't make a difference
3. **How important is it for the system to provide real-time updates on your progress (e.g., which items have been collected, remaining items)?**
    - Very important
    - **Moderately important**
    - Not important
4. **How important is it for the system to provide real-time updates on progress (e.g., which items have been collected, remaining items)?**
    - Very important
    - **Moderately important**
    - Not important

Section 3: Usability

1. **How comfortable are you using digital tools (e.g., apps, software) for work-related tasks?**
    - **Very comfortable**
    - Somewhat comfortable
    - Not comfortable
2. **How important is it for the system to have a simple and intuitive interface?**
    - **Very important**
    - Moderately important
    - Not important
3. **How frequently would you use such a system if it were available?**
    - **Every day**
    - A few times a week
    - Rarely

Other comments:

It simply should speed up our processing times. Customers often complain that they aren't receiving their items quick enough, and in some cases decide to leave and go elsewhere. Your software should aim to speed up our process so we can retain and serve more customers.

Usability is key: we don't want to waste time training new employees how to use the software. Because we don't have strict requirements for qualifications, we get a range of people with varying capabilities, and we don't want to spend time explaining intricacies, bugs and common issues: we need something that just works and is self-explanatory.

I think the primary feature should be the path-finding algorithm. It's what will have the most impact on us, since our main problem is timing. Obstacle placement would also be useful as we often do have obstacles like piled-up crates. Anything else is secondary to everything else mentioned - main features should be quick pathfinding, obstacle placement, easy-to-use interface and some sort of feedback.

Minash Khambhaita
Employee @ Currys PLC

1.5.2 Secondary Stakeholder Questionnaire & Responses

Purpose

This questionnaire is designed to gather feedback from my secondary stakeholders (teachers) on the features and usability of the proposed solution. The goal is to ensure the project aligns with client expectations as well as offer excellent usability. As such, this questionnaire focuses more on usability features as well as assessing whether possible features presented to them would be of use and as such incorporated into the solution.

Section 1: General Feedback

1. **Based on the project outline, which feature do you think is most critical for the success of the system?**
    - Stock checker (verifies stock availability before retrieval)
    - **Shortest path algorithm (calculates the fastest route to collect items)**
    - Direct feedback (provides real-time updates on progress)
    - Graphical User Interface (GUI) for ease of use
2. **How important is it for the system to integrate with a database for managing stock levels?**
    - Very important
    - **Moderately important**
    - Not important
3. **Which aspect of the system do you think would benefit the client the most?**
    - **Algorithm efficiency (e.g., finding the shortest path)**
    - Database management (e.g., updating stock levels)
    - User interface design (e.g., ease of use, visual clarity)
    - Error handling (e.g., dealing with invalid inputs or out-of-stock items)

Section 2: Feature Prioritisation

1. **How useful would it be for the system to allow users to input multiple waypoints (locations of items) at once?**
   - **Very useful**
   - Somewhat useful
   - Not useful
2. **Would you prioritise a system that focuses on simplicity or one that includes advanced features (e.g., handling weighted paths, dynamic obstacles)?**
   - **Simplicity**
   - Advanced features
3. **How important is it for the system to provide visual feedback (e.g., a grid showing the warehouse layout and path)?**
   - **Very important**
   - Moderately important
   - Not important
4. **Should the system include a help section or tooltips to guide users through its features?**
   - Yes, very important
   - **Yes, but only moderately important**
   - No, not necessary

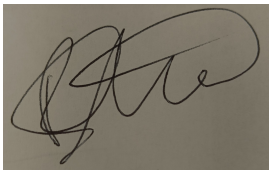
Section 3: Usability and Design

1. **How important is it for the system to follow established usability guidelines (e.g., visibility of system status, error prevention)?**
   - **Very important**
   - Moderately important
   - Not important
2. **How important is it for the system to be scalable (i.e., adaptable to larger warehouses or more complex operations)?**
   - Very important
   - **Moderately important**
   - Not important

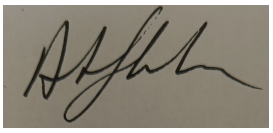
Other comments:

We agree that the shortest path algorithm is critical to the solution, without it the program is useless. It should aim to be efficient enough to be a good replacement for current operations. Integration with a database could be good, but it is not as essential. If it can be done, then excellent. Same with scalability, it is excellent to have but your program should aim to be usable first-and-foremost. Also, you might want to look at a way that people can input their own data quickly, as I guess manually filling out inventory levels is quite time-consuming

Mr Ryan Slater



Mr Alex Johnson



1.5.3 Conclusion

Based on the feedback received, I will focus on simplicity rather than overly complex features, ensuring that StockBot provides greater value through faster collection times rather than more features which are already available and not unique. The time measurements I collected provide a clear baseline against which I can measure StockBot's performance. My goal is to substantially reduce the average time of 7:31 minutes to locate and retrieve an item, which will directly address my primary stakeholder's concern of losing customers due to timing.

Primary Features

1. **Advanced Pathfinding Algorithm**
   My research shows that implementing an efficient pathfinding algorithm must be my highest priority. Both my father (primary stakeholder) and my teachers (secondary stakeholders) emphasised this as the most critical component. The time data collected over 5 days confirms this need, with locating items consistently taking the longest time across all 25 recorded interactions.

   I will implement a pathfinding algorithm that can:

   - Calculate the optimal route between multiple item locations
   - Generate the shortest possible path to minimise collection time
   - Operate efficiently within the warehouse's layout constraints

2. **Intuitive Graphical User Interface**
   The primary stakeholder emphasised that "usability is key" and that the system must be "self-explanatory" for employees with varying technical abilities. The interface will include:

   - Visual representation of the warehouse layout
   - Simple controls for entering item locations
   - Colour-coding for each item

   See section 3.3 for usability features of the GUI.

3. **Multi-Item Collection Capability**
   The data reveals that employees typically retrieve 1-5 items per warehouse trip. Enabling StockBot to handle multiple items in a single operation will significantly reduce the overall time spent. This feature was rated as "very useful" by the secondary stakeholders and will address my father's concern about collection times.

Secondary Features

1. **Database Integration**
   While my father indicated that the current proprietary stock system is accurate "most of the time," integrating StockBot with a database will enhance its functionality. Benefits:

   - Store item locations for quick retrieval
   - Maintain a record of stock levels for verification
   - Support future expansions of the system's capabilities

2. **Progress Tracking**
   Real-time updates on collection progress were rated as "moderately important" by my father. Implementing this feature will provide valuable feedback during the collection process and help address his concern that "customers often complain that they aren't receiving their items quick enough."

3. **Minimal Help System**
   A simple help system was rated as "moderately important" by the secondary stakeholders. This aligns with my father's requirement for a system that "just works." I will include detailed help tooltips within the interface, as well as a brief guide to ensure new users can quickly understand the system without extensive training.

Desirable/Optional Features

1. **JSON Import/Export**
   My secondary stakeholders mentioned that some sort of way to input data other than the interface/manually could be good: hence I settled on a JSON import and export feature, where data is saved to a file and can be inputted/outputted easily.

2. **Output as image**
   My father mentioned that usability is essential: I think it would be useful if the path could be printed as an image so that it can also be manually followed and provides an easy reference.

3. **Path animation**
   As usability is key for this software, a path animation could be a method of easily following the path as it is constructed, and is an abstraction of an actual robot moving in the warehouse.

Based on the research into both current implementations and shortest path algorithms I will conduct, I will make an informed decision about what I believe are the best features and algorithm(s) to incorporate into the final solution from this larger list. Not all of these features listed above will be included, as I plan to refine this into a fixed set of features that would benefit the client the most.

Section 2

Exploring the solution

2.1 Research on current implementations

Companies like Amazon and Ocado have been at the forefront of technological advancement, deploying sophisticated robotic systems to improve efficiency, reduce costs, and enhance safety.

Fig. 2.1 - Amazon's robots [9]

2.1.1 History of warehouse robots

The history of warehouse robotics is relatively recent. General Motors pioneered the field in 1961 by installing the first robotic arm, known as Unimate [10], to automate tasks in manufacturing. For several decades, warehouse robots weren't sophisticated enough to go beyond pre-programmed tasks. Around the 21st century, advances in computing power and sensors enabled robots to perform tasks with greater autonomy and independence. The development of real-time image recognition software and improvements in robotic movement eventually led to the creation of robots capable of navigating warehouse environments.

2.1.2 Amazon Robotics (formerly Kiva Systems)

Amazon's robotics infrastructure is perhaps the most well-known large-scale implementation of automated warehouse management. Following Amazon's acquisition of Kiva Systems in 2012, the technology has been deployed across hundreds of fulfilment centres globally, with over 200,000 robots in operation as of 2020 [11], with an average of 15,000 added each year to warehouses around the world. This figure will no doubt increase as investments in robotics and AI are part of the e-commerce giant's $100bn planned capital expenditure this year. [12]

Centralised Control with Distributed Execution

The system architecture employs a hierarchical control structure where high-level planning occurs centrally while individual robots maintain some autonomy for local decision-making. This hybrid approach enables the system to manage thousands of robots simultaneously while remaining responsive to local conditions. The central system breaks the warehouse floor into a grid of approximately 1 square meter cells, each uniquely identifiable through QR-code markers.

Reservation-Based Traffic Management

Rather than simply calculating shortest paths, the robots use reservation-based path planning, where robots not only find optimal routes but also reserve specific grid cells for specific time intervals, which prevents conflicts before they occur rather than relying solely on reactive collision avoidance. Each robot is aware of where they should be at exact times.

Dynamic Re-slotting and Inventory Positioning

The system continuously analyses product demand patterns and adjusts inventory positions to minimise average travel distances. Frequently ordered items move closer to packing stations, while seasonal or rarely ordered products move to other storage locations further away, meaning overall collection times are reduced.

Multi-Objective Path Optimisation

Amazon's robots consider multiple factors in their routing algorithms, including battery efficiency and charge status, task priority and delivery deadlines, current congestion patterns, predicted future congestion based on planned robot movements and historical traffic density data. This multi-objective optimisation reduces robot travel distance compared to standard shortest-path algorithms, significantly extending battery life and increasing system throughput.

While some of these features are beyond the scope of my project due to complexity, I believe a less advanced multi-objective path optimisation could be implemented into the A* algorithm as part of refinements to the program, dependent on performance and time remaining.
References: [13]

2.1.3 Ocado Smart Platform: The Grid-Based Approach

The Ocado Smart Platform (OSP) takes a fundamentally different approach to warehouse automation compared to Amazon's system, using a dense grid structure where robots move across the top of a storage grid rather than moving the storage units themselves.

The Hive Mind Architecture

Ocado's system uses a proprietary control system called "The Hive Mind", which coordinates hundreds of robots moving on a three-dimensional grid. The system uses a combination of centralised planning for global optimisation and edge computing for rapid local decisions. Each robot communicates with nearby robots multiple times per second to coordinate movements, while the central system performs broader optimisation at a lower frequency.

4D Path Planning

Similar to Amazon but a more sophisticated, robots reserve both physical slots as well as time — specific grid positions at specific moments. This allows robots to schedule crossings through the same physical space by assigning different time slots, meaning they have a greater number of robots at any one point.

Swarm Behaviour Implementation

While the system uses a central system for control, it also implements swarm principles; robots communicate with neighbouring units to form general patterns of movement that distribute across the grid faster. This approach allows the warehouse to gracefully degrade performance when components fail, rather than experiencing catastrophic failures due to a single failure.

While some of these implementations by Ocado are impressive, I do not believe I will be able to implement anything similar from this, other than the model of robots moving in a 2D space despite a 3D warehouse. Since Ocado store the same item vertically, I could apply the same principle to StockBot, reducing the need for 3D considerations, which would make my project much more complex and beyond the scope of my aim.
References: [10], [14], [15], [16]

2.1.4 AutoStore: Cube Storage

AutoStore's approach to automated storage and retrieval is similar to Ocado, using a three-dimensional storage cube accessed by robots operating on the top surface. Ironically, they were in a 3-year legal battle with Ocado over the patents surrounding these robots, before settling to share the patent in a mutually beneficial agreement [17]

Cube Storage Efficiency

The AutoStore system achieves remarkable storage density by eliminating aisles entirely. Bins are stacked directly on top of each other in a dense grid, with robots traversing the top surface and retrieving bins through vertical shafts. This approach achieves significantly higher storage density than traditional automated systems and conventional shelf storage.

Grid-Based Robot Movement

The robots move on an aluminium grid structure along straight paths, but with a unique feature; when two robots need to traverse the same grid space, one robot can physically drive on top of another, then continue its journey when the bottom robot moves to its destination.

Bin Retrieval Algorithms

The system employs sophisticated algorithms to determine which bins to retrieve and in what order, especially when a single order requires items from multiple bins:

1. The system first establishes if multiple items from an order might be in the same bin.
2. It then calculates the optimal sequence to retrieve bins, considering their vertical positions.
3. Bins needed for imminent orders are often pre-positioned near the top of stacks.
4. The algorithm continuously balances retrieval efficiency against grid traffic density.

Decentralised Control with Hierarchical Oversight

Unlike Amazon's more centralised control, AutoStore employs a more distributed approach where individual robots have substantial influence in path selection. A central system provides high-level task allocation, but robots resolve movement conflicts directly with neighbouring units. This architecture provides fall-backs against central system failures.
References: [18], [19], [20], [21]

2.2 Research on pathfinding algorithms

2.2.1 Breadth-First Search (BFS) [1]

Breadth-First Search is an uninformed algorithm which begins at a specified start node and systematically explores all neighbouring nodes at the present depth level before moving to nodes at the next depth level. The boundary of this level is often referred to as the frontier. BFS maintains a queue of nodes to visit, implementing a first-in-first-out (FIFO) system that ensures all nodes at a given depth are processed before any nodes at a greater depth. This guarantees that when a goal node is discovered, the path to it will have traversed the minimum number of edges possible.

Properties

- **Time Complexity:** The time complexity of BFS is $O(b^d)$, where b represents the branching factor (the average number of successors per node) and d represents the depth of the shallowest solution. In the worst case, BFS must explore all nodes up to the depth of the solution.
- **Space Complexity:** Space complexity is also $O(b^d)$, as BFS must store all nodes at the current depth level in its queue.

A queue manages the frontier of nodes to be explored, while a set or hash table tracks visited nodes to prevent cycles and redundant exploration. For each node dequeued, all unvisited neighbours are enqueued and marked as visited. This process continues until either the goal node is found or the queue becomes empty, indicating that no path exists. The best scenarios for BFS are unweighted graphs or where the shortest path is defined by the number of edges traversed. For these scenarios, BFS always finds the most optimal solution, but in other cases it is sub-optimal.

2.2.2 Depth-First Search (DFS) [2]

Depth-First Search, another uninformed algorithm, prioritises depth over breadth in its traversal pattern, a different approach to BFS. This algorithm begins at a designated start node and recursively explores along each branch to its fullest extent before returning to explore alternatives.

Properties

- **Time Complexity:** The time complexity of DFS is $O(b^m)$, where b represents the branching factor and m represents the maximum depth of the search space. In the worst case, DFS must explore all possible paths to the maximum depth before finding a solution or determining that none exists.
- **Space Complexity:** The space complexity of DFS is $O(bm)$, which is generally more favourable than BFS. DFS needs to store only the nodes on the current path from the start node to the current node, plus the siblings of nodes on this path that are waiting to be explored. This more efficient memory usage makes DFS applicable to deeper graphs where BFS would exhaust available memory.

This algorithm is typically implemented using a stack data structure, as it is a LIFO structure. When a branch reaches a dead end or a previously visited node, the algorithm backtracks to the most recent node with unexplored neighbours and continues the exploration process. DFS is excellent at finding all possible paths, but not necessarily the shortest one.

2.2.3 A* Search [3], [4]

A*, an **informed** algorithm, evaluates nodes based on a composite function $f(n) = g(n) + h(n)$, where $g(n)$ represents the cost of the path from the start node to node n, and $h(n)$ is a heuristic function that estimates the cost from node n to the goal. The algorithm maintains a priority queue of nodes ordered by their f-values, always expanding the node with the lowest f-value.

Properties

- **Time Complexity:** The time complexity of A* depends on the quality of the heuristic function. In the worst case, with a poor heuristic, A* degenerates to Dijkstra's algorithm with a time complexity of $O((V + E) \log V)$. With a perfect heuristic, A* can achieve $O(d)$ time complexity, where d is the length of the optimal path.
- **Space Complexity:** The space complexity of A* is $O(b^d)$, where b is the branching factor and d is the depth of the solution. This is because A* must store all generated nodes in memory.

The choice of heuristic significantly impacts the algorithm's performance. Common heuristics for A* include the Manhattan distance for grid-based problems, the Euclidean distance for geometric problems, and the straight-line distance for geographic pathfinding.

2.2.4 Genetic Algorithms [5], [6]

Genetic Algorithms (GAs) are radically different compared to traditional algorithms like Dijkstra. Based on the principles of natural selection and genetic evolution, these algorithms tackle pathfinding as an optimisation problem, evolving solutions over multiple generations rather than systematically exploring the search space. They mimic biological evolution — selection, crossover, mutation — to gradually improve solutions.

Core Concepts

- **Population:** A collection of individual candidate solutions (chromosomes or genomes), each representing a possible path through the environment.
- **Chromosome Representation:** An encoding scheme that represents paths as genomes. Common representations include direct path encoding (a sequence of nodes or waypoints) and direction-based encoding (a series of movement instructions).
- **Fitness Function:** A measure of solution quality that evaluates each candidate path based on criteria such as path length or travel time.
- **Selection:** The process of choosing individuals from the current population to serve as parents for the next generation, with higher-fitness individuals having a greater probability of selection.
- **Crossover (Recombination):** The creation of offspring by combining genetic material from two parent solutions, potentially preserving beneficial characteristics from each.
- **Mutation:** Random alterations to individual solutions that introduce diversity and prevent premature convergence to suboptimal solutions.
- **Elitism:** The practice of preserving the best individuals from each generation to ensure that solution quality never decreases.

2.2.5 Ratings

These ratings are my opinion of how viable the algorithms are in terms of implementation and usage. [22]

Algorithm	Implementation	Usage
BFS	8/10	6.5/10
DFS	8/10	3/10
A*	6/10	9.5/10
Genetic	5/10	7/10

From this table's ratings, I will implement BFS first, then add on A* as time progresses/permits.

2.3 Proposed features for my solution

While implementing a full-scale robotic warehouse automation system may be challenging within the scope and constraints of this project, I have identified several key features that are not only feasible to implement but also of the most use to my client, through the questionnaires(Section 1.4).

2.3.1 Stock checker

Purpose and Functionality

The stock checker component will integrate with the warehouse database to verify product availability before dispatching the robot. This critical feature ensures operational efficiency by preventing wasted journeys to locations with zero inventory.

Implementation Details

- Real-time database queries to verify stock before adding positions to the collection route
- Stock-based filtering that removes out-of-stock items from collection paths
- Automated notification system that alerts warehouse management about stock discrepancies
- Stock threshold monitoring to flag items approaching minimum levels

Benefits

This feature will significantly reduce inefficiencies in the collection process, allowing the robot to focus only on available items. For items identified as out of stock, the system will notify personnel in some manner, which could, in a real-life scenario, enable prompt restocking or perhaps customer communication about alternatives.

2.3.2 Shortest path algorithm

Purpose and Functionality

Instead of relying on a sequential list-based approach commonly used by human workers, I will implement an advanced pathfinding algorithm to determine the most efficient route for collecting multiple items across the warehouse.

Implementation Details

- Implementation of a pathfinding algorithm, most likely A* and/or BFS, for a balance of speed and optimal solution
- Dynamic path recalculation when obstacles are encountered or stock status changes
- Multi-objective optimisation considering both distance and collection priorities

Benefits

This algorithmic approach provides a significant advantage over human collection methods, which typically follow a list-based sequence regardless of physical layout. By calculating optimal routes, the system can reduce travel time, particularly for orders containing multiple items from different warehouse sections.

2.3.3 Obstacle Placement

Purpose and Functionality

To create a realistic simulation environment, I will implement an obstacle placement system that allows operators to block off specific items or areas in the warehouse, simulating real-world scenarios where certain parts of the warehouse may be temporarily inaccessible.

Implementation Details

- Grid-based obstacle representation with occupied/free cell designation
- User interface for operators to mark cells or regions as obstructed
- Path recalculation to automatically route around blocked areas

Benefits

This feature will ensure the robot can operate effectively in a realistic warehouse environment where certain pathways may be blocked or items may be temporarily inaccessible. By allowing operators to place obstacles, the system can be tested under various scenarios, demonstrating how automated systems can adapt to changing warehouse conditions.

2.3.4 Direct Feedback

Purpose and Functionality

I will implement a comprehensive feedback mechanism that provides real-time status updates to the central management system throughout the collection process.

Implementation Details

- Event-driven architecture for immediate status propagation
- Progress tracking with percentage completion and time estimates
- Status reporting at multiple stages (travelling to location, item collection, returning)
- Exception handling with detailed error reporting

Benefits

This feature enables warehouse management to track the status of each order in real-time, facilitating better coordination between automated systems and human workers. The detailed feedback will also provide valuable data for system optimisation, allowing for continuous improvement of pathfinding algorithms and warehouse layout based on actual performance metrics.

2.3.5 Environment persistence

Purpose and functionality

The system will have a basic form of preservation, so that the user can load into a warehouse environment without repeated manual configuration.

Implementation Details

- JSON files for ease-of-use, compatibility & modularity
- Key details are saved in a single JSON file in a user-selected directory

Benefits

This feature allows users to easily load and save their warehouse environments, allowing for multiple layouts to be saved and the environment can easily be transferred to another system.

2.4 Limitations of the solution

Despite the functionality and benefits of the proposed warehouse stock management software, several limitations must be acknowledged. These constraints affect various aspects of the system's performance, functionality, and integration capabilities. These limitations are due to a number of reasons, including complexity, time & my choice of architectures/computational methods to develop my solution.

2.4.1 Limitations from code

Pathfinding Algorithm Constraints

The Breadth-First Search (BFS) algorithm, while comprehensive in finding the shortest path, has a time complexity of $O(V + E)$, where V represents the number of vertices and E represents the number of edges in the graph [23]. This becomes problematic for large warehouses because the algorithm must explore all possible paths at each level before moving to the next depth level. In warehouses exceeding 50x50 grid positions, BFS calculations could take several seconds or more, creating noticeable delays in the software's response time. The A* algorithm offers improvements over BFS through its use of heuristics to guide the search toward promising paths. However, it still faces limitations in very large warehouses. For particularly complex warehouse layouts with numerous obstacles, A* may struggle to maintain performance.

Stock Checking Performance

Database queries for stock verification increase linearly with the number of items requested, creating significant performance overhead. Each stock check operation requires a separate database query, and without advanced optimisation techniques like batch processing or prepared statements, these queries introduce latency. As well as this, the integration of real-time inventory checks during pathfinding adds computational overhead that can increase path calculation time. This performance impact becomes more pronounced as the number of potential collection points increases.

2.4.2 Limitations from software & library choice

SQLite Constraints

SQLite's file-based architecture imposes significant limitations on concurrent access patterns when multiple system instances attempt to read from or write to the database simultaneously. Unlike client-server database systems, SQLite uses file-level locking mechanisms [24] that restrict concurrent write operations, creating potential bottlenecks when a user enters a large number of items. These locks prevent other processes from modifying the database until the current transaction completes.

Performance Bottlenecks

Python's Global Interpreter Lock (GIL) [25] restricts parallelism for CPU-bound operations like complex pathfinding calculations or heavy data processing, which are both aspects of my solution. Combined with GIL, string operations used extensively for database queries, logging, and data formatting create significant memory overhead as strings are immutable (unable to be changed)[26]. This structure overall reduces the efficiency of my solution; the interpreted nature of Python fundamentally limits certain algorithmic optimisations that would be possible in compiled languages like C++ or Rust.

2.4.3 Limitations from time & software choice

Algorithm optimisation

Since I intend to start with Breadth-First Search (due to its speed/reduced time complexity), I may not have time to add the path optimisations I mentioned previously, but rather opt for the addition of different algorithms such as Dijkstra, and offer different options to the user based on their needs.

Performance

As I am using high-level frameworks and languages like Python and Tkinter, performance will decrease as Tkinter runs on a single thread, and does not support multi-core usage [27]. As well as this, more intensive processes like database access will also run on this thread. Hence, the program may be a bit more sluggish. While there are other options like Pygame, I believe their complexity

would cause issues when it came to development, as I am unfamiliar with them and they have a steeper learning curve.

2.5 Software specifications

Below are the tools and software I plan to use to develop my solution.

- IDE: IntelliJ IDEA
- Libraries: tkinter, threading, heapq, logging, sys (see Section 3.1.3 for the reasoning)
- Methodology: Custom - see section 2.7
- Version Control: GitHub
- Code Analysis: Qodana
- Task tracking for iterative development: Linear
- Development OS: Fedora 40
- Testing environment: Windows via Parallels (ARM), macOS (Apple Silicon), Fedora (x64)

2.6 Requirements

2.6.1 Hardware

For the program itself, requirements are quite achievable for most people.

