

SWINBURNE UNIVERSITY OF TECHNOLOGY
COS30019

ASSIGNMENT 1
PATH FINDING

NAME & STUDENT ID : Nguyen Gia Binh (104219428)

CLASS : COS30019

LECTURER : PHAM THI KIM DUNG
(dtpham@swin.edu.au)

TUTOR : NGUYEN THE PHUC
(pnguyen2@swin.edu.au)

Hanoi, March 2, 2024

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 1: INSTRUCTION

CHAPTER 2: INTRODUCTION

CHAPTER 3: SEARCH ALGORITHM

- 3.1. Depth First Search
- 3.2. Breath First Search
- 3.3. Greedy Best First Search
- 3.4. A star Search
- 3.5. Weighted A star Search
- 3.6. Iterative Deepening Depth First Search

CHAPTER 4: IMPLEMENTATION

- 4.1. Depth First Search
- 4.2. Breath First Search
- 4.3. Greedy Best First Search
- 4.4. A star Search
- 4.5. Weighted A star Search
- 4.6. Iterative Deepening Depth First Search

CHAPTER5:FEATURE/BUGS/MISSING

CHAPTER 6: RESEARCH

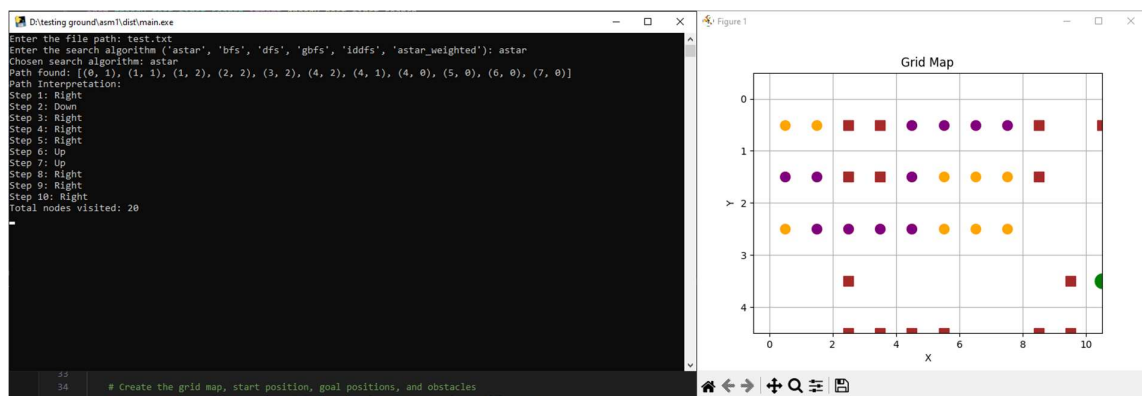
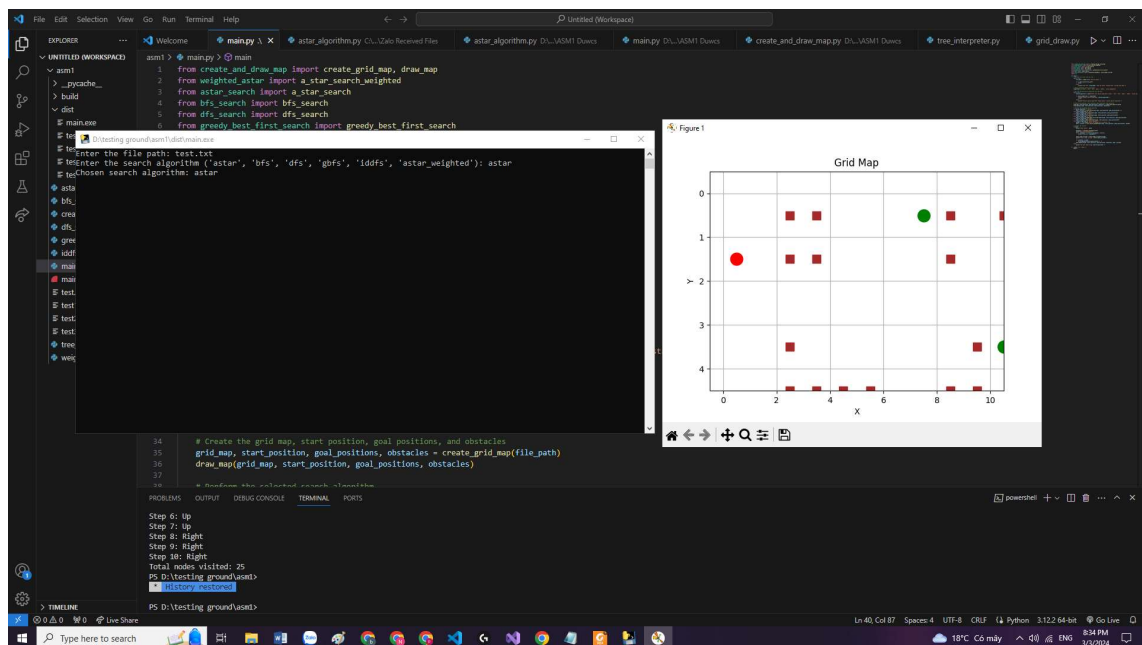
- 6.1. Map drawing
- 6.2. Navigation robot

