

Assignment 1: Tree-based search

Intelligent Agent – Robot Navigation World



Dang khoa Le

Student ID: 103844421

Tutorial Wednesday – 2.20PM

Major: Bachelor of Software Engineer

Table of Contents

[A. Abstract 2](#_Toc162095527)

[B. Introduction 2](#_Toc162095528)

[C. Instruction 3](#_Toc162095529)

[1. Input command lines: 3](#_Toc162095530)

[2. Output command lines: 3](#_Toc162095531)

[D. Map 4](#_Toc162095532)

[E. Implementation 4](#_Toc162095533)

[1. Grid Initialization (class Grid) 4](#_Toc162095534)

[2. Node Initialization (class Node) 5](#_Toc162095535)

[3. Method Obtaining Path 5](#_Toc162095536)

[4. Reading Map Configuration 6](#_Toc162095537)

[5. Search Method Invocation 6](#_Toc162095538)

[F. Algorithms 7](#_Toc162095539)

[1. Uninformed Search Methods 7](#_Toc162095540)

[a. Depth First Search (DFS): 7](#_Toc162095541)

[b. Breadth First Search (BFS): 7](#_Toc162095542)

[c. Custom Search 1 (CUS1): 8](#_Toc162095543)

[2. Informed Search Methods 9](#_Toc162095544)

[a. Greedy Best First Search (GBFS): 9](#_Toc162095545)

[b. A Star (AS): 9](#_Toc162095546)

[c. Custom Search 2 (CUS2): 10](#_Toc162095547)

[G. Heuristic 11](#_Toc162095548)

[1. Manhattan Distance 11](#_Toc162095549)

[2. Euclidean Distance 11](#_Toc162095550)

[3. Code functions 11](#_Toc162095551)

[a. Manhattan Distance Heuristic 11](#_Toc162095552)

[b. Euclidean Distance Heuristic 11](#_Toc162095553)

[H. Testing 12](#_Toc162095554)

[1. Analysis: 13](#_Toc162095555)

[2. Application: 13](#_Toc162095556)

[I. Conclusion 13](#_Toc162095557)

[J. Research Initiatives 14](#_Toc162095558)

[K. Acknowledgements/Resources 14](#_Toc162095559)

[L. References 15](#_Toc162095560)

# A. Abstract

This assignment focuses on creating an intelligent agent – system for robot navigation using various search algorithms, including both uninformed and informed methods, as also using customized methods. The implementation involves algorithms such as Depth First Search (DFS), Breadth First Search (BFS), Greedy Best First Search (GBFS), A\* Search (AS), and the two custom methods Custom Search 1 (CUS1) and Custom Search 2 (CUS2). The goal is to analyse and compare the performance of these algorithms in navigating a map to reach predefined goal coordinates.

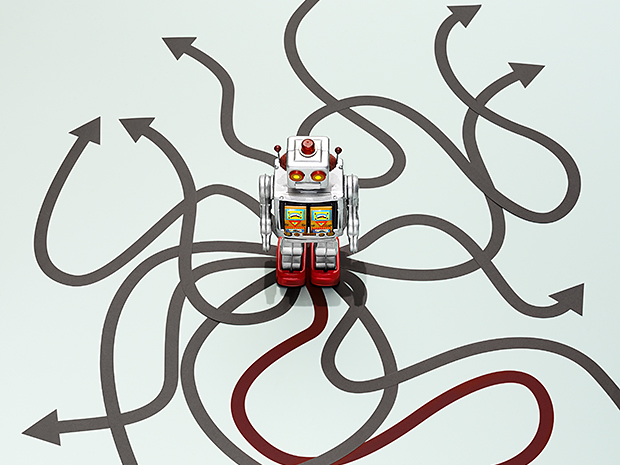
Robot navigation plays a crucial role in various real-world applications, ranging from autonomous vehicles to warehouse logistics and search and rescue operations. In these contexts, the ability to navigate efficiently and safely through complex environments is essential for achieving tasks effectively. By developing intelligent agents capable of navigating autonomously, we can enhance efficiency, reduce human intervention, and improve safety in various domains.

Figure. Illustration for the robot navigation context.