	Minimum Requirements	Recommended Requirements
CPU	1 GHz 64-bit dual-core processor	>1.5 GHz 64-bit quad-core processor
RAM	2GB	4GB
Disk space	10GB	20GB
OS	Windows, macOS, Linux	Windows, macOS, Linux

I have based these requirements on the recommended Python specifications, accounting for the fact I may use a complex AI algorithm (the shortest path algorithm, see Section 2.1.5).

These requirements are by no means restricted. These only apply to the deployment of the software. Hardware requirements may vary.

2.6.2 Software

In terms of software, the user will only require a valid Python install on their computer; I plan to write a script that automatically installs all required/recommended libraries on the device, using the requirements.txt and Python package manager (pip).

2.7 Success criteria

#	Success Criterion	Justification
1	Displays the interface successfully with no errors	This makes sure the user can actually use the program's functions to their max ability.
2	The system will identify and flag all items with stock levels below 2 units at the end of a run.	The threshold of 2 items provides sufficient buffer for restocking before complete depletion.
3	The pathfinding algorithm will calculate the optimal collection route in under 5 seconds for orders containing up to 10 items.	This balances speed requirements with allowing time for database operations, crucial for the efficiency and performance criterion.
4	The system will reduce item collection time by 30-40% compared to the manual method (baseline average: 7:31 minutes).	This is to prove that human-based list collection is inefficient compared to computational approaches.
5	The application will run without crashes and minimal performance degradation for a minimum of 10 consecutive minutes during testing.	This demonstrates system stability & reliability, which are required in any program.
6	The system will successfully detect and navigate around 100% of placed obstacles, while maintaining the high speed element as mentioned in Criterion 3.	Ensures the robot can adapt to changing warehouse conditions, a key capability for real-world implementation.
7	The system will provide real-time position tracking of the robot with updates at least every 5 seconds and positional accuracy within 1 grid cell.	Essential for warehouse management to coordinate operations effectively through visual representation of robot position.
8	The stock verification system will prevent 100% of journey attempts to locations with zero inventory items.	Critical for preventing wasted journeys and improving operational efficiency.
9	All system errors will be logged with detailed information including timestamp, error type, context, and affected component.	Comprehensive error reporting is essential for troubleshooting and system improvement.
10	The graphical user interface will respond to all user interactions within 2 seconds under normal operating conditions.	Responsive UI is critical for quick access and runtime for the program, since the main objective is to speed up operations.

#	**Success Criterion**	**Justification**
11	The system will successfully process and complete 95% of assigned collection tasks without human intervention.	Measures the core functionality of autonomous performance of warehouse tasks, supporting the goal of reducing human operations.
12	The robot will correctly identify and report discrepancies between database stock levels and actual inventory with 98% accuracy.	Essential for maintaining accurate inventory records through automated notification of stock discrepancies.
13	The system will provide collection completion estimates accurate to within ±15% of actual completion time for 90% of tasks.	Allows warehouse management to effectively coordinate operations based on realistic time estimates.
14	The application will successfully connect to and query the SQLite database with an average response time under 500ms.	Efficient database operations are essential for the inventory management and stock verification features.
15	The path calculation algorithm will optimise routes to reduce total travel distance by at least 25% compared to sequential collection.	Measures the effectiveness of the shortest path algorithm, a key differentiator from manual collection methods.
16	The system will ensure all calculated paths use only orthogonal movements (up, down, left, right) and avoid diagonal navigation.	Models the layout of a warehouse, as you cannot cross items to reach others
17	The system will reject invalid inputs, such as non-existent points (e.g., Point 101 in a 10×10 grid) or malformed data (e.g., "abc"), with clear error messages.	Ensures the program is robust and does not fail, accounting for most if not all inputs
18	The system will maintain stable memory usage during continuous operation for up to 10 minutes, without exceeding a 15% increase from baseline.	Measures the footprint and performance of the program; it is focused on speed, but it must also be reasonably memory-efficient.
19	The system will recover gracefully from configuration errors (e.g., corrupted JSON files) by rejecting the invalid configuration and maintaining the previous state.	This is a fail-safe to ensure data is preserved and does not damage the current operating state.
20	The system will return the stock level of any queried item within 1 second, regardless of grid size or database load.	Ensures the database is responsive enough.
21	The database will update stock levels on every run with 100% accuracy for all warehouse items.	Updating constantly is ineffective for a large program. The run-time interval prevents database overload from constant accessing.

This table provides 21 detailed success criteria with specific, measurable targets and comprehensive justifications. The criteria specifications are based on my previous research of current implementations & stakeholder requirements. From the data I gathered, I established these success criteria so that each criterion is actionable and includes clear metrics for evaluation, making it easy to accurately measure whether my solution would be more efficient and meet client expectations.

2.8 Methodology

2.8.1 Outline

As a single developer, I have decided to adopt my own version of the Scrum methodology, to optimise my workflow and encourage more planned decisions. While Scrum is typically associated with teams, I have applied its core principles to my individual workflow.

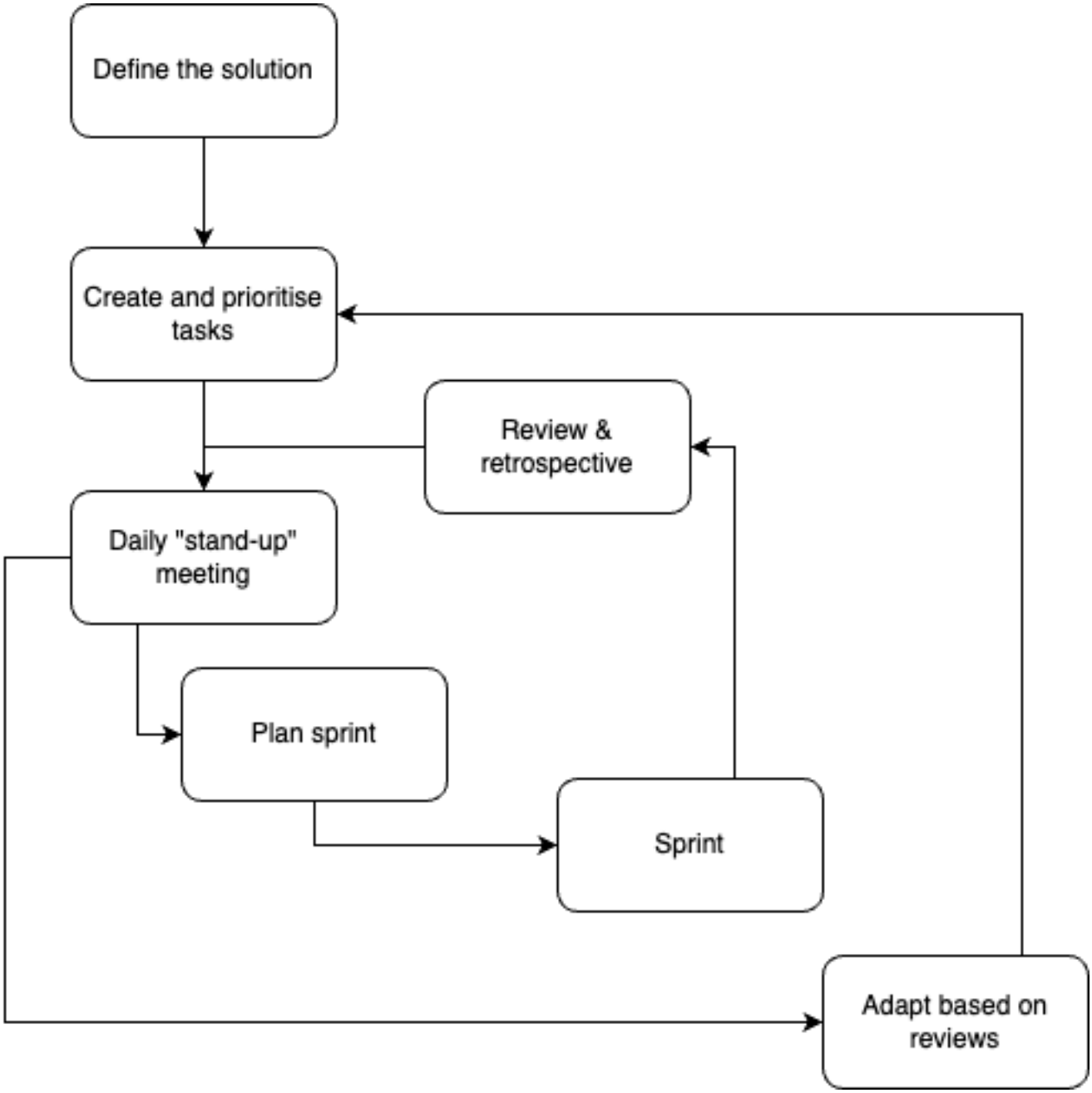


Fig. 2.2 - The stages of my methodology

2.8.2 A breakdown of each stage

- Define the solution: Clearly outline the project's goals/objectives and identify the key features and functionalities the robot should possess. Break down the project into smaller, manageable tasks. This is the main part of my documentation: the analysis & design stage.
- Create and prioritise tasks: prioritise the tasks based on their importance and urgency. Create a backlog of these prioritised tasks.
- Plan sprint: Divide the project into short, iterative development cycles (sprints). As a single developer, sprints will most likely take between a few hours and a few days. Select a subset of tasks from the backlog for each sprint.
- Conduct daily "stand-up" meetings with myself, noting what I accomplished the previous sprint, what I plan to do today, and any obstacles I face. This is essentially a "mini-review".
- Sprint: Focus solely on the tasks I assigned for the sprint; I will use the stand-up notes to ensure the backlog of tasks is cleared or allocated a time to be resolved.
- Review and retrospective: At the end of each sprint, I will review the completed tasks and assess progress, identifying any areas for improvement and adjusting my plan accordingly.
- Adapt and review: Once the backlog of each category of tasks is cleared, implement each section into the central program, regularly testing the code to ensure it meets the requirements. I will use version control systems (VCS) such as GitHub to track changes and document the problems I faced.

2.8.3 Key considerations:

- Time Management: Effectively managing my time to ensure tasks are completed within the sprint time-frame.
- Communication: I plan to use tools like project management software or even a simple to-do list to track progress.
- Documentation: Using VCS, I aim to document any changes I make, as well as the reasons why.

Section 3

Designing the solution

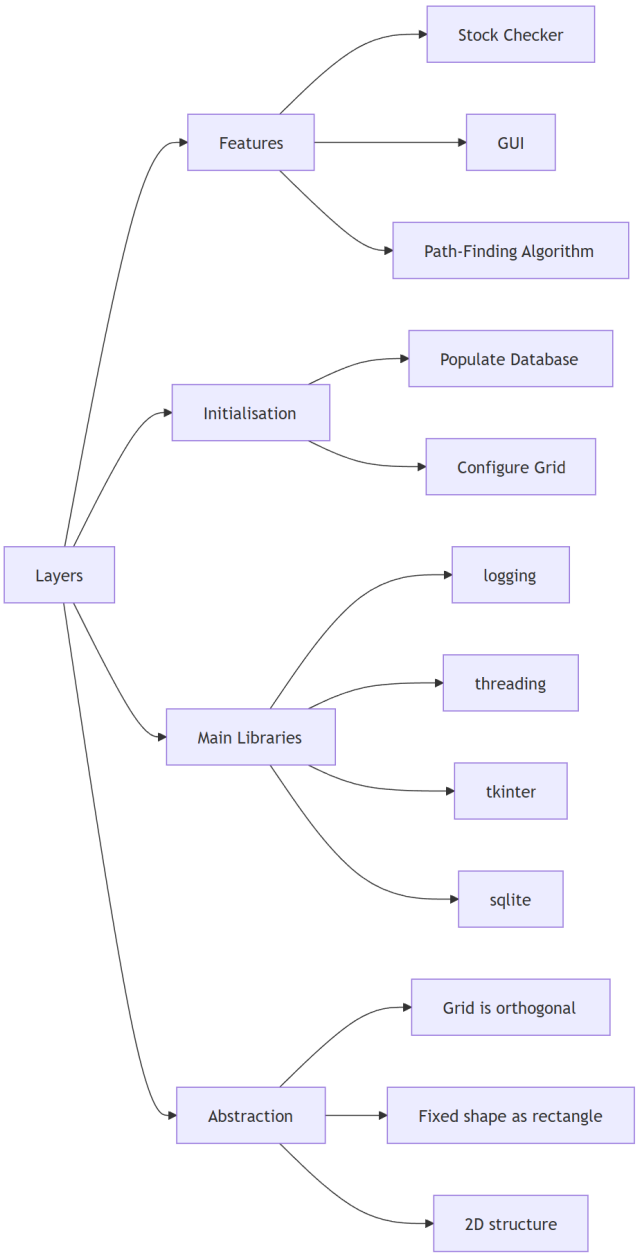


Fig. 3.1 A further breakdown/structure of the pathfinding feature

3.1 An initial breakdown of my solution

Due to the complexity of my solution, I have decomposed my problem down into the fundamentals I believe will benefit me when designing a computational approach to this problem. I have broken down my complex solution into the initial layers of my design outline, and written justifications for each decision. See previous page for the chart outlining the basics.

1. Features - this is the central part of the program - what will my program be able to do?
2. Initialisation and Startup - this is how the program will configure itself - what must happen for my program to begin functioning?
3. Main Libraries - this is how I will achieve the creation of this program - what is required to make my idea a reality?
4. Abstraction - this is how I will break down the problem into more manageable chunks appropriate for a computational approach - how can I simplify the program and remove anything unnecessary?

3.1.1 Features

This layer outlines the central features that the system offers in order for my client to benefit from this solution. This allows me to tackle each feature at a time, building a functional solution a section at a time.

- Stock Checker - check whether an item is available before fetching it.
- Shortest Path Algorithm - find the fastest route to pick up all items
- GUI - make my solution more user-friendly and interactive
- Direct feedback to user - keep the user informed on what is happening, such as what path is being traced, what items are out of stock and so on.

See section x.x.x for a detailed explanation of these features, and see the next few pages for a basic outline of a further breakdown.

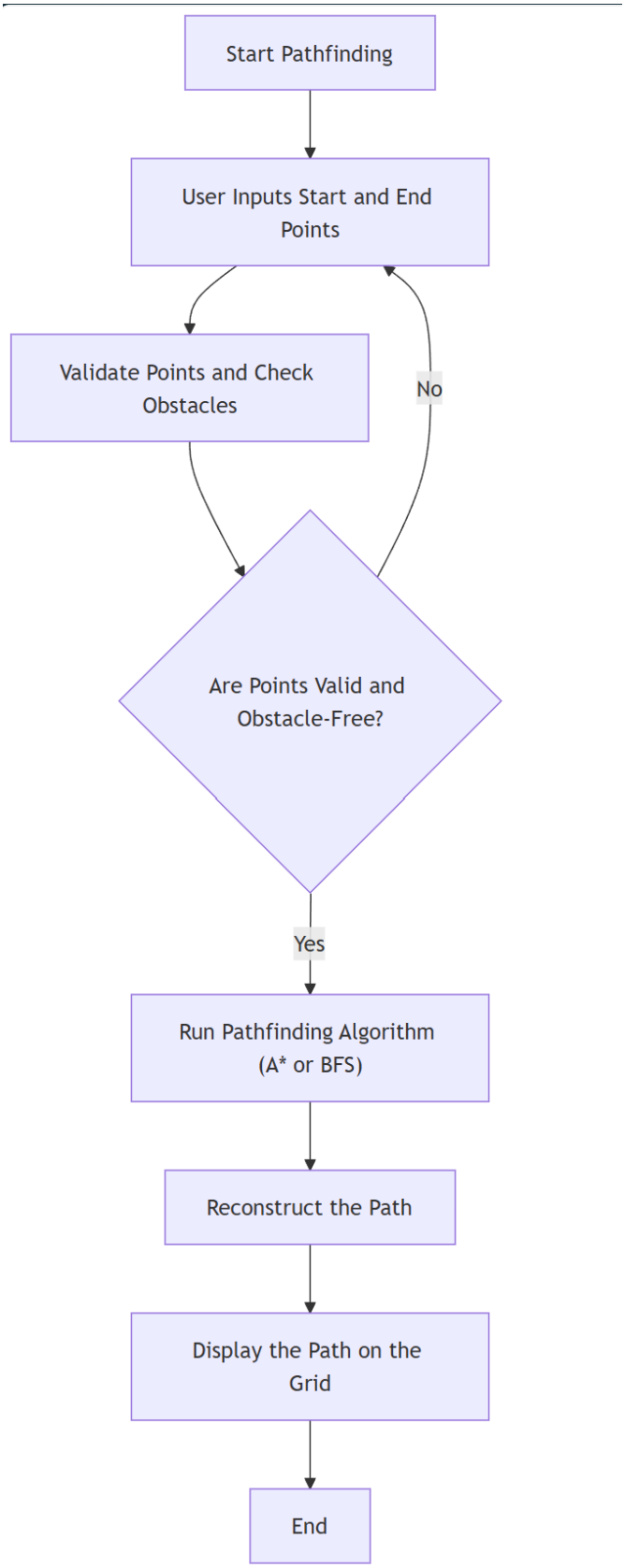
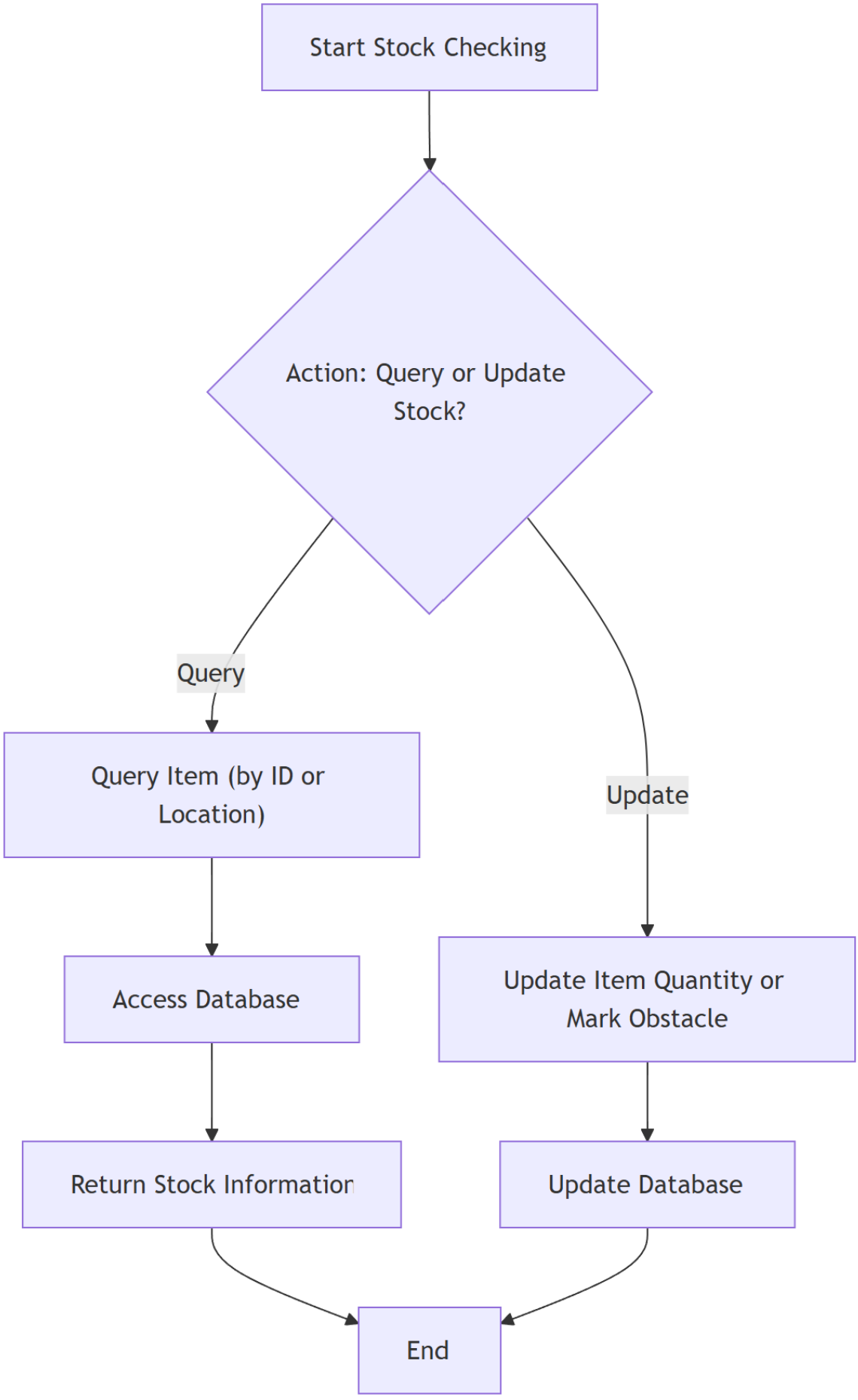


Fig. 3.2 A further breakdown/structure of the pathfinding feature


Start Stock Checking
Action: Query or Update Stock?
Query
Update
Query Item (by ID or Location)
Access Database
Return Stock Information
Update Item Quantity or Mark Obstacle
Update Database
End

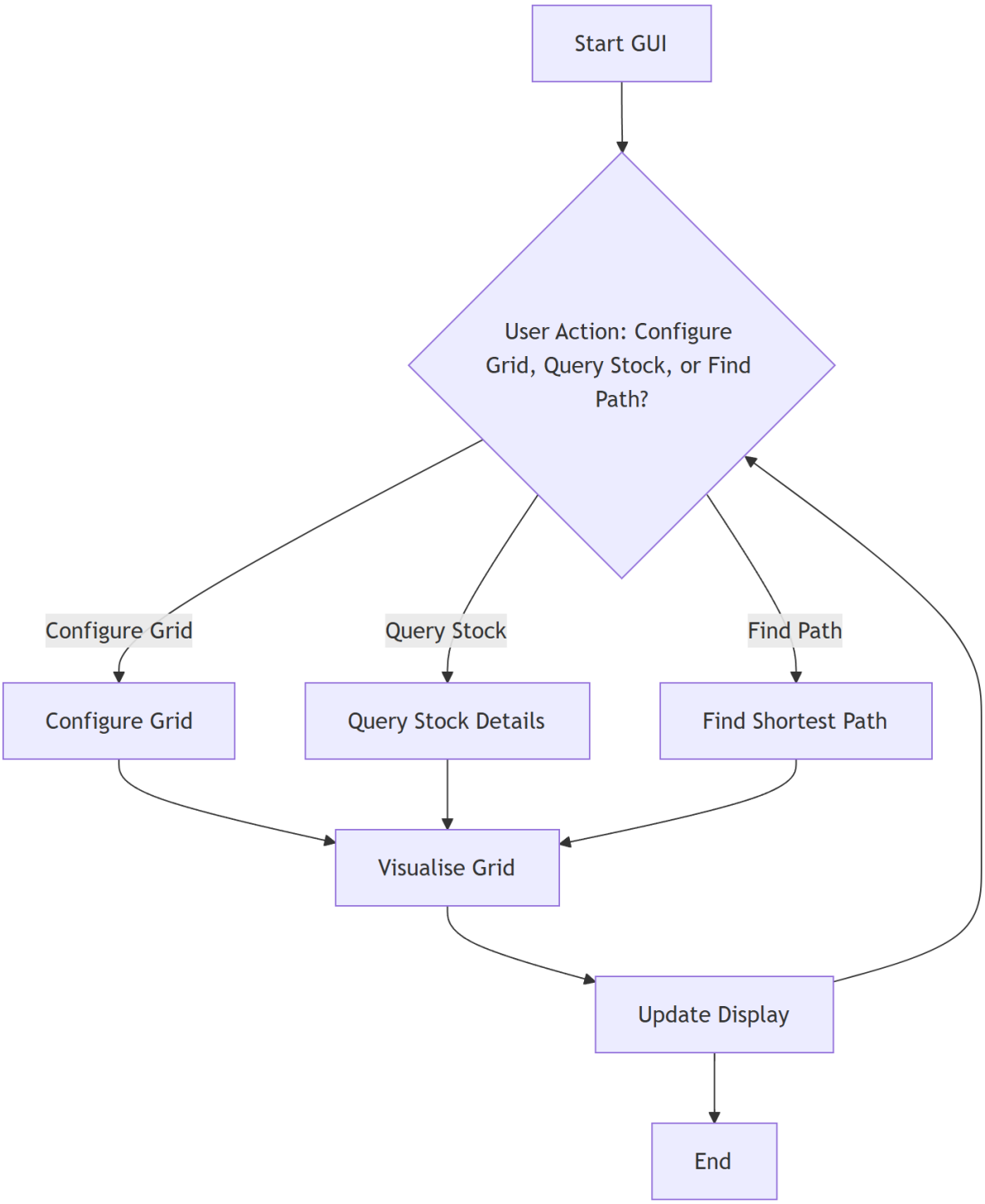


Fig. 3.4 A further breakdown/structure of the GUI

3.1.2 Initialisation and Startup

This layer details the steps that will be involved in setting up and starting the system. This includes tasks like initialising the grid, ensuring the database exists and contains values, checking the shortest path algorithm, and displaying all elements of the GUI correctly.

