UNIVERSITY OF HERTFORDSHIRE

Department of Computer Science

Modular BSc Honours in Computer Science

6COM2018 - Computer Science Project

Final Report

March 2024

TITLE OF PROJECT

Visualising Pathfinding Algorithms

Author's initials and surname

D.Karimi

Student ID

19034243

Supervised by: Supervisors Name

Gani Nashi

Abstract

This report presents the development and evaluation of a pathfinding visualisation tool aimed at enhancing understanding and engagement with various pathfinding. The project objective encompassed the design, implementation and testing of an interactive tool capable of showcasing Dijkstra’s algorithm, A* algorithm, and Depth-First search algorithm on a 2-d grid-based world. Using Python and the Pygame library, the tool offers users the ability to observe algorithm execution in real-time. Through testing and analysis, the project identified key insights into the performance and behaviour of these algorithms. The success of this project underscores the significance of algorithm optimisation in real-world application, offering a platform for further research and development in computation problem solving.

Acknowledgements

I would like to extend my sincere appreciation to my supervisor and mentor, Gani Nashi, for his invaluable support and guidance through this project. His expertise, insight, and encouragement were essential in shaping the direction of my research and overcoming challenges along the way. I am thankful for his dedication, patience, which greatly contributed to the completion of this project.

Table of Contents

List of figures

List of tables

Introduction

1.1 Overview

This project presents the pathfinding visualising tool, outlining its aims, motivations, objectives, and structure. The project aims to develop an interactive tool for visualising various pathfinding algorithms on a grid, with the intention of enhancing the understanding and engagement among users. Drawing from personal experience and educational insights, the project seeks to bridge the gap between theoretical concepts and practical applications in path finding algorithm solving. Through clear objectives and user-centred approach, the visualiser aims to provide a valuable resource for students and enthusiasts seeking to explore and comprehend pathfinding effectively.

1.2 Project Aim

The primary goal of this project is to develop a tool that showcases how various algorithms work on finding the shortest path on a grid. Through interactive visualisation, the aim is to provide users with a practical understanding of different pathfinding algorithms and how they are used in real life.

1.3 Motivation

The motivation behind this project stems from the need to make the learning process of pathfinding algorithms more accessible and engaging. Traditional ways of learning often do not have interactive tools or pictures, which can make it harder to understand. By developing an interactive aim to bridge this gap, helping users actively learn and comprehending pathfinding algorithms.

1.4 Project goals

To ensure a successful development of the pathfinding visualisation tool, specific objectives have been outlined to provide a clear guidance and direction for the project.

- Develop an interactive visualisation tool: design and implement an interactive visualisation tool capable of depicting the functionality of multiple pathfinding algorithms in real time providing users with an engaging learning experience.
- Enhance understanding of pathfinding algorithms: to promote a deeper understanding of pathfinding algorithms, including Dijkstra’s algorithm, A*, and DFS, by enabling users to observe their execution and effects on pathfinding outcomes with the visualised grid environment.
- Bridge theoretical concepts with practical applications: bridge the gap between theoretical concepts and practical applications in pathfinding algorithm solving by illustrating how algorithmic principles translate into effective solutions for real-world navigation problems.
- Foster user engagement and learning resource: foster user engagement and learning by incorporating clear objectives, intuitive user interfaces, and interactive features that encourage exploration and experimentation with different algorithms, ultimately promoting a self-directed and enriching learning experience.
- Provide a valuable educational resource: provide a valuable educational resource for students, educators, and enthusiasts interested in algorithmic problem-solving, offering a platform for learning, experimentation, and skill development in the field of pathfinding algorithms.

1.5 Objectives

The primary objective of this project is to develop a pathfinding visualisation tool using python and pygame library. The tool aims to implement and visualise various pathfinding algorithms, including Dijkstra's algorithm, A* algorithm, and Depth-First search (DFS) algorithm, allowing users to interactively observe the process of finding the shortest path between two points on a grid-based environment.

Initial research and planning (October 2023 – November 2023)
Conduct thorough research on pathfinding algorithms, visualisation techniques, and user interface design principles. Gather relevant literature resource, and existing implementation to inform the project's direction and requirements.

Design (December 2023 – January 2024)
Develop a detail design plan for the visualisation tool, outlining its structure, components, and interactions. Define the user interface layout, colour scheme, visual elements to ensure clarity and usability.

Prototype development (January 2024)
Build a prototype of the visualisation tool, focusing on implementing core features and functionalities. Develop the grid rendering system, algorithmic logic, and basic user interaction capabilities to demonstrate the tool's potential.

Algorithm Implementation (January 2024 – February 2024)
Implement Dijkstra's algorithm, A*, and Depth-First search algorithm within the visualisation tool. Ensure accurate execution of each algorithm and proper visualisation of their outcomes on the grid environment.

Documentation and finalisation (March 2024)
Prepare a documentation covering the tool's features, functionalities, usage instructions, and technical specifications. Finalise the project deliverables, including code, documentation, and materials, for submission.

1.6 Report Structure

The report is structured into six main chapters, each with a specific focus on contributing to a comprehensive understanding of the project.

The first chapter is the introduction which serves as an entry point to the project, giving an overview of its objectives, motivations and aims. Additionally, it will structure the report.

The second chapter is the literature review, which provides a concise overview of the historical context and current state of knowledge relevant to the project. The objective of this chapter is to describe the topics, analyse various approaches, and highlight their strength and weaknesses.

The third chapter is project work and methodology will be organised by time, task, and processes. This chapter describes what was done during the project, including what worked well and what did not work well, and any other methods which were considered. It will explain why certain choices were made, backed by research and results.

The fourth chapter is testing, which will focus on the algorithm visualiser tool. This section will show the testing procedures, datasets used, and environments simulated to assess the tool's performance and accuracy.

The fifth chapter is results and evaluation, where the findings from testing are analysed and discussed in depths. This section will evaluate strength and weaknesses of the algorithm visualiser tool.

The sixth chapter will be the conclusion which will summarise the main findings of the report and highlights key findings for future research and development.

The seventh and final chapter is project management review section provides an evaluation of the project management comparing Gantt charts and discussing modifications that were made. It will examine aspects such as time, and workload management, quality of work, and meeting schedules. This segment aims to identify strengths and weaknesses in project management and recommendations for improvements for future projects.

Literature Review

2.1 Introduction to pathfinding algorithms

The primary aim is to identify the most suitable algorithm for implementation within the project's scope. Extensive analysis of past studies will be conducted to discern key attributes and performance measures associated with prominent pathfinding algorithms, including Dijkstra's, A*, and Depth-First search. To review will evaluate the strengths and limitations of each algorithm to determine the optimal choice that alights with the project's objective and requirements.

Pathfinding algorithms play a critical role in a wide range of fields, including computer science, robots, and even gaming, by providing efficient solutions to navigation and routing problems. Pathfinding algorithms are all about finding the best route to get from point A to point B, with or without obstacles in their way. In computer science these algorithms are fundamental components of figuring out the best way to plan routes on a network, and help computers make smart decisions (Shen et al. 2022). In robotics, pathfinding algorithms allow autonomous vehicles (Luo et al. 2023) or drones (Chen et al. 2022) to navigate through environment's safely and efficiently. Lastly in gaming pathfinding algorithms allow character movements, enemy AI behaviour and movements and helps with level design (Pathfinding Algorithms in Game Development. [2021]). Overall pathfinding algorithms are crucial tools for solving navigation problems in different areas, helping drive technology forward with fresh ideas and enhancement.

2.2 History and background

Pathfinding algorithms have been evolving for many decades, thanks to ideas from computer science, maths, and engineering (Pan and Ching Pun-Cheng 2020). It all started with the earliest attempts to solve routing problems manually, where individuals planned simple strategies to navigate through physical landscapes (Schrijver 2012). However, pathfinding algorithms became more official when computers came around and needed to automate planning routes for different things.

One of the pivotal works in pathfinding algorithms is Dijkstra's algorithm, proposed by Dutch computer scientist Edsger W. Dijkstra in 1956 (*Edsger Wybe Dijkstra: His Life, Work, and Legacy - Krzysztof R. Apt, Tony Hoare - Google Books* [2022]). Dijkstra's algorithm aimed to find the shortest path between two nodes in a graph by iteratively expanding the search from the initial node to all reachable nodes, updating the distance along the way. Dijkstra's algorithm laid the groundwork for following pathfinding algorithms and remains a corner stone in the field.

Another significant advancement in the history of pathfinding algorithms is the development of the A* algorithm introduced by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968. A* algorithm is built upon Dijkstra's algorithm but incorporates a heuristic function to guide the search more efficiently towards the goal node. By combining the benefits of Greedy search strategies and breadth-first. A* became wildly used in various applications due to its effectiveness in finding optimal paths while minimising computational resources (Barnouti et al. 2016).

Alongside Dijkstra and A*, the Depth-First Search (DFS) algorithm has played an important role in pathfinding. Depth-First Search, dating back to the 19th century, is a classic graph traversal algorithm that explores as far as possible along each branch before backtracking. While Depth-First Search is not typically used for finding the shortest paths due to it not being optimal, it remains relevant in certain scenarios such as maze generation and topological sorting (View of A COMPREHENSIVE AND COMPARATIVE STUDY OF DFS, BFS, AND A* SEARCH ALGORITHMS IN A SOLVING THE MAZE TRANSVERSAL PROBLEM 2022).

Over time, many improvements have been made to these basic algorithms, resulting in new methods like mixing different techniques, running tasks at the same time, and spread them out over multiple systems for finding paths. Nowadays Dijkstra, A*, and Depth-First Search algorithms are still important in planning routes for GPS systems (Oh and Ashuri 2023), robots, games, and more, playing a huge role in how we research and use pathfinding today.

2.3 Overview of pathfinding algorithms

The A* algorithm is a renowned pathfinding algorithm known for its efficiency and optimality in finding the shortest path between two points on a graph or grid. It integrates the strengths of Dijkstra's algorithm and Greedy best-first search by employing heuristics to guide its search towards the goal node while considering the cost of reaching each node from the start node.

Key to the A* algorithm is using a heuristic function to estimate the cost of reaching the goal node from any given node in the graph. This heuristic, often symbolised as h(n), provides an optimistic estimate of the remaining cost to reach the goal node, allowing A* algorithm to prioritise nodes that are closer to the goal. A* maintains two lists which are open and closed. It starts with the initial node and continuously evaluates adjacent nodes, considering both the actual cost from the start node g(n), and the heuristic cost to the goal h(n). This algorithm selects nodes with the lowest combine cost $f(n) = g(n) + h(n)$ from the open list for expansions until it reaches the goal or all possible paths (Noto and Sato 2000). The A* algorithm finds applications in various segments, including robots, gaming, logistics, and route planning systems. In robotics, A* is used for path planning of mobile robots (Kumar Guruji et al. 2016) to navigate through dynamic environments efficiently (Li et al. 2023). In gaming, it is used to create realistic NPC movement and to find the best paths for player characters (Cui and Shi 2011). Logistics companies use A* algorithm for vehicle routing and scheduling to optimise delivery routes and minimise transportation costs (Fu [2001]). Numerous research papers and case studies have highlighted the effectiveness, and efficiency of the A* algorithm in solving pathfinding problems. An example is a study Russel and Norvig (Ponce et al. 2021) demonstrates how A* outperforms other search algorithms in terms of computational efficiency and path optimality. Additionally, utilisation of A* in real-world applications such as Google Maps and GPS navigation systems (Mehta et al. 2019) underscore its real-world importance and extensive use.

In summary, the A* algorithm serves as a key tool in pathfinding, offering a strong and flexible approach to solving navigation problems efficiently and optimally. Its theoretical foundations, along with its many uses and proven success, this shows how important A* algorithm is in computer science and related fields.

Dijkstra's algorithm is a classic pathfinding algorithm is known for its effectiveness in finding the shortest path between nodes in a graph. Unlike A*, which includes heuristic guidance, Dijkstra's algorithm explores all possible paths from the start node to all other nodes, updating the shortest path but may involve redundant exploration in some scenarios due to its lack of heuristic guidance.