Python was chosen as the core language for implementing the navigation system due to its versatility, ease of use, extensive libraries for data manipulation, algorithm implementation, and visualization. Python's readability and expressiveness make it an ideal choice for prototyping and developing complex algorithms, making it well-suited for tackling the challenges of robot navigation.

# B. Introduction

In uninformed search methods, the agent explores the search space systematically without considering any knowledge about the goal location or the path to reach it. Depth First Search (DFS) traverses as far as possible along each branch before backtracking, while Breadth First Search (BFS) explores all neighbour nodes at the present depth prior to moving on to the nodes at the next depth level. Custom Search 1 (CUS1) is a hybrid approach inspired by DFS and BFS, incorporating depth limitation to improve efficiency.

On the other hand, informed search methods employ heuristic functions to estimate the cost of reaching the goal from a given state. Greedy Best First Search (GBFS) prioritizes expanding nodes that are closest to the goal based on the heuristic value, while A\* Search (AS) evaluates nodes by combining the actual cost to reach them from the start and the estimated cost to reach the goal. Custom Search 2 represents an informed method and an optimised combination between GBFS and A\* method.

In this Robot Navigation design, a GUI display interface is also created as an innovative idea for Research Initiatives, associated with the main components of the navigation algorithms to visualise the navigation of a robot agent through obstacles to reach the goal node on the map.

These search algorithms are fundamental in artificial intelligence and have applications in various domains, including robotics, path planning, and game playing. In the context of map navigation, these algorithms help the agent to efficiently explore the search space and find an optimal or near-optimal path to the goal.

# C. Instruction

## 1. Input command lines:

The main searching code, searchmain.py, allows users to execute different search algorithms on a given map file to navigate a robot from a starting position to a goal location. The usage of the program is straightforward, requiring the execution of a command in the terminal console with the following format:

python searchmain.py [map\_file] [method]

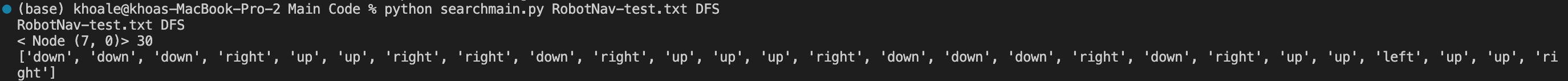
Here, [map\_file] refers to the text file containing the map configuration, and [method] represents the chosen search method.

Figure. Example of an input command using DFS method.

## 2. Output command lines:

If the robot agent successfully navigates to the goal target, the output will be in the following format:

[filename] [method]

[goal] [number\_of\_nodes]

[path]

* Where:

[filename] is the name of the map file.

[method] is the chosen search method.

[goal] is the goal node the search method reached.

[number\_of\_nodes] is the number of nodes the program has created during the search.

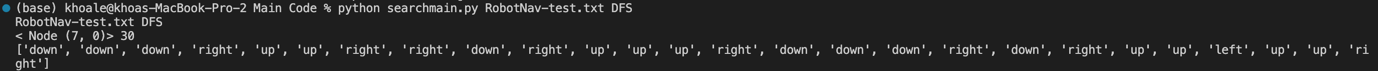
[path] is a sequence of moves in the solution that brings the robot from the start to the end configuration.

Figure. Example of a valid output command using DFS method.

* A black background with white text

  Description automatically generatedIf the goal cannot be reached, the output will be:

[filename] [method]

No goal is reachable; [number\_of\_nodes]

Figure. Example of an invalid output result using AS method.

# D. Map

The map file, "RobotNav-test.txt," serves as the environment in which the robot operates. It provides essential information about the dimensions of the map, the start and goal coordinates, and the locations of walls that serve as obstacles.

By analysing the provided map - text file, we can understand the layout of the environment and the placement of obstacles, which influence the robot's navigation path.

Dimensions: (5 rows and 11 columns) [5,11]

Start coordinates: (0,1).

Goal coordinates: (7,0) and (10,3).

A black background with white text

Description automatically generatedWalls: Defined by tuples indicating the top-left corner coordinates and the width and height of each wall (the remaining lines).