CHAPTER 7: CONCLUSION

REFERENCES

CHAPTER 1: INSTRUCTION

The main code is in the root folder and the main.exe file is in a folder name “dist”. After the program start running, user need to wait a little bit for it to compile. Next is choosing the file to use as testing and choosing the search algorithm user want to perform, after the user press enter the initial map is shown through a separate window and closing the window that show the map by pressing the “x” symbol at the upper right corner will show the path found (purple) and all the visited node (orange). Not only the map but the required information such as specific path chosen and visited node count will be print out on the Terminal interface as required by the assignment.



CHAPTER 2: INTRODUCTION

The challenge of Robot Navigation involves a customary maze problem, evaluating diverse search algorithms on search trees. Typically, a designated starting point and endpoint are given, and the effectiveness of different search algorithms is assessed in their ability to discover a solution relative to one another.

In the given task, a starting point, destination, and several obstacles, was supplied. The objective was to reconstruct this information into a traversable environment for search algorithms. Subsequently, four distinct search algorithms were to be implemented, along with the option to create two customized search algorithms. These included Depth-First Search, Breadth-First Search, Greedy Best-First Search, and A* Search. The desired outcome involved choosing the test file name, method employed, nodes searched, and the pathway taken. To overcome this challenge, a clear understanding of graph and tree principles was needed. The fundamental notion is that a node may possess multiple descendant nodes, and the expansion of all potential nodes results in a tree graph representing the entirety of the dataset.