At its core, Dijkstra's algorithm maintains a priority queue of nodes, initially assigning infinite distance to all nodes except the start node, which is assigned a distance of zero. It iteratively selects the node with the smallest distance from the priority queue and updates its neighbouring nodes' distance if the shorter path is found. This process continue until the priority queue is empty, making sure the shortest path to each node is found (Rahayuda and Santiari 2021). Unlike A* algorithm, Dijkstra's algorithm does not utilise heuristic functions to guide the search process. Instead, it relies solely on the actual distance travelled from the start node to evaluate paths, making it less efficient in certain scenarios where heuristic guidance could speed up the search (Wayahdi et al. 2021). However, Dijkstra's algorithm guarantees the discovery of the shortest path in a non-negative-weighted graphs, making it reliable for finding the optimal routes. Many success stories demonstrate the practical significance of Dijkstra's algorithm in solving routing and navigation problems. A big example is its implementation in GPS navigation devices enables users to find the shortest route between locations accurately. Additionally, network routing protocols rely on Dijkstra's algorithm to determine the most efficient paths for data transmission in computer networks, ensuring optimal network performance (Nasiboglu 2022).

In summary, Dijkstra's algorithm remains a fundamental method in pathfinding, providing a reliable way to find the shortest path in graphs. While it lacks the heuristic guidance of A*, its predictable behaviour and guaranteed optimality make it indispensable in various real-world applications requiring route optimisation and pathfinding.

The Depth-First Search (DFS) algorithm provides an alternative approach to pathfinding, notable for its simplicity and versatility. Unlike A* and Dijkstra, Depth-First search does not prioritise finding the shortest path but rather focuses on exploring as far as possible along each branch before backtracking. This characteristic makes Depth-First search well suited for scenarios where the primary objective is to exhaustively search through all possible paths or to traverse entire graphs without necessarily finding the shortest route (Papamanthou 2024).

Depth-First search operates by recursively exploring each possible path until it reaches a dead end, at which point it backtracks to the most recent unexplored branch and continues its exploration. This process continues until all paths have been explored or until the desired node or condition is reached (Zhang and Korf 1993). While Depth-First search does not guarantee optimality in finding the shortest path, it is often preferred in scenarios where memory usage is a concern or when exploring all possible solutions is valuable, such as in maze-solving or cycle detection in graphs.

In exploration, Depth-First search plays a crucial role in finding connected. Parts detecting cycles and performing topological sorting. Its ability to systematically explore all reachable nodes from a given starting point makes it an essential tool in various graph related applications. However, Depth-First search may encounter issues such as infinite loops in graphs with cycles or suboptimal paths in scenarios where the shortest path is desired (Bonet and Geffner [2006]). Despite these limitations, Depth-First search remains a fundamental algorithm in graph theory and serves as a building block for more complex pathfinding algorithms.

2.4 Comparison of pathfinding algorithms

In selecting the most suitable pathfinding algorithms for implementation in this project, several key criteria were considered to ensure alignment with the projects aims, objectives, and requirements. Firstly, the algorithm’s ability to effectively demonstrate pathfinding concepts and principles in an educational context was crucial, as the project aims to develop an educational tool for teaching these algorithms. Additionally, the scalability of the algorithm was a determining factor, as the system is expected to handle the consistent grid size and varying obstacle configurations without compromising performance, providing a seamless learning experience for users. Furthermore, the algorithms adaptability to different types of scenarios and obstacles, such as various obstacle configurations and complexities in the graph structure, played a significant role in the decision-making process. By carefully considering these factors and aligning them with the project’s objectives, the selection of the A*, Dijkstra’s, and Depth-First search algorithms was made to ensure the development of an effective and comprehensive educational tool for teaching pathfinding algorithms.

A comparative analysis of the DFS, Dijkstra and A* algorithms reveal clear strengths, weaknesses, and compromises that influence their suitability for various applications.

A* stands out for its efficiency in finding the shortest path while considering heuristic information to guide its search, making it particularly effective in scenarios where optimality is crucial. However, A* may suffer from increased memory usage and computational overhead especially when dealing with large graphs or complex heuristic functions. Its reliance on heuristics also means that the quality of the solution heavily depends on the accuracy of the heuristic function. On the other hand, Dijkstra’s algorithm guarantees optimality by systematically exploring all possible paths from the start node to all other nodes. Its simplicity and effectiveness in finding the shortest path make it a popular choice for scenarios where finding the optimal solution is crucial. However, Dijkstra’s algorithm may become computationally expensive in dense graphs or graphs with negative edge weights, as it explores all reachable nodes without considering any heuristics. Lastly Depth-First search offers simplicity and versatility, performing in scenarios where exhaustively exploring all possible paths or traversing entire graphs is the primary objective. Its memory efficiency and straightforward implementation make it suitable for tasks such as cycle detection and graph traversal. However, Depth-First search does not guarantee optimality and may encounter issues such as infinite loop in graphs with cycles or suboptimal paths in scenarios where the shortest path is preferred (Iloh 2022).

Practical studies and performances analyses provide valuable insights into the performance and efficiency of these algorithms in different scenarios. These analyses often compare factors such as runtime, memory usage, and solution quality across various scenarios, helping practitioners select the most appropriate algorithm based on the specific requirements of their application. Overall, understanding the strengths, and weaknesses, and compromises of A*, Dijkstra’s, and Depth-First search algorithms is essential for informed decision making in pathfinding and graph traversal tasks.

The choice of A*, Dijkstra’s, and Depth-First search, algorithms for this project is based on their distinct characteristics and practical relevance in the field of pathfinding. Each algorithm offers unique strengths and weaknesses that make them suitable for different scenarios and applications.

In the evaluation of pathfinding algorithms within a grid world setting, it’s crucial to ensure a fair comparison. To achieve this, each algorithm will be assessed within a grid world where each cell represents an equal distance. This approach offers several advantages.

Firstly, utilising a grid world with unform cell distances ensures that each algorithm operates within the same spatial limitations. This uniformity eliminates potential biases that may arise from variations in cell sizes or distances, providing a playing field for evaluation. Moreover, by maintaining consistent conditions across all evaluations, we can accurately compare the performance of A*, Dijkstra’s, and Depth-First search algorithms. Standardised conditions help to eliminate confounding variables, allowing for a more reliable assessment of each algorithm’s efficacy. In a grid world with unform cell distances, the objective of finding the shortest path between two points remains consistent. This consistency enables an objective evaluation of each algorithm’s efficiency and optimality, facilitating meaningful comparisons of their performances. Lastly, the simplicity of the grid world with unform cell distances simplifies the comparison process. With equal spatial intervals throughout the grid, it becomes easier to interpret algorithmic behaviours and outcomes, enhancing the understanding of the evaluation process (Theses et al. 2019).

A* algorithm is known for its efficiency in navigating the shortest path by using heuristic information to guide its search. In a grid world with uniform distance per cell, A* explores neighbouring cells, prioritising those closer to the goal based on heuristic estimates. This typically results in quicker arrival at the goal compared to Dijkstra's and Depth-First search algorithms. A* algorithm guarantees optimality when using admissible heuristics. In a grid world, A* is likely to efficiently find the shortest path due to its heuristic guided search strategy. A* algorithm demonstrates a balanced exploration behaviour, prioritising paths nearer to the goal while considering the actual cost from the start node. In a grid world, A* efficiently navigates towards the goal while minimising unnecessary exploration.

Dijkstra's algorithm systematically explores all possible paths from start node to all others, adjusting the shortest path accordingly. In a grid world, it may display slower performance than A* due to its lack of heuristic guidance. However, in obstacle free scenarios or where the heuristic information us unreliable, Dijkstra's algorithm may offer comparable performance. Dijkstra's algorithm ensures optimality in finding the shortest path in graphs with non-negative edge weights. In a grid world, it will consistently find the shortest path potentially at the cost of increased computational resources compared to A*. Dijkstra's algorithm explores all paths uniformly without considering their proximity to the goal. In a grid world, it may explore a larger portion of the grid before converging to the shortest path, especially in the presence of obstacles (Afshani 2015).

Depth-First search algorithm traverses' paths in ta depth-first manner, often delving into lengthy paths before retracing its steps. In a grid world, Depth-First search may show inefficient behaviour as it explores paths regardless of their lengths or proximity to the goal. This can result in suboptimal paths in potentially longer search times compared to A* and Dijkstra's algorithm.

By conducting the comparisons within the same grid world environment characterised by unform cell distances, we can effectively assess the relative performance and suitability of A*, Dijkstra's, and Depth-First search algorithms for pathfinding tasks. This approach ensures a robust evaluation methodology, allowing for informed decisions regarding algorithms selection in various applications.

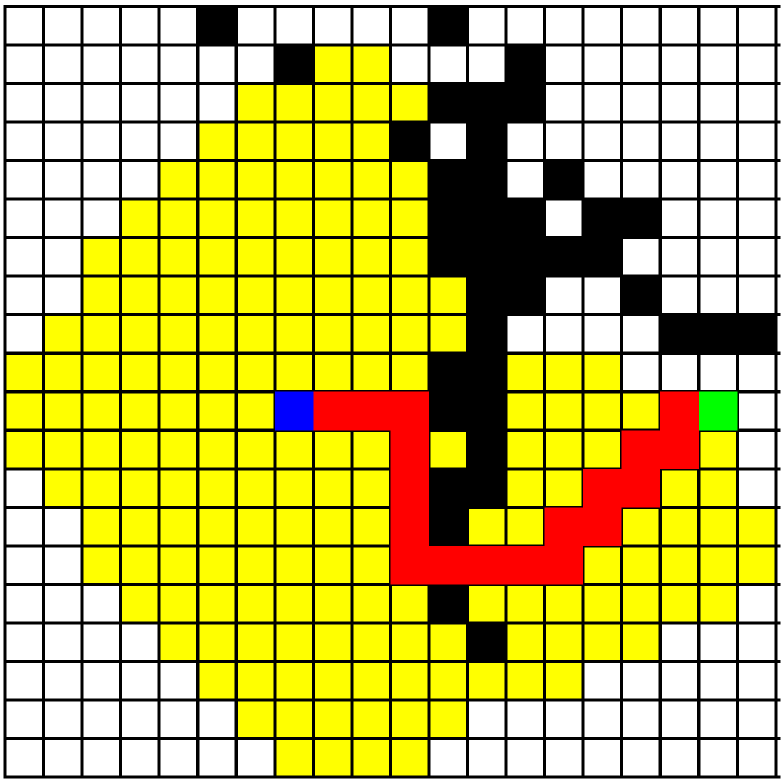


Figure 1: visualisation of A algorithm in a grid world.*

The provide figure shows a visual representation of the A* algorithm in action within a grid world environment. This environment is structured with each cell serving as a unique location, forming the foundation for the pathfinding problem. Within this grid, the start and end points are prominently marked providing clear indication of the initial and final positions between which the A* algorithm aims to discover the optimal path. Additionally, obstacles within the grid are visibly marked, represented by coloured cells. These obstacles present challenges for the pathfinding algorithm to navigate around, highlighting the complexity of real-world scenarios where paths may be obstructed by physical barriers or obstacles.

The figure effectively illustrates the step-by-step process of the A* algorithm as it traverses the grid world, searching for the shortest path from the start to the end point. Each iteration of the algorithm, including node evaluation and path selection is visually presented, providing understanding into the algorithm's decision making. Moreover, the outcome of the A* algorithm's execution is illustrated through the visualisation of the optimal path from start to end point. This path, distinguished by a distinct colour, represents the most efficient route identified by the algorithm while considering the layout of the grid and obstacles along the way.

By observing the figure, it becomes apparent how the A* algorithm adeptly manoeuvres through the grid world, avoiding obstacles and selecting the shortest path to reach the destination. This demonstration effectively showcases the effective and efficiency of the A* algorithm in solving pathfinding problems within the grid world, highlighting its practical applicability in various real-world scenarios (Afshani 2015).