Setup of environment

One objective is to configure the program initially - setting up the warehouse environment. It should be decomposed into the following:

- Defining the rows and columns: As my program is designed to be used in a typical warehouse environment (one that is comprised of a grid structure), the user should input their warehouse specifications in the form of rows and columns.
- Initialise the database: Based on the number of rows and columns, an ID should be assigned to each square as an item to serve as a hypothetical product. The initial quantity should be randomised, then modified by the user.
- Load the interface: create all elements of the interface ready for user interaction: this should be done quite quickly.

Graphical User Interface

Another objective is to create a functioning and responsive GUI with the following characteristics:

- Responsive - The interface should be responsive to the user's clicks/presses and should update in an appropriate amount of time (e.g. 750ms) to display new content.
- Functional - The interface should allow users to access all features at the press of a button. The GUI will display key elements, including simple buttons to launch any subprogram of the solution such as a query system. It will also be a legend for the meanings of certain aspects of the visualisation of the warehouse, such as red for out-of-stock, green for the path, etc.
- Easy to use - The interface should be very simple and accessible to all users. The layout will be designed with simplicity and intuitiveness in mind, using a logical arrangement of buttons and visual elements. Tooltips and a help toolbar will be integrated to help users understand each feature without requiring detailed training.

See sections x.x.x for a prospective GUI

3.1.3 Main Libraries

This layer lists the underlying software libraries or tools that are used to build the solution; these libraries provide the building blocks for each feature. Some examples of libraries include:

- SQLite: This is a database management system that is used to store and manage data. In this case, I will be using it to store the warehouse layout, item IDs and quantities, as well as a boolean/binary property for whether a coordinate is an obstacle or not.
- Tkinter: This is Python's de-facto standard GUI package. It is a thin object-oriented layer on top of Tcl/Tk. Tkinter will be used to construct the GUI due to its simplicity and compatibility with Python. I will specifically using the 'grid' display sub-manager to create each aspect of the interface - this is due to the greater control it offers over positioning.
- Threading: This library is used to run multiple threads concurrently within a single process. Threads allow for parallel execution of tasks, enabling efficient utilization of system resources and the ability to perform multiple operations simultaneously. In my case, my algorithm is complex, requiring a large amount of processing power; threading allows the algorithm to run efficiently on the device by not restricting all processes like the GUI and database functions to run on a single thread - separating the algorithm will improve performance.
- Logging: This is fairly self-explanatory - the logging library is a built-in module used for generating and managing log messages. It provides a flexible framework for tracking events, debugging, and recording errors in a standardized and structured manner. I plan to use this to instrument my code - this will allow users to see exactly what is happening behind the GUI and allows me to target and debug any issues that may arise.

3.1.4 Abstraction

This layer outlines how I have simplified the solution down to essential components that need to be implemented - this is because some contextual parts of the solution are not necessary to craft a prototype and solution, as is the nature of abstraction. For example, current implementations use complex database systems to continually update and handle more advanced cases - since my implementation is designed to be scalable for smaller warehouses and businesses, some features can be abstracted, such as constant database updates and more complex elements of the database.

3.1.5 Summary/Justifications

Using the computational methods/principles, I felt it was best to split my solution into these layers, then further into what may be required to implement each section of the solution. By breaking it down like this, I can easily focus my efforts on a single part of my solution and then not have to worry in the future. Especially with OOP, this structure means I can easily diagnose and fix any errors that occur, while still leaving my code maintainable and clear. See the next page for the structure of my solution modelled as a top-down systems diagram and a class diagram so interactions between components are clear.

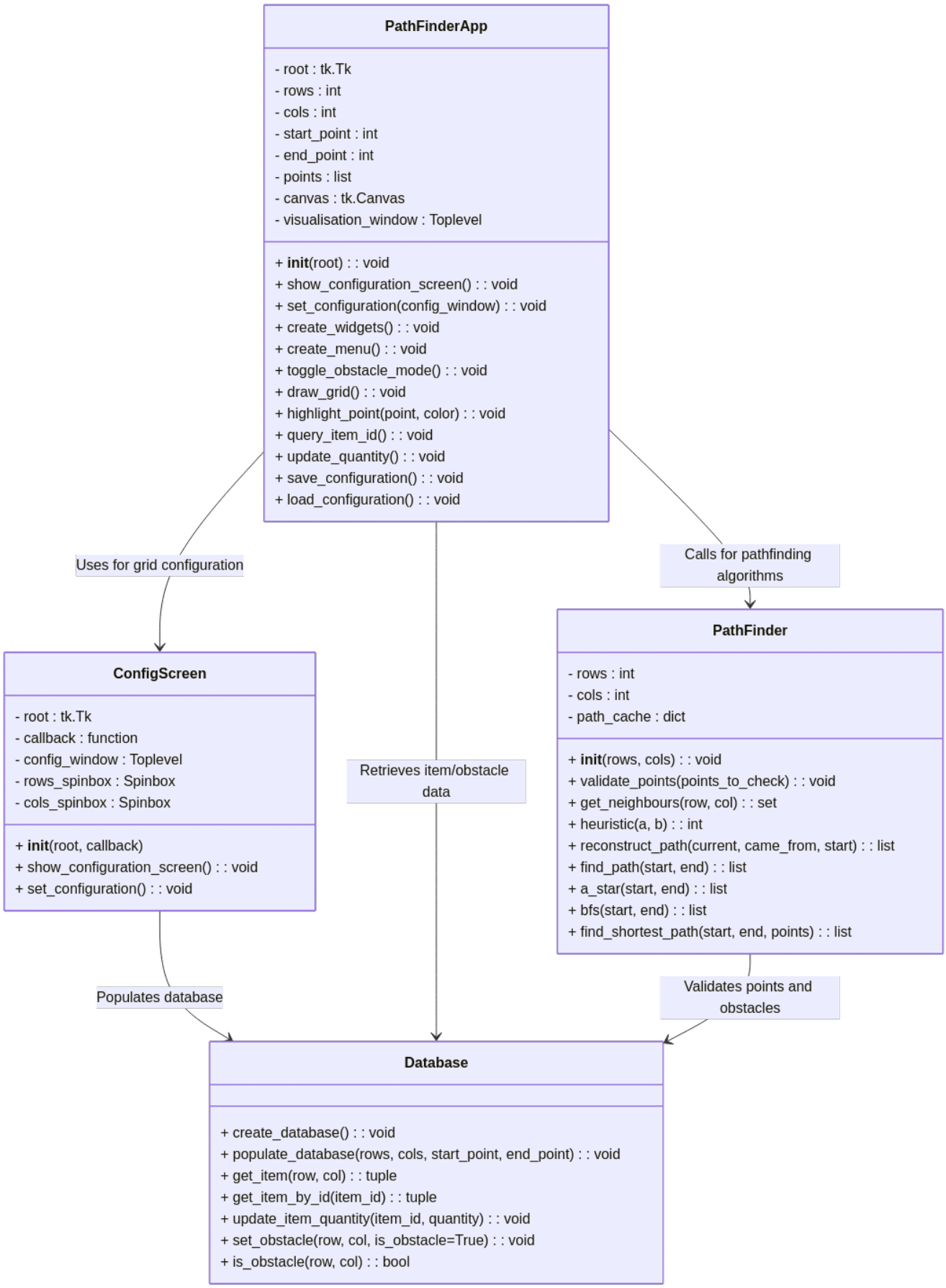


Fig. 3.5 A hypothetical structure of my solution as a class diagram

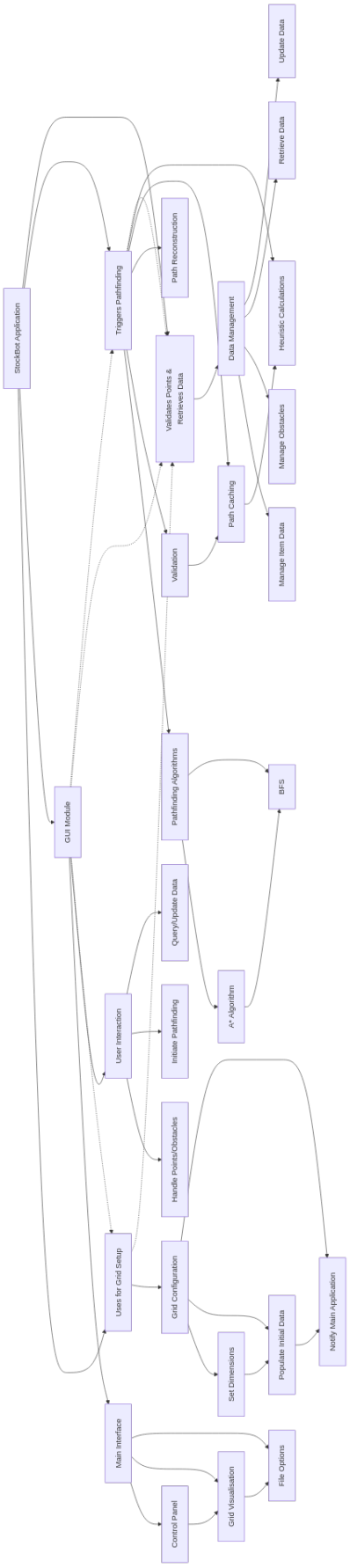


Fig. 3.6 A basic top-down system diagram

3.2 Algorithm prototyping

From the design outline, I will break down each section into the following subsections to begin prototyping my solution using a mixture of pseudocode and flowcharts (in chronological order):

1. Create the primary shortest path algorithm (BFS).
2. Build a rudimentary GUI complementing the SPA.
3. Implement the database operations and the stock checker.
4. Unify all the algorithms into a single prototype.

Once a fundamental program has been established, I will then proceed to do the following:

1. Refine the algorithm to improve performance.
2. Refine the GUI and add a visualisation of the path for ease of use and accessibility.
3. Add help sections and more sophisticated database operations to improve user experience.

3.2.1 Creating the shortest path algorithm - BFS

A good starting point is the central feature of this program: the shortest path algorithm. As my solution depends on this feature to outperform a human counterpart, it seems like a sensible point to start. I plan to implement a breadth-first search for the initial program, as it offers easy implementation, albeit at the cost of an optimum route - I can refine this later on as I am taking an object-oriented approach; I will separate the program into its respective features and place them in different files and classes, then call the required methods in a central file.

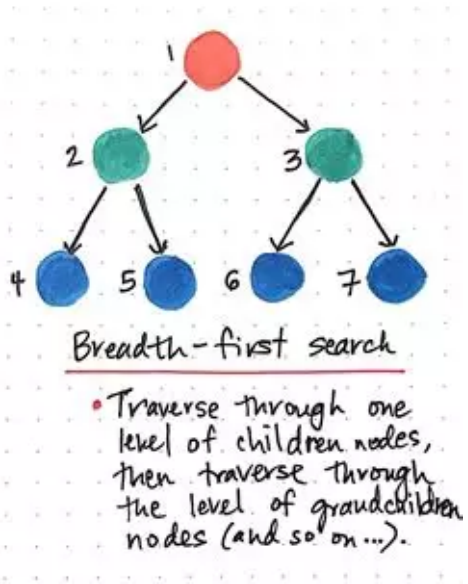
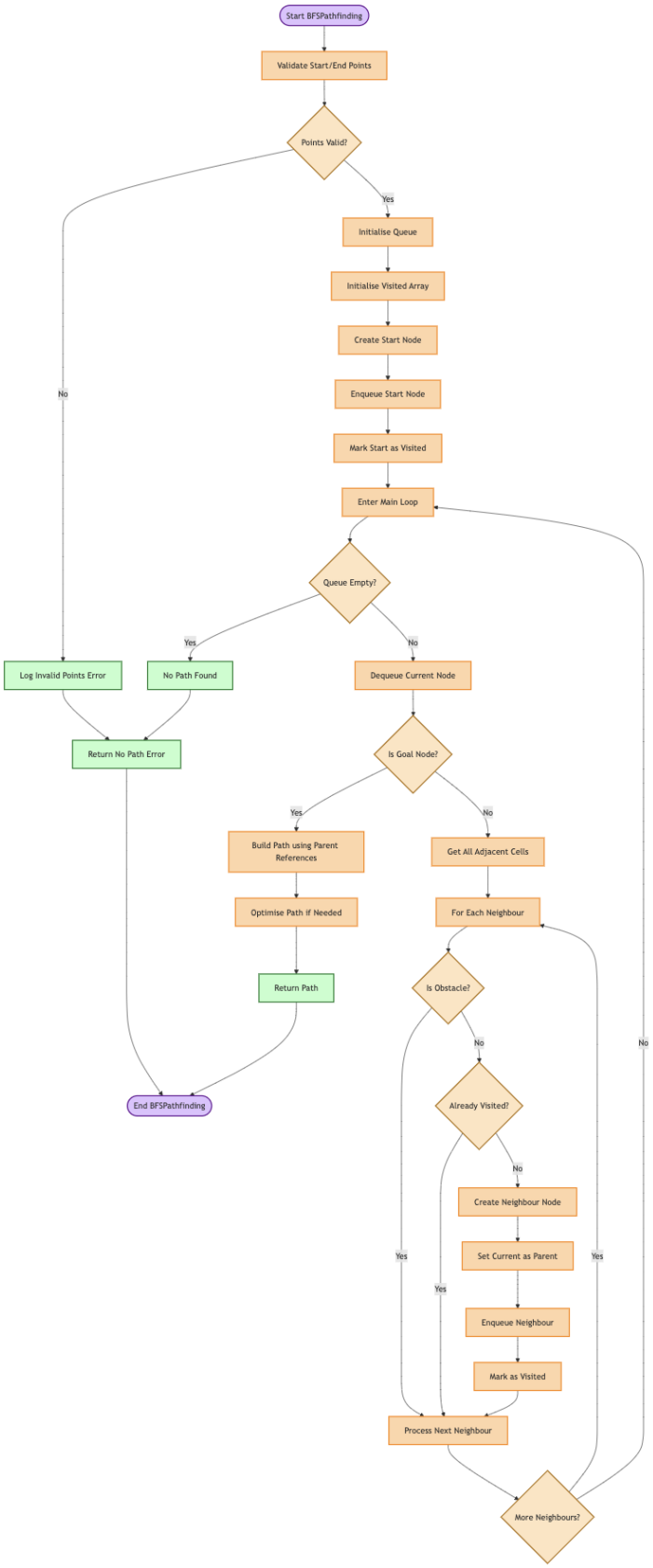


Fig. 3.7 - BFS demonstration [28]

BFS Flowchart



Grid Class

The Grid class serves as a spatial abstraction layer that encapsulates the environment representation. This separation of concerns follows sound object-oriented principles by isolating the terrain data structure from the pathfinding algorithm. This approach enables independent modification of the environment's implementation without affecting the pathfinding logic, supporting the open-closed principle and enhancing maintainability.

- **initialiseGrid**: Creates the initial grid structure with empty cells.
  - *Justification*: Critical foundation method that establishes the environment's structure. Without proper initialisation, the entire pathfinding process would have no valid space to operate in.
  - *Variables*: width: integer, height: integer, grid: 2D array
  - *Type Justification*: Integers provide precise dimensional control with minimal memory footprint. The 2D array structure provides O(1) access time to any cell using coordinates.
- **isWithinBounds**: See Validation Functions section.
- **isObstacle**: See Validation Functions section.
- **getNeighbours**: Returns all traversable adjacent cells.
  - *Justification*: Core traversal method that implements the graph connectivity model. By defining neighbours in the cardinal directions, this method establishes the movement rules for the entire pathfinding system.
  - *Variables*: x: integer, y: integer, neighbours: array
  - *Type Justification*: Integer coordinates allow precise adjacency calculations. The returned array of position pairs provides a clean interface for iteration.
- **setObstacle**: Sets a cell's obstacle status.
  - *Justification*: Provides essential mutability to the environment, allowing for dynamic map changes, responsive obstacles, or initial map setup.
  - *Variables*: x: integer, y: integer, isObstacle: boolean, grid[y][x]: boolean
  - *Type Justification*: Integer coordinates locate the target cell. The boolean parameter explicitly defines the intended state, preventing type coercion issues.

Pathfinder Class

The Pathfinder class encapsulates the complete BFS algorithm implementation, separating the pathfinding logic from the environmental representation. This design choice promotes modularity and follows the single responsibility principle by focusing solely on path discovery. The class structure allows for straightforward extension to other pathfinding algorithms (such as A* or Dijkstra's) without modifying the Grid class or dependent systems.

- **findPath**: Main BFS implementation to discover path.
  - *Justification*: The algorithmic core that translates the abstract BFS concept into executable code. This method orchestrates the entire search process, managing the queue, tracking visited states, and coordinating the exploration of the search space.
  - *Variables*: startX: integer, startY: integer, endX: integer, endY: integer, queue: array, visited: set
  - *Type Justification*: Integer coordinates provide precise position specification. Array implementation for the queue ensures FIFO ordering critical to BFS correctness. Set for visited tracking provides O(1) lookup time.
- **validatePoints**: See Validation Functions section.
- **createNode**: Creates node objects with parent references.
  - *Justification*: Architectural keystone that establishes the parent-child relationships essential for path reconstruction. This method standardises the node structure throughout the algorithm.
  - *Variables*: x: integer, y: integer, parent: object or null, node: object
  - *Type Justification*: Integer coordinates locate the node in space. The recursive object structure creates a linked list that captures the entire path history.
- **buildPath**: Reconstructs the path from goal to start.
  - *Justification*: Translation layer that converts the linked node structure into a usable, ordered path representation. Without this method, the algorithm would find the goal but be unable to report the actual path.
  - *Variables*: endNode: object, path: array
  - *Type Justification*: Node object with its chain of parent references contains the complete solution information, allowing backward path reconstruction.
- **optimisePath**: Potential optimisation of the raw path.
  - *Justification*: Enhancement method that prepares for future expansion. While BFS provides the shortest path in terms of steps, real applications often benefit from smoothing, diagonal shortcuts, or other optimisations.
  - *Variables*: path: array, optimisedPath: array

  - *Type Justification*: Array of coordinate pairs offers sequential access and modification capabilities needed for path transformation algorithms.

- **logError**: Reports pathfinding errors.
  - *Justification*: Diagnostic interface that improves debuggability and user feedback. Instead of silently failing or throwing exceptions, this method provides structured error reporting.
  - *Variables*: message: string, errorType: enum
  - *Type Justification*: String type provides flexible, human-readable error descriptions. The error type enumeration allows for programmatic handling of different error categories.

Validation Functions

Validation functions are critical components that ensure the integrity and correctness of the pathfinding process. They act as gatekeepers that prevent algorithm failure, detect impossible scenarios early, and provide meaningful feedback when errors occur. These functions collectively create a robust system that fails gracefully and provides useful diagnostic information.

- **isWithinBounds** (Grid): Checks if coordinates are within grid boundaries.
  - *Justification*: Fundamental safety mechanism that prevents array index out-of-bounds errors which would crash the application. This method acts as a gatekeeper for all grid operations.
  - *Variables*: x: integer, y: integer, width: integer, height: integer
  - *Type Justification*: Integer coordinates allow for direct comparison operations against dimensions. Using the same integer type for all spatial values ensures consistent boundary checks.

- **isObstacle** (Grid): Determines if a cell contains an obstacle.
  - *Justification*: Essential for path viability assessment. Without obstacle detection, the algorithm would potentially generate invalid paths through impassable terrain.
  - *Variables*: x: integer, y: integer, grid[y][x]: boolean
  - *Type Justification*: Integer coordinates provide direct access to the specific cell. The boolean value in each grid cell provides an unambiguous obstacle status.

- **validatePoints** (Pathfinder): Validates start and end points.
  - *Justification*: Crucial pre-execution safeguard that prevents wasted computation on impossible scenarios. By checking validity upfront, this method fails fast when given invalid inputs.
  - *Variables*: startX: integer, startY: integer, endX: integer, endY: integer
  - *Type Justification*: Integer coordinates enable direct validation against grid dimensions and obstacle status. Using primitive types simplifies validation logic with standard comparison operators.

Example pseudocode of functions

```
// Class representing the grid environment
class Grid {
    // Properties
    private cells[][]       // 2D array representing the grid
    private width           // Width of the grid
    private height          // Height of the grid

    // Constructor
    constructor(width, height) {
        this.width = width
        this.height = height
        this.cells = initialiseGrid(width, height)
    }

    // initialise grid with empty cells
    private initialiseGrid(width, height) {
        // Create a grid of specified dimensions
        // Default cells are traversable (not obstacles)
        return new Array(height).fill().map(() => new Array(width).fill(false))
    }

    // Check if given coordinates are within grid boundaries
    public isWithinBounds(x, y) {
        return x >= 0 && x < this.width && y >= 0 && y < this.height
    }

    // Check if cell at (x,y) is an obstacle
    public isObstacle(x, y) {
        if (!this.isWithinBounds(x, y)) return true
        return this.cells[y][x]
    }

    // Set a cell as an obstacle
    public setObstacle(x, y, isObstacle) {
        if (this.isWithinBounds(x, y)) {
            this.cells[y][x] = isObstacle
        }
    }
```

```
    // Get all neighbours of a given position that are not obstacles
    public getneighbours(x, y) {
        const neighbours = []
        const directions = [
            {dx: 0, dy: -1},  // Up
            {dx: 1, dy: 0},   // Right
            {dx: 0, dy: 1},   // Down
            {dx: -1, dy: 0}   // Left
        ]

        for (const dir of directions) {
            const newX = x + dir.dx
            const newY = y + dir.dy

            if (!this.isObstacle(newX, newY)) {
                neighbours.push({x: newX, y: newY})
            }
        }

        return neighbours
    }
}

// Class implementing the BFS pathfinding algorithm
class Pathfinder {
    // Properties
    private grid            // Reference to the Grid object

    // Constructor
    constructor(grid) {
        this.grid = grid
    }

    // Find path using BFS
    public findPath(startX, startY, endX, endY) {
        // Validate start and end points
        if (!this.validatePoints(startX, startY, endX, endY)) {
            this.logError("Invalid start or end points")
            return null
        }
```

```
        // initialise queue, visited array, and nodes
        const queue = []
        const visited = {}
        const startNode = this.createNode(startX, startY, null)

        // Enqueue start node and mark as visited
        queue.push(startNode)
        visited[`${startX},${startY}`] = true

        // Main BFS loop
        while (queue.length > 0) {
            // Dequeue current node
            const currentNode = queue.shift()

            // Check if goal reached
            if (currentNode.x === endX && currentNode.y === endY) {
                return this.buildPath(currentNode)
            }

            // Get all neighbours
            const neighbours = this.grid.getneighbours(currentNode.x, currentNode.y)

            // Process each neighbour
            for (const neighbour of neighbours) {
                const key = `${neighbour.x},${neighbour.y}`

                // Skip if already visited
                if (visited[key]) continue

                // Create neighbour node and set parent
                const neighbourNode = this.createNode(
                neighbour.x, neighbour.y, currentNode)

                // Enqueue neighbour and mark as visited
                queue.push(neighbourNode)
                visited[key] = true
            }
        }
        // No path found
        this.logError("No path found")
        return null
    }
```

```
    // Validate start and end points
    private validatePoints(startX, startY, endX, endY) {
        // Check if points are within grid boundaries
        if (!this.grid.isWithinBounds(startX, startY) ||
        !this.grid.isWithinBounds(endX, endY)) {
            return false
        }

        // Check if points are not obstacles
        if (this.grid.isObstacle(startX, startY) ||
        this.grid.isObstacle(endX, endY)) {
            return false
        }

        return true
    }

    // Create a new node
    private createNode(x, y, parent) {
        return {
            x: x,
            y: y,
            parent: parent
        }
    }

    // Build path by traversing parent references
    private buildPath(endNode) {
        const path = []
        let currentNode = endNode

        // Traverse from end node to start node
        while (currentNode !== null) {
            path.unshift({x: currentNode.x, y: currentNode.y})
            currentNode = currentNode.parent
        }
    }
    // Log an error message
    private logError(message) {
        console.log("Error: " + message)
    }
}
```

Class Structure

The implementation consists of two classes:

1. **Grid Class**: Manages the environment representation and spatial operations
2. **Pathfinder Class**: Implements the BFS algorithm for finding the shortest path

These are separated as they deal with 2 different areas of the solution, making debugging easier and making the code more readable, as functions are now grouped as methods under classes.

Grid Class

- **Core Properties**:
  - `cells[][]`: A 2D array representing traversable spaces and obstacles
  - `width` and `height`: Define the dimensions of the grid
- **Key Methods**:
  - `isWithinBounds()`: Ensures coordinates remain within grid limits
  - `isObstacle()`: Determines if a cell is blocked and cannot be traversed
  - `getneighbours()`: Returns all adjacent traversable cells in orthogonal directions (Up, down, left, right)

I used a 2D array as it is similar to a grid in most aspects, and is a good abstraction of a warehouse layout, as each element could represent a shelf/area in a warehouse. I passed width & height as attributes as they are required to calculate start & end points.