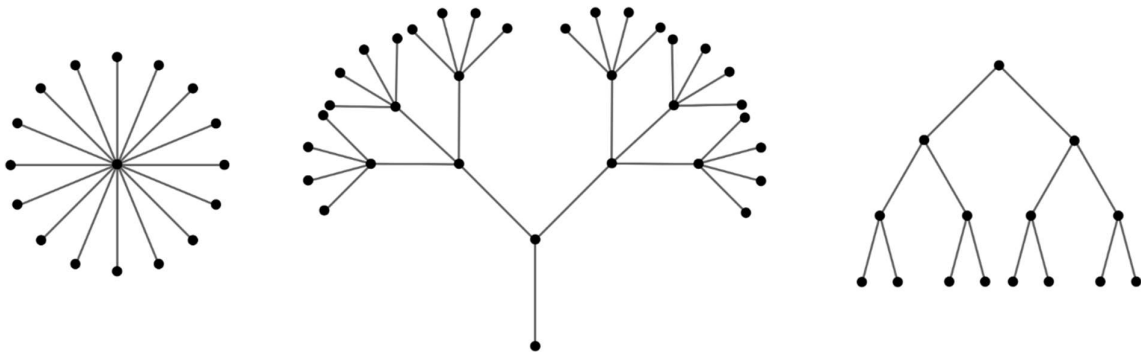
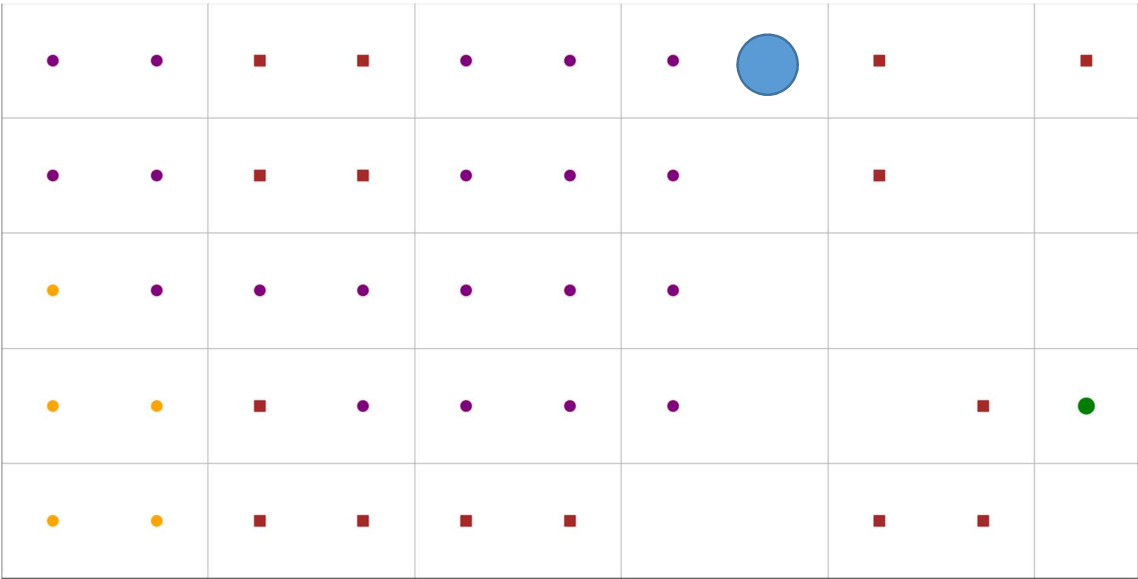


Fig1: Source: <https://ptwiddle.github.io/Graph-Theory-Notes/images/forest.png>

CHAPTER 3: SEARCH ALGORITHM


















































3.1: Depth First Search

In simpler terms, what makes DFS (Depth-First Search) unique is how it searches. It goes as deep as it can along one path in a maze-like structure before going back and trying other paths. It does this by visiting a point, then checking all its unvisited neighbors, one by one, until it finds the target or runs out of options. This method is great for complex problems, but you need to be careful not to get stuck in endless loops.













































3.2: Breath First Search

Unlike DFS which dives deep into one path at a time, BFS takes a broader approach. It's like exploring a neighborhood, checking every house on one street before moving on to the next. This guarantees that BFS thoroughly examines all close options before venturing further out. This strategy makes BFS efficient for finding things likely nearby, like the quickest route in a simple map (where all paths take the same time). BFS is also a popular tool for solving many different problems due to its organized and thorough exploration style.