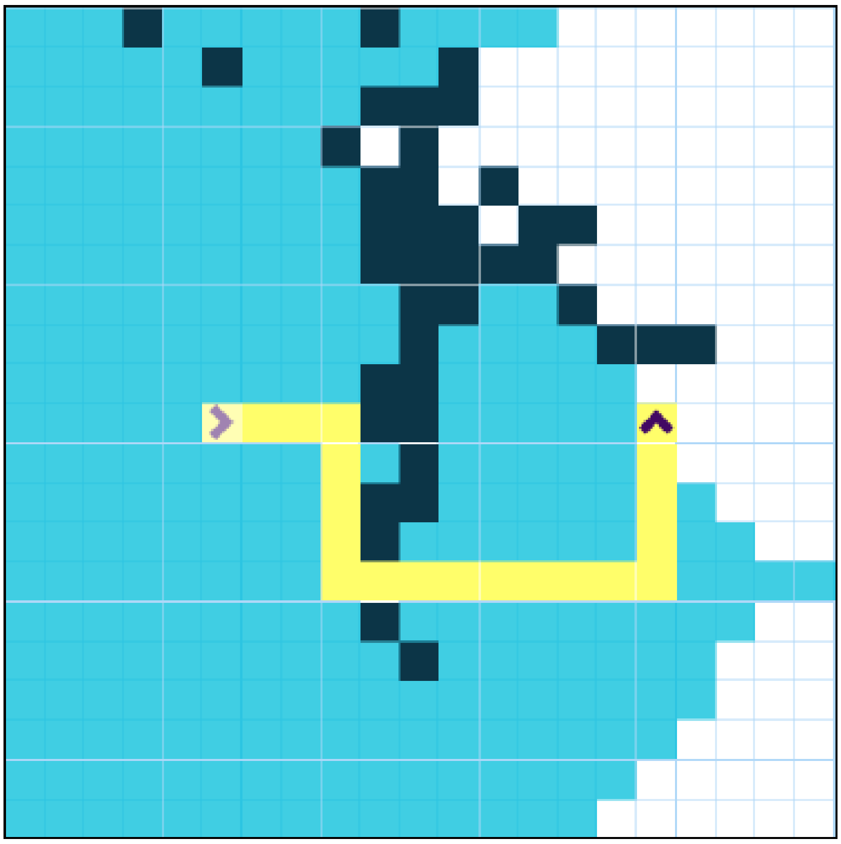


Figure 2: visualisation of Dijkstra in a grid world (Sharma et al. [2023]).

The provided figure showcases Dijkstra's algorithm operating within a grid world environment which resembles the previous one. In the showcased grid world environment, Dijkstra algorithm efficiently operates from the designed start point to the specified end point while considering the presence of obstacles. Dijkstra's algorithm systematically explores the grid, evaluating the shortest path from the start point to all other reachable nodes, despite not using heuristics guidance like the A* algorithm. Overall, the depiction of Dijkstra's algorithm in the grid world environment highlights its reliability and effectiveness in solving pathfinding problems, particularly in scenarios where exhaustive exploration of all possible paths is essential.

2.5 Practises of pathfinding algorithms

Studying past research shows different ways to make A*, Dijkstra, and Depth-First search algorithms better. It also suggests new ideas for improving these algorithms or using them in different ways.

For A* algorithm usage, using accurate heuristics is seen as a top approach to estimate the remaining cost to reach the goal node. Researchers stress the need to pick up heuristic's functions suited to the problem domain to balance complexity and solution quality. Additionally, optimising by removing unnecessary nodes and using data structures like priority queues can greatly enhance A*'s runtime performance, especially in scenarios with large graphs or complex search spaces (Sigurdson et al. [2019]). In the case of Dijkstra's algorithm, best practices include employing data structures like priority queues or Fibonacci heaps to efficiently extract the node with the minimum distance during each iteration (Xu et al. [2007]). Moreover, researchers highlight the significance of implementing optimisation such as early termination or bidirectional search techniques to reduce the algorithm's computational overhead (Sedeño-Noda and Colebrook 2019), especially in scenarios with widely spread-out graphs or non-negative edge weights. Additionally, exploring parallel and distributed computing paradigms presents promising avenues for accelerating Dijkstra's algorithm on modern computing architectures. For Depth-First search algorithm implementation, optimising memory usage and preventing infinite loops are key considerations. Techniques such as cycle detection and backtracking can help mitigate the risk of encountering infinite loops, ensuring termination in finite graphs. Moreover, researchers advocate for utilising parallelisation and concurrency to exploit modern multi-core processors effectively (Acar et al. [2015]), enhancing Depth-First search scalability and performance in large-scale graph traversal tasks. Furthermore, exploring the potential of machine learning and artificial intelligence techniques to improve the performance and adaptability of pathfinding algorithms presents promising research opportunities for future studies (Graham Hugh McCabe Stephen Sheridan 2003). By utilising advanced computational methods and collaborative approaches, researchers can continue to expand the horizons of pathfinding algorithms and discover new possibilities for applications in various domains.

Overall, the examination of existing research highlights the significance of adopting best practices for implementing and optimising A*, Dijkstra, and Depth-First search algorithms, while also pointing out potential areas for further improvement in their efficiency and effectiveness.

2.6 Conclusion

In conclusion. The comparative analysis of the Depth-First search, Dijkstra, and A* algorithms reveal their distinct characteristics, strengths, weaknesses, and applicability in various scenarios. Each algorithm offers unique advantages and trade-offs, which must be carefully considered on the specific requirements of the application.

A* algorithm demonstrates efficiency in finding optimal paths, especially in scenarios where heuristics guidance can expedite the search process. Its ability to balance between actual cost and heuristic estimates makes it suitable for dynamic environments with changing road conditions and varying terrain.

Dijkstra’s algorithm guarantees optimality by systematically exploring all possible paths, making it reliable for scenarios with non-negative edge weights. While it may exhibit slower performance compared to A* in certain scenarios, its simplicity and effectiveness in finding the shortest path make it a valuable choice for route planning applications.

Depth-First search algorithm offers simplicity and versatility, making it suitable for exhaustive exploration of all possible paths. However, its lack of optimality and potential for suboptimal paths make it less favourable for scenarios where finding the shortest path is crucial.

Based on the findings and comparative analysis, it is recommended to select the A*, Dijkstra’s, and Depth-First search algorithms for further evaluation. Each algorithm will undergo testing within a grid world environment to assess its performance in navigation and route planning tasks.

Project work and methodology

3.1 Introduction to project work

The methodology employed in developing the pathfinding visualisation tool is introduced in this section, emphasising the rationale behind its selection. This choice of methodology plays a crucial role in shaping the workflow of the entire project, from its beginning to its conclusion.

The agile methodology was chosen as the guiding framework for this project. Agile methodology is characterised by its iterative and incremental approach, emphasising flexibility. Collaboration, and responsiveness to change throughout the development process.

The decision to use agile methodology was guided by the requirements and objectives of the project. Agile’s iterative nature aligns well with the dynamic nature of software development, allowing for continuous feedback and adaptation. This approach enables us to respond promptly to emerging challenges and refine the approach iteratively. Agile methodology provides a structured framework to manage projects efficiently, prioritise tasks, and maintain a clear focus on project objectives. The iterative nature of agile allows for continuous improvement and adjustment based on evolving needs and insights gained throughout the development process.

In summary, Agile methodology was chosen because it alights with the project's need. It emphasises flexibility and responsiveness which is crucial for its success. After consideration, agile emerged as the best approach to guide the project effectively.

The project involves designing, implementing, testing, and evaluation the pathfinding visualisation tool. It includes creating an interactive graphical interface where users can set obstacles, define start, and end points, select different pathfinding algorithms, and observe the algorithm's execution in real-time. Additionally, the project entails developing algorithms for pathfinding, ensuring correctness, efficiency, and optimality in finding the shortest path.

In computer science and technology, pathfinding algorithms serve as foundational problem-solving tools. They enable efficient route planning and navigation across various scenarios, supporting critical applications such as network routing and robotic independence. While the theoretical principles of these algorithms are well-established, practical implementation and visualisation play a crucial role in connecting theory with real-world applications.

The project followed a structured methodology covering several key phases: design, planning, testing, and evaluation. Beginning with the planning phase, careful consideration was given to designing the user interface, algorithmic implementations, and overall architecture of the pathfinding visualisation tool. Subsequently the implementation phase involved translating these plans into functional code, leveraging python and the pygame library alongside other essential technologies, comprehensive testing procedures were then conducted to ensure the accuracy, efficiency, and reliability of the implemented algorithms and graphical interface. Finally, the evaluation phase involved assessing the tools performance, usability, and effectiveness in achieving its objectives, with iterative improvements made based on feedback and testing results.

3.2 Design

Delving into the design phase of the project, focusing on planning, and conceptualising the pathfinding algorithm visualisation tool. Primarily centred around the creation of flowcharts to visualise the pathfinding algorithm. The flowcharts serve as blueprints, illustrating step by step the process of each algorithm. By mapping out the algorithm's functionalities and interactions, the design phase aims to provide a clear and comprehensive understanding of the algorithms approach.

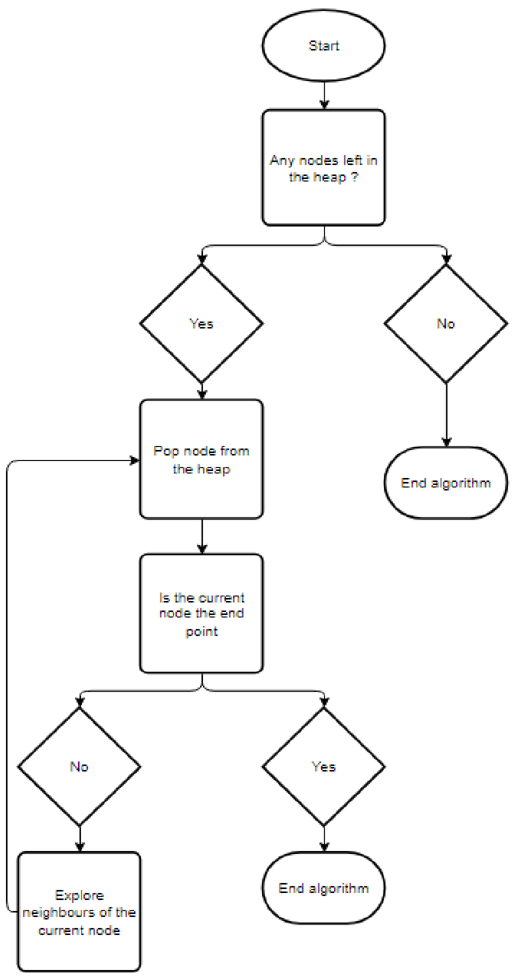

Start
Any nodes left in the heap ?
Yes
No
Pop node from the heap
End algorithm
Is the current node the end point
No
Yes
Explore neighbours of the current node
End algorithm

*Figure 3:Flowchart for Dijkstra and A**

The flowchart provides a representation of both Dijkstra's algorithm and A* algorithm, illustrating their shared approach to pathfinding. The algorithm begins with an initial check to determine if there are any nodes remaining in the priority queue or heap, representing unexplored nodes in the graph. If nodes are present, the algorithm proceeds by selecting the node with the shortest distance or the lowest combined cost, including heuristic for A* from the priority queue. It then evaluates whether the current node is the destination node, terminating the algorithm if the destination is reach. If the current node is not the destination, the algorithm explores its neighbouring nodes, updating their distance and previous nodes if a shorter path is found. This process continues iteratively until the destination node is reached or there are no more nodes left to explore, while both Dijkstra's and A* algorithm aim to find the shortest path, A* incorporates heuristic information to guide its search, often resulting in more efficient solutions, especially in grid-based environments. Conversely, Dijkstra's algorithm guarantees optimality regardless of heuristic information, making it suitable for scenarios prioritising optimal solution over computational efficiency. The flowchart provides a clear visualisation of the iterative nature of both

Dijkstra's and A* algorithm, highlighting their fundamental steps in finding the shortest path within a graph.