Figure. The “RobotNav-test.txt” file’s data

The map provides obstacles and defines the match starting point and goal locations for the robot's navigation task.

# E. Implementation

## 1. Grid Initialization (class Grid)

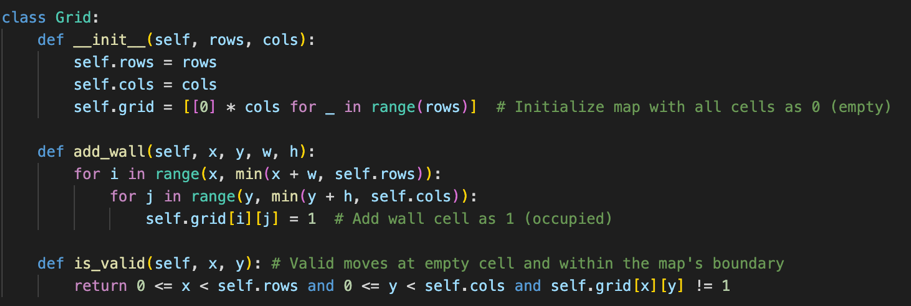
**Description:** The Grid class initializes the map grid, sets walls, and checks the validity of moves within the grid boundaries.

Figure. Code handling the Grid initialization section

**Components:**

\_\_init\_\_(self, rows, cols): Initializes the grid with empty cells represented by 0s.

add\_wall(self, x, y, w, h): Marks cells within a specified rectangular area as walls (occupied cells).

is\_valid(self, x, y): Checks if a move to the specified coordinates is valid within the grid boundaries and not obstructed by a wall.

## 2. Node Initialization (class Node)

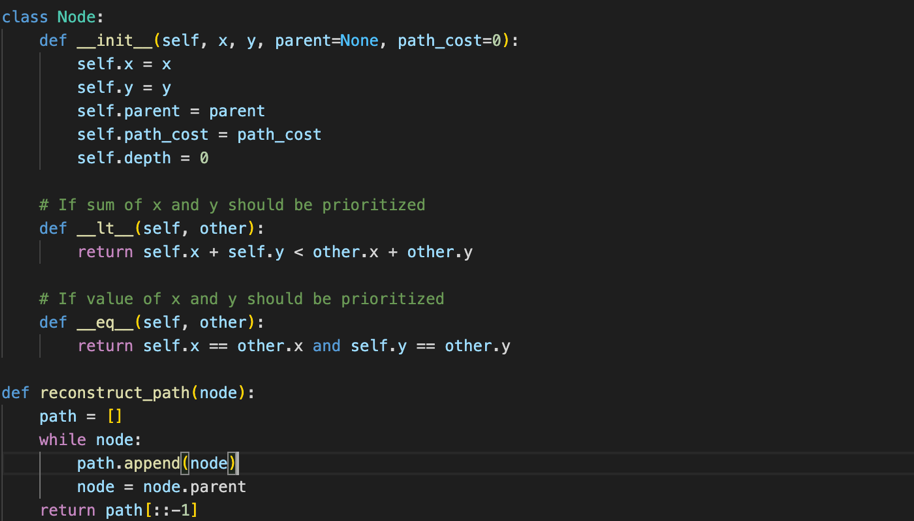
**Description:** The Node class represents a node in the search space, containing information about its position, parent node, path cost, and depth.

Figure. Code handling the Node initialization section (excluding the methods’ algorithms.

**Components:**

\_\_init\_\_(self, x, y, parent=None, path\_cost=0): Initializes a node with its coordinates, parent node, path cost, and depth.

\_\_lt\_\_(self, other): Defines the less than comparison based on the sum of x and y coordinates.

\_\_eq\_\_(self, other): Defines equality based on x and y coordinates.

reconstruct\_path(node): Reconstructs the path from the goal node back to the start node.

## 3. Method Obtaining Path

A screen shot of a computer program

Description automatically generated**Description:** Print out the path (direction as string) from the current node to the next node based on their coordinates.

Figure. Code handling the path (direction) determination.

**Components:**

get\_direction(current, next): Determines the direction ('up', 'down', 'left', 'right') from the current node to the next node. This algorithm is set based on comparing the current and next x and y coordinates.