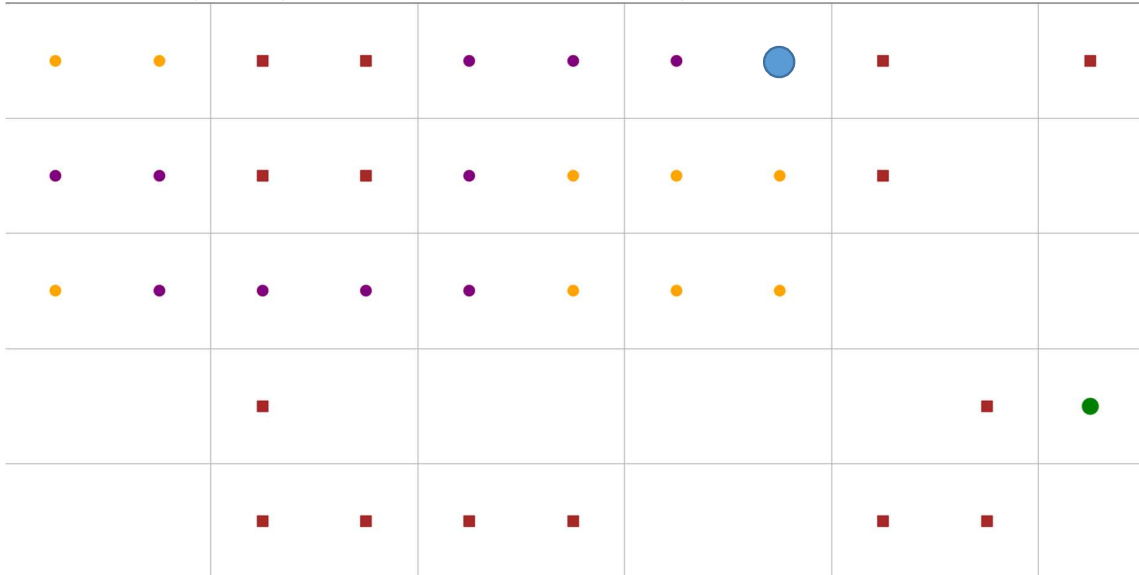
3.3: Greedy Best First Search

In contrast to traditional search methods, Greedy Best-First Search prioritizes exploration based on an educated guess of how close each option is to the goal. It always chooses the path that currently appears most promising for reaching the destination quickly. This approach involves making locally optimal choices at each step, focusing on minimizing the estimated cost to the goal, rather than considering the entire problem structure. While Greedy Best-First Search can be efficient in specific situations, its reliance on estimations can lead to suboptimal solutions if the estimations are inaccurate or fail to capture the complete problem complexity.

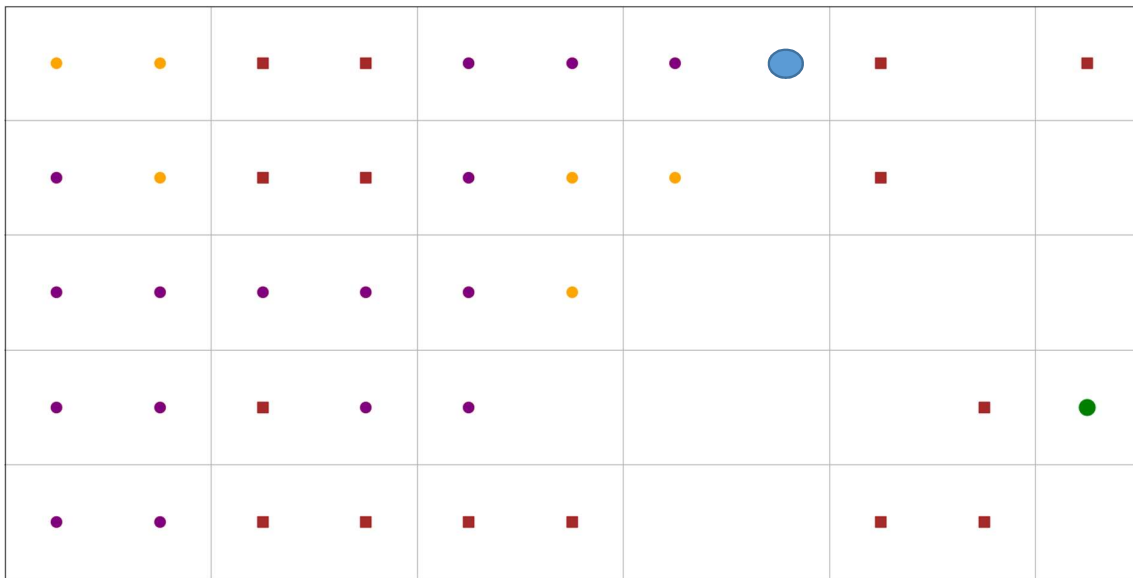
3.4: A star search

A* Search combines the best of both worlds!* It takes into account both how far you've come (like Uniform Cost Search) and how close you seem to be to the goal (like Greedy Best-First Search). It uses a special trick (a heuristic) to guess how promising each option is, then picks the one that looks best overall. This way, A* Search avoids getting stuck on bad paths and efficiently finds the real best route. It's like having a map and a compass in a maze - you know where you've been and where you're likely headed! This makes A* Search a superstar in many situations where you need to find the best path or solution, especially in mazes and other network problems.