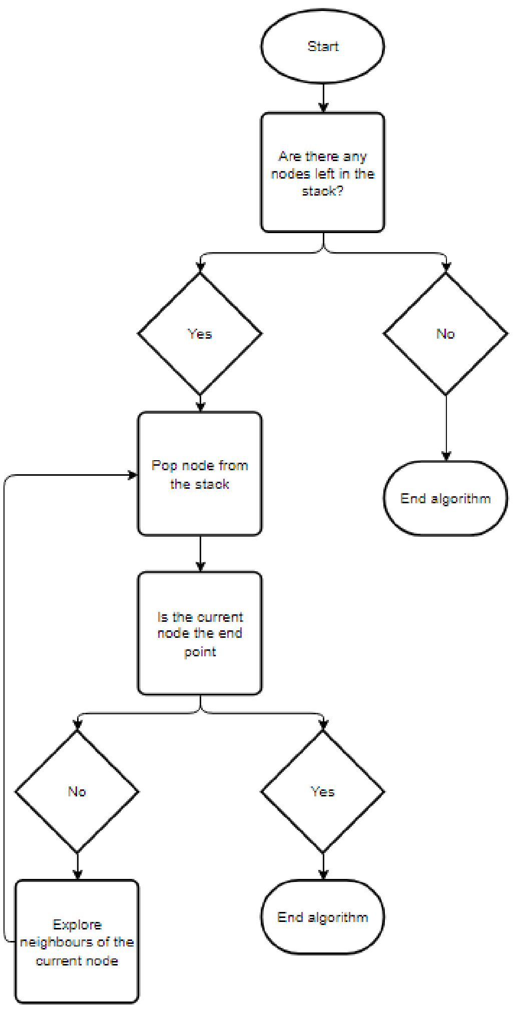

Start
Are there any nodes left in the stack?
Yes
No
Pop node from the stack
End algorithm
Is the current node the end point
No
Yes
Explore neighbours of the current node
End algorithm

Figure 4: Flowchart for DFS

The depicted flowchart illustrates the process of the Depth-First search algorithm, a traversal method used to explore all possible paths from the starting node to the end node. Initiated by checking for remaining nodes in the stack, the algorithm progresses by popping the last added node for exploration. It then evaluates whether the current node corresponds to the end point, branching out accordingly to explore neighbouring nodes or terminate the algorithm's execution upon reaching the end point. This sequential process continues until all possible paths have been explored or the algorithm reaches the end point, facilitating comprehensive traversal of the graph structures.

3.3 Implementation

For the graphical interface design, the design process for the pathfinding algorithm visualising tool involved creating an intuitive graphical interface that allows user to interact with the grid-based environment effortlessly. The layout includes components for setting obstacles, defining start and end points, selecting pathfinding algorithms, and observing the algorithms execution in real-time. The user interface was designed to be visually appealing and user friendly, utilising familiar controls and user-friendly interactions to enhance usability.

As for the interaction, users can interact with the graphical interface through mouse clicks and keyboard inputs to set obstacles, define start and end point, and select pathfinding algorithms. Mouse clicks are used to toggle between setting obstacles, defining start and end points, and removing obstacles. Keyboard inputs allow users to select different pathfinding algorithms and initiate the pathfinding process. Real-time visual feedback is provided to users, allowing them to observe the algorithms execution and the resulting path dynamically.

In the pathfinding algorithm implementation, the implementation of each pathfinding algorithm like Dijkstra's, A*, and Depth-First search, and involved translating their respective algorithms into code to achieve the desired functionality within the grid-based environment. Each algorithm was implemented as a separate function, utilising appropriate data structures and algorithmic techniques to navigate the grid, evaluate nodes, and identify the shortest path from the start to the end point.

During the implementation phase, there was various challenges like handling complex algorithms, managing data structure, and ensuring smooth graphical performances. To tackle these issues, there was a focus on optimising algorithms for better efficiency and reducing computational load. Debugging tools were utilised to identify and resolve coding errors, while collaboration with peers facilitated problem solving. These challenges contributed to enhancement of the tool's performance through incremental adjustment.

3.4 Choice of Python libraries

Python was chosen as the programming language for this project due to its simplicity, readability, and versatility. Pythons' clear syntax and extensive library make it an ideal choice for rapid development and prototyping, especially in the context of this visualisation project. Additionally, python's popularity ensures robust community support and a wealth of resources for troubleshooting and learning.

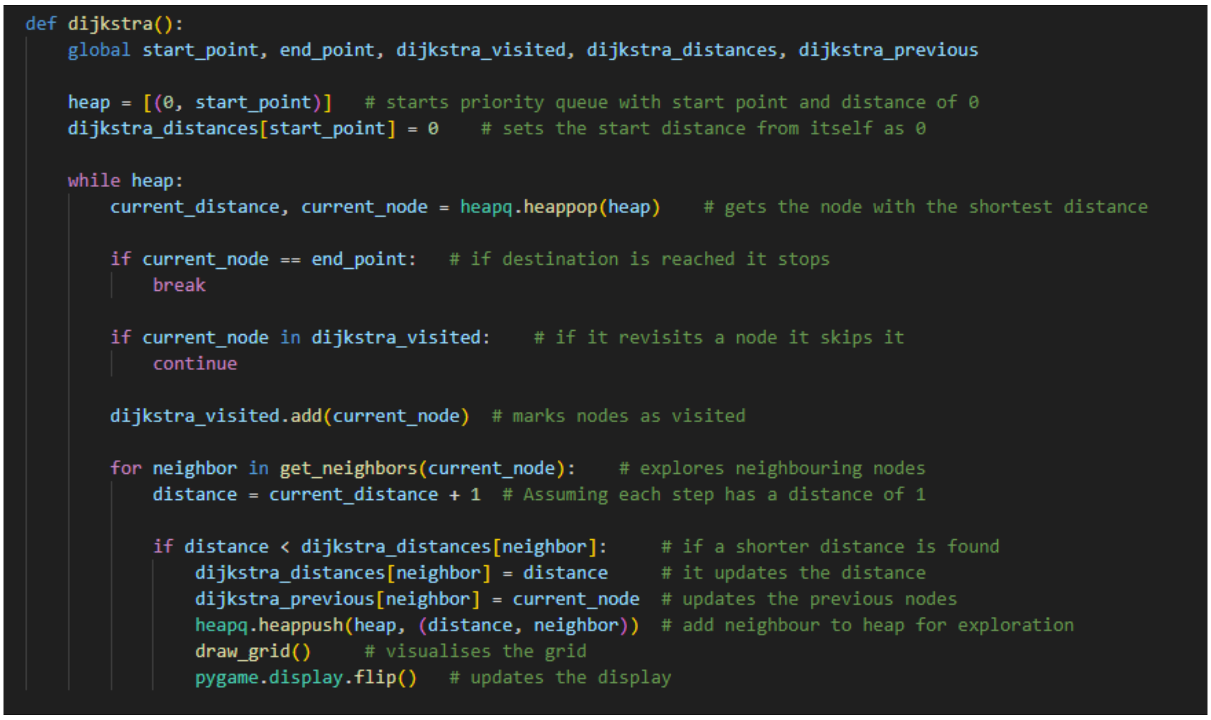
Pygame is a cross-platform set of python modules designed for writing video games, was selected as the primary library for creating the visualisation tool. Several factors contributed to this decision. Firstly, pygame provides a straightforward interface for graphics making it accessible for developers. Secondly, pygame offers robust graphics rendering capabilities, allowing for the creation of interactive visualisation with customisable shapes, colours, and animations. Thirdly, pygame application can run seamlessly on multiple platforms, including Windows, macOS, and Linux, ensuring broad accessibility to users. Lastly. Pygame's is instrumental for creating the graphical user interface. pygame benefits from a large and active community of developers who contribute tutorials, documentation, and libraries, which can expedite the development process and address potential challenges (*Beginning Game Development with Python and Pygame: From Novice to Professional - Will McGugan - Google Books* [2007]).

Python, alongside the pygame library, played a crucial role in achieving the objectives of this project. Firstly, Python's simplicity combined with pygame's graphics capabilities enabled the creation of an intuitive and visually appealing interface for visually appealing interface for visualising pathfinding algorithms. Secondly, pygame's event handling features allowed for user interaction, such as selecting start and end points on the grid and toggling walls. Thirdly Python's flexibility facilitated the implementation of various pathfinding algorithms, including Dijkstra's, A*, and Depth-First search, providing a clear and concise syntax for expressing complex logic. Lastly. Although Python is not as fast as low-level languages like C++ or Java, pygame's efficient rendering engine ensured smooth performance, even when visualising large grids or complex algorithms.

The inclusion of the 'sys', 'heapq', and 'math' libraries in the project significantly augmented its functionality and efficiency. The 'sys' library provided essential system-specific parameters and functions, enabling robust error handling and smooth termination of the application. Additionally, the 'heapq' library's implementation of the heap queue algorithms facilitated priority queue operations crucial for efficient pathfinding algorithms like Dijkstra's and A*. Furthermore, the 'math' library's mathematical functions, particularly the 'sqrt' function, were instrumental in calculating heuristic values for the A* algorithm, enhancing its accuracy and effectiveness. Overall, the incorporation of these libraries enhanced the project's performance, reliability, and computational capabilities, contributing to a more robust and versatile pathfinding visualisation tool.

3.5 Algorithm Functions

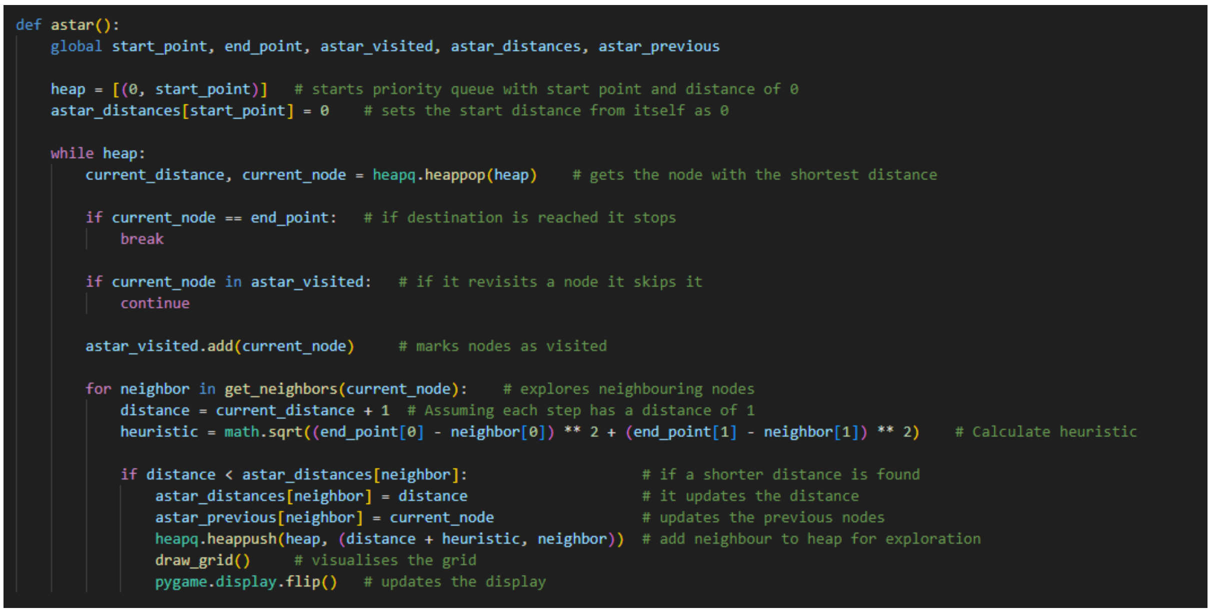
The 'dijkstra()' function implements Dijkstra's algorithm to find the shortest path from a start point to an end point on the grid. It begins by initialise a priority queue 'heap' with the start point and a distance of 0. Additionally, the distance from the start point to itself is set to 0. The function then enters a loop. Popping nodes from the priority queue based on their current distance. If the current node is the end point, the function terminates. If the current node has already been visited, it skipped to avoid revisiting nodes unnecessarily. Otherwise, the node is marked as visited, and its neighbours are explored. For each neighbour, the function calculates the distance from the start point through the current node. If this distance is shorter than the previously recorded distance, it updates the distance and the previous node accordingly. The neighbour is then added to the priority queue for further exploration. Throughout the process, the function updates the visualisation of the grid to reflect the current state of exploration.



```python
def dijkstra():
    global start_point, end_point, dijkstra_visited, dijkstra_distances, dijkstra_previous

    heap = [(0, start_point)]  # starts priority queue with start point and distance of 0
    dijkstra_distances[start_point] = 0    # sets the start distance from itself as 0

    while heap:
        current_distance, current_node = heapq.heappop(heap)    # gets the node with the shortest distance

        if current_node == end_point:   # if destination is reached it stops
            break

        if current_node in dijkstra_visited:    # if it revisits a node it skips it
            continue

        dijkstra_visited.add(current_node)  # marks nodes as visited

        for neighbor in get_neighbors(current_node):    # explores neighbouring nodes
            distance = current_distance + 1  # Assuming each step has a distance of 1

            if distance < dijkstra_distances[neighbor]:     # if a shorter distance is found
                dijkstra_distances[neighbor] = distance     # it updates the distance
                dijkstra_previous[neighbor] = current_node  # updates the previous nodes
                heapq.heappush(heap, (distance, neighbor))  # add neighbour to heap for exploration
                draw_grid()     # visualises the grid
                pygame.display.flip()   # updates the display
```

Figure 5: function for Dijkstra's algorithm.