The `isWithinBounds()` and `isObstacle()` methods are used to validate points so that the program does not fail and is robust.

Pathfinder Class

- **Core Methods**:
  - `findPath()`: The main method that orchestrates the entire BFS process
  - Uses a queue to ensure cells are visited in order of increasing distance from start (FIFO)
- **Supporting Methods**:
  - `validatePoints()`: Performs comprehensive validation of start and end coordinates
  - `createNode()`: Constructs node objects with parent references for path reconstruction
  - `buildPath()`: Reconstructs the complete path by traversing parent references backward

The methods in this class are mostly self-explanatory. The `findPath()` method is used to find a path between 2 points. The `validatePoints()` method ensures the points are valid, using the `grid` methods. The `createNode()` and `buildPath()` methods are used to backtrack and construct the actual path.

BFS Implementation Details

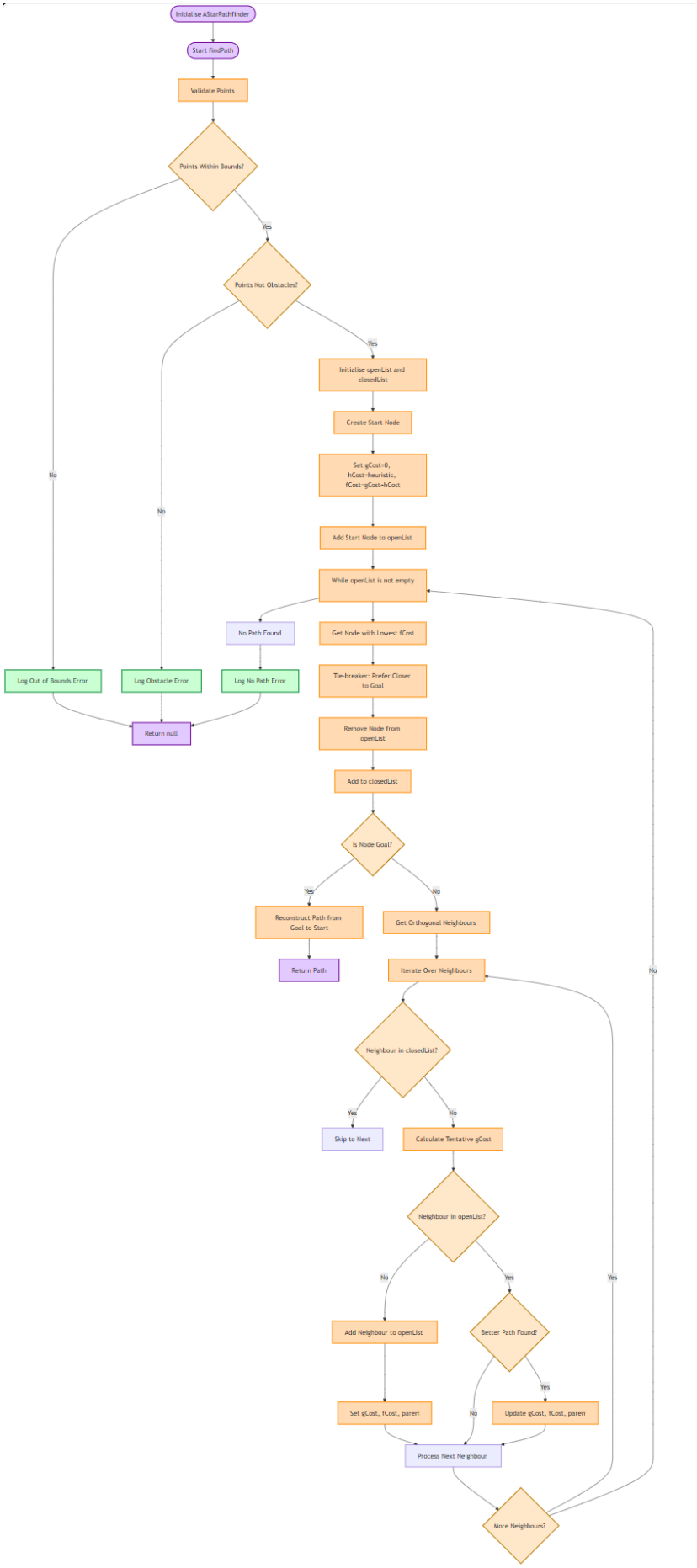
1. **Initialisation Phase**:
    - Validates input points to ensure they're within bounds and not obstacles
    - Sets up the queue with the starting node and marks it as visited
2. **Exploration Phase**:
    - Processes nodes in breadth-first order (nearest nodes first)
    - For each node, checks if it's the goal before exploring neighbours
    - Avoids revisiting cells by maintaining a visited state dictionary
3. **Path Reconstruction Phase**:
    - When the goal is found, traces back through parent references
    - Builds the path from start to end in the correct order

Validation Strategy

- **Input Validation**: Checks that start and end points are valid before beginning the search
- **Boundary Validation**: Prevents accessing cells outside the grid boundaries
- **Obstacle Validation**: Ensures only traversable cells are considered during pathfinding
- **Duplicate Prevention**: Avoids processing the same cell multiple times

This is an example of how the BFS algorithm could be implemented in pseudocode. As my program grows in complexity, I will most likely deviate from this fixed template of classes but still maintain all the features. For example, when adding the A* algorithm, I can extract the validation features and have a single class dedicated to pathfinding operations only, while another class can handle validation.

3.2.2 Creating the shortest path algorithm - A*


Initialise AStarPathfinder
Start findPath
Validate Points
Points Within Bounds?
Yes
No
Points Not Obstacles?
Yes
No
Initialise openList and closedList
Create Start Node
Set gCost=0, hCost=heuristic, fCost=gCost+hCost
Add Start Node to openList
While openList is not empty
No Path Found
Get Node with Lowest fCost
Log Out of Bounds Error
Log Obstacle Error
Log No Path Error
Tie-breaker: Prefer Closer to Goal
Return null
Remove Node from openList
Add to closedList
Is Node Goal?
Yes
No
Reconstruct Path from Goal to Start
Get Orthogonal Neighbours
Return Path
Iterate Over Neighbours
No
Neighbour in closedList?
Yes
No
Skip to Next
Calculate Tentative gCost
Neighbour in openList?
No
Yes
Yes
Add Neighbour to openList
Better Path Found?
Yes
Set gCost, fCost, parent
No
Update gCost, fCost, parent
Process Next Neighbour
More Neighbours?

The A* algorithm is a refinement in my program, a step above BFS in terms of performance. While more complex to implement, it often finds the path faster as it incorporates a heuristic system.

- **findPath**: Main A* implementation that orchestrates the pathfinding process.
  - *Justification*: Serves as the algorithmic controller that coordinates all aspects of the A* search, from initialisation through node exploration to final path construction. The method systematically explores the grid by always selecting the most promising nodes first, ensuring an optimal solution whilst minimising the search space.
  - *Variables*: startX: integer, startY: integer, endX: integer, endY: integer, openList: array, closedList: set
  - *Type Justification*: Integer coordinates provide precise position specification on the grid. The array implementation for openList is straightforward to manipulate, whilst the set data structure for closedList provides efficient lookup for previously explored nodes.
- **createNode**: Constructs node objects with cost values and parent references.
  - *Justification*: Standardises the node structure throughout the algorithm and encapsulates the creation of the node object that must track multiple cost values along with position and parentage. Centralising this construction ensures consistency across all node instances.
  - *Variables*: x: integer, y: integer, parent: object or null, gCost: integer, hCost: integer, fCost: integer
  - *Type Justification*: Integer coordinates identify spatial position on the grid. Integer cost values enable mathematical operations for comparative pathfinding whilst avoiding floating-point precision issues. The parent reference creates a linked structure essential for path reconstruction.
- **calculateHeuristic**: Estimates the distance from a node to the goal.
  - *Justification*: Provides the crucial admissible heuristic function that gives A* its efficiency advantage over algorithms like Dijkstra's. For orthogonal movement, the Manhattan distance perfectly estimates the minimum possible cost to reach the goal without overestimation, guiding exploration towards the target.
  - *Variables*: x: integer, y: integer, endX: integer, endY: integer
  - *Type Justification*: Integer coordinates allow for precise Manhattan distance calculation, which is the sum of horizontal and vertical distances, matching exactly the movement constraints of our orthogonal-only system.

- **getBestNode**: Finds the node with the lowest f-cost in the open list.
  - *Justification*: This method implements the greedy selection aspect of A* by ensuring that the most promising nodes are explored first. This prioritisation is fundamental to the algorithm's efficiency and optimality guarantees.
  - *Variables*: openList: array, bestNode: object
  - *Type Justification*: The array structure allows for iteration through all candidate nodes, whilst the node object provides access to the cost values needed for comparison. The tie-breaking mechanism using h-cost biases exploration toward the goal when f-costs are equal.
- **getOrthogonalNeighbours**: Returns valid adjacent nodes in the four cardinal directions.
  - *Justification*: Implements the graph connectivity model for a grid with orthogonal-only movement. By limiting movement to the four cardinal directions, this method enforces the movement constraints of our simplified pathfinding system.
  - *Variables*: x: integer, y: integer, directions: array, neighbours: array
  - *Type Justification*: Integer coordinates enable precise adjacency calculations. The array of direction vectors provides a clean way to represent and iterate through the four cardinal directions, whilst the resulting neighbours array offers a standard format for the main algorithm to process.
- **buildPath**: Reconstructs the path from goal to start using node parentage.
  - *Justification*: Transforms the linked structure of parent references into a usable, sequential path representation. This backward traversal efficiently constructs the optimal path once the goal is reached, without requiring separate bookkeeping during the search phase.
  - *Variables*: endNode: object, path: array, currentNode: object
  - *Type Justification*: The node object contains the complete solution through its parent chain. The resulting array provides a clean, ordered representation of waypoints that can be directly used by path-following systems.

Validation Functions

These are identical to BFS validation.

Example pseudocode for A*

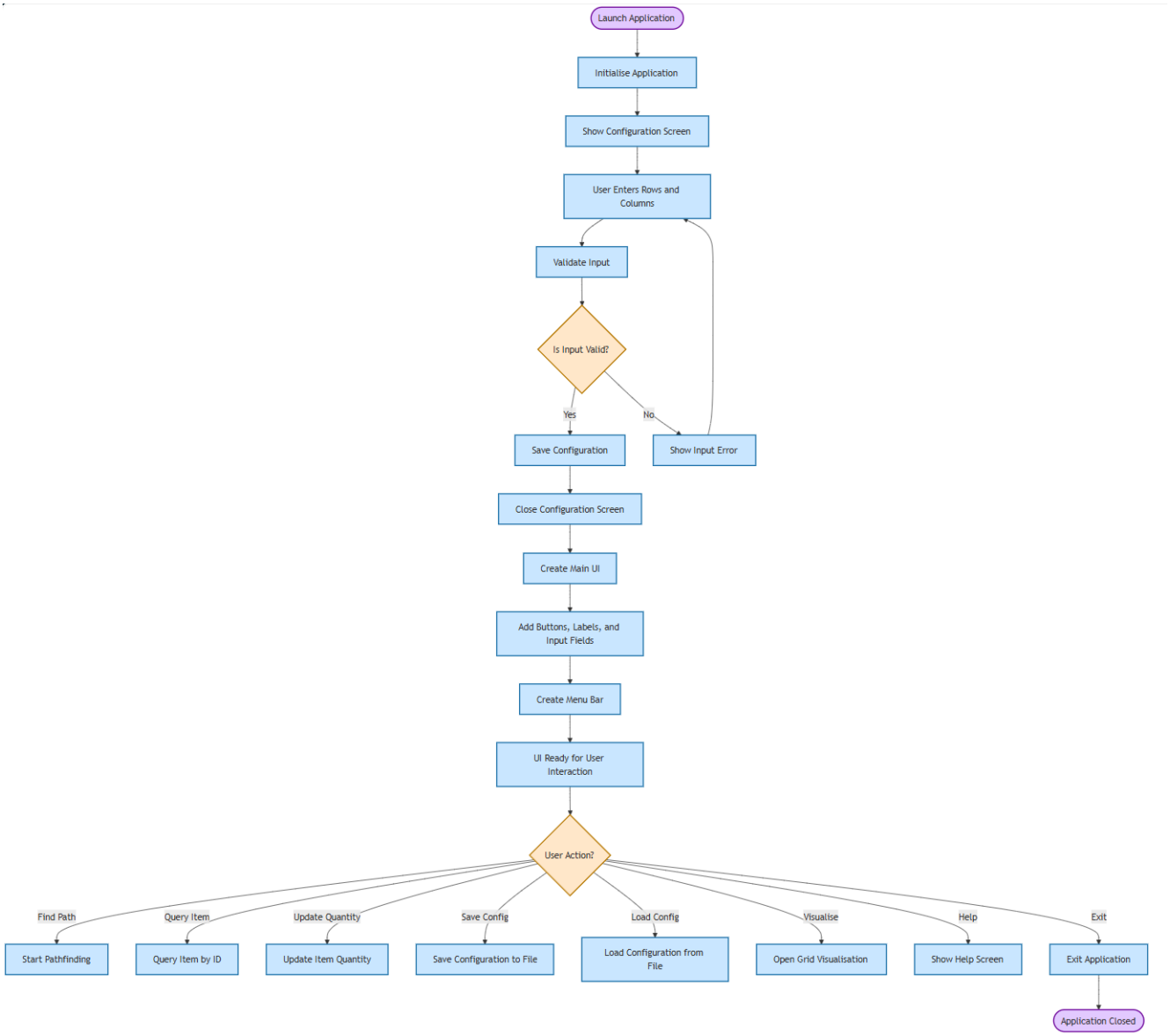
```
class AStarPathfinder:
    # Class variables
    grid: Grid            # Reference to Grid class for terrain information
    openList: Array       # Nodes to be evaluated, manually sorted by fCost
    closedList: Set       # Nodes already evaluated

    # Constructor
    function constructor(grid):
        this.grid = grid

    # Main pathfinding method
    function findPath(startX, startY, endX, endY):
        # Validate inputs
        if not this.validatePoints(startX, startY, endX, endY):
            this.logError("Invalid start or end point")
            return null

        # Initialise data structures
        this.openList = []
        this.closedList = new Set()

        # Create and add start node
        startNode = this.createNode(startX, startY, null)
        startNode.gCost = 0
        startNode.hCost = this.calculateHeuristic(startX, startY, endX, endY)
        startNode.fCost = startNode.gCost + startNode.hCost
        this.openList.push(startNode)

        # Main loop
        while this.openList.length > 0:
            # Get node with lowest fCost
            currentNode = this.getBestNode()

            # Check if goal reached
            if currentNode.x == endX and currentNode.y == endY:
                return this.buildPath(currentNode)

            # Remove current from open list and add to closed list
            this.openList.splice(this.openList.indexOf(currentNode), 1)
            this.closedList.add(currentNode.x + "," + currentNode.y)
```

```
        # Process all neighbours (only orthogonal: up, right, down, left)
        neighbours = this.getOrthogonalNeighbours(currentNode.x, currentNode.y)

        for each neighbour in neighbours:
            neighbourKey = neighbour.x + "," + neighbour.y

            # Skip nodes in closed list
            if this.closedList.has(neighbourKey):
                continue

            # Each step costs 1 unit in this simplified version
            tentativeGCost = currentNode.gCost + 1

            # Check if this node is already in the open list
            existingNode = null
            for each node in this.openList:
                if node.x == neighbour.x and node.y == neighbour.y:
                    existingNode = node
                    break

            if not existingNode:
                # New node, add to open list
                neighbourNode = this.createNode(
                neighbour.x, neighbour.y, currentNode
                )
                neighbourNode.gCost = tentativeGCost
                neighbourNode.hCost = this.calculateHeuristic(
                neighbour.x, neighbour.y, endX, endY
                )
                neighbourNode.fCost = neighbourNode.gCost + neighbourNode.hCost
                this.openList.push(neighbourNode)
            elif tentativeGCost < existingNode.gCost:
                # Better path found, update existing node
                existingNode.gCost = tentativeGCost
                existingNode.fCost = tentativeGCost + existingNode.hCost
                existingNode.parent = currentNode
            }
    }

    # No path found
    this.logError("No path exists between start and end points")
```

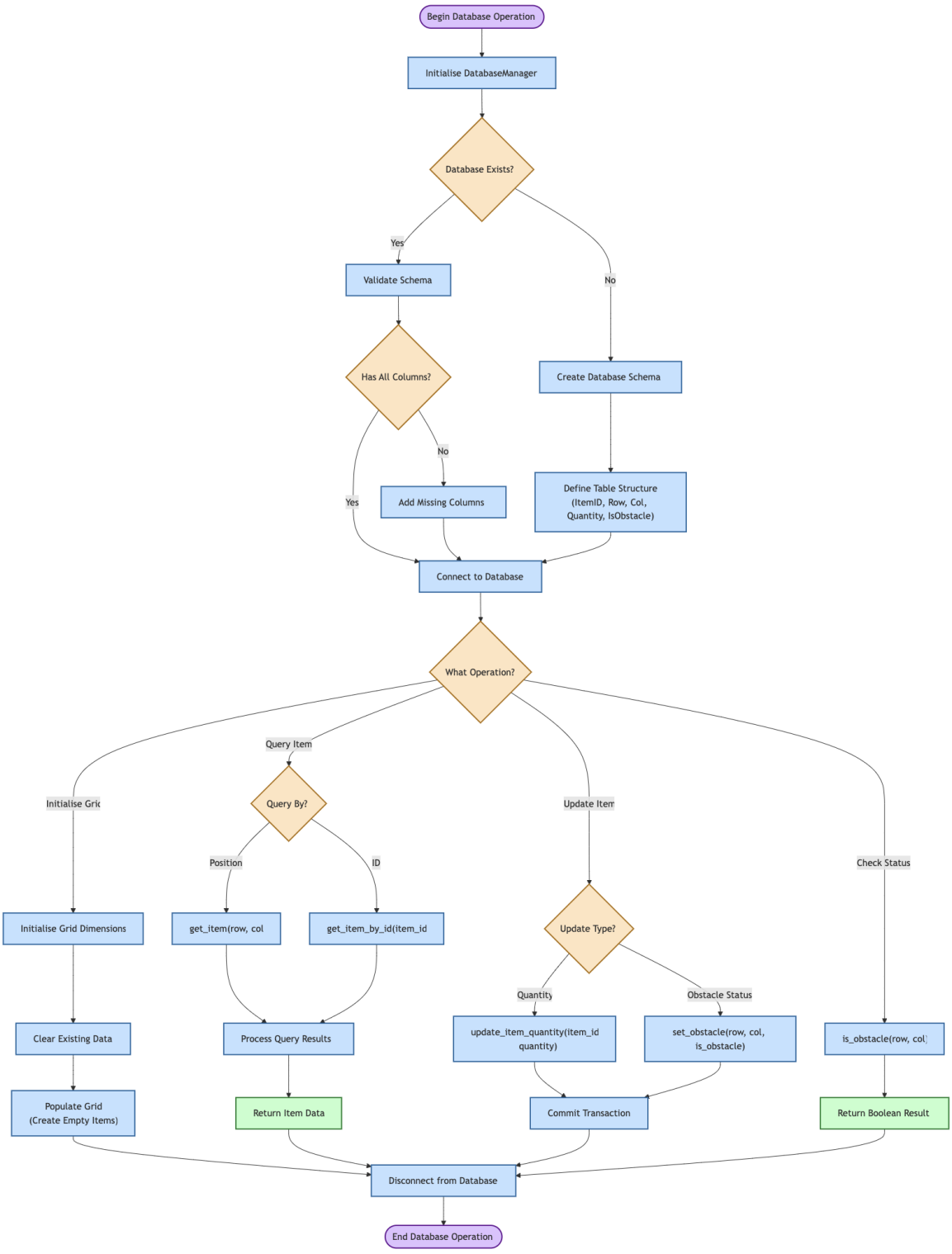
```
    return null

# Create a node object with cost values
function createNode(x, y, parent):
    return {
        x: x,
        y: y,
        gCost: 0,     # Cost from start to this node
        hCost: 0,     # Manhattan distance from this node to goal
        fCost: 0,     # Total cost (g + h)
        parent: parent # Reference to parent node for path reconstruction
    }

# Calculate Manhattan distance from node to goal
function calculateHeuristic(x, y, endX, endY):
    # Manhattan distance (absolute difference in x plus absolute difference in y)
    return abs(x - endX) + abs(y - endY)

# Find node with lowest fCost in the open list
function getBestNode():
    bestNode = this.openList[0]

    for each node in this.openList:
        if node.fCost < bestNode.fCost:
            bestNode = node
        # Tie-breaker: prefer nodes closer to the goal
        elif node.fCost == bestNode.fCost and node.hCost < bestNode.hCost:
            bestNode = node

    return bestNode

# Get orthogonal neighbouring positions (up, right, down, left)
function getOrthogonalNeighbours(x, y):
    neighbours = []

    # The four orthogonal directions
    directions = [
        {x: 0, y: -1},  # Up
        {x: 1, y: 0},   # Right
        {x: 0, y: 1},   # Down
        {x: -1, y: 0}   # Left
    ]
```

```
    for each dir in directions:
        newX = x + dir.x
        newY = y + dir.y

        # Check if position is valid (within bounds and not an obstacle)
        if this.grid.isWithinBounds(newX, newY) and not this.grid.isObstacle(newX, n
            neighbours.push({x: newX, y: newY})

    return neighbours

# Reconstruct path from goal node to start node
function buildPath(endNode):
    path = []
    currentNode = endNode

    # Traverse parent chain from end to start
    while currentNode:
        path.unshift([currentNode.x, currentNode.y])  # Add to front
        currentNode = currentNode.parent

    return path

# Validate start and end points
function validatePoints(startX, startY, endX, endY):
    # Check if points are within grid bounds
    if not this.grid.isWithinBounds(startX, startY) or not
    this.grid.isWithinBounds(endX, endY):

        return false

    # Check if points are not obstacles
    if this.grid.isObstacle(startX, startY) or this.grid.isObstacle(endX, endY):
        return false

    return true

# Log errors for diagnostics
function logError(message):
    console.log("AStarPathfinder Error: " + message)
```