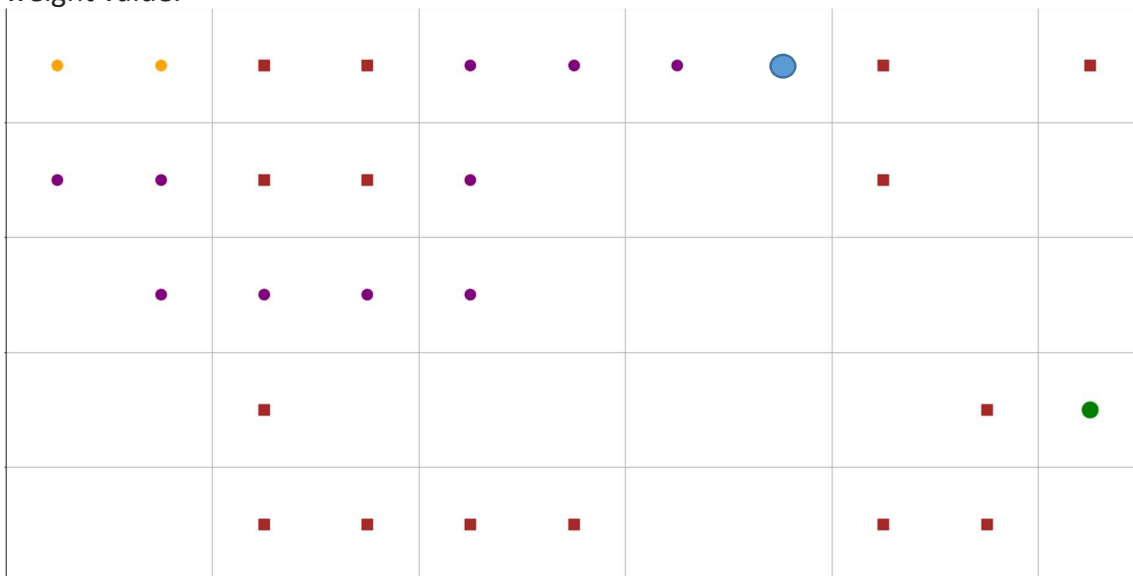
3.5 Iterative Deepening Depth First Search

A hybrid search strategy for unknown depths. This algorithm merges the strengths of two popular search approaches: depth-first search (DFS) and breadth-first search (BFS). It tackles the search space by performing repeated DFS iterations, each with a progressively increasing depth limit. This iterative process allows IDDFS to explore nodes in a DFS manner, minimizing memory usage compared to BFS. Notably, IDDFS is particularly valuable when the depth of the solution within the search tree is unknown. It facilitates a balanced exploration strategy, efficiently utilizing resources while searching for the goal.



3.6 Weighted A star search

Tailoring exploration through a weighted balance.* This variant of A* Search introduces a weight parameter to control the influence of the heuristic function. This weight determines the trade-off between prioritizing the actual cost incurred from the start node ($g(n)$) and the estimated cost to the goal ($h(n)$). A higher weight emphasizes the heuristic, leading the search towards more promising paths based on the estimate, potentially sacrificing optimality for efficiency. Conversely, a lower weight focuses on the actual cost, ensuring a guaranteed optimal solution but potentially requiring more exploration. This flexibility allows Weighted A* Search to adapt to specific application needs, balancing solution quality and computational efficiency based on the chosen weight value.



Overall, I think that A Search often outshines its rivals for the reason follow:

- **Guaranteed best path:** Unlike Depth-First Search (DFS) and Greedy Best-First Search (GBFS), A* always finds the most cost-effective solution, provided the guidance it uses (heuristic) is reliable (admissible). It considers both the actual path taken and an estimate of what's to come.
- **Memory-friendly optimality:** While Breadth-First Search (BFS) is also optimal, it can gobble up memory. A* avoids this by using the heuristic to focus on promising paths first, making it more efficient in this regard.
- **Adaptable to the situation:** A* can adjust its focus on actual cost versus estimated cost depending on the situation, making it more flexible than DFS and GBFS, which can be either less efficient (BFS) or less optimal (DFS, GBFS).
- **Smart exploration:** A* uses the heuristic information to guide its search, striking a balance between exploring new possibilities and exploiting promising ones. This makes it more effective in complex situations.