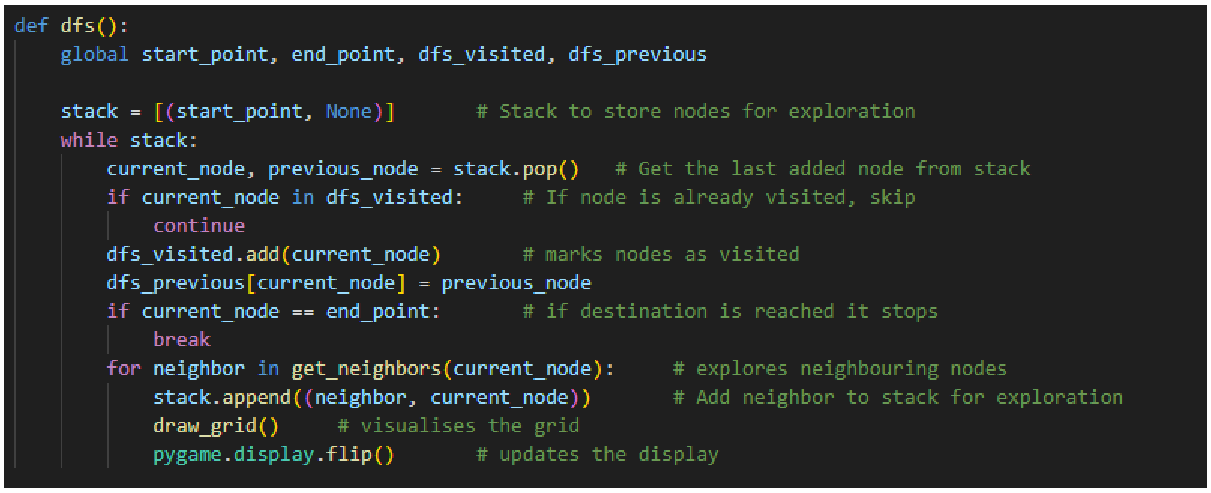
The ‘astar()’ function implements the A* algorithm, which is an informed search algorithm that utilises a heuristic to find the shortest path from a start point to an end point. Like Dijkstra’s algorithm, it begins by initialising a priority queue ‘heap’ with the start point and a distance of 0. Additionally, the distance from the start point to itself is set to 0. The function then enters a loop, popping nodes from the priority queue based on their current distance, If the current node is the end point, the function terminates. If the current node has already been visited, it is skipped to avoid revisiting nodes unnecessarily. Otherwise, the node is marked as visited, and its neighbours are explored. For each neighbour, the function calculates the distance from the start point through the current node and adds it to the heuristic value, which estimates the remaining distance to the end point. If this combined distance is shorter than the previously recorded distance, it updates the distance and the previous node accordingly. The neighbour is then added to the priority queue for further exploration. As with Dijkstra’s algorithm. The function updates the visualisation of the grid to reflect the current state of exploration.



```python
def astar():
    global start_point, end_point, astar_visited, astar_distances, astar_previous

    heap = [(0, start_point)]   # starts priority queue with start point and distance of 0
    astar_distances[start_point] = 0    # sets the start distance from itself as 0

    while heap:
        current_distance, current_node = heapq.heappop(heap)    # gets the node with the shortest distance

        if current_node == end_point:   # if destination is reached it stops
            break

        if current_node in astar_visited:   # if it revisits a node it skips it
            continue

        astar_visited.add(current_node)     # marks nodes as visited

        for neighbor in get_neighbors(current_node):    # explores neighbouring nodes
            distance = current_distance + 1  # Assuming each step has a distance of 1
            heuristic = math.sqrt((end_point[0] - neighbor[0]) ** 2 + (end_point[1] - neighbor[1]) ** 2)    # Calculate heuristic

            if distance < astar_distances[neighbor]:                    # if a shorter distance is found
                astar_distances[neighbor] = distance                    # it updates the distance
                astar_previous[neighbor] = current_node                 # updates the previous nodes
                heapq.heappush(heap, (distance + heuristic, neighbor))  # add neighbour to heap for exploration
                draw_grid()     # visualises the grid
                pygame.display.flip()   # updates the display
```

Figure 6: Function for A algorithm.*

The ‘dfs()’ function implements the Depth-First search algorithm, a fundamental graph traversal algorithm. It begins by initialising a stack ‘stack’ with the start point and ‘none’ to represent the absence of the previous node. The function then enters a loop, popping nodes from the stack. If the current node has already been visited, it is skipped to prevent revisiting nodes. Otherwise, the node is marked as visited, and its previous node is recorded. If the current node is the end point, the function terminates. Otherwise, the function explores the neighbouring nodes of the current node, adding them to the stack for further exploration. Like the previous algorithms, the function updates the visualisation of the grid to reflect the current state of exploration after each iteration of the loop.



```python
def dfs():
    global start_point, end_point, dfs_visited, dfs_previous

    stack = [(start_point, None)]       # Stack to store nodes for exploration
    while stack:
        current_node, previous_node = stack.pop()   # Get the last added node from stack
        if current_node in dfs_visited:     # If node is already visited, skip
            continue
        dfs_visited.add(current_node)       # marks nodes as visited
        dfs_previous[current_node] = previous_node
        if current_node == end_point:       # if destination is reached it stops
            break
        for neighbor in get_neighbors(current_node):     # explores neighbouring nodes
            stack.append((neighbor, current_node))       # Add neighbor to stack for exploration
            draw_grid()     # visualises the grid
            pygame.display.flip()       # updates the display
```

Figure 7: Function for DFS algorithm.

Testing, results, discussion, and evaluation

4.1 Testing

The testing approach for evaluating the pathfinding algorithms and visualisation tool involved a combination of iterative testing, unit test and manual testing. Throughout the iterative development process, the code underwent continuous testing and debugging to identify and rectify any errors or bugs. Unit tests were utilised to verify the correctness of individual algorithmic components, while manual testing was employed to assess the overall functionality and user experience of the visualisation tool. The goal was to ensure that the algorithms produced correct results and that the visualisation accurately depicted the pathfinding algorithm.

This test plan outlines the procedures for testing the functionality and usability of the pathfinding algorithm visualisation tool. The goal is to ensure that the algorithms produce correct results, and that the visualisation accurately depicts the pathfinding progress.

Table 1: Algorithm testing and evaluation results.

Test case	Test case description	Actions	Expected Results	Actual results	Error handling
1	Visualisation tool	Set obstacles, start, and end point	Set obstacles, start, and end point	Kept only placing the start node	Debugged and fixed the issue
		Select pathfinding algorithm	Select the right algorithm with the right key	Selected the algorithm	
		Observe algorithm execution in real time	Observe algorithm execution in real time	Observe algorithm execution in real time	Hardware issue had to greatly reduce the grid size
2	Dijkstra's algorithm	Run Dijkstra's algorithm	Dijkstra's algorithm to run	Dijkstra's algorithm did run.	
		Verify shortest path is found	The shortest path to be found	Failed to identify shortest path	Code error resulting in inaccurate path. Fixed the mistake.
3	A* algorithm	Run A* algorithm	A* algorithm to run	when used it crashed the software	Debugged and resolved
		Verify shortest path is found with heuristic	The shortest path to be found with heuristic cost	The shortest path was found with heuristic cost	

4	Depth-First search algorithm	Run Depth-First search algorithm	Run Depth-First search algorithm	Depth-First search successfully run	
		Verify path is found	To see a path has been found	A path was found	

The test plan contains various aspects of the pathfinding algorithm and visualisation tool, including Dijkstra's, Depth-First search, A* algorithms, and the visual user interface functionalities. Each test case involves executing the respective algorithm or tool functionality and comparing the actual results against the expected outcomes. Error handling strategies including debugging code to correct the issue such as when only the starting node could be placed in the visualisation tool, and the software crashing for when A* algorithm was used. These measures ensure the reliability and accuracy of the implemented algorithms and visualisation tool, enhancing their usability.

For unit testing, specific test cases for Dijkstra's algorithm and A* algorithm. In the unit test for Dijkstra's algorithm. In the unit test for Dijkstra's algorithm, a grid environment was created with defined started and end points, obstacles. The expect outcome was to ensure that Dijkstra's algorithm was correctly computed the shortest path from the start to the end point while following predefined obstacles, such as termination within a reasonable time frame and handling of unreachable endpoints or invalid grid configurations. Similarly, the unit test for A* algorithm to confirm the algorithm's optimality in finding the shortest path by considering both the actual cost and heuristic estimation, while maintain admissibility and consistency properties.

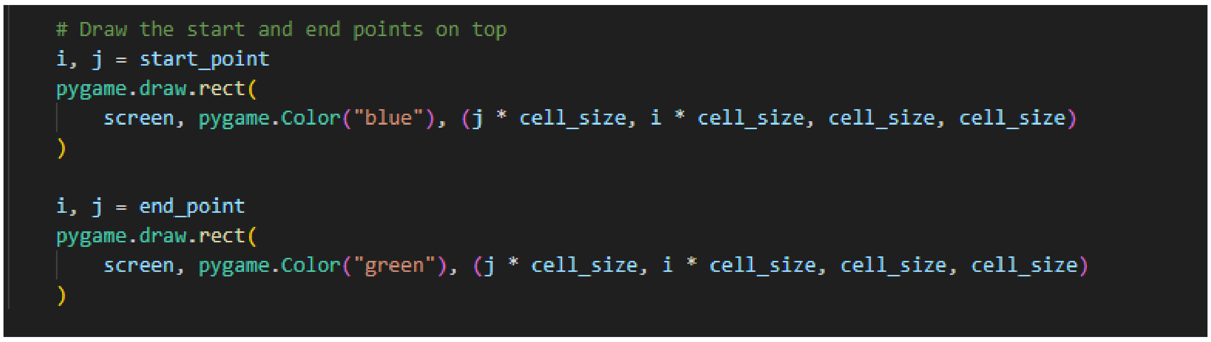
Test scenarios were created to evaluate the tool's performance under various conditions, such as different grid sizes, obstacle placemen, and algorithm selections. These scenarios were used to gauge the tool's effectiveness in finding the shortest path between the specific start and end points. Additionally. Scenarios with different obstacles densities were tested to assess the algorithms' adaptability to vary environmental complexities.

Integration testing aimed to assess how well the visualisation tool worked with the implement algorithms. Testing involved different grid setups, obstacles, and algorithm choices to evaluate if it accurately depicted algorithm operations and resulting path. Evaluation criteria included responsiveness, clarity of visualisation and usability across various scenarios.

The results of the testing phase were primarily qualitive, focusing on the functionality and usability of the visualisation tool. Any issues encountered during testing, such as algorithmic errors or interface glitches, were documented and addressed through code modifications. The emphasis was on ensuring that the tool provided the expected visual representation of the pathfinding process and accurately depicted the shortest path between the specified points.

The discussion of results centred on the observed behaviour of the pathfinding algorithm and the visualisation tool during testing. Any challenges that were encountered, such as graphic anomalies, the algorithm malfunctioning, and the occasional software crashes, were analysed, and appropriate adjustments were made to improve performance for user experience. The iterative nature of the testing process allowed for continuous refinement of the tool, leading to enhanced functionality and reliability overtime.