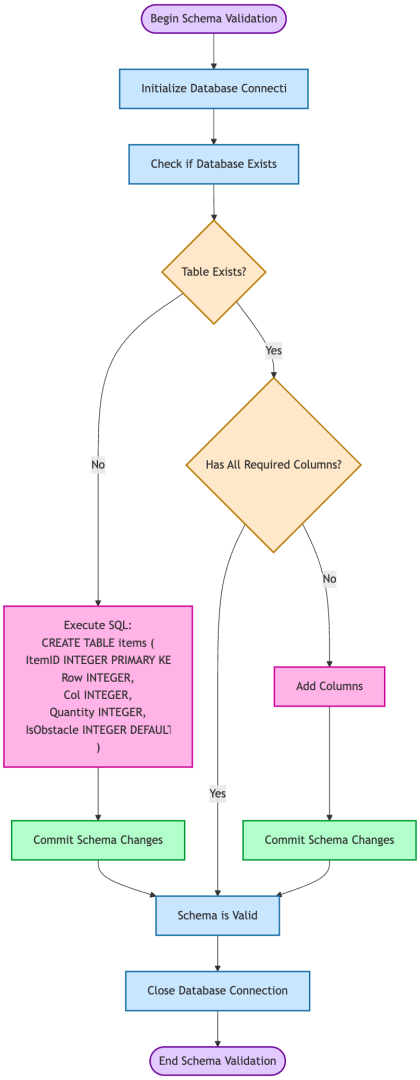
3.2.3 GUI Logic



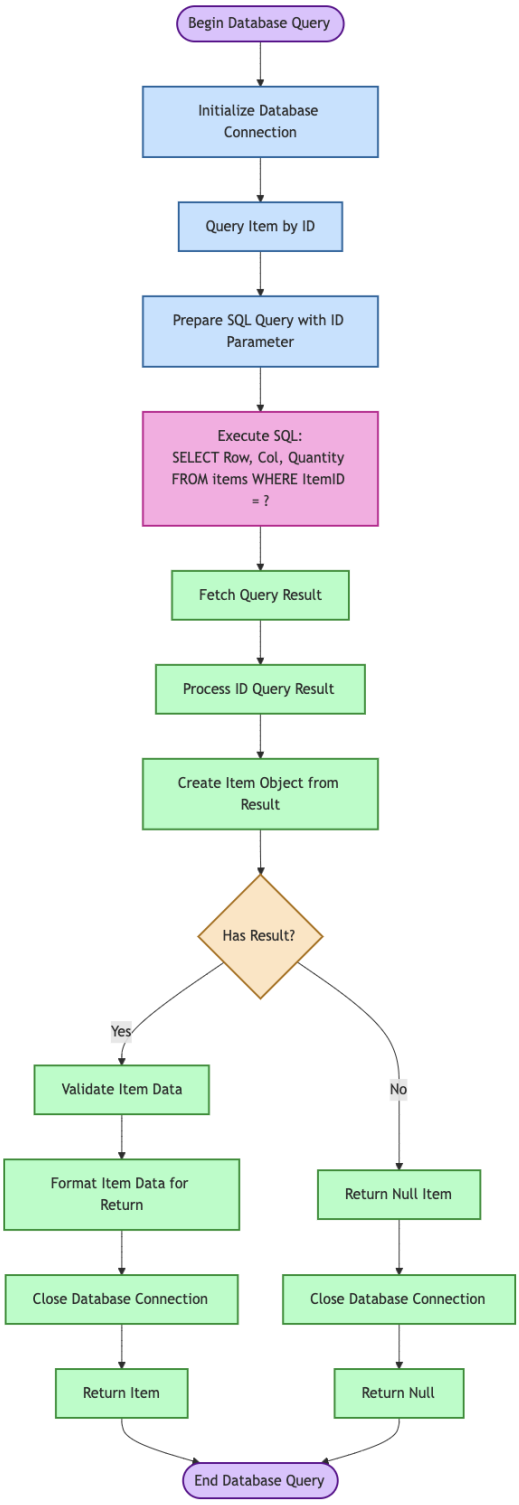
3.2.4 Database operations



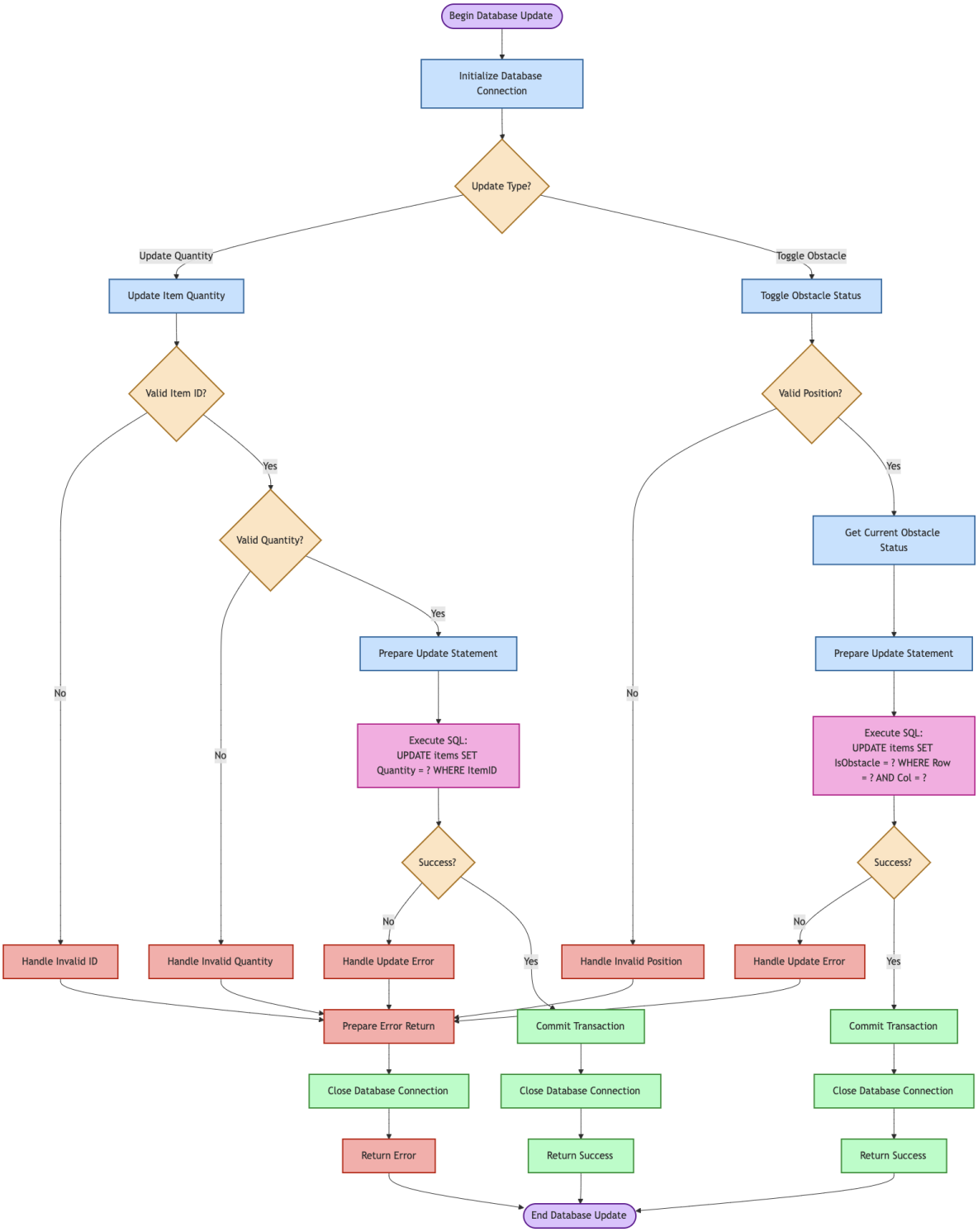
Database validation



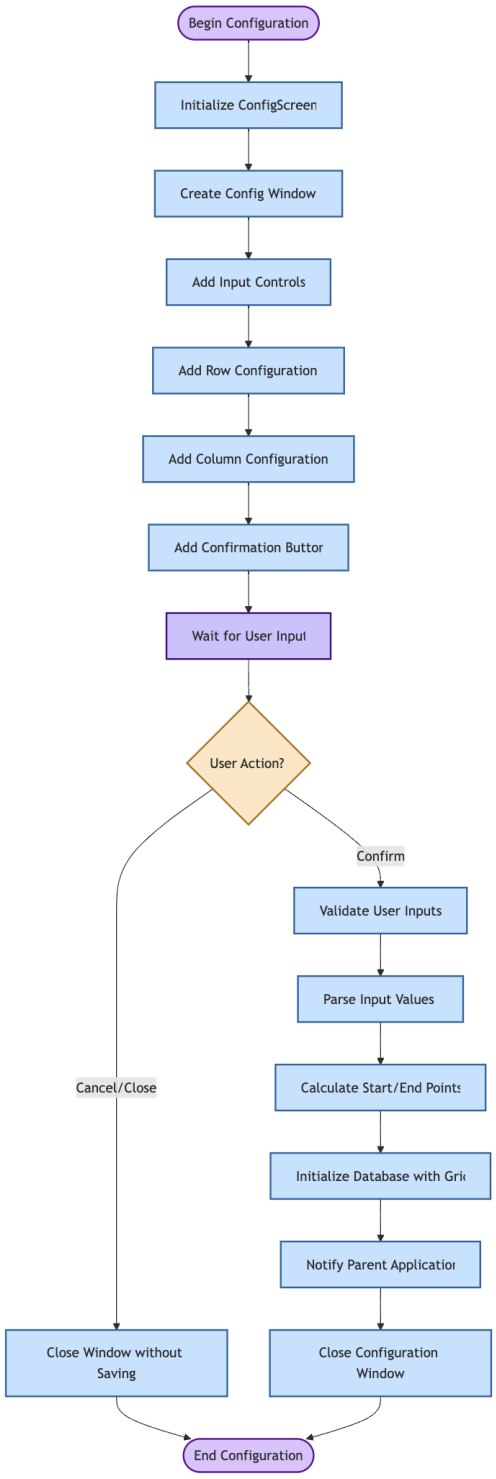
Database query



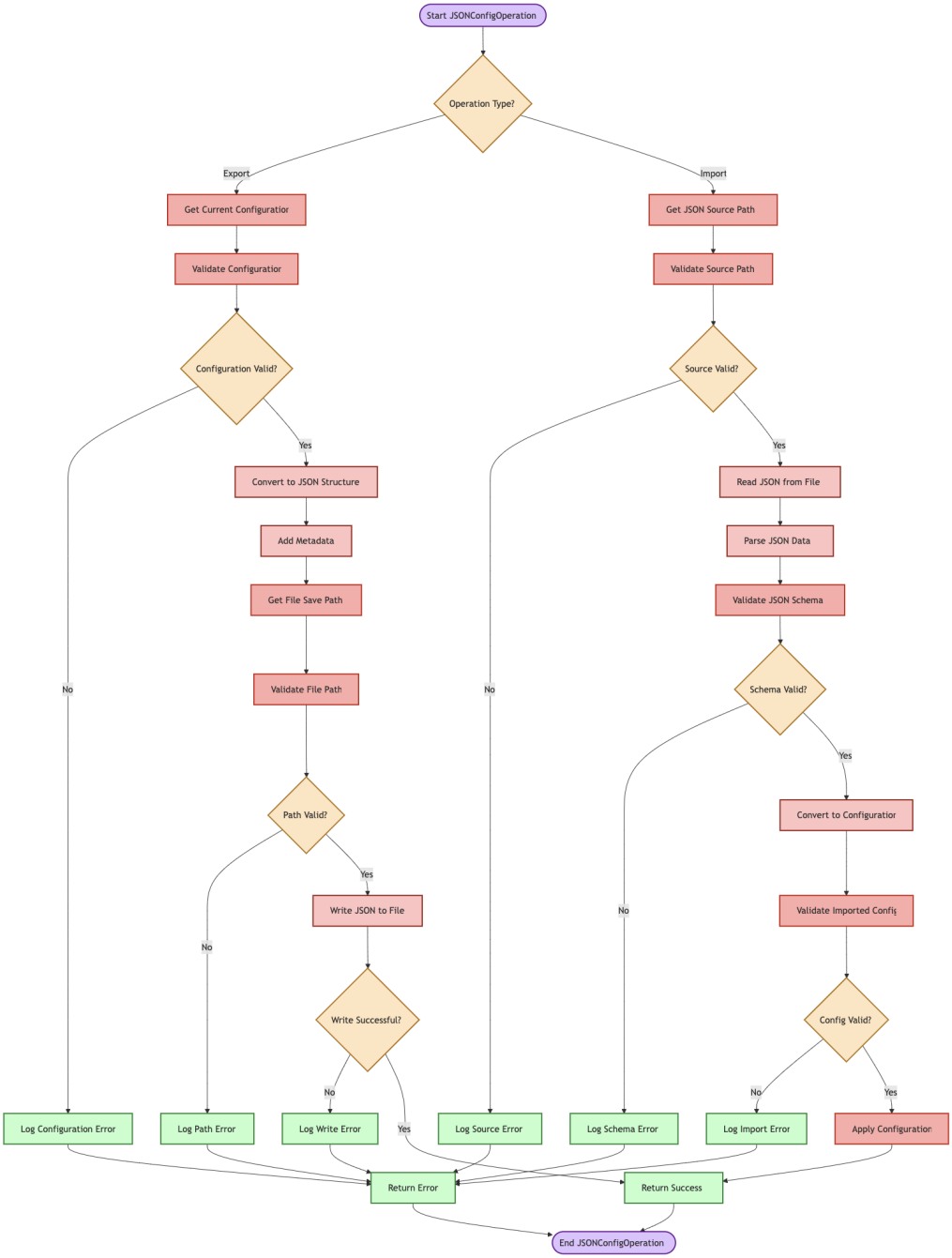
Database update



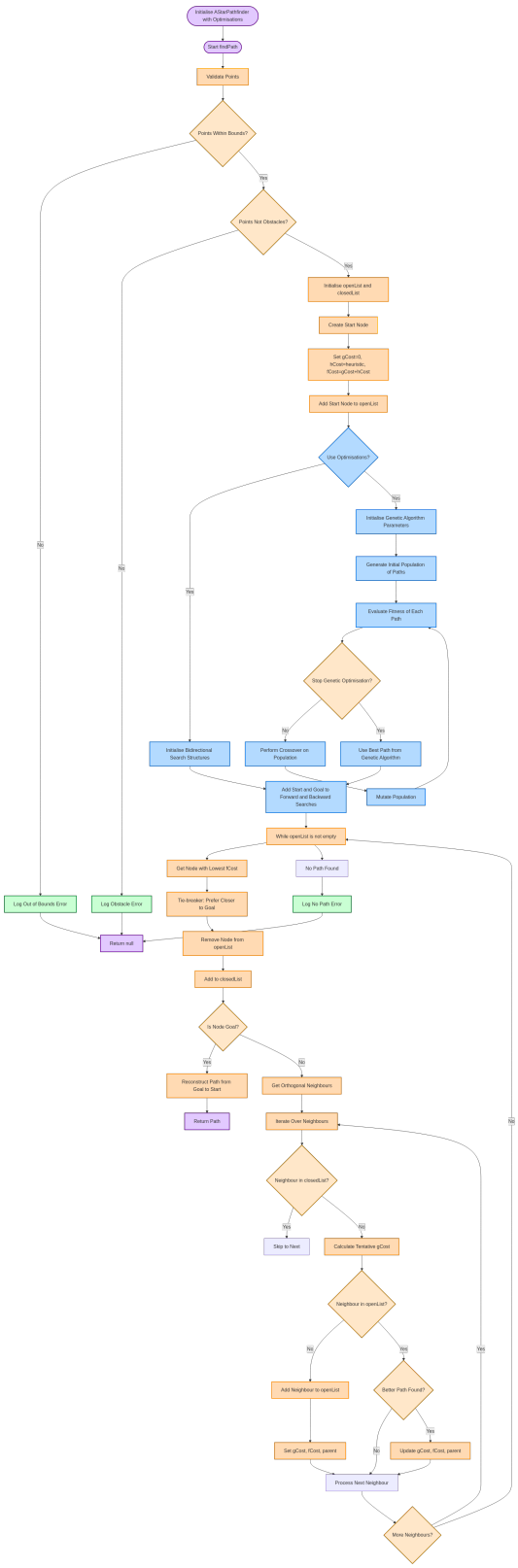
3.2.5 Configuration



3.2.6 JSON import/export



3.2.7 A* refinements if possible



3.3 Usability

On the subject of GUIs, what rules should a GUI actually follow to be optimal for the end user? The guidelines below are a subset of the ones that form the guides of the Google design team, and hence appropriate due to Google's success and accessibility.

3.3.1 Guidelines

- **Visibility of system status:** Users should always be informed of system operations with easy to understand and highly visible status displayed on the screen within a reasonable amount of time.
- **Consistency and standards:** Interface designers should ensure that both the graphic elements and terminology are maintained across similar platforms. For example, an icon that represents one category or concept should not represent a different concept when used on a different screen.
- **Error prevention:** Whenever possible, design systems so that potential errors are kept to a minimum. Users do not like being called upon to detect and remedy problems, which may on occasion be beyond their level of expertise. Eliminating or flagging actions that may result in errors are two possible means of achieving error prevention.
- **Recognition rather than recall:** Minimize cognitive load by maintaining task-relevant information within the display while users explore the interface. Human attention is limited and we are only capable of maintaining around five items in our short-term memory at one time. Due to the limitations of short-term memory, designers should ensure users can simply employ recognition instead of recalling information across parts of the dialogue. Recognizing something is always easier than recall because recognition involves perceiving cues that help us reach into our vast memory and allowing relevant information to surface. For example, we often find the format of multiple choice questions easier than short answer questions on a test because it only requires us to recognize the answer rather than recall it from our memory.
- **Aesthetic and minimalist design:** Keep clutter to a minimum. All unnecessary information competes for the user's limited attentional resources, which could inhibit user's memory retrieval of relevant information. Therefore, the display must be reduced to only the necessary components for the current tasks, whilst providing clearly visible and unambiguous means of navigating to other content.

3.3.2 Configuration Screen Analysis

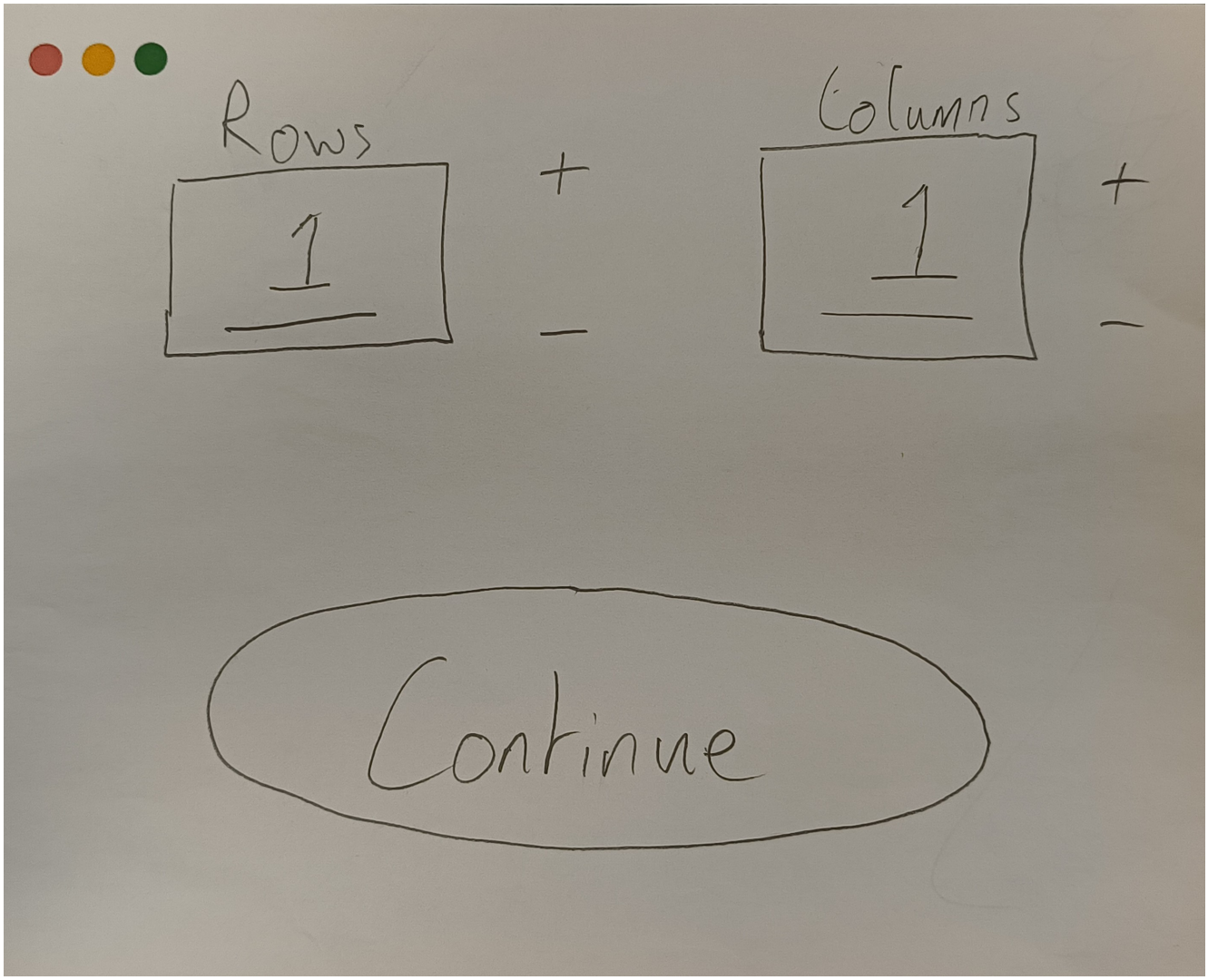


Fig. 3.8 - A hypothetical configuration window prior to running the main program.

Guideline alignment:

- Visibility of System Status
  - Clear numeric displays for rows/columns
  - Plus/minus controls provide immediate feedback
  - "Continue" button shows clear next step
- Error Prevention
  - Bounded input fields prevent invalid grid sizes
  - Plus/minus buttons prevent text input errors
  - Single "Continue" path reduces navigation errors
- Recognition vs Recall
  - Labeled "Rows" and "Columns" fields
  - Simple numeric input reduces cognitive load
  - Clear action button labeled "Continue"
- Aesthetic/Minimalist
  - Only essential grid setup controls
  - Clean layout without distracting elements
  - Logical grouping of related controls
- Consistency
  - Uniform input field styling
  - Consistent plus/minus button placement
  - Standard window controls

Usability Analysis:

The numeric input fields with increment/decrement controls represent an optimal solution for grid dimension entry, as there are many variants of warehouse that can be constructed via row and column definition - this allows for faster input of the desired warehouse dimensions.
The interface's spatial organization leverages natural reading patterns and mental models by arranging elements in a logical left-to-right, top-to-bottom flow that correlates with the final grid structure. This spatial relationship subconsciously enhances user understanding of the configuration process, as this format has been typical of most cultures across the world for millenia.
The centrally positioned continue button serves as a clear visual endpoint, while the minimalist two-field design effectively mitigates a cluttered and hard-to-navigate interface that would otherwise arise if this and the main program was incorporated as one window.

3.3.3 Main Screen Analysis

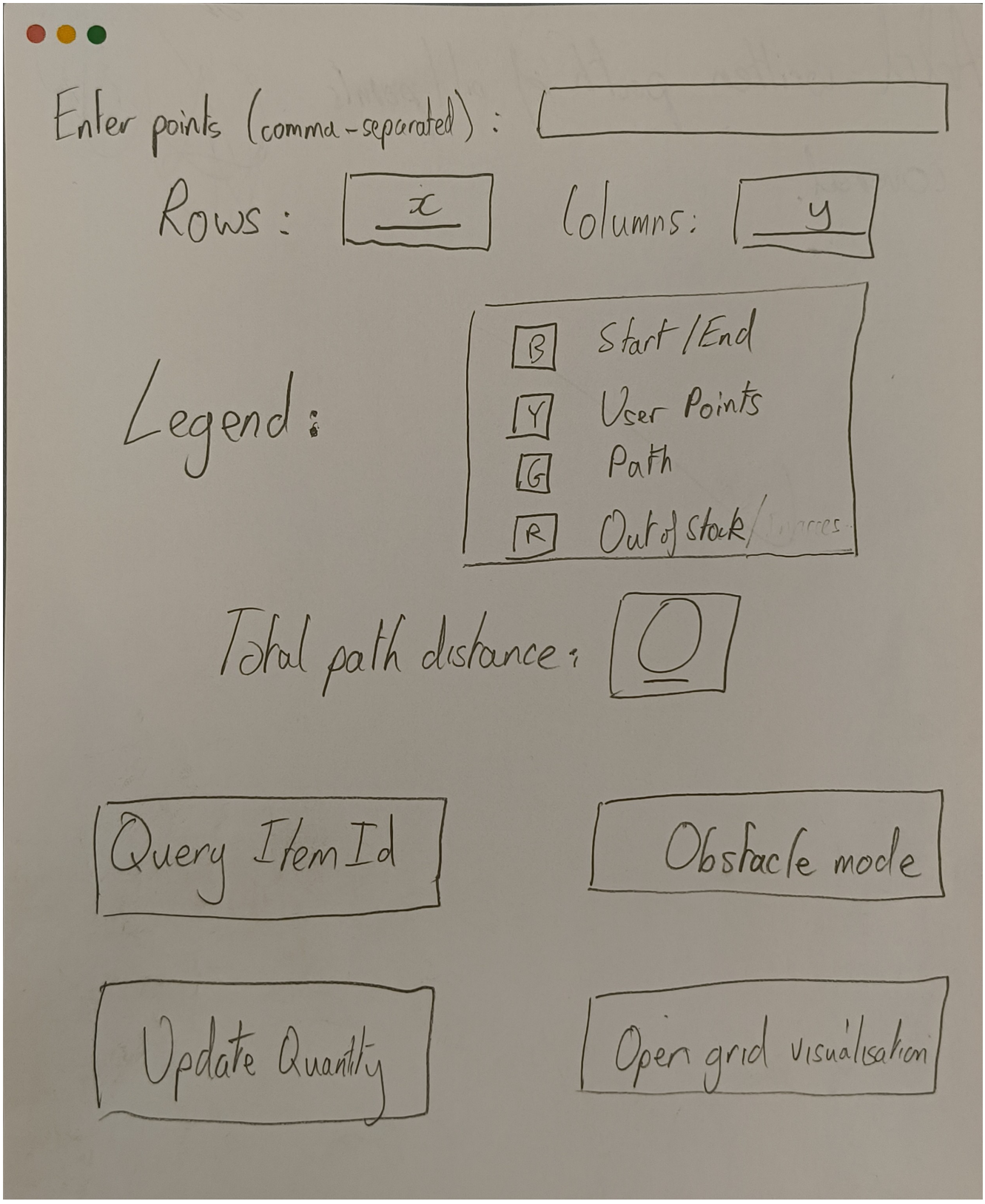


Fig. 3.9 - The main window of StockBot

Guideline alignment:

- Visibility of System Status
  - Legend shows current grid state meanings
  - Total path distance display
  - Clear input field for points
- Error Prevention
  - Explicit format guidance ("comma-separated")
  - Legend prevents color interpretation errors
  - Clear button labels prevent navigation mistakes
- Recognition vs Recall
  - Complete legend eliminates color code memorization
  - Input format shown directly above field
  - Descriptive button labels
- Aesthetic/Minimalist
  - Logical grouping of controls
  - Clean separation of input/visualization areas
  - Essential information only

This interface is consistent for the same reasons as the configuration screen.