CHAPTER 4: IMPLEMENTATION

I started by reading the text file containing the information about the map. Based on the details in the file, like the size of the grid, where the starting and ending points are, and where the obstacles are located, I built a representation of the map itself. This is done by parsing the file information, initializes a 2D grid map, and marks the start position, goal positions, and obstacles accordingly. The `draw_map` function is then used to visually represent the grid. The resulting grid map is displayed using matplotlib's plot functions, offering a visual representation of the specified map configuration.

The program prompts the user for a file path and verifies its existence. Next, the user chooses a search algorithm from a predefined list. The chosen algorithm is then used to find a path from the start position to the goal. The program outputs the path, interprets the movements, counts the visited nodes, and displays the final grid map with the path highlighted. If no path is found, a corresponding message is printed.

4.1: Depth First search

1. **Initialization:** The dimensions of the grid, the set of visited positions, and the possible movements (UP, LEFT, DOWN, RIGHT) are established.
2. **Function for Prioritizing Movements:** An internal function is defined to prioritize neighboring positions based on the predefined order of movements.
3. **Recursive Depth-First Search Function:** The core of the algorithm is a recursive function that marks the current position as visited. It checks if the goal is reached; if not, it explores neighboring positions in the prioritized order.

4. DFS Algorithm Execution: The DFS function is initiated with the starting position and an empty path. The result is a tuple containing the path from start to goal and the set of visited positions.

In essence, the DFS algorithm systematically explores the grid, backtracking when necessary, and ultimately provides a path from the start to the goal along with the set of visited positions.

4.2: Breath First Search

1. Initialization: The BFS algorithm utilizes a queue (imported from the collections module) to manage positions in a first-in-first-out manner. A set, visited, is employed to track visited positions.
2. Queue Initialization: The queue is initialized with the starting position and an empty path.
3. BFS Algorithm Execution: The algorithm uses a while loop that continues until the queue is empty. The front node (position and path) is dequeued, and the position is marked as visited. If the goal is reached, the path and visited set are returned. If the current position is not in the visited set, it is added to both visited and the visited_set.
4. Exploring Neighbors: BFS explores neighbors in a predefined order (UP, DOWN, LEFT, RIGHT) using a loop over possible movements. Neighbors are enqueued with an updated path.

4.3: Greedy Best First Search

1. Initialization: The number of rows and columns in the grid is determined. A set, visited, keeps track of visited positions. A priority queue, priority_queue, is initialized. A dictionary, came_from, stores the parent position of each position in the path.
2. Heuristic Function: A Euclidean heuristic function estimates the cost from a position to the goal. The function combines Euclidean distance with an obstacle cost.
3. Sort Key Function: A function, sort_key, determines the sorting order in the priority queue based on heuristic values, movement priorities, and coordinates.
4. Main Loop: The main loop continues while the priority queue is not empty. The current position with the lowest cost is dequeued from the priority queue. If the current position is the goal, the path is reconstructed and returned. If the current position is not visited, it is marked as visited. Neighbors of the current position are obtained and sorted using the sort_key function. Each neighbor is explored, and its cost is added to the priority queue if it is not visited.

5. Reconstructing Path: The `reconstruct_path` function reconstructs the path from the goal to the start using the `came_from` dictionary.
6. Get Neighbors Function: The `get_neighbors` function obtains the neighbors of a position in the grid based on possible movements (UP, LEFT, DOWN, RIGHT).

4.4: A star Search

1. Node Class: Represents a position in the grid with cost and heuristic information. Defines a comparison function (`__lt__`) for ordering in the priority queue based on total cost.
2. Heuristic Function (Manhattan): The `heuristic_cost_estimate` function calculates the Manhattan distance heuristic between the current and goal positions.
3. Reconstruct Path Function: The `reconstruct_path` function reconstructs the path from the start to the current position using the `came_from` dictionary.
4. A Search Algorithm: Initializes data structures, including a priority queue (`open_set`), dictionaries (`came_from` and `g_score`), and a set (`visited_set`). Utilizes a priority queue to store nodes based on their total cost (`g_score + heuristic`). The main loop continues until the priority queue is empty. Dequeues the node with the lowest total cost from the priority queue. If the current position is the goal, the path is reconstructed and returned. Explores neighboring positions and updates the cost if a better path is found. Nodes are added to the priority queue for exploration. Visited nodes are tracked in the `visited_set`.