Another notable challenge that was encountered was related to the highlighting of the start and end nodes in the visualisation tool. Initially the highlighting function used to mark the shortest path also inadvertently highlighted the start and end nodes, causing visual confusion. To address this issue a few lines of code were implemented within the highlighting function, which made the start and end nodes remain their colour of red and green. Separating the highlighting of the path from the start and end nodes made the tool easer to use and understand. The change improved how users interacted with the tool, making it more user-friendly. It shows how we fixed problems step by step to make the tool better.



```
# Draw the start and end points on top
i, j = start_point
pygame.draw.rect(
    screen, pygame.Color("blue"), (j * cell_size, i * cell_size, cell_size, cell_size)
)

i, j = end_point
pygame.draw.rect(
    screen, pygame.Color("green"), (j * cell_size, i * cell_size, cell_size, cell_size)
)
```

Figure 8: additional lines added to draw_shortest_path function.

4.2 Results

The experimentation provided a valuable insight into the performance and behaviour of the implemented pathfinding algorithms and visualisation tool. Through a combination of qualitative observations and quantitative measurements, a comprehensive understanding of their effectiveness in solving the task at hand.

Quantitative analysis means measuring important factors like how fast the algorithm run, how long the paths they find are, and how good those paths are. Dijkstra's algorithm consistently found paths quickly and efficiently across different grid sizes and obstacle amounts. A* algorithm, which uses estimates to speed up its search, was even faster than Dijkstra's, especially in more complicated situations.

Meanwhile, Depth-First search algorithm, while not as fast, show different ways of finding paths, giving us new ideas.

Qualitative observations provided valuable insights into algorithm behaviour and traversal patterns in diverse environments. Dijkstra's algorithm exhibited a systematic approach to exploring the search space, prioritising nodes based on their distance from the start point. A* algorithm demonstrated a balance between exploration and exploitation, leveraging heuristic information to guide search towards the goal while efficiently trimming unproductive branches. Depth-First search algorithm, characterised by depth-first exploration, exhibited a tendency to explore paths deeply before backtracking, making it suitable for certain problem domains with specific constraints or objectives.

Comparing the performance of different algorithms revealed trade-offs between computational complexity, solution quality, and optimality. Dijkstra's algorithm. Consistently provided optimal solutions but exhibit higher computational cost, particularly noticeable in larger grids due to its exhaustive exploration. On the other hand, A* algorithms integration of heuristic estimation resulted in expedited pathfinding, especially evident in environment s with complex obstacles. The quality of solutions was reliably high, contingent upon the accuracy of the employed heuristic function. In contrast, Depth-First search algorithm, though less efficient in terms of optimality compared to Dijkstra's and A*, showcased a unique exploration strategy suited for specific scenarios, such as maze-like environments. Despite its non-optimality, Depth-First search offered valuable insights into alternative pathfinding strategies, enriching the understanding of algorithmic approaches.

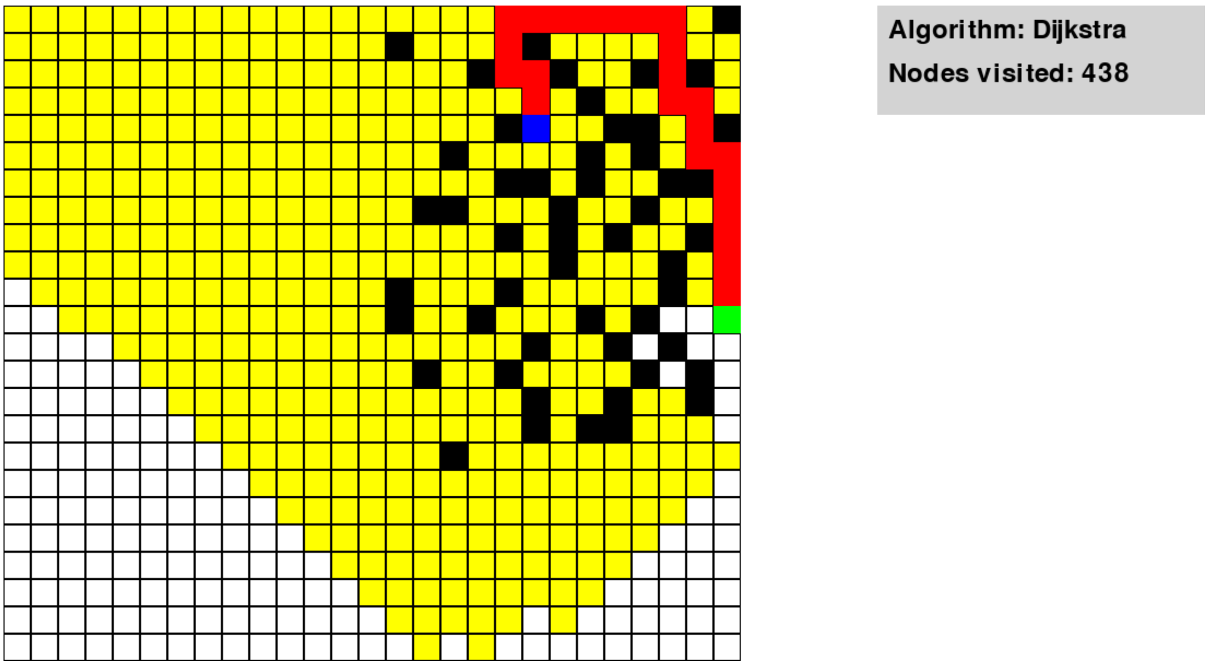

Algorithm: Dijkstra
Nodes visited: 438

Figure 9: Dijkstra and A comparison.*

.

A notable observation emerges from the experimentation regarding efficiency contrast between Dijkstra's and A* algorithms. While Dijkstra's algorithm systematically explored a total of 438 nodes to reach the goal, A* algorithm exhibited remarkable efficiency, traversing only 162 nodes. This substantial reduction in node exploration underscores the efficacy of A* Algorithm's heuristic-guided approach, intelligently directing the search process towards the goal while efficiently discarding unproductive paths. The stark contrast in node exploration highlights the practical advantages of A8 algorithm in real-world applications, where computational efficiency plays a crucial role in performance optimisation.

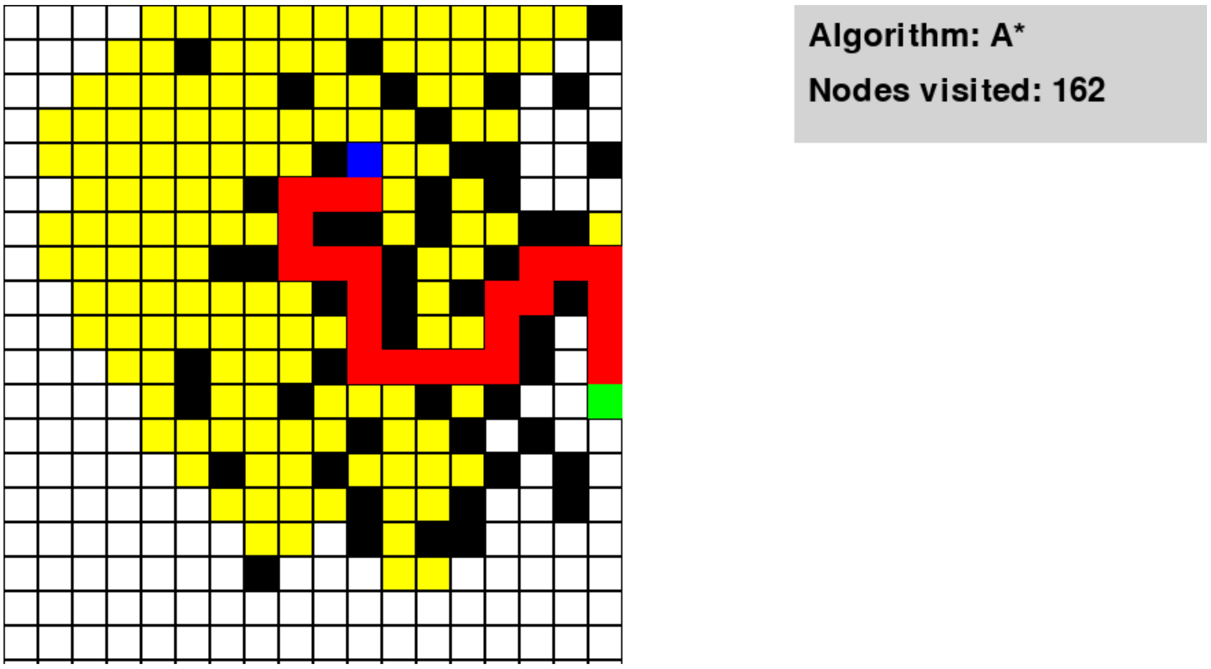


Figure 10: A and Dijkstra's comparison.*

The quantitative measures and qualitive observations provide a comprehensive evaluation of each algorithm's performance, facilitating informed decisions regarding their applicability to diverse problem domains. These findings not only contribute to the optimisation of pathfinding algorithms but also lay the groundwork for future research efforts in computational problem-solving.

In reflection, the experimentation yielded valuable insights into the performance and behaviour of pathfinding algorithms, highlighting several key lessons. Firstly, and understanding of the trade-offs between computational complexity, solution quality, and optimality is essential for selecting the most suitable algorithm for specific applications. Secondly, the accuracy of heuristic functions significantly impacts the effectiveness of algorithms like A*, emphasising the importance of fine-tuning and validating these functions. Additionally, recognising the unique characteristics and traversal patterns of each algorithm enables informed decision-making in diverse problem domains. Furthermore, balancing efficiency and optimality is crucial for optimising algorithm performance within resource constraints. Insights from alternative strategies, such as Depth-First search, offer valuable perspectives and enrich the understanding of algorithmic approaches. Ultimately, these lessons inform practical implications for real-world applications, guiding decision-making in fields like robotics, navigation systems, and video game development.

4.3 Discussion

The experimentation and testing phase provided valuable insights into the performance and behaviour of the implemented pathfinding algorithms and visualisation tool. In this section, we interpret the results obtained and discuss their significance in the context of our research objectives.

The quantitative and qualitive analysis of the pathfinding algorithms revealed distinct characteristics and performance measures. Dijkstra's algorithm consistently demonstrated efficiency in finding paths quickly and reliably across different grid size and obstacles configuration. Its systematic approach to exploring the search space, prioritising nodes based on their distance from the start point, contributed to its effectiveness in finding optimal solutions.

Conversely, the A* algorithm, leveraging heuristic estimation to guide search towards the goal while efficiently trimming unproductive branches, exhibited even faster performance, particularly in more complex environments. The balance between exploration and exploitation facilitated by A* resulted in expedited pathfinding without compromising solution quality.

Depth-First search algorithm, while not as fast as Dijkstra's and A*, showcased alternative pathfinding strategies characterised by Depth-First search. Despite its non-optimality, Depth-First search offered valuable insights into unique traversal patterns suited for specific problem domain, such as maze-like environments.

The findings align with existing literature on pathfinding algorithms, confirming the effectiveness of Dijkstra's and A* algorithms in solving shortest path problems. Additionally, the exploration of Depth-First search highlights its potential applicability in certain scenarios, complementing previous studies that primarily focus on Dijkstra's and A* algorithms.

Throughout the testing phase, several challenges were encountered, including algorithmic errors, interface glitches, and unexpected behaviours. These challenges were systematically address through iterative testing and debugging, leading to continuous refinement of the visualisation tool. Notable improvements, such as the separating of path highlighting from start and end nodes, enhanced the usability of the user experience tool.

4.4 Evaluation

This section, the assessment of the approach's validity is conducted, reflections on the strengths and weaknesses of the project are provided, and discussions on its contributions to the field of pathfinding algorithms and visualisation technique are presented.

One of the key strengths of our project lies in the comprehensive testing and evaluation of pathfinding algorithms across diverse grid configurations and obstacle placements. The combination of quantitative measures and qualitative observations provided a detailed understanding of algorithmic behaviours and performance. The visualisation tool emerged as an asset, assisting in the clear representation of algorithm operations, and facilitating user interaction. Additionally, while the project focused primarily on classic pathfinding algorithms, future iterations could explore more advanced techniques or hybrid approaches to further enhance performance and versatility.

The comparison of A*, Dijkstra's and Depth-First search algorithm sheds light on their respective strengths and weaknesses informing practitioners and researchers alike in algorithm selection for various problem domains. Furthermore, the development of the visualisation tool offers a valuable resource for teaching and learning pathfinding concepts. Its user-friendly interface and real-time visualisation capabilities enable users to explore and understand the complexities of pathfinding algorithms in an interactive manner.