## 4. Reading Map Configuration

**Description:** Reads the map configuration from a file, including dimensions, start and goal coordinates, and wall positions, then add and construct these data to prepare for the methods’ executions.

**Components:**

Reads map dimensions, start coordinates, and goal coordinates from the file.

Parses wall coordinates and initializes walls in the grid.

Initializes the grid with provided dimensions and walls.

## 5. Search Method Invocation

Figure. Code handling map file reader and constructor.

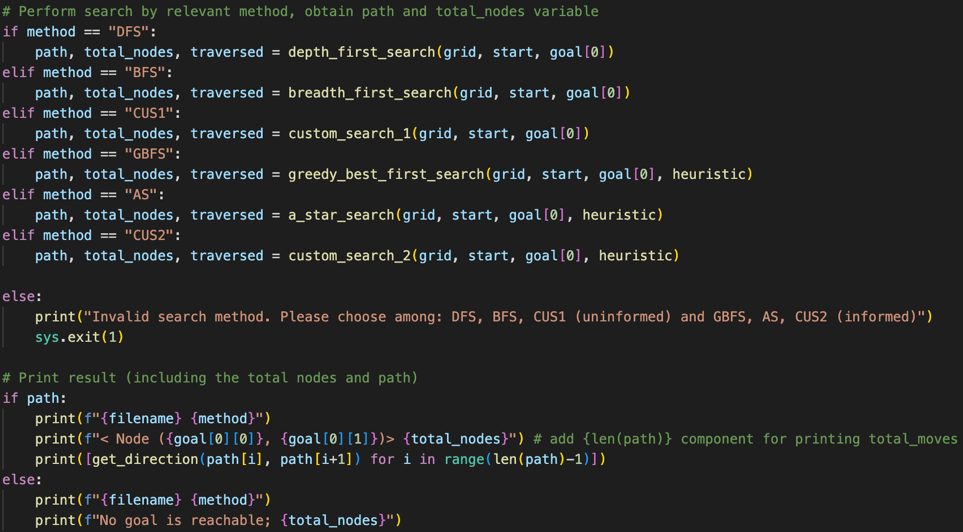
**Description:** Invokes the specified search method (DFS, BFS, CUS1, GBFS, AS, CUS2) based on user input and executes the search algorithm on the initialized grid.

Figure. Code handling method search execution.

**Components:**

Reads user input for the map file name and search method.

Invokes the corresponding search method function based on the input.

Executes the selected search algorithm (method) on the initialized grid with the provided start and goal coordinates.

Prints the result output to the terminal console, including the goal node, total number of nodes explored, traversed nodes’ coordination and the path if a solution is found, else returns as unreachable.

# F. Algorithms

## 1. Uninformed Search Methods:

### a. Depth First Search (DFS):

**Description:** DFS explores as far as possible along each branch before backtracking. It utilizes a stack to store nodes and backtracks when it reaches a dead end.

**Function:**

+ The depth\_first\_search function explores as far as possible along each branch before backtracking.

+ DFS algorithm follows Stack Data structure, and LIFO (Last In First Out) principle.

+ It maintains a stack to store nodes and their respective paths.

+ The algorithm pops nodes from the stack, explores adjacent unvisited nodes, and continues until it reaches the goal or exhausts all possibilities.

+ While searching, it records and returns with total\_nodes explored (integer) and the set of traversed nodes’ coordination.

+ If a dead end is reached, it backtracks by popping nodes from the stack.

+ The algorithm prioritizes deep exploration, potentially leading to long paths.