4.5: Weighted A star search

1. Node Class: Represents a search node with position, cost, and heuristic information. Defines a custom comparison method (`__lt__`) for priority queue ordering based on the weighted total cost.
2. Heuristic Function (Manhattan): The `heuristic_cost_estimate` function calculates the Manhattan distance heuristic between the current and goal positions.
3. Reconstruct Path Function: The `reconstruct_path` function reconstructs the path from the start to the current position using the `came_from` dictionary.
4. Weighted A Search Algorithm:* Initializes data structures, including a priority queue (`open_set`), dictionaries (`came_from` and `g_score`), and a set (`visited_set`). Utilizes a priority queue to store nodes based on their weighted total cost (`g_score + weight * heuristic`). The main loop continues until the priority queue is empty. Dequeues the node with the lowest weighted total cost

from the priority queue. If the current position is the goal, the path is reconstructed and returned. Explores neighboring positions and updates the cost if a better path is found. Nodes are added to the priority queue for exploration. Visited nodes are tracked in the visited_set.

4.6: Iterative Deepening Depth First Search

1. IDDFS Function (iddfs_search): Uses sys.maxsize to represent a large integer value as the initial depth limit. Iterates through depth limits starting from 0 up to max_depth_limit. Calls the dfs_search_with_depth_limit function with the current depth limit. If the goal is found within the depth limit, returns the result and visited set.
2. Depth-Limited DFS Function (dfs_search_with_depth_limit): Takes parameters for the grid, start and goal positions, and the depth limit. Initializes a set, visited, to keep track of visited positions.
3. Recursive DFS Function (dfs): Explores the grid using depth-first search with recursion. Takes parameters for the current position, path, and current depth. Adds the current position to the visited set. If the current position is the goal, returns the path and visited set. If the current depth exceeds the depth limit, stops the DFS exploration. Explores neighbors within the depth limit, recursively calling the function. Returns the result and visited set if the goal is found.
4. Starting the DFS Search: Calls the dfs function to start the DFS search from the start position with an initial depth of 0.

CHAPTER 5: FEATURE/BUGS/MISSING

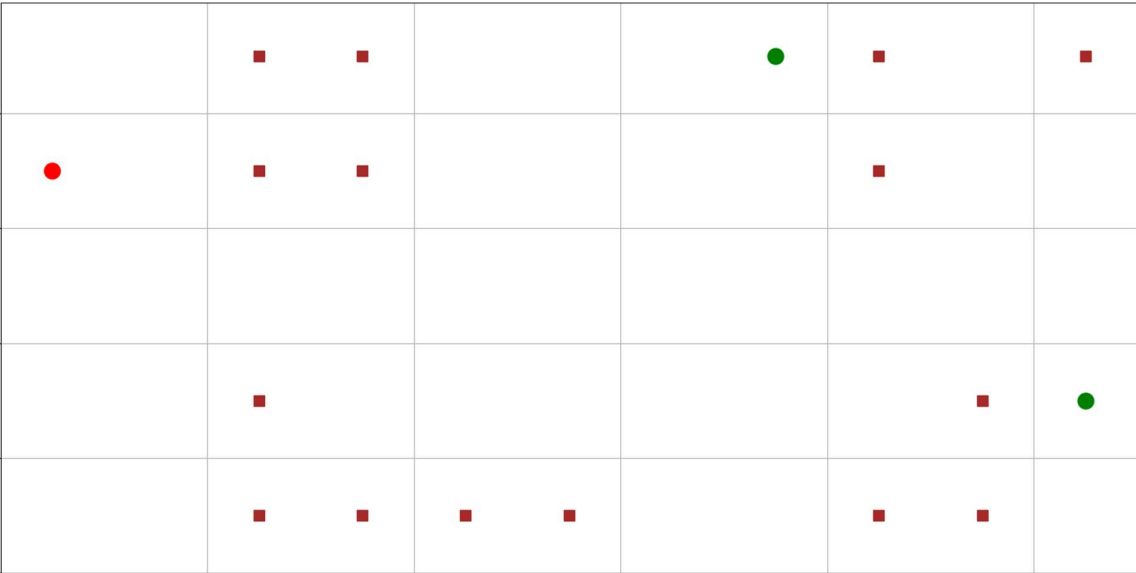
I choose to make it so that the x and y value of the node in path found is shown and then interpret it to up down right left format later.