Conclusion and future work

In conclusion, this project set out to design and implement a pathfinding visualisation tool, evaluating the effectiveness of various algorithms in solving navigation problems. Through iterative development, rigorous testing, and thorough analysis, several key findings have emerged. Firstly, the comparison between Dijkstra's and A* algorithms revealed trade-oofs between computational complexity and solution quality, highlighting the importance of heuristic-guided approaches in optimising efficiency. Secondly, the exploration of alternative strategies like Depth-First search provided valuable insights into algorithmic diversity and traversal patterns. This project's success in achieving its objectives underscores the significant of algorithmic optimisation in real-world applications such as robotics and navigation systems. Moving forward, further research could explore advanced heuristic functions, algorithmic enhancements, and integration with emerging technologies to enhance performance and applicability. Furthermore, for improvements on this project, one section that should be explored involves incorporating different values for each cell in the grid. By assigning varying costs or weights to different cells based on factors such as terrain type, or traversal difficulty, the pathfinding algorithms can be enhanced to produce more realistic and context-aware paths. This addition would enable the algorithms to adapt to diverse environment's more effectively leading to improved pathfinding solutions for wider range of scenarios.

By leveraging the lessons learned from this project, future endeavours in computational problem-solving can build upon these foundations to address increasingly complex challenges and drive innovation in the field. This conclusion summarises the project's objectives, outcomes, and implications serving as a concise summary and roadmap for future studies in pathfinding algorithms and visualisation tool.

Project management review

The project management review provides a comprehensive assessment of the project's management, examining key aspects such as time and workload management, quality of work, schedules, meetings, communication, resources, and documentation.

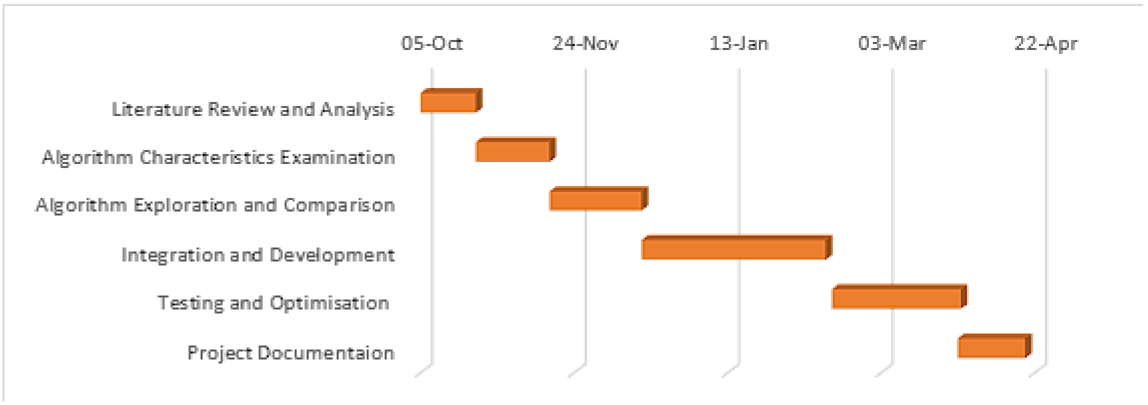


Figure 11: Gantt chart of work timeline.

This Gantt chart allowed me to ay attention to the timeline, ensuring tasks were completed within the allocated timeframe. Regular monitoring of progress allowed for timely adjustment and task reallocation to manage workload effectively. Additionally, the quality of work was prioritised, with a focus on delivering robust and well tested solutions. Meetings were held regularly, but while the inability to attend every meeting, while not hindering progress, did make it uncertain on what to do next and what to do for certain aspects of the project, but the meetings were highly productive, meeting schedules and communication were made evident. An improvement would be to keep the timeline to the Gantt chart scheduled correctly, there were instances where deviations occurred due to unforeseen circumstances such as personal timetable, exams and other assignments and projects, as this Gantt chart was made without the thought of other projects clashing. Improving the schedule of the Gantt chart could enhance project efficiency and help ensure timely completion of task.

References

Acar, U.A., Charguéraud, A. and Rainey Inria, M. [2015]. A Work-Efficient Algorithm for Parallel Unordered Depth-First Search. Available at: http://dx.doi.org/10.1145/2807591.2807651 [Accessed: 13 March 2024].

Afshani, P. 2015. *Pathfinding in Two-dimensional Worlds*.

Barnouti, N.H., Al-Dabbagh, S.S.M., Naser, M.A.S., Barnouti, N.H., Al-Dabbagh, S.S.M. and Naser, M.A.S. 2016. Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm. *Journal of Computer and Communications* 4(11), pp. 15–25. Available at: http://www.scirp.org/journal/PaperInformation.aspx?PaperID=70460 [Accessed: 10 March 2024].

Beginning Game Development with Python and Pygame: From Novice to Professional - Will McGugan - Google Books. [2007]. Available at: https://books.google.co.uk/books?hl=en&lr=&id=Kn8nCgAAQBAJ&oi=fnd&pg=PR14&dq=pygame&ots=eNRSqpAlgE&sig=eiPPqeRI8gYFbT9FB_OwBUCd6IQ&redir_esc=y#v=onepage&q=pygame&f=false [Accessed: 24 March 2024].

Bonet, B. and Geffner, H. [2006]. Learning Depth-First Search: A Unified Approach to Heuristic Search in Deterministic and Non-Deterministic Settings, and Its Application to MDPs. Available at: www.aaai.org [Accessed: 12 March 2024].

Chen, Q., Chen, J. and Huang, W. 2022. Pathfinding method for an indoor drone based on a BIM-semantic model. Available at: https://doi.org/10.1016/j.aei.2022.101686 [Accessed: 10 March 2024].

Cui, X. and Shi, H. 2011. A*-based Pathfinding in Modern Computer Games. *IJCSNS International Journal of Computer Science and Network Security* 11(1). Available at: https://www.researchgate.net/publication/267809499 [Accessed: 10 March 2024].

Edsger Wybe Dijkstra: His Life, Work, and Legacy - Krzysztof R. Apt, Tony Hoare - Google Books. [2022]. Available at: https://books.google.co.uk/books?hl=en&lr=&id=d8x8EAAAQBAJ&oi=fnd&pg=PR9&dq=history+of+dijkstra&ots=MEPio6LbLI&sig=h8B4DJh9FBJG29tZjAYRe3rHpC4&redir_esc=y#v=onepage&q=history%20of%20dijkstra&f=false [Accessed: 10 March 2024].

Fu, L. [2001]. An adaptive routing algorithm for in-vehicle route guidance systems with real-time information. Available at: www.elsevier.com/locate/trb [Accessed: 10 March 2024].

Graham Hugh McCabe Stephen Sheridan, R. 2003. Issue 2 Article 6 2003 Part of the Computer and Systems Architecture Commons. *The ITB Journal* 4(2). Available at: https://arrow.tudublin.ie/itbjhttps://arrow.tudublin.ie/itbj/vol4/iss2/6 [Accessed: 13 March 2024].

Iloh, P.C. 2022. A COMPREHENSIVE AND COMPARATIVE STUDY OF DFS, BFS, AND A* SEARCH ALGORITHMS IN A SOLVING THE MAZE TRANSVERSAL PROBLEM. *International Journal of Social Sciences and Scientific Studies* 2(2), pp. 482–490. Available at: https://www.ijssass.com/index.php/ijssass/article/view/54 [Accessed: 12 March 2024].

Kumar Guruji, A., Agarwal, H. and Parsediya, D.K. 2016. ScienceDirect Time-Efficient A* Algorithm for Robot Path Planning. *Procedia Technology* 23, pp. 144–149. Available at: www.sciencedirect.com [Accessed: 10 March 2024].

Li, H., Chu, Z., Fang, Y., Liu, H., Zhang, M., Wang, K. and Huang, J. 2023. Available online 5. *Expert Systems With Applications* 233, pp. 957–4174. Available at: https://doi.org/10.1016/j.eswa.2023.120932 [Accessed: 10 March 2024].

Luo, L., Zhao, N., Zhu, Y. and Sun, Y. 2023. A* guiding DQN algorithm for automated guided vehicle pathfinding problem of robotic mobile fulfillment systems. Available at: https://doi.org/10.1016/j.cie.2023.109112 [Accessed: 10 March 2024].

Mehta, H., Kanani, P. and Lande, P. 2019. Google Maps. *Google Maps Article in International Journal of Computer Applications* 178(8), pp. 975–8887. Available at: https://www.researchgate.net/publication/333117435 [Accessed: 10 March 2024].

Nasiboglu, R. 2022. Dijkstra solution algorithm considering fuzzy accessibility degree for patch optimization problem. *Applied Soft Computing* 130, p. 109674. Available at: https://doi.org/10.1016/j.asoc.2022.109674 [Accessed: 11 March 2024].

Noto, M. and Sato, H. 2000. Method for the shortest path search by extended Dijkstra algorithm. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics* 3, pp. 2316–2320. doi: 10.1109/ICSMC.2000.886462.

Oh, H.J. and Ashuri, B. 2023. Sustainable Cities and Society 95 (2023) 104600 Enriching GPS data for expanding interpretation of emergency vehicles using a pathfinding algorithm and spatial data harvesting methods. Available at: https://doi.org/10.1016/j.scs.2023.104600 [Accessed: 10 March 2024].

Pan, T. and Ching Pun-Cheng, S. 2020. A Discussion on the Evolution of the Pathfinding Algorithms. Available at: www.preprints.org.

Papamanthou, C. [2004]. Depth First Search & Directed Acyclic Graphs.

Pathfinding Algorithms in Game Development. [2021]. doi: 10.1088/1757-899X/769/1/012021.

Ponce, F.& et al. 2021. Artificial Intelligence A Modern Approach Fourth Edition. Available at: https://lccn.loc.gov/2019047498 [Accessed: 10 March 2024].

Rahayuda, I.G.S. and Santiari, N.P.L. 2021. Dijkstra and Bidirectional Dijkstra on Determining Evacuation Routes. In: *Journal of Physics: Conference Series*. IOP Publishing Ltd. doi: 10.1088/1742-6596/1803/1/012018.

Schrijver, A. 2012. *On the History of the Shortest Path Problem*.

Sedeño-Noda, A. and Colebrook, M. 2019. Discrete Optimization A biobjective Dijkstra algorithm. *European Journal of Operational Research* 276, pp. 106–118. Available at: https://doi.org/10.1016/j.ejor.2019.01.007 [Accessed: 13 March 2024].

Sharma, S., Singh, R. and Kumar, R. [2023]. *Pathfinding Visualizer*. Available at: https://www.researchgate.net/publication/374848532.

Shen, B., Cheema, M.A., Harabor, D.D. and Stuckey, P.J. 2022. Fast optimal and bounded suboptimal Euclidean pathfinding. *Artificial Intelligence* 302, p. 103624. Available at: www.elsevier.com/locate/artint [Accessed: 10 March 2024].

Sigurdson, D., Bulitko, V., Yeoh, W., Hernández, C. and Koenig, S. [2019]. Automatic Algorithm Selection in Multi-Agent Pathfinding.

Theses, E., Theses, D., Papers, M. and Kaur Sidhu, H. 2019. *Scholarship at UWindsor Scholarship at UWindsor Performance Evaluation of Pathfinding Algorithms Performance Evaluation of Pathfinding Algorithms*. Available at: https://scholar.uwindsor.ca/etd.

View of A COMPREHENSIVE AND COMPARATIVE STUDY OF DFS, BFS, AND A SEARCH ALGORITHMS IN A SOLVING THE MAZE TRANSVERSAL PROBLEM*. [2022]. Available at: https://www.ijssass.com/index.php/ijssass/article/view/54/64 [Accessed: 10 March 2024].

Wayahdi, M.R., Ginting, S.H.N. and Syahputra, D. 2021. Greedy, A-Star, and Dijkstra's Algorithms in Finding Shortest Path. *International Journal of Advances in Data and Information Systems* 2(1), pp. 45–52. doi: 10.25008/ijadis.v2i1.1206.

Xu, M.H., Liu, Y.Q., Huang, Q.L., Zhang, Y.X. and Luan, G.F. 2007]. An improved Dijkstra's shortest path algorithm for sparse network q. Available at: www.elsevier.com/locate/amc [Accessed: 13 March 2024].