Usability Analysis

The implementation of a text entry field for input is superior as it offers flexibility and choice to the user, while remaining as simple as possible. While I did consider multiple drop-down menus, one for the number of points and the others for items, I dismissed the idea on the fact that a large warehouse would lead to a very cluttered UI. This design choice also allows for easier parsing, as the entire entry can be taken at once rather than extraction from each drop-down menu.
The interface architecture establishes clear visual hierarchies that guide users through complex tasks while maintaining accessibility to all functions. The prominent placement of primary operations (point entry and grid visualization) balanced against the discrete positioning of secondary functions (querying, updating) creates an intuitive interaction flow.
While I initially did not want to include a legend, adding one would benefit the user; it eliminates cognitive overhead associated with colour mapping recall.

3.3.4 Grid Visualisation

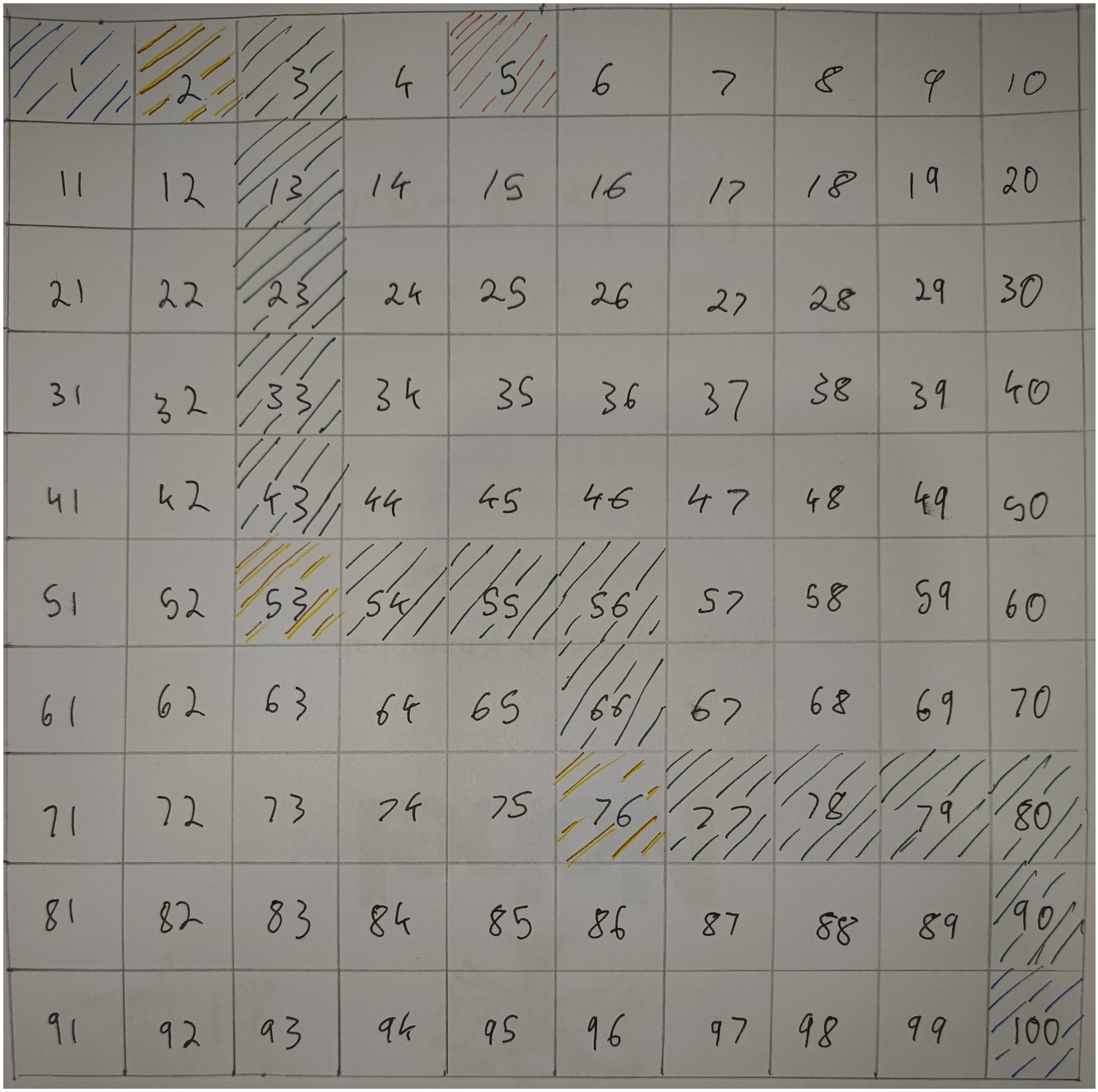


Fig. 3.10 - The visualisation of the path.

3.4 Unifying the algorithms

While the algorithms are complete, the solution has not yet been unified into a functional app; as I have used OOP principles in the code, it will be much easier to create a working prototype. See section x.x.x for the benefits OOP provides. The goal is to combine the shortest path algorithms (SPA), the graphical user interface (GUI), and the database management system into a cohesive application structured around these three core features. I have broken down the unification process into the module breakdown, the role of each module, the key functions and the interactions between modules.

3.4.1 Key modules breakdown

GUI

This serves as the main entry point for the program. It will manage user input, visualization, and overall coordination of the application. When necessary, the program will call methods or functions from the SPA and Database modules for computation and data management respectively.

- Primary Role: Acts as the main control system and interface between the user and the computational/backend modules.
- Key Functions:
  - Input Collection: Accepts user inputs like setting obstacles, defining waypoints, and initiating pathfinding.
  - Visualization: Displays the grid and highlights paths, obstacles, and waypoints for the user.
  - Interaction with SPA: Passes grid configurations and waypoints to the SPA module to compute the shortest path.
  - Interaction with Database: Calls Database to save or load grid states, configurations, or paths.
- Effect of unification:
  The GUI acts as a bridge between the user and the underlying logic. It passes the data it collects (like grid configurations or waypoints) to SPA for computations and calls Database for storage tasks. All modules are coordinated through the GUI, making it the central orchestrator.

SPA

This contains the core algorithms for computing the shortest path. It contains logic for handling grid data, waypoints and pathfinding. In the program, it will act as a processing unit, receiving data from the GUI, performing computations, and returning results.

- Primary Role: Implements and executes the algorithms that compute the shortest path on a grid.
- Key Functions:
  - Algorithm Logic: Encapsulates shortest path algorithms.
  - Data Handling: Accepts grid configuration and waypoints from GUI and processes them to compute the desired path.
  - Result Output: Returns the computed path to the GUI for visualization.
- Effect of unification:
  SPA is isolated from the GUI and database logic, focusing purely on computation. It accepts data from GUI and processes it using the appropriate algorithm. By keeping the logic separate, the module remains reusable and scalable, allowing future algorithms to be added without modifying the GUI or database.

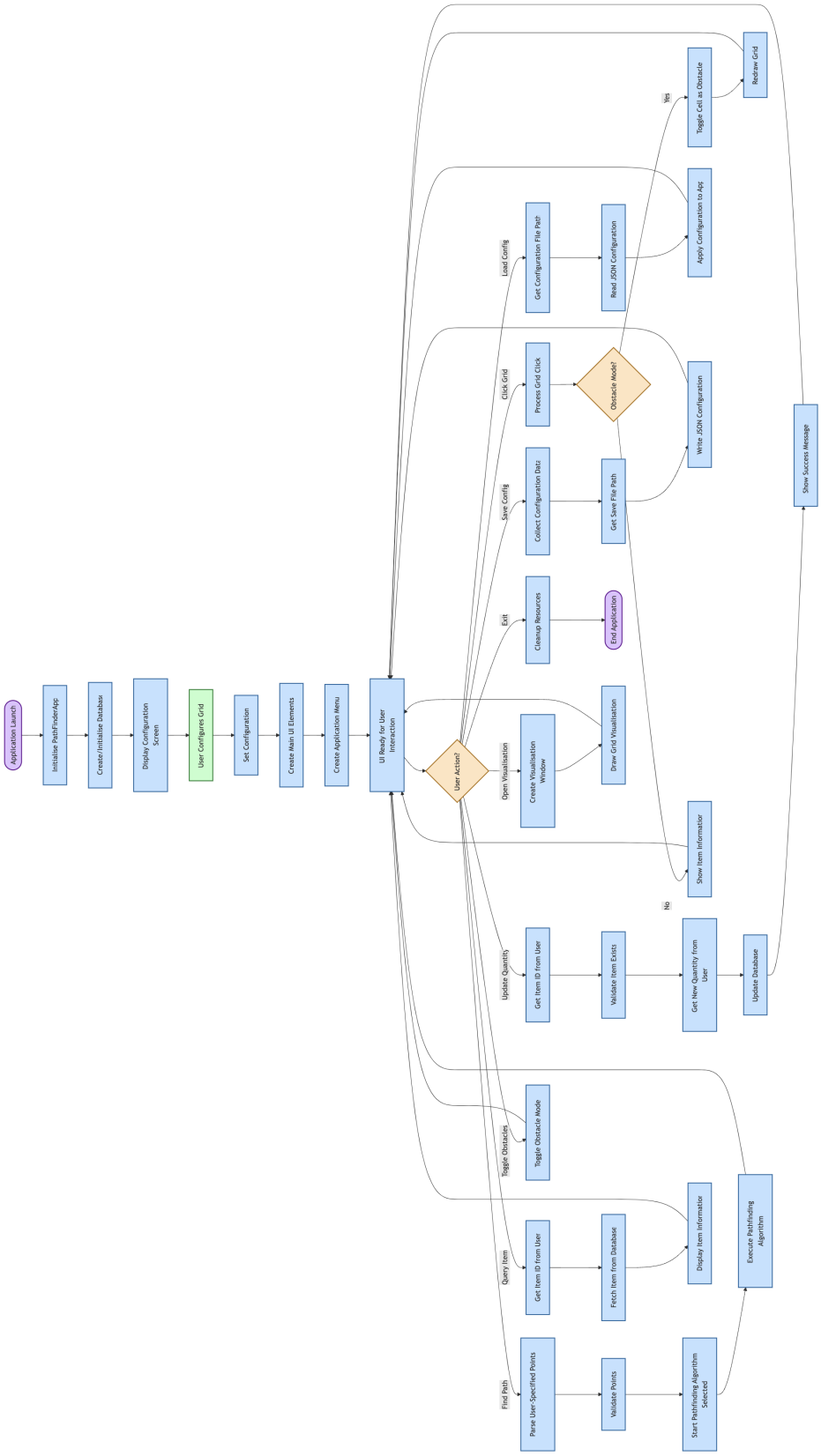
Database

This will provides data persistence by saving and loading grid configurations, waypoints, and paths. It will ensures the program can maintain state across sessions, enabling users to resume from where they left off.

- Primary Role: Stock management - ensuring stock is present and if not, to skip over the item.
- Key Functions:
  - Data Integrity: Ensures that saved data remains consistent and valid.
  - Stock assignment: Allows for items to be assigned to positions in the warehouse
  - Stock verification: Verifies that stock exists and if not, to remove the item from queue.
- Effect of unification:
  Database connects with GUI to store or retrieve data as requested by the user. This separation of concerns allows the GUI to focus on interaction and visualization, while the database handles storage operations independently.

These are just the key features that are integral to my solution. I have also included some quality-of-life features that I feel the stakeholders would benefit from, which I will include if time permits.

3.4.2 A unified implementation



3.5 Test data

3.5.1 During development

ID	Description	Expected	Actual	Pass?
T1.1	Input 0,0 and 9,9	Direct path between both points	Direct path between points	X
T1.2	Input 1,2 and 5,7	Direct path between defined points only	Direct path between 1,2 and 5,7	X
T1.3	Input -1,-1 and 4,8	Returns error	Error and break	~
T1.4	Input 0,1 and 10,10	Return error	Error and break	~

3.5.2 Post-development

#	Test	Expected Behaviour	Justification	Test Type
1	Input item 101 in a 10×10 grid via points entry field	Error message: "Point 101 does not exist in current grid configuration"	Tests error handling for out-of-bounds items to ensure the system validates input before attempting path calculation	Erroneous
2	Input "abc" in the points entry field	Error message: "Invalid Input: Please enter valid points"	Validates non-numeric input handling to prevent type errors in the processing logic	Erroneous
3	Configure a 0×5 grid using configuration screen	Error message: "Invalid grid dimensions, rows and columns must be at least 1"	Tests boundary validation for minimum grid size to ensure system prevents invalid configurations	Boundary
4	Configure a 101×50 grid using configuration screen	Error message: "Invalid grid dimensions. Maximum rows allowed is 100"	Tests upper boundary limits for grid configuration to ensure system prevents excessive resource consumption	Boundary

#	Test	Expected Behaviour	Justification	Test Type
5	Set quantity of ItemID 25 to 0, then attempt to include it in path by entering points "20,25,30"	Error message: "Cannot calculate path: Points [25] are obstacles"	Tests zero-stock validation to ensure the system prevents journeys to zero-inventory locations	Functional
6	Set quantity of ItemID 25 to -5 using update quantity function	Error message: "Invalid quantity. Value must be 0 or positive"	Tests negative value validation to ensure database integrity for stock quantities	Erroneous
7	Set obstacles to completely block all paths between points 1 and 25 in a 5×5 grid, then attempt pathfinding	Error message: "No path found between the given points" within 5 seconds	Verifies system correctly identifies impossible path scenarios and provides appropriate feedback	Functional
8	Enter start point (1) in points field for a 10×10 grid	Error message: "You have inputted a start and/or end point. Please remove it!"	Tests validation logic for preventing the selection of reserved points (start/end)	Functional
9	Enter end point (100) in points field for a 10×10 grid	Error message: "You have inputted a start and/or end point. Please remove it!"	Tests validation logic for preventing the selection of reserved points (start/end)	Functional
10	Enter comma-separated points with trailing comma "5,10,15,"	Error message: "Invalid Input: Please enter valid points"	Tests parsing robustness for malformed input to ensure the system handles common input errors	Erroneous
11	Enter a valid path with points "5,10,15" in a 10×10 grid with all items in stock	System calculates and displays optimal path including points 1,5,10,15,100 within 5 seconds	Tests core pathfinding functionality under normal operating conditions	Functional
12	Update quantity of non-existent ItemID 101 in a 10×10 grid	Error message: "Error: ItemID 101 does not exist"	Tests boundary validation for database operations to prevent manipulation of non-existent records	Boundary

#	Test	Expected Behaviour	Justification	Test Type
13	Query information for ItemID 0	Error message: "No item found" or "ItemID must be at least 1"	Tests lower boundary validation for item query functionality	Boundary
14	Configure a 1×1 grid (minimum possible size)	System should create grid with single cell, start and end both at position 1	Tests minimum boundary case for grid configuration to ensure the system handles edge cases	Boundary
15	Configure a 100×100 grid (maximum allowed size)	System should create grid with 10,000 cells without freezing or crashing	Tests maximum boundary case for grid configuration to assess system stability under maximum load	Boundary
16	Set all intermediate points to have 0 quantity in a 5×5 grid, then attempt pathfinding	Error message indicating all points are unavailable or out of stock	Tests comprehensive validation when all requested points are invalid	Functional
17	Create a path where points 5,10,15 are visited, then run again after setting point 10 to quantity 0	Second run should find alternative path excluding point 10	Tests system adaptation to changing inventory conditions	Functional
18	Save configuration with specific setup to JSON file, then load the same file	System should restore exact grid dimensions, obstacles, and item quantities with 100% accuracy	Tests data persistence functionality to ensure configurations can be reliably saved and restored	Functional
19	Load a deliberately corrupted JSON configuration file	Error message indicating invalid configuration file, system should maintain previous state	Tests error handling for data integrity issues to prevent system corruption	Erroneous
20	Rapidly click the "Find Path" button 10 times in succession	System should either queue requests or ignore additional clicks while processing, without crashing	Tests UI stability under rapid user interaction to ensure the system handles potential race conditions	Stress

#	Test	Expected Behaviour	Justification	Test Type
21	Set obstacles in a pattern forcing a very long path (>50 steps) in a 10×10 grid	System should calculate correct path navigating all obstacles within reasonable time (<10 seconds)	Tests pathfinding algorithm performance in worst-case scenarios	Performance
22	Request path through 20 intermediate points in a 20×20 grid	System should calculate optimal path visiting all points within 10 seconds	Tests algorithm scalability for complex routing scenarios	Performance
23	Enter points with mixed spacing "5, 10,15, 20"	System should correctly parse input and calculate path through points 5,10,15,20	Tests input parsing flexibility to handle various user input formats	Functional
24	Use obstacle mode to create a complex maze with only one valid orthogonal path, then request pathfinding	System should find the only valid path through maze without attempting diagonal moves	Tests path finding in highly constrained environments	Functional
25	Click on a grid cell with quantity 5, verify information displayed	Dialog should show correct ItemID, row, column, and quantity (5)	Tests item information display functionality	Functional
26	Set ItemID 42 to quantity 0 and verify its visual representation in grid	Cell should be highlighted in red color in the visualization	Tests correct visual indication of out-of-stock items	Functional
27	Update quantity of an item to 1, then traverse path through this item	Item should be flagged as below threshold (<2) after path completion, with quantity updated to 0 and color changed to red	Tests low stock detection, flagging and visual indication	Functional
28	Query an item after updating its quantity	Dialog should show updated quantity value, not original value	Tests database consistency after update operations	Functional
29	Set quantity of an item to maximum integer value (2,147,483,647)	System should accept and store this value without errors	Tests handling of extreme values within valid range	Boundary

#	Test	Expected Behaviour	Justification	Test Type
30	Find path in a 10×10 grid with obstacles placed to create exactly 2 possible orthogonal paths	System should find the shorter of the two possible paths	Tests optimal path selection when multiple valid paths exist	Functional
31	Run pathfinding with valid points "5,10,15" then immediately change grid configuration	Either current operation should complete with original grid or error should indicate configuration changed during operation	Tests system stability during concurrent operations	Functional
32	Export grid visualization to image file after calculating complex path	Exported image should accurately represent grid with all cells, paths, obstacles and point highlights matching on-screen display	Tests data visualization consistency between display and export	Functional
33	Create a grid with obstacles in positions that would make start or end points inaccessible	System should detect that no path is possible and display appropriate error message	Tests validation for critical point accessibility	Functional
34	Configure grid with alternating obstacles creating a "checkerboard" pattern	System should correctly calculate path navigating through the pattern using only orthogonal movements	Tests pathfinding in highly constrained but solvable scenarios	Functional
35	Set item quantities for points along a path to exactly 1, then run path twice	First run should succeed with all cells in path, second run should omit previously visited points that now have 0 quantity	Tests stock depletion scenario handling	Functional
36	Configure grid with obstacles along all edges except start and end points	System should calculate path navigating through the internal grid area using only orthogonal movements	Tests boundary obstacle handling	Functional
37	Open visualization window, close it, then request pathfinding	System should reopen visualization window and display calculated path correctly	Tests window management and state recovery	Functional

#	Test	Expected Behaviour	Justification	Test Type
38	Load configuration with large grid (50×50), switch to small grid (5×5), then back to large grid	System should correctly render both grid sizes without display artifacts or sizing issues	Tests UI scaling and flexibility	Functional
39	Leave points entry field blank and click "Find Path"	System should handle empty input gracefully, calculating direct path from start to end	Tests handling of minimal/empty input	Boundary
40	Verify color coding in grid visualization for start point (position 1)	Start point should be highlighted in blue color	Tests correct visual indication of start point	Functional
41	Verify color coding in grid visualization for end point (position 25 in 5×5 grid)	End point should be highlighted in blue color	Tests correct visual indication of end point	Functional
42	Verify color coding in grid visualization for user-specified points (5,10,15)	User points should be highlighted in yellow color	Tests correct visual indication of user-specified points	Functional
43	Verify color coding in grid visualization for calculated path	Path cells should be highlighted in green color	Tests correct visual indication of calculated path	Functional
44	Verify color coding in grid visualization for obstacle cells	Obstacle cells should be highlighted in gray color	Tests correct visual indication of obstacles	Functional
45	Attempt to move diagonally in path by editing database constraints	System should only use orthogonal movements (up, down, left, right) in calculated path	Tests movement constraints enforcement	Boundary
46	Configure ItemID 50 as an obstacle, then attempt to include it in a path	System should exclude the obstacle from path calculation or return error if path is not possible	Tests obstacle avoidance in path calculation	Functional
47	Set multiple consecutive obstacles to form a line, leaving exactly one path	System should find the only available orthogonal path around the obstacle line	Tests complex obstacle navigation	Functional

#	Test	Expected Behaviour	Justification	Test Type
48	Set every other item in alternating pattern to quantity 0, then request a path	System should find path avoiding all zero-quantity items, using only available items	Tests comprehensive avoidance of out-of-stock items	Functional
49	Click "Toggle Obstacle Mode" button and verify its visual state	Button should change appearance (sunken, gray background) when obstacle mode is active	Tests UI state indicators	Functional
50	In a 10×10 grid, create a path with 8 intermediate points forming a spiral pattern	System should calculate optimal path visiting all points in correct sequence using only orthogonal movements	Tests complex path optimization	Functional
51	Verify legend in visualization window	Legend should display correct color coding: blue for start/end, yellow for user points, green for path, red for out of stock, gray for obstacles	Tests visual guidance elements	Functional
52	After path calculation, verify path output text area	Text area should display correct sequence of points in format "1 − > 5 − > 10 − > 15 − > 25"	Tests path reporting functionality	Functional
53	Verify path distance label after calculation	Label should show "Total Path Distance: X" where X equals (number of points in path - 1)	Tests path metrics reporting	Functional
54	With a 5×5 grid where ItemID 13 has quantity 1, perform path through this point and verify quantity update	Database query should show quantity 0 for ItemID 13 after path completion	Tests stock level update functionality	Functional
55	Enter multiple items with same position in points field, e.g., "5,5,5"	System should deduplicate points and calculate path as if single point "5" was entered	Tests handling of duplicate inputs	Erroneous
56	Click "Help" menu and verify help window contents	Help window should display comprehensive application usage instructions	Tests documentation accessibility	Functional

#	Test	Expected Behaviour	Justification	Test Type
57	Enter a mixture of valid and invalid points, e.g., "5,999,15" in 10×10 grid	Error message should identify specific invalid point: "Point 999 does not exist"	Tests granular error reporting	Erroneous
58	Verify grid numbering in visualization for 5×5 grid	Cells should be numbered 1-25 in row-major order (row 0, col 0 = 1; row 4, col 4 = 25)	Tests grid coordinate system	Functional
59	Set an item to quantity 0, verify it appears red, then update to quantity 5	Cell should change from red to normal color (white) after quantity update	Tests dynamic visual updates based on stock changes	Functional
60	Click "Query ItemID" and enter valid ID, then verify dialog contents	Dialog should display correct row, column and current quantity information	Tests item information retrieval	Functional
61	Verify database update after loading configuration file with specific quantities	Database quantities should exactly match those in the configuration file for all items	Tests data integrity during configuration loading	Functional
62	Test cell size scaling by comparing 5×5 grid to 50×50 grid visualization	Cells should be appropriately scaled to fit visualization window while maintaining readability	Tests UI scaling algorithms	Functional
63	In a 10×10 grid with all items in stock, measure time to calculate path with 1, 5, and 9 intermediate points	Time should remain under 5 seconds for all three scenarios, with time increasing proportionally to complexity	Tests performance scaling	Performance
64	Test system memory usage during continuous operation for 10 minutes with repeated path calculations	Memory usage should remain stable without significant growth over time	Tests resource management	Performance

#	Test	Expected Behaviour	Justification	Test Type
65	Verify font scaling in grid visualization for different grid sizes	Text should remain readable across all supported grid sizes (1×1 to 100×100)	Tests accessibility and usability	Functional
66	Calculate path through points that create a rectangular pattern, verify optimality	System should find shortest path to visit all points using only orthogonal movements	Tests optimization effectiveness	Functional
67	After obstacle placement, attempt to query item information for obstacle cell	System should indicate cell is an obstacle, not a regular item	Tests data representation consistency	Functional
68	Enter floating point numbers in points field, e.g., "5.5, 10.7"	System should reject input as invalid and display appropriate error message	Tests numeric input validation	Erroneous
69	Enter negative numbers in points field, e.g., "-5, -10"	System should reject input as invalid and display appropriate error message	Tests numeric input validation	Erroneous
70	Set all items to quantity 1, execute path through all items, verify database update	All visited items should be updated to quantity 0 and marked as out of stock (red)	Tests comprehensive stock update	Functional

3.6 Further data

I will be using the following data for post-development in addition to the test data above. These are the parameters I will set for generic testing for the stakeholders. They will test it as they see fit and report back with any final feedback. I will not be providing them with exact data as I wish to simulate a real environment and hence test this software in a less controlled manner; this is why this data is more ambiguous.