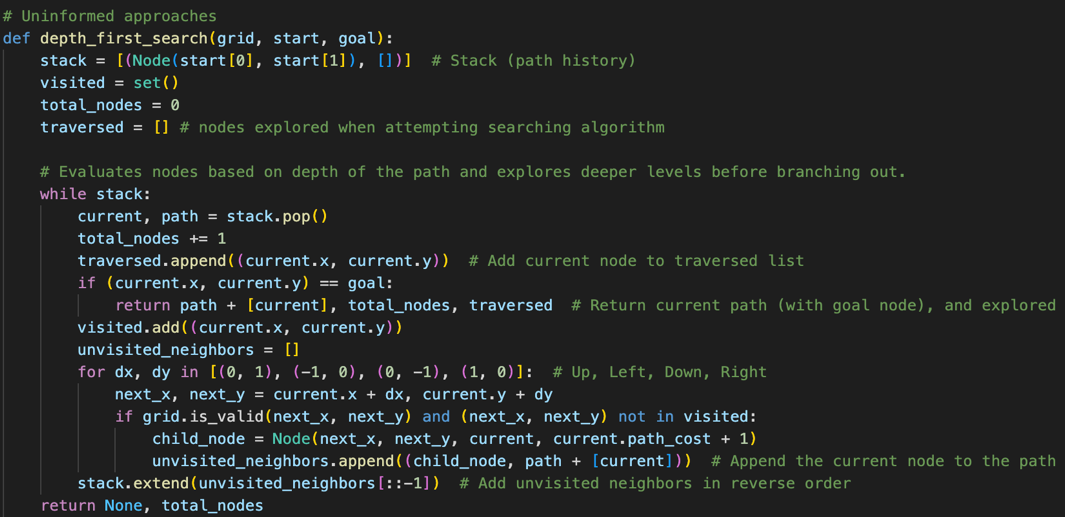
+ The code implementation uses a stack data structure to mimic the recursive nature of DFS, enabling efficient backtracking and exploration of deep branches.

Figure. Code handling DFS search method’s algorithm.

### b. Breadth First Search (BFS):

**Description:** BFS explores all neighbour nodes at the current depth level before moving on to nodes at the next depth level. It uses a queue to store nodes for exploration.

**Function:**

+ The breadth\_first\_search function explores all neighbour nodes at the current depth level before moving on to nodes at the next depth level.

+ BFS algorithm follows Queue Data structure, and FIFO (First In First Out) principle.

+ While searching, it also records and returns with total\_nodes explored (integer) and the set of traversed nodes’ coordination.

+ It uses a queue to store paths and explores nodes level by level.

+ The algorithm starts by enqueueing the start node and continues until it finds the goal node or explores the entire graph.

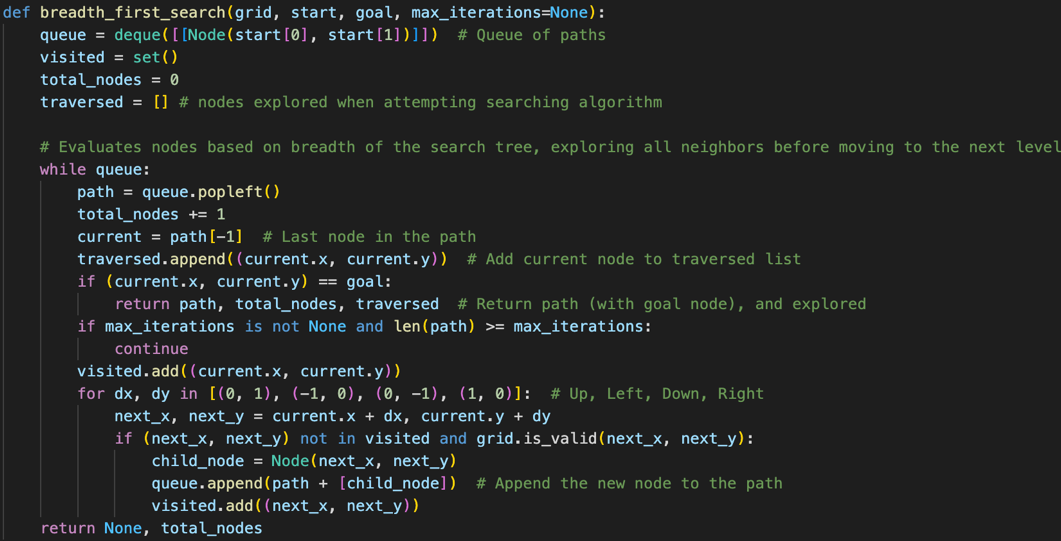
+ The code utilizes a deque (double-ended queue) to efficiently maintain the queue of paths, ensuring that nodes at the current depth level are explored before moving to the next level.