Next, I only made the map so that it would show the initial map and final result and not showing how the algorithm is traversing the map in real time.

CHAPTER 6: RESEARCH

6.1:Map drawing

My idea for this is map drawing, drawing the map with what node was visited give way to efficient debugging. This visualization is using matplotlib to show start, goal, obstacles,visited node and the path found.



The `create_grid_map` function is the cornerstone of our algorithm, responsible for translating raw map data from an external file into a structured 2D grid. It meticulously parses essential information such as map dimensions, starting and goal positions, and obstacle configurations. This function establishes the foundational grid map, marking the start position as 'R,' goal positions as 'G,' and obstacles as 'B.' Its modular design ensures an accurate representation of the navigational terrain, laying the groundwork for subsequent algorithmic exploration.

The `draw_map` function plays a pivotal role in providing a visual representation of the algorithmic exploration. Employing the Matplotlib library, it systematically plots each cell of the grid with distinct colors and markers. Obstacles are visualized in brown, the start position in red, goal positions in green, and visited cells in orange. The inclusion of purple circles traces algorithmic paths, facilitating a comprehensive understanding of the exploration process. This function's flexibility in adjusting axis limits, grid visibility, and labels enhances its utility in visualizing diverse scenarios.

Example of bfs:

●	●	■	■	●	●	■	■
●	●	■	■	●	●	■	
●	●	●	●	●	●	●	
●	●	■	●	●	●	●	■
●	●	■	■	●	●	■	■

6.2: Navigation robot

Testing demo: <https://youtube.com/shorts/RXkefzLcxus?si=F7BxkDf9K1iKwkL>

Everything is now functioning as expected. The next step is to output the discovered path to a text file, which will then be converted into a motor file by the IoT students. To be honest, this is significantly less work than I initially anticipated.

Since I've already coded the program to store the discovered path as a list and print out the (x,y) coordinates of each node along the optimal path, adding just two more lines of code is all that's required. These lines will specify the file where the program should write the results and write the actual results to the text file.

It's important to note that the robot's movement based on the values computed by my algorithm falls outside the scope of my project and will be handled by the IoT students. The video linked above is a demonstration; while the map displays the correct path, it's solely for monitoring the robot's movement and not part of the final implementation

CHAPTER 7: CONCLUSION

In a maze where only up, down, left, and right movements are allowed, A* (A star) stands out as a superior choice among DFS (Depth-First Search), BFS (Breadth-First Search), and GBFS (Greedy Best-First Search). A* combines the benefits of both informed and uninformed search algorithms, offering a heuristic-driven approach for optimal pathfinding.

A* Search, while a powerful tool, isn't without its limitations. One challenge lies in finding an "admissible" heuristic, a crucial guiding factor for the algorithm. The quality

of this heuristic heavily influences A* Search's performance, and crafting an effective one can be difficult for specific problems. Additionally, the algorithm requires storing information about all explored and potential nodes, leading to potentially high memory usage, especially in resource-constrained environments. Finally, A* Search's efficiency relies on the problem domain itself. If the heuristic provides minimal guidance, the algorithm might perform suboptimally, resembling uninformed search methods. In essence, while A* Search offers significant advantages, its effectiveness is contingent on crafting a good heuristic and the nature of the problem it's tackling.

In the future I would improve my map so that it would show which step is being taken and which step is next as well as which step is in the queue waiting to be visit, all in real time. Additionally, I would try to fix my searching algorithm since some of them are still working not as intended.

REFERENCES

MAS341: Graph theory. Trees. (n.d.). https://ptwiddle.github.io/Graph-Theory-Notes/s_intro_trees.html