Zhang, W. and Korf, R.E. 1993. Depth-First vs. est-First Search: New Results *. Available at: www.aaai.org [Accessed: 12 March 2024].

Appendix code:

```
import pygame
import sys
import heapq
import math

# Initialize Pygame
pygame.init()

# Set up the grid
width, height = 800, 600
cell_size = 25
rows, cols = height // cell_size, (width // cell_size) - 5  # Reduced columns
for more space

# Define the area for displaying text
text_area_width = 300
text_area_height = 100
text_area_x = width
text_area_y = 0

# Initialize the display
screen = pygame.display.set_mode((width + text_area_width, height +
text_area_height))
pygame.display.set_caption("Pathfinding Visualization")

# Initialize font
pygame.font.init()
font = pygame.font.Font(None, 36)

# Variables for text display
algorithm_text = font.render("Algorithm: Dijkstra", True,
pygame.Color("black"))
nodes_visited_text = font.render("Nodes visited: 0", True,
pygame.Color("black"))

# Initialize the grid with zeros, where 0 represents an empty cell
grid = [[0] * cols for _ in range(rows)]
```

```
# Variables to store the positions of start and end points
start_point = None
end_point = None

# Variables for Dijkstra's algorithm
dijkstra_visited = set()
dijkstra_distances = {(i, j): float('inf') for j in range(cols) for i in
range(rows)}
dijkstra_previous = {(i, j): None for j in range(cols) for i in range(rows)}

# Variables for A* algorithm
astar_visited = set()
astar_distances = {(i, j): float('inf') for j in range(cols) for i in
range(rows)}
astar_previous = {(i, j): None for j in range(cols) for i in range(rows)}

# Variables for DFS algorithm
dfs_visited = set()
dfs_previous = {(i, j): None for j in range(cols) for i in range(rows)}

# Variable to store the currently selected algorithm
current_algorithm = "Dijkstra"

def dijkstra():
    global start_point, end_point, dijkstra_visited, dijkstra_distances,
dijkstra_previous

    heap = [(0, start_point)]   # starts priority queue with start point and
distance of 0
    dijkstra_distances[start_point] = 0    # sets the start distance from
itself as 0

    while heap:
        current_distance, current_node = heapq.heappop(heap)    # gets the
node with the shortest distance

        if current_node == end_point:   # if destination is reached it stops
            break

        if current_node in dijkstra_visited:    # if it revisits a node it
skips it
            continue

        dijkstra_visited.add(current_node)  # marks nodes as visited

        for neighbor in get_neighbors(current_node):    # explores
neighbouring nodes
```

```
            distance = current_distance + 1  # Assuming each step has a
distance of 1

            if distance < dijkstra_distances[neighbor]:     # if a shorter
distance is found
                dijkstra_distances[neighbor] = distance     # it updates the
distance
                dijkstra_previous[neighbor] = current_node  # updates the
previous nodes
                heapq.heappush(heap, (distance, neighbor))  # add neighbour to
heap for exploration
                draw_grid()     # visualises the grid
                pygame.display.flip()   # updates the display

def astar():
    global start_point, end_point, astar_visited, astar_distances,
astar_previous

    heap = [(0, start_point)]   # starts priority queue with start point and
distance of 0
    astar_distances[start_point] = 0    # sets the start distance from itself
as 0

    while heap:
        current_distance, current_node = heapq.heappop(heap)    # gets the
node with the shortest distance

        if current_node == end_point:   # if destination is reached it stops
            break

        if current_node in astar_visited:   # if it revisits a node it skips
it
            continue

        astar_visited.add(current_node)     # marks nodes as visited

        for neighbor in get_neighbors(current_node):    # explores
neighbouring nodes
            distance = current_distance + 1  # Assuming each step has a
distance of 1
            heuristic = math.sqrt((end_point[0] - neighbor[0]) ** 2 +
(end_point[1] - neighbor[1]) ** 2)    # Calculate heuristic

            if distance < astar_distances[neighbor]:                    # if a
shorter distance is found
                astar_distances[neighbor] = distance                    # it
updates the distance
```

```
                astar_previous[neighbor] = current_node                 #
updates the previous nodes
                heapq.heappush(heap, (distance + heuristic, neighbor))  # add
neighbour to heap for exploration
                draw_grid()     # visualises the grid
                pygame.display.flip()   # updates the display

def dfs():
    global start_point, end_point, dfs_visited, dfs_previous

    stack = [(start_point, None)]       # Stack to store nodes for exploration
    while stack:
        current_node, previous_node = stack.pop()   # Get the last added node
from stack
        if current_node in dfs_visited:     # If node is already visited, skip
            continue
        dfs_visited.add(current_node)       # marks nodes as visited
        dfs_previous[current_node] = previous_node
        if current_node == end_point:       # if destination is reached it
stops
            break
        for neighbor in get_neighbors(current_node):     # explores
neighbouring nodes
            stack.append((neighbor, current_node))       # Add neighbor to
stack for exploration
            draw_grid()     # visualises the grid
            pygame.display.flip()       # updates the display

def get_neighbors(node):
    i, j = node
    neighbors = []

    # Check the four cardinal directions
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    for x, y in directions:
        new_i, new_j = i + x, j + y

        if 0 <= new_i < rows and 0 <= new_j < cols and grid[new_i][new_j] != -
1:    # checks bounds and see's if the cell is a wall
            neighbors.append((new_i, new_j))        # Add neighbor to the list

    return neighbors

# Function to draw the grid with colors
def draw_grid():
    screen.fill(pygame.Color("white"))  # Clear the screen

    for row in range(rows):
        for col in range(cols):
            if (row, col) == start_point:
                color = pygame.Color("blue")  # Start point
            elif (row, col) == end_point:
                color = pygame.Color("green")  # End point
            elif grid[row][col] == -1:
                color = pygame.Color("black")  # Wall
            elif current_algorithm == "Dijkstra":
                if (row, col) in dijkstra_visited and (row, col) !=
start_point and (row, col) != end_point:
                    color = pygame.Color("yellow")  # Visited node during
Dijkstra's search
                elif (row, col) in get_shortest_path("Dijkstra"):
                    color = pygame.Color(255, 0, 0, 100)  # Shortest path with
transparency for Dijkstra
                else:
                    color = pygame.Color("white")  # Empty cell
            elif current_algorithm == "A*":
                if (row, col) in astar_visited and (row, col) != start_point
and (row, col) != end_point:
                    color = pygame.Color("yellow")  # Visited node during A*
search
                elif (row, col) in get_shortest_path("A*"):
                    color = pygame.Color(255, 0, 0, 100)  # Shortest path with
transparency for A*
                else:
                    color = pygame.Color("white")  # Empty cell
            elif current_algorithm == "DFS":
                if (row, col) in dfs_visited and (row, col) != start_point and
(row, col) != end_point:
                    color = pygame.Color("yellow")  # Visited node during DFS
search
                elif (row, col) in get_shortest_path("DFS"):
                    color = pygame.Color(255, 0, 0, 100)  # Shortest path with
transparency for DFS
                else:
                    color = pygame.Color("white")  # Empty cell

            pygame.draw.rect(
                screen, color, (col * cell_size, row * cell_size, cell_size,
cell_size)
            )
            pygame.draw.rect(
                screen, pygame.Color("black"), (col * cell_size, row *
cell_size, cell_size, cell_size), 1
            )

    # Draw the text area background
    pygame.draw.rect(screen, pygame.Color("lightgray"), (width, 0,
text_area_width, text_area_height))

    # Draw the text
    screen.blit(algorithm_text, (width + 10, 10))
    if current_algorithm == "DFS":
        nodes_visited_text = font.render(f"Nodes visited: {len(dfs_visited)}",
True, pygame.Color("black"))
    else:
        nodes_visited_text = font.render(f"Nodes visited:
{len(dijkstra_visited if current_algorithm == 'Dijkstra' else
astar_visited)}", True, pygame.Color("black"))
    screen.blit(nodes_visited_text, (width + 10, 50))

# Function to draw the shortest path found by the algorithm.
def draw_shortest_path(algorithm):
    global end_point, dijkstra_previous, astar_previous, dfs_previous
    current_node = end_point
    if algorithm == "Dijkstra":
        previous = dijkstra_previous
    elif algorithm == "A*":
        previous = astar_previous
    else:
        previous = dfs_previous

    while current_node is not None:
        i, j = current_node
        pygame.draw.rect(
            screen, pygame.Color(255, 0, 0, 100), (j * cell_size, i *
cell_size, cell_size, cell_size)
        )
        current_node = previous[current_node]

    # Draw the start and end points on top
    i, j = start_point
    pygame.draw.rect(
        screen, pygame.Color("blue"), (j * cell_size, i * cell_size,
cell_size, cell_size)
    )

    i, j = end_point
    pygame.draw.rect(
        screen, pygame.Color("green"), (j * cell_size, i * cell_size,
cell_size, cell_size)
    )

# Function to get the shortest path found by the algorithm.
def get_shortest_path(algorithm):
    global end_point, dijkstra_previous, astar_previous, dfs_previous
    path = []
    current_node = end_point
    if algorithm == "Dijkstra":
        previous = dijkstra_previous
    elif algorithm == "A*":
        previous = astar_previous
    else:
        previous = dfs_previous

    while current_node is not None:
        path.append(current_node)
        current_node = previous[current_node]
    return path

# Function to reset the grid
def reset_grid():
    global grid, start_point, end_point, dijkstra_visited, dijkstra_distances,
dijkstra_previous, astar_visited, astar_distances, astar_previous,
dfs_visited, dfs_previous
    grid = [[0] * cols for _ in range(rows)]    # Reset grid to all empty
cells
    start_point = None  # Reset start point
    end_point = None    # Reset end point
    dijkstra_visited = set()    # Clear visited nodes for Dijkstra
    dijkstra_distances = {(i, j): float('inf') for j in range(cols) for i in
range(rows)}
    dijkstra_previous = {(i, j): None for j in range(cols) for i in
range(rows)}
    astar_visited = set()       # Clear visited nodes for A*
    astar_distances = {(i, j): float('inf') for j in range(cols) for i in
range(rows)}
    astar_previous = {(i, j): None for j in range(cols) for i in range(rows)}
    dfs_visited = set()         # Clear visited nodes for DFS
    dfs_previous = {(i, j): None for j in range(cols) for i in range(rows)}

# Main loop
running = True
path_found = False

while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False     # Quit the main loop if the window is closed
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_pos = pygame.mouse.get_pos()
            row = mouse_pos[1] // cell_size
            col = mouse_pos[0] // cell_size
            # Toggle between start point, end point, and wall on mouse click
            if grid[row][col] == 0 and start_point is None:
                grid[row][col] = 1  # Set as start point
                start_point = (row, col)
            elif grid[row][col] == 0 and end_point is None:
                grid[row][col] = 2  # Set as end point
                end_point = (row, col)
            elif grid[row][col] == -1:
                grid[row][col] = 0  # Remove wall
            elif grid[row][col] == 0 and start_point is not None and end_point
is not None:
                grid[row][col] = -1  # Set as wall
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_d:
                current_algorithm = "Dijkstra"
                algorithm_text = font.render("Algorithm: Dijkstra", True,
pygame.Color("black"))
            elif event.key == pygame.K_a:
                current_algorithm = "A*"
                algorithm_text = font.render("Algorithm: A*", True,
pygame.Color("black"))
            elif event.key == pygame.K_s:
                current_algorithm = "DFS"
                algorithm_text = font.render("Algorithm: DFS", True,
pygame.Color("black"))
            elif event.key == pygame.K_f and start_point is not None and
end_point is not None:
                if current_algorithm == "Dijkstra":
                    dijkstra()
                elif current_algorithm == "A*":
                    astar()
                elif current_algorithm == "DFS":
                    dfs()
                path_found = True
            elif event.key == pygame.K_g:
                reset_grid()  # Reset the grid when "G" is pressed
                path_found = False

    draw_grid()     # Visualize the grid

    if path_found:
        draw_shortest_path(current_algorithm)   # Visualize the shortest path
found by the algorithm

    pygame.display.flip()   # Update the display

# Quit Pygame
```

```
pygame.quit()
sys.exit()
```