- Grid layout: 15 x 15 default, user discretion to change
- Random database generation
- Point selection varying between 2 and 224, using a random number generator
- Number of points varying between 1 and 25, using a random number generator

Section 4

Developing the solution

4.1 Sprint Ada

This is Sprint Ada, the first iteration of my program. This iteration will be terminal-based, and mainly for creating the BFS implementation and adding in relevant functionality such as being able to enter multiple points, a basic visualisation etc.

4.1.1 Tasks

Task ID	Task Description
OCSP-001	BFS Initialisation: Compare the representation methods available for BFS and then find a way to program that representation into the first version of the program. Main focus on a CLI and only 2 points
OCSP-002	Terminal Visualisation: Create a basic visualisation of a path between 2 points in the terminal
OCSP-003	Multi-point entry: Allow user to enter more than 2 points and find the shortest path between each, forming a large path from start to end.

4.1.2 Purpose

This sprint entails me creating the first, basic algorithm for the SPA feature, and adding the multi-point entry so that it is useful in this scenario, as I intend for multiple items to be picked up at once.

4.1.3 Sprint Planning Details

Technical Approach

As this is still a very basic program and is in the alpha stages of development, this prototype will start with procedural programming, and once the main BFS features have been confirmed to work in the first few iterations, I will move to an object-oriented approach. (OCSP-001)

1. Create a 10 × 10 2D array to represent the graph.
2. Define the start and end nodes.
3. Initialize the queue with the start node.
4. Create an empty set for visited nodes.
5. Define possible movement directions (up, down, left, right).
6. While the queue is not empty:
   (a) Pop the first path from the queue.
   (b) Get the last node in the current path.
   (c) If the last node is the end node, return the current path.
   (d) If the last node has not been visited, mark it as visited.
   (e) For each possible direction:
      i. Calculate the new node.
      ii. If the new node is within bounds and not visited:
         - Create a new path including the new node
         - Add the new path to the queue
7. Create a function to repeat the pathfinding process for all points the user defines to find the shortest path between each pair of nodes. (OCSP-003)
8. Display the path in a pretty format. (OCSP-002)

Architecture & Structural Considerations

Below are the data structures I plan to use.

- 2D Array (List of Lists): The graph is represented as a 10 x 10 2D array (list of lists in Python), where each element can be either 0 (indicating a free cell) or 1 (indicating an obstacle).
- Queue: A list is used to implement the queue for BFS, where paths are stored and processed in a first-in, first-out (FIFO) manner.
- Set: A set is used to keep track of visited nodes to avoid processing the same node multiple times.
- List: Lists are used to store paths and directions.

Dependencies

There are no dependencies as such currently, I have opted to use Python's built-in functions (namely the list) for the queue rather than the external library `queue`. This may change in future iterations if the code becomes too complex.

4.1.4 Development Summary

Iteration 1 - Hours: 3

- **Progress made:**
  - OCSP-001: Created a fully working implementation of BFS that outputs the path it took, tested on a simple 10x10 grid.
  - OCSP-002: Came up with an approach on how to output the path in a more interactive format, similar to the interface I presented in the usability section (see section X.X.X). I plan to use placeholder characters in the array to interpret the terminal output, as colours are not supported in most terminal emulators. [*] represents a user input point, and [=] represents the path taken.
- **Blockers identified:**
  - I used my knowledge from the CS50AI course to create the basic BFS implementation between 2 points. However, I struggled to think about how I could implement multiple points.
- **Plan for next day:**
  - Find a way to calculate the shortest path between more than 2 points.
  - Add the visualisations using placeholder characters.

Iteration 2 - Hours: 1.5

- **Progress made:**
  - OCSP-002: I completed the visualisation using my placeholder characters, displaying a grid in the terminal.
  - OCSP-003: I managed to come up with an approach where the BFS algorithm is run between each pair of points, and the path is then connected together.
- **Blockers identified:**
  - I found that the grid did not display correctly as the user could not differentiate between their chosen points and the path followed. This was a very quick fix of highlighting user points AFTER the path was formed, overwriting the path syntax with the point syntax.

4.1.5 Sprint Ada Implementation

Iteration 1: The BFS algorithm

Code Changes:

- **GitHub Commit(s):** d2fe05e, ad6ff95, e2c67a46
- **Modular Structure:**
  - I enclosed the BFS algorithm in a function called `bfs` with the parameters `graph_in`, `start` and `end`. I originally intended to use consistent names, however Qodana flagged that this is not conventional. The naming scheme would have been in violation of PEP 3104, which addressed this issue. As such, I changed variable names like `graph` to `graph_in`, which is still an appropriate name (as it refers to the parameter being passed INto the function) but does not violate Python conventions.
  - I chose a basic function structure for now, as I have only made a small part and single feature of my solution. However, I did implement sub-programs to organise my code better and allow for better debugging.

Code Quality:

- **Annotations added:** As I was planning to continue this at a later time, I annotated each step of the BFS algorithm so that I could easily backtrack and visualise what was happening. I annotated most lines to ensure I would understand exactly how the algorithm worked and I could dry-run the algorithm in my head.
- **Variable/Structure naming:** I followed the lower-case underscore convention as defined by PEP 8. I focused on using industry terminology as my variable names, for example `path` and `node`
- **Modular approach:** I have encapsulated all BFS-related code in a single BFS function. I have opted for this approach as BFS is a relatively simple algorithm, meaning the code is quite short and is appropriate to group into a single function.

Code Implementation:

```
rows, cols = 10, 10
graph = [[0 for _ in range(cols)] for _ in range(rows)]

def bfs(graph_in, start, end):
    queue = [[start]] # Start with the start node
    visited = set() # Keep track of visited nodes
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    while queue:
        path = queue.pop(0) # Get the first path in the queue
        x, y = path[-1] # Get the last node in the path

        if (x, y) == end:
            return path # Return the path if we reach the end

        if (x, y) not in visited: # If the node has not been visited
            visited.add((x, y))  # Mark the node as visited
            for dx, dy in directions:  # Check all possible directions
                nx, ny = x + dx, y + dy  # Calculate the new node
                if 0 <= nx < rows and 0 <= ny < cols:
                # Check if the new node is within the bounds
                    if (nx, ny) not in visited:
                    # Check if the new node is not visited
                        new_path = list(path) + [(nx, ny)]  # Add new node to path
                        queue.append(new_path) # Add new path to queue

    return None  # Return None if no path is found


start_node = (0, 0)
end_node = (4, 8)

path = bfs(graph, start_node, end_node)
print(path)
```

```python
rows, cols = 10, 10
graph = [[0 for _ in range(cols)] for _ in range(rows)]

def bfs(graph_in, start, end):
    queue = [[start]] # Start with the start node
    visited = set() # Keep track of visited nodes
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    while queue:
        path = queue.pop(0) # Get first path in the queue
        x, y = path[-1] # Get last node in the path

        if (x, y) == end:
            return path # Return path if we reach the end

        if (x, y) not in visited: # If node has not been visited
            visited.add((x, y))  # Mark node as visited
            for dx, dy in directions:  # Check all possible directions
                nx, ny = x + dx, y + dy  # Calculate the new node
                if 0 <= nx < rows and 0 <= ny < cols:
                  # Check if the new node is within bounds
                    if (nx, ny) not in visited:
                      # Check if not visited
                        new_path = list(path) + [(nx, ny)]  # Add new node to path
                        queue.append(new_path) # Add new path to the queue

    return None  # Return None if no path is found


start_node = (0, 0)
end_node = (4, 8)

path_in = bfs(graph, start_node, end_node)
print(path_in)
```

Fig. 4.1 A coloured screenshot of the code

Prototype Version:

Currently, the BFS algorithm is working well for 2 defined points: a start and end node. It outputs a basic list of the coordinates that were followed to reach the end node from the start node. However, there is no visualisation as of yet, this will be implemented in the next iteration after some planning. As well as this, the BFS algorithm can currently handle only 2 points at a time, therefore I will be researching into how I can implement more points and allow the user to be able to set points to stop at.

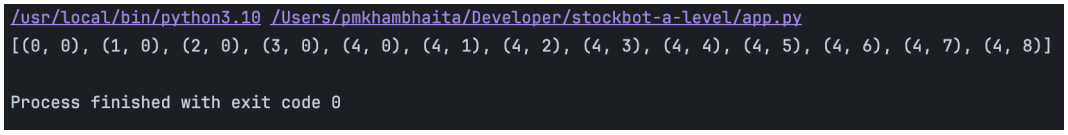


Fig. 4.2 The output of my algorithm with start at (0,0) and end at (4,8)

Testing:

ID	Description	Expected	Actual	Pass?
T1.1	Input 0,0 and 9,9	Direct path between both points	Direct path between points	X
T1.2	Input 1,2 and 5,7	Direct path between defined points only	Direct path between 1,2 and 5,7	X
T1.3	Input -1,-1 and 4,8	Returns error	Error and break	~
T1.4	Input 0,1 and 10,10	Return error	Error and break	~

Table 4.1 Testing results for iteration 1

T1.1 and T1.2 were successful, meaning the core functionality of the program is functional as expected. However, T1.3 and T1.4 were partially successful. While I did include the validation, I did not add a graceful error message, it was left to the basic python error-catching mechanisms. This will be fixed in the next iteration.

Screenshots of tests/program

```
/usr/local/bin/python3.10 /Users/pmkhambhaita/Developer/stockbot-a-level/app.py
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9)]

Process finished with exit code 0
```

Fig. 4.3 T1.1 Output

```
/usr/local/bin/python3.10 /Users/pmkhambhaita/Developer/stockbot-a-level/app.py
[(1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6), (5, 7)]

Process finished with exit code 0
```

Fig. 4.4 T1.2 Output

```
/usr/local/bin/python3.10 /Users/pmkhambhaita/Developer/stockbot-a-level/app.py
None

Process finished with exit code 0
```

Fig. 4.5 T1.2 Output

```
/usr/local/bin/python3.10 /Users/pmkhambhaita/Developer/stockbot-a-level/app.py
None

Process finished with exit code 0
```

Fig. 4.6 T1.2 Output

Validation:

- Boundary check: I ensured that the new node `(nx, ny)` is within the bounds of the graph
- Visited check: I checked that the new node `(nx, ny)` has not been visited before.

Qodana Analysis

- Issues identified: Shadowed name from outer scope
- Resolved issues: Modified variable name to prevent this

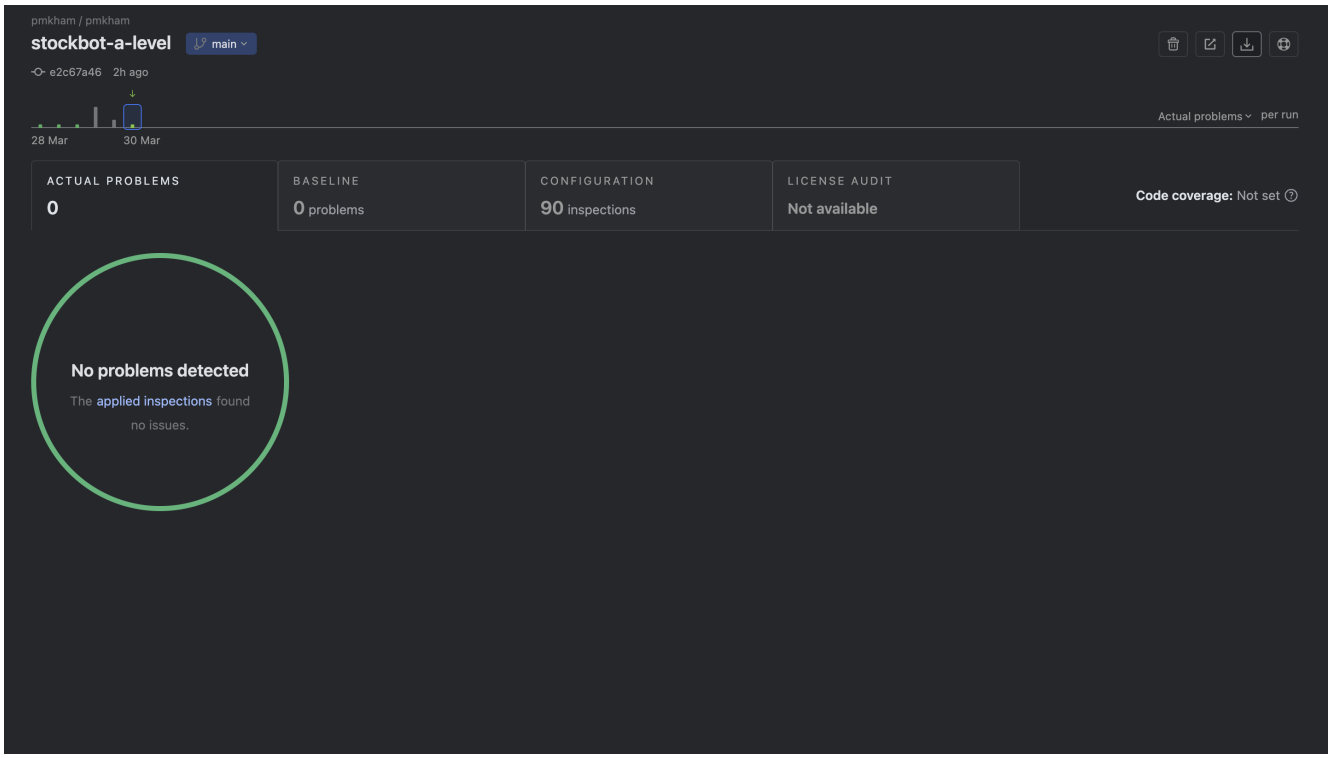


Fig. 4.7 As of commit e2c67a46, no problems were identified.

Review:

- Overall this iteration was quite successful. I managed to get a functional BFS algorithm working between 2 points, however I must be careful in not only validating but adding error messages for specific cases.
- Since this is still quite basic, I stuck to a simple procedural format rather than object-oriented principles. In the next iteration, I will be applying object-oriented principles as the program greatens in complexity.

Iteration 2: Multi-point BFS and Visualisation

Code Changes:

- **GitHub Commit(s):** x,x,x
- **Modular Structure:**
  - I have now implemented an object-oriented approach to the problem, separating each section into their respective classes. Now that my solution is developing in complexity, I decided to separate the elements of my program further into distinct classes, referencing each other when necessary.

Code Quality:

- **Annotations added:** As I was planning to continue this at a later time, I annotated each step of the BFS algorithm so that I could easily backtrack and visualise what was happening. I annotated most lines to ensure I would understand exactly how the algorithm worked and I could dry-run the algorithm in my head.
- **Variable/Structure naming:** I followed the lower-case underscore convention as defined by PEP 8. I focused on using industry terminology as my variable names, for example `path` and `node`
- **Modular approach:** I have encapsulated all BFS-related code in a single BFS function. I have opted for this approach as BFS is a relatively simple algorithm, meaning the code is quite short and is appropriate to group into a single function.

Code Implementation:

```
rows, cols = 10, 10
graph = [[0 for _ in range(cols)] for _ in range(rows)]

def bfs(graph_in, start, end):
    queue = [[start]] # Start with the start node
    visited = set() # Keep track of visited nodes
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    while queue:
        path = queue.pop(0) # Get the first path in the queue
        x, y = path[-1] # Get the last node in the path

        if (x, y) == end:
            return path # Return the path if we reach the end

        if (x, y) not in visited: # If the node has not been visited
            visited.add((x, y))  # Mark the node as visited
            for dx, dy in directions:  # Check all possible directions
                nx, ny = x + dx, y + dy  # Calculate the new node
                if 0 <= nx < rows and 0 <= ny < cols:
                # Check if the new node is within the bounds
                    if (nx, ny) not in visited:
                    # Check if the new node is not visited
                        new_path = list(path) + [(nx, ny)]  # Add new node to path
                        queue.append(new_path) # Add new path to queue

    return None  # Return None if no path is found


start_node = (0, 0)
end_node = (4, 8)

path = bfs(graph, start_node, end_node)
print(path)
```

```python
rows, cols = 10, 10
graph = [[0 for _ in range(cols)] for _ in range(rows)]

def bfs(graph_in, start, end):
    queue = [[start]] # Start with the start node
    visited = set() # Keep track of visited nodes
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    while queue:
        path = queue.pop(0) # Get first path in the queue
        x, y = path[-1] # Get last node in the path

        if (x, y) == end:
            return path # Return path if we reach the end

        if (x, y) not in visited: # If node has not been visited
            visited.add((x, y))  # Mark node as visited
            for dx, dy in directions:  # Check all possible directions
                nx, ny = x + dx, y + dy  # Calculate the new node
                if 0 <= nx < rows and 0 <= ny < cols:
                  # Check if the new node is within bounds
                    if (nx, ny) not in visited:
                      # Check if not visited
                        new_path = list(path) + [(nx, ny)]  # Add new node to path
                        queue.append(new_path) # Add new path to the queue

    return None  # Return None if no path is found


start_node = (0, 0)
end_node = (4, 8)

path_in = bfs(graph, start_node, end_node)
print(path_in)
```

Fig. 4.8 A coloured screenshot of the code

Prototype Version:

Currently, the BFS algorithm is working well for 2 defined points: a start and end node. It outputs a basic list of the coordinates that were followed to reach the end node from the start node. However, there is no visualisation as of yet, this will be implemented in the next iteration after some planning. As well as this, the BFS algorithm can currently handle only 2 points at a time, therefore I will be researching into how I can implement more points and allow the user to be able to set points to stop at.

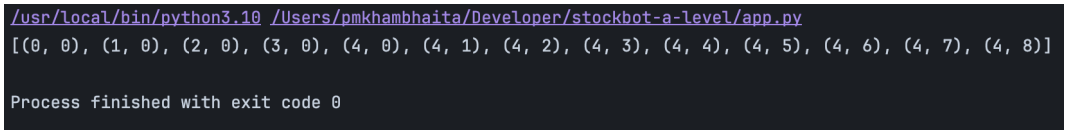


Fig. 4.9 The output of my algorithm with start at (0,0) and end at (4,8)

Testing:

ID	Description	Expected	Actual	Pass?
T1.1	Input 0,0 and 9,9	Direct path between both points	Direct path between points	X
T1.2	Input 1,2 and 5,7	Direct path between defined points only	Direct path between 1,2 and 5,7	X
T1.3	Input -1,-1 and 4,8	Returns error	Error and break	~
T1.4	Input 0,1 and 10,10	Return error	Error and break	~

Table 4.2 Testing results for iteration 1

T1.1 and T1.2 were successful, meaning the core functionality of the program is working as expected. However, T1.3 and T1.4 were partially successful. While I did include the validation, I did not add a graceful error message, it was left to the basic python error-catching mechanisms. This will be fixed in the next iteration.

Screenshots of tests/program

Fig. 4.10 T1.1 Output

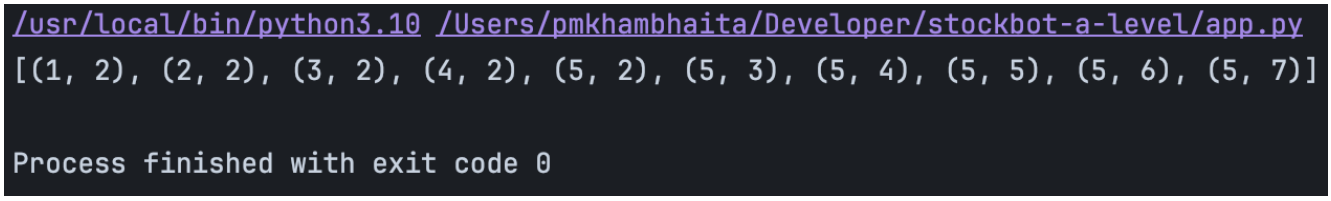


Fig. 4.11 T1.2 Output

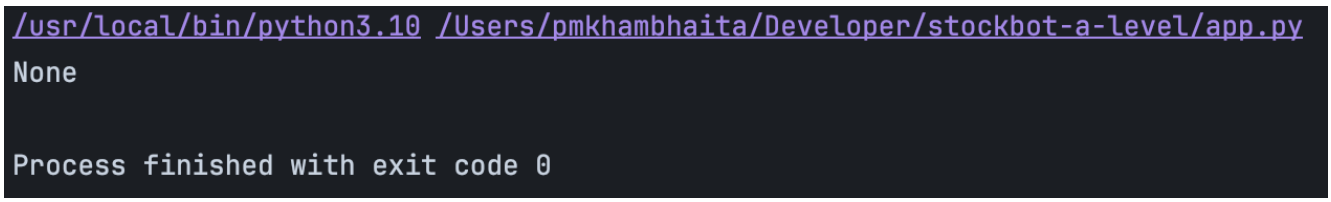


Fig. 4.12 T1.2 Output

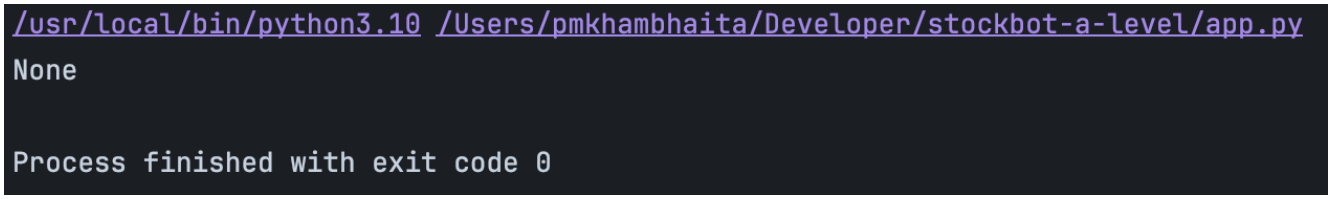


Fig. 4.13 T1.2 Output

Validation:

- Boundary check: I ensured that the new node `(nx, ny)` is within the bounds of the graph
- Visited check: I checked that the new node `(nx, ny)` has not been visited before.

Qodana Analysis

- Issues identified: Shadowed name from outer scope
- Resolved issues: Modified variable name to prevent this

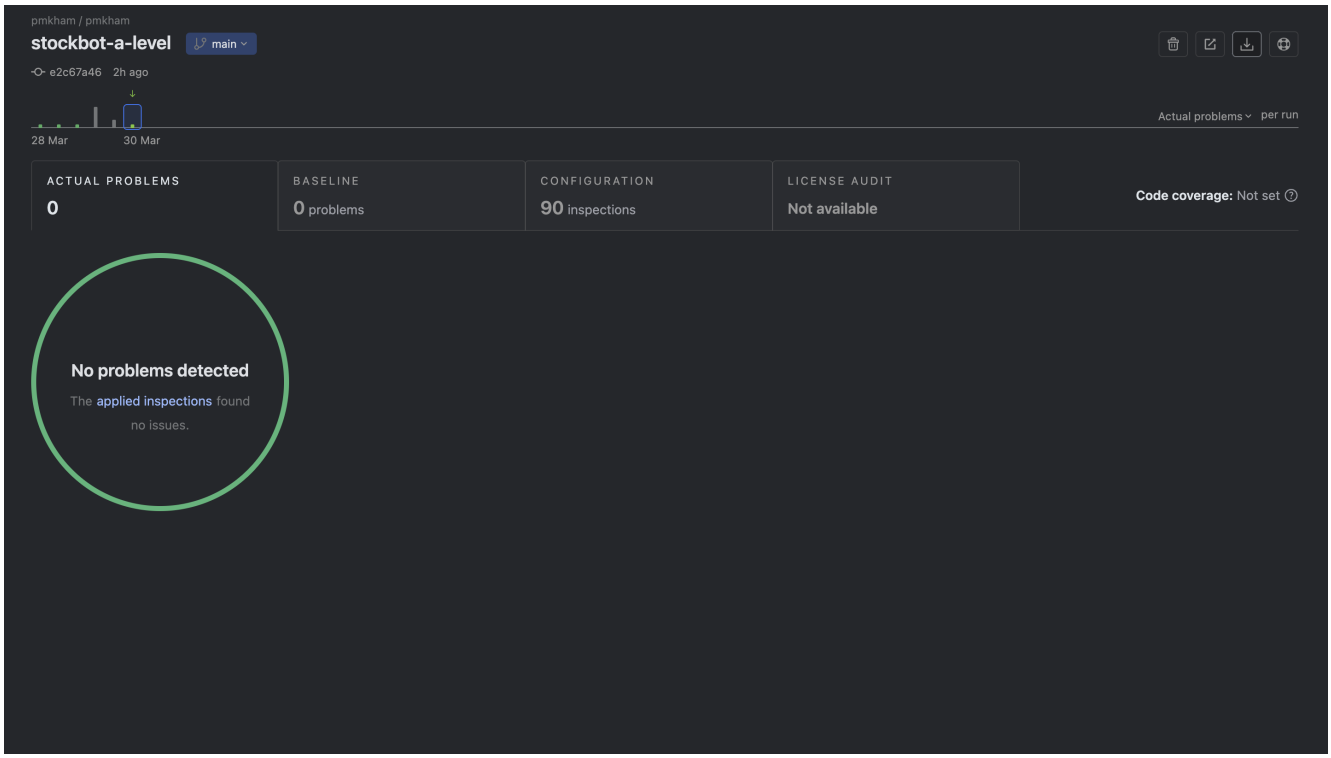


Fig. 4.14 As of commit e2c67a46, no problems were identified.

Review:

- Overall this iteration was quite successful. I managed to get a functional BFS algorithm working between 2 points, however I must be careful in not only validating but adding error messages for specific cases.
- Since this is still quite basic, I stuck to a simple procedural format rather than object-oriented principles. In the next iteration, I will be applying object-oriented principles as the program greatens in complexity.

4.1.6 Sprint Review and Retrospective

Accomplishments

- List completed Linear tasks with IDs
- Summary of specific features implemented with reference to requirements

Testing Summary

Metric	Count
Total tests conducted	Number
Tests passed	Number
Tests failed	Number
Fixed issues	Number

Table 4.3 Sprint 1 testing summary

- **Key validation implementations:**
  - Describe specific validation techniques implemented

Code Quality Metrics

- **Qodana final report summary:**
  - Specific code quality metrics
  - List of issues resolved with before/after comparisons
  - Technical debt measurements

Sprint Review Outcomes

- Specific feedback received
- Decisions made for next sprint

Retrospective

- **What went well:**
  - Specific processes or implementations that were successful
- **What could be improved:**
  - Specific areas for improvement with technical details
- **Action items for next sprint:**
  - Specific technical actions to take in next sprint

4.1.7 Mark Scheme Alignment Evidence

Iterative Development (15 marks potential)

Criterion	Evidence Provided
Stages of iterative process	List specific documentation elements showing all stages
Prototype versions	List all prototype versions with specific features
Modular structure	Describe how code structure demonstrates modularity
Code annotations	Detail annotation approach used for maintenance
Variable naming	Explain naming conventions with examples
Validation approaches	Summarize all validation techniques used
Review stages	List all review points with outcomes

Table 4.4 Evidence for iterative development mark criteria

Testing to Inform Development (10 marks potential)

Criterion	Evidence Provided
Testing at each stage	Summarize testing at each iteration with specific tests
Failed tests documentation	Explain how failed tests were documented with examples
Remedial actions	Summarize specific remedial actions taken with justifications

Table 4.5 Evidence for testing mark criteria

4.2 Sprint Number: 2

4.3 Final Solution Review

4.3.1 Solution Overview

Describe the complete solution with reference to requirements from analysis.

4.3.2 Code Structure and Modularity

Explain the final structure with diagrams if helpful.

4.3.3 Key Implementation Features

Highlight innovative or complex aspects of the solution.

4.3.4 Comprehensive Validation Strategy

Summarize the validation approach across the entire project.

4.3.5 Maintenance Considerations

Explain how the code is designed for future maintenance.

Section 5

Evaluation

References

[1] T. Point, "Data structure - breadth first traversal - tutorialspoint." https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm.

[2] GeeksforGeeks, "Depth first search - geeksforgeeks." https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/, 02 2019.

[3] A. Patel, "Introduction to a*." http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html, 2019.

[4] R. A. S, "A* search algorithm explained: Applications & uses." https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm#algorithm, 07 2021.

[5] GeeksforGeeks, "Genetic algorithms." https://www.geeksforgeeks.org/genetic-algorithms/, 06 2017.

[6] GeeksforGeeks, "Introduction to optimization with genetic algorithm." https://www.geeksforgeeks.org/introduction-to-optimization-with-genetic-algorithm/, 09 2024.

[7] "Warehouse worker." https://www.ioscm.com/blog/warehousing-strategies-for-building-a-productive-work-environment/.

[8] "Warehouse productivity." https://us.blog.kardex-remstar.com/hs-fs/hubfs/Imported%20sitepage%20images/Infographic_Blog_Update_Productivity.jpeg?width=800&height=444&name=Infographic_Blog_Update_Productivity.jpeg.

[9] A. Robotics, "Close-up of kiva robots." https://blog.scallog.com/en/amazon-robotics-robots-logistiques-supply-chain, 02 2024.

[10] "Joseph engelberger and unimate: Pioneering the robotics revolution." https://www.automate.org/robotics/engelberger/joseph-engelberger-unimate, 2024.

[11] D. Edwards, "Amazon now has 200,000 robots working in its warehouses." https://roboticsandautomationnews.com/2020/01/21/amazon-now-has-200000-robots-working-in-its-warehouses/28840/, 01 2020.

[12] C. Clifford, "Amazon robots in warehouse." https://www.ft.com/content/31ec6a78-97cf-47a2-b229-d63c44b81073, 03 2025.

[13] Scallog, "Amazon robotics and logistics robots in the supply chain." https://blog.scallog.com/en/amazon-robotics-robots-logistiques-supply-chain, 02 2024.

[14] O. Group, "The ocado smart platform (osp) | ocado group." https://www.ocadogroup.com/solutions/online-grocery, 2024.

[15] O. Group, "The ocado smart platform: Automating online grocery." https://www.youtube.com/watch?v=Eu3jgy2-tL8, 06 2023.

[16] O. Group, "Explained: The tech powering the ocado smart platform." https://www.youtube.com/watch?v=g2uMfpAHdGY, 09 2020.

[17] M. Speed, "Ocado and autostore settle three-year row over ecommerce tech." https://www.ft.com/content/7038e2f4-0201-4632-a79e-8613890e8c99, 07 2023.

[18] AutoStore, "Autostore | the facts behind the stacked design." https://www.youtube.com/watch?v=9UDgSofF_AY, 03 2023.

[19] B. Solutions, "Autostore system | goods to person automated storage & retrieval | bastian solutions." https://www.bastiansolutions.com/solutions/technology/goods-to-person/autostore/?srsltid=AfmBOop3Qs8YAH6JU69_gX8hp7hUlfPbCx6afmk7b3cxcemB0Ox5VlOW, 2025.

[20] A. Systems, "Autostore | introduction: Stop airhousing, start warehousing." https://www.youtube.com/watch?v=iHC9ec591lI, 03 2018.

[21] A. Systems, "Autostore robotic warehouse storage and inventory retrieval system." https://www.autostoresystem.com/.

[22] B. Elkari, L. OURABAH, H. SEKKAT, A. HSAINE, C. ESSAIOUAD, Y. BOUARGANE, and K. E. Moutaouakil, "Exploring maze navigation: A comparative study of dfs, bfs, and a* search algorithms.," *Statistics Optimization & Information Computing*, vol. 12, pp. 761–781, 02 2024.

[23] S. Overflow, "Time complexity analysis of bfs." https://stackoverflow.com/questions/26549140/breadth-first-search-time-complexity-analysis, 11 2014.

[24] SQLite, "File locking and concurrency in sqlite version 3." https://www.sqlite.org/lockingv3.html.

[25] A. Ajitsaria, "What is the python global interpreter lock (gil)? – real python." https://realpython.com/python-gil/.

[26] G. Dev, "Python strings | python education." https://developers.google.com/edu/python/strings.

[27] P. S. Foundation, "tkinter — python interface to tcl/tk — python 3.7.2 documentation." https://docs.python.org/3/library/tkinter.html, 2019.

[28] V. Joshi, "Breaking down breadth-first search by vaidehi joshi - basecs - medium." https://miro.medium.com/v2/resize:fit:1400/1*VM84VPcCQe0gSy44l9S5yA.jpeg.

Appendix A

Complete pseudocode in OCR ERL

This is a basic approach to my solution using the OCR Exam Reference Language. This does not incorporate all features that may be present in the final solution, but rather served as a starting point for me to build my solution. I used snippets and/or versions of this pseudocode within my documentation, to illustrate certain subprograms & how they might run.

A.0.1 BFS

Approach reasoning

Below is a rudimentary implementation of BFS in pseudocode. I have centralised all methods under the `BFS` class. By using an object-oriented approach, some benefits include:

- **Encapsulation** - Encapsulation is bundling data and methods that operate on the data within a single unit (class) and restricting direct access to some components. The `graph`, `visited`, and `queue` are private fields within the `BFS` class, meaning that only the class's methods can access and modify them directly. This ensures that the internal state of these objects is protected from exploits by individuals with malicious intent.
- **Abstraction** - Abstraction hides implementation details and exposes only essential functionality. Users of the `BFS` class don't need to understand the internal mechanics of how the queue is managed or how the path is found and reconstructed. This simplification makes the code easier to use and understand, as the complexity is hidden behind a simple interface.
- **Future Scalability** - OOP systems can be extended with minimal modification to existing code. If new features need to be added in the future, such as handling weighted edges or implementing different search strategies, the current structure allows for extending the `BFS` class by adding new methods or modifying existing ones.

Pseudocode

```
CLASS BFSPath
    PRIVATE graph
    PRIVATE visited
    PRIVATE queue
    PRIVATE database
    PRIVATE start = 1
    PRIVATE end
    PRIVATE width
    PRIVATE length

    CONSTRUCTOR BFSPath(graph, database, warehouseWidth, warehouseLength)
        THIS.graph = graph
        THIS.visited = EMPTY SET
        THIS.queue = NEW Queue()
        THIS.database = database
        THIS.width = warehouseWidth
        THIS.length = warehouseLength
        THIS.end = warehouseWidth * warehouseLength
    END CONSTRUCTOR



    FUNCTION FindPathWithWaypoints(waypoints)
        validWaypoints = EMPTY LIST
        FOR point IN waypoints
            IF point = start OR point = end THEN
                CONTINUE
            END IF
            IF database.HasStock(point) AND IsValidWaypoint(point) THEN
                validWaypoints.APPEND(point)
            END IF
        END FOR
        # Always start from start and end at end
        finalPath = EMPTY LIST
        currentPoint = start
        validWaypoints.INSERT(0, start)
        validWaypoints.APPEND(end)
        FOR i = 0 TO LENGTH(validWaypoints) - 1
            nextPoint = validWaypoints[i + 1]
            subPath = FindPath(currentPoint, nextPoint)
            IF subPath IS EMPTY THEN
```

```
            CONTINUE
        END IF
        IF LENGTH(finalPath) > 0 AND LENGTH(subPath) > 0 THEN
            subPath.RemoveFirst()
        END IF
        finalPath.EXTEND(subPath)
        currentPoint = nextPoint
    END FOR
    RETURN finalPath
END FUNCTION

FUNCTION FindPath(startVertex, endVertex)
    parentMap = EMPTY DICTIONARY
    THIS.visited = EMPTY SET
    THIS.queue = NEW Queue()
    THIS.queue.Enqueue(startVertex)
    ADD startVertex TO THIS.visited
    parentMap[startVertex] = ""
    WHILE NOT THIS.queue.IsEmpty()
        currentVertex = THIS.queue.Dequeue()
        IF currentVertex = endVertex THEN
            RETURN ReconstructPath(parentMap, startVertex, endVertex)
        END IF
        FOR EACH neighbor IN THIS.graph[currentVertex]
            IF neighbor NOT IN THIS.visited THEN
                THIS.queue.Enqueue(neighbor)
                ADD neighbor TO THIS.visited
                parentMap[neighbor] = currentVertex
            END IF
        END FOR
    END WHILE
    RETURN EMPTY LIST
END FUNCTION

FUNCTION ReconstructPath(parentMap, startVertex, endVertex)
    path = EMPTY LIST
    currentVertex = endVertex
    # Work backwards from end to start
    WHILE currentVertex != startVertex
        INSERT currentVertex AT START OF path
        currentVertex = parentMap[currentVertex]
        # Safety check in case of disconnected path
```

```
            IF currentVertex IS NULL THEN
                RETURN EMPTY LIST
            END IF
        END WHILE
        # Add the start vertex
        INSERT startVertex AT START OF path
        RETURN path
    END FUNCTION

    FUNCTION IsValidWaypoint(position) RETURNS BOOLEAN
        RETURN position > start AND
               position < end AND
               position <= (width * length)
    END FUNCTION
END CLASS
```

Analysis

- **Purpose and Functionality**
  The `BFSPath` component is responsible for computing the shortest path from the start position to the end position in the warehouse, incorporating user-defined waypoints if they are valid.

- **Class Attributes**
  - `graph`: Represents the warehouse layout as a graph where each position is a vertex, and edges represent valid paths between positions. This abstraction allows for flexible changes in warehouse structure.
  - `visited`: Keeps track of visited vertices during the BFS traversal, preventing redundant computations and infinite loops.
  - `queue`: Implements the BFS algorithm by maintaining vertices to be explored in a FIFO (First-In-First-Out) manner.
  - `database`: Provides access to the warehouse inventory for waypoint validation, ensuring the pathfinding logic dynamically incorporates inventory constraints.
  - `start` and `end`: Represent fixed entry and exit points for the warehouse. These points act as anchors for the pathfinding process.
  - `width` and `length`: Define the dimensions of the warehouse grid, enabling validation of waypoint positions against physical constraints.

- **Key Functions:**
  1. **Constructor:** Initializes the class with the warehouse `graph`, inventory `database`, and grid dimensions. Ensures consistent access to these components for other methods and establishes the fixed start and end positions.
  2. `FindPathWithWaypoints`: Computes the shortest path through user-defined waypoints. Key steps include:
     - Filtering waypoints to ensure validity (within bounds, not fixed positions, and in stock).
     - Including the start and end positions in the waypoint list to guarantee connectivity.
     - Using the BFS algorithm to compute sub-paths between consecutive waypoints. This modular approach simplifies debugging and ensures each sub-path is independently optimal.
     - Combining sub-paths into a single final path by removing redundant positions at the boundaries of sub-paths.
  3. `FindPath`: Implements BFS to compute the shortest path between two vertices. Maintains a parent map to reconstruct the path once the end vertex is reached. If no path is found, it returns an empty list to signify failure.
  4. `IsValidWaypoint`: Validates a waypoint by ensuring it lies within the grid dimensions, is not a fixed position, and satisfies logical constraints for pathfinding.

- **Design Decisions:**
  - **Private Attributes:** Ensure encapsulation and prevent unauthorized access to internal data structures like the `queue` or `visited` set. This promotes modularity and prevents unintended side effects. See Approach reasoning for the benefits of OOP.
  - **Dynamic Validation:** The `database` integration ensures waypoints are checked for stock dynamically, reflecting real-time inventory. This is critical in a warehouse setting where stock levels frequently change.
  - **Reconstruction of Path:** BFS guarantees shortest path computation between 2 points, hence the parent map is required to simplify the reconstruction process as more than 2 points will be entered. This is done by tracing each sub-path back from the end vertex, forming a single path with a sufficient solution.
- **Justifications:**
  - **Use of BFS:** BFS is optimal for unweighted graphs like the warehouse grid, ensuring the shortest path is found efficiently.
  - **Waypoint Validation:** Prevents unnecessary computation for invalid or unreachable points, improving performance and accuracy.
  - **Integration with Database:** Enhances functionality by dynamically considering stock availability during pathfinding, ensuring practical relevance and application.

A.0.2 Database operations

Approach reasoning

1. **Minimal Data Structure**
    - Only position and quantity stored
    - Simplifies database operations
    - Reduces potential for errors
    - Perfect for basic inventory tracking
2. **Error Handling**
    - TRY/CATCH blocks throughout
    - Clear error messages
    - Prevents database corruption
    - Helps with debugging
3. **Validation**
    - Position validation before all operations
    - Database constraints as secondary validation
    - Prevents invalid data entry
4. **Encapsulation**
    - Private attributes
    - Public methods for controlled access
    - Maintains data integrity
5. **Flexibility**
    - Adaptable to different warehouse sizes
    - Easy to extend with additional features
    - Clear interface for integration with routing system

```
CLASS WarehouseDatabase
    PRIVATE dbConnection
    PRIVATE width
    PRIVATE length
    PRIVATE start = 1
    PRIVATE end

    CONSTRUCTOR WarehouseDatabase(warehouseWidth, warehouseLength)
        width = warehouseWidth
        length = warehouseLength
        end = width * length
        InitializeDatabase()
    END CONSTRUCTOR

    PROCEDURE InitializeDatabase()
        TRY
            dbConnection = NEW SQLiteConnection("warehouse.db")
            ExecuteSQL("
                CREATE TABLE IF NOT EXISTS inventory (
                    position INTEGER PRIMARY KEY,
                    quantity INTEGER,
                    item_name TEXT,
                    last_updated TIMESTAMP,
                    CONSTRAINT valid_position CHECK (
                        position >= 1 AND position <= ? * ?
                    )
                )
            ", [width, length])

            # Initialize start and end positions if not exists
            InitializeFixedPositions()
        CATCH error
            OUTPUT "Failed to initialize database: " + error.message
            RAISE error
        END TRY
    END PROCEDURE
```

```
PROCEDURE InitializeFixedPositions()
    # Start position always has stock
    ExecuteSQL("
        INSERT OR IGNORE INTO inventory (position, quantity, item_name)
        VALUES (?, 1, 'Start Position')
    ", [start])

    # End position always has stock
    ExecuteSQL("
        INSERT OR IGNORE INTO inventory (position, quantity, item_name)
        VALUES (?, 1, 'End Position')
    ", [end])
END PROCEDURE

FUNCTION HasStock(position) RETURNS BOOLEAN
    TRY
        # Start and end positions always considered to have stock
        IF position = start OR position = end THEN
            RETURN TRUE
        END IF

        result = dbConnection.Query("
            SELECT quantity
            FROM inventory
            WHERE position = ? AND quantity > 0
        ", [position])
        RETURN LENGTH(result) > 0
    CATCH error
        OUTPUT "Failed to check stock: " + error.message
        RETURN FALSE
    END TRY
END FUNCTION
```

```
    PROCEDURE AddItem(position, quantity, itemName)
        IF position = start OR position = end THEN
            OUTPUT "Cannot modify fixed positions"
            RETURN
        END IF
        IF NOT IsValidPosition(position) THEN
            OUTPUT "Invalid position"
            RETURN
        END IF
        TRY
            ExecuteSQL("
                INSERT OR REPLACE INTO inventory
                (position, quantity, item_name, last_updated)
                VALUES (?, ?, ?, CURRENT_TIMESTAMP)
            ", [position, quantity, itemName])
            OUTPUT "Item added successfully at position " + ToString(position)
        CATCH error
            OUTPUT "Failed to add item: " + error.message
        END TRY
    END PROCEDURE

    FUNCTION IsValidPosition(position) RETURNS BOOLEAN
        RETURN position >= 1 AND position <= (width * length)
    END FUNCTION
END CLASS
```

Analysis

- **Purpose and Functionality:**
  The `WarehouseDatabase` component manages inventory data, ensuring accurate stock tracking and supporting operations like waypoint validation for pathfinding. This component abstracts inventory management into a dedicated class, simplifying the overall system architecture.

- **Class Attributes:**
  - `dbConnection`: Handles the connection to the SQLite database. This attribute centralizes database operations and ensures consistent access throughout the class.
  - `width` and `length`: Define the dimensions of the warehouse grid for position validation. These attributes are used to enforce logical constraints on inventory positions.
  - `start` and `end`: Represent fixed entry and exit points. These positions are immutable to ensure consistency across different components of the system.

- **Key Procedures:**

  1. `InitializeDatabase`: Sets up the SQLite database, creating an inventory table with constraints on valid positions. This procedure ensures the database schema aligns with the warehouse's logical structure.
  2. `InitializeFixedPositions`: Ensures the start and end positions are pre-defined with default stock. These positions are critical for pathfinding and must always exist.
  3. `HasStock`: Checks if a given position has stock by querying the database. This function considers fixed positions as always in stock, simplifying edge-case handling.
  4. `AddItem`: Adds or updates an item at a specified position in the inventory, provided it is a valid position. This procedure includes safeguards against modifying fixed positions, preserving system integrity.
  5. `IsValidPosition`: Validates that a position is within the warehouse bounds. This function ensures that all inventory operations respect physical constraints.

- **Design Decisions:**

  - **Use of SQLite:** Chosen for its lightweight nature, SQLite is sufficient for the scale of the warehouse inventory. It offers robust data handling without the overhead of a full-fledged database management system.
  - **Fixed Positions:** The start and end positions are immutable to maintain consistency across components. This design choice simplifies logic in the pathfinding and GUI components.
  - **Position Validation:** Constraints ensure data integrity, preventing invalid entries and reducing potential errors in downstream operations.

- **Justifications:**

  - **Encapsulation:** Keeps inventory management logic within the `WarehouseDatabase` class, simplifying integration with other components.
  - **Real-Time Stock Validation:** Enhances the pathfinding component by ensuring only valid positions are considered, reflecting real-world scenarios accurately.
  - **Error Handling:** Provides feedback on invalid operations, improving robustness and user experience.

A.0.3 Rudimentary GUI

Approach reasoning

1. **Visual Representation**
    - Grid-based visualization of warehouse
    - Color-coded positions for clarity
    - Intuitive user interface elements
    - Simplifies warehouse management
2. **User Interaction**
    - Button-based command interface
    - Input validation for waypoints
    - Clear feedback messages
    - Prevents user errors
3. **Integration with Database**
    - Seamless connection to inventory data
    - Real-time updates of warehouse state
    - Consistent representation of data
4. **Waypoint Management**
    - Simple waypoint addition
    - Position validation
    - Visual feedback through highlighting
    - Supports pathfinding operations
5. **Flexibility**
    - Adaptable to different warehouse dimensions
    - Easy to extend with additional features
    - Clear separation of GUI and logic

```
CLASS WarehouseGUI
    PRIVATE width
    PRIVATE length
    PRIVATE canvas
    PRIVATE waypointsList
    PRIVATE start = 1
    PRIVATE end
    CONSTRUCTOR WarehouseGUI(warehouseWidth, warehouseLength)
        width = warehouseWidth
        length = warehouseLength
        end = width * length
        waypointsList = EMPTY LIST
        InitializeGUI()
    END CONSTRUCTOR
    PROCEDURE InitializeGUI()
        canvas = CREATE Canvas(width * 50, length * 50)
        DrawGrid()
        CREATE Button("Add Waypoint", AddWaypoint)
        CREATE Button("Find Path", FindPath)
        CREATE Button("Update Stock", UpdateStock)
        CREATE Button("View Inventory", ViewInventory)
        # Highlight fixed positions
        HighlightCell(start, "green")
        HighlightCell(end, "red")
    END PROCEDURE
    PROCEDURE AddWaypoint()
        position = INPUT "Enter position for waypoint: "
        IF position = start OR position = end THEN
            OUTPUT "Cannot add start or end positions as waypoints"
            RETURN
        END IF
        IF IsValidWaypoint(position) THEN
            waypointsList.APPEND(position)
            HighlightCell(position, "blue")
        ELSE
            OUTPUT "Invalid waypoint position"
        END IF
    END PROCEDURE
    FUNCTION IsValidWaypoint(position) RETURNS BOOLEAN
        RETURN position > start AND position < end AND position <= (width * length)
    END FUNCTION
END CLASS
```

Analysis

- **Purpose and Functionality:**
  The `WarehouseGUI` component provides a visual interface for the warehouse management system, allowing users to interact with the warehouse grid, add waypoints, find paths, and manage inventory. This component bridges the gap between the user and the underlying database and pathfinding algorithms.

- **Class Attributes:**
  - `width` and `length`: Define the dimensions of the warehouse grid for visualization. These attributes ensure the GUI accurately represents the physical warehouse layout.
  - `canvas`: Manages the graphical representation of the warehouse grid. This attribute centralizes all drawing operations for consistency.
  - `waypointsList`: Stores the user-defined waypoints for pathfinding. This list maintains the sequence of positions for the robot to visit.
  - `start` and `end`: Represent fixed entry and exit points. These positions are immutable to ensure consistency across different components of the system.

- **Key Procedures:**
  1. `InitializeGUI`: Sets up the visual components of the interface, including the canvas and control buttons. This procedure establishes the foundation for user interaction with the system.
  2. `DrawGrid`: Creates the visual grid representation of the warehouse. This function translates the logical warehouse structure into a visual format that users can easily understand.
  3. `AddWaypoint`: Allows users to add waypoints for pathfinding, with validation to ensure they are within valid bounds. This procedure includes safeguards against selecting fixed positions as waypoints.
  4. `IsValidWaypoint`: Validates that a position can be used as a waypoint by checking it against logical constraints. This function ensures that all waypoint operations respect system integrity.
  5. `HighlightCell`: Provides visual feedback by highlighting specific cells on the grid. This function enhances user experience by clearly indicating selected positions and statuses.

- **Design Decisions:**
  - **Canvas Sizing:** The canvas size is proportional to the warehouse dimensions, ensuring a consistent visual representation regardless of warehouse size.
  - **Color Coding:** Different colors are used to distinguish between start positions, end positions, and waypoints, improving visual clarity.
  - **Button-Based Interface:** A simple button-based interface was chosen for its familiarity and ease of use, reducing the learning curve for users.
  - **Waypoint Validation:** Constraints ensure that only valid positions can be added as waypoints, preventing invalid path calculations.
- **Justifications:**
  - **Encapsulation:** Keeps GUI-related logic within the `WarehouseGUI` class, simplifying integration with other components.
  - **Real-Time Feedback:** Provides immediate visual feedback on user actions, enhancing the user experience and reducing errors.
  - **Error Handling:** Clear error messages guide users when invalid operations are attempted, improving system usability.

Appendix B

Complete final code in Python