Figure. Code handling BFS search method’s algorithm.

### c. Custom Search 1 (CUS1):

**Description:** CUS1 is a hybrid approach inspired by DFS and BFS. It incorporates depth limitation to improve efficiency by preventing excessive exploration of deep branches.

Function:

+ The custom\_search\_1 function iteratively increases the depth limit until it finds a solution using depth-limited search.

+ While searching, it store and returns with total\_nodes explored (integer) and the set of traversed nodes’ coordination.

+ It iteratively increases the depth limit until a solution is found.

+ The algorithm calls a depth-limited search function with an increasing depth limit.

+ depth\_limited\_search explores nodes up to a specified depth limit, similar to DFS, but avoids excessive exploration of deep branches.

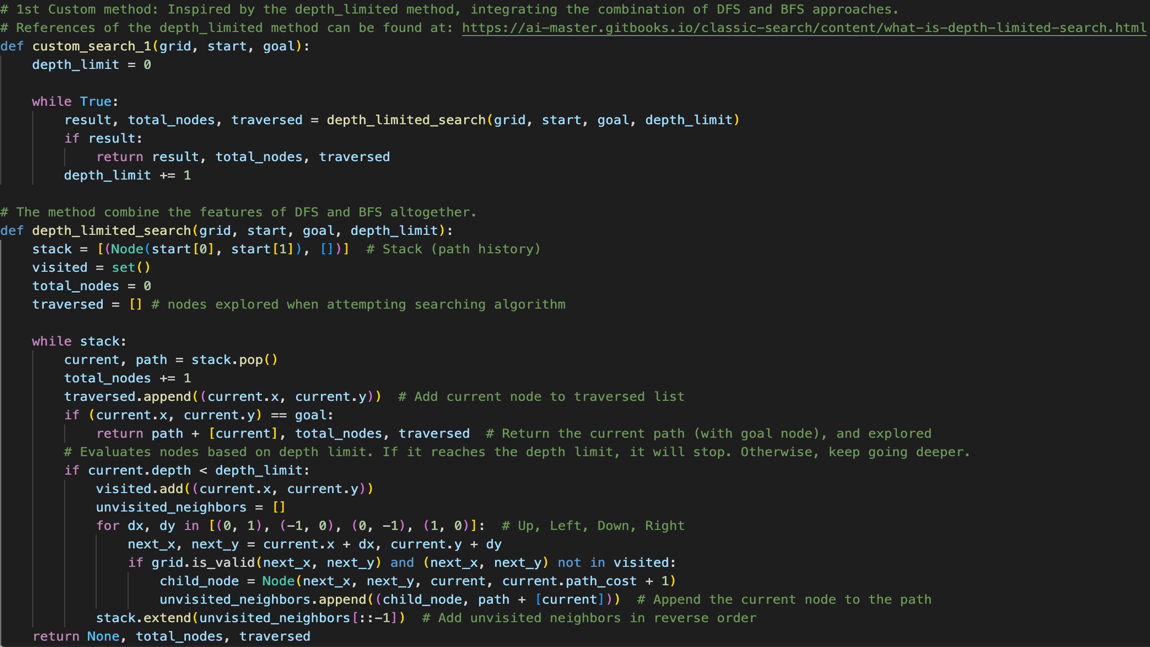
+ The code implementation iterates over increasing depth limits and utilizes the depth-limited search function to explore nodes within each depth limit.

Figure. Code handling CUS1 – custom search method’s algorithm.

## 2. Informed Search Methods:

### a. Greedy Best First Search (GBFS):

**Description:** GBFS prioritizes expanding nodes that are closest to the goal based on the heuristic value. It uses a priority queue to select nodes for expansion.

**Function:**

+ The greedy\_best\_first\_search function takes the grid, start and goal coordinates, and a heuristic function as input.

+ It prioritizes expanding nodes that are closest to the goal based on the heuristic value.

+ It uses a priority queue to select nodes for expansion, where the priority is determined by the heuristic estimate of the distance to the goal.

+ The algorithm expands nodes greedily, always choosing the node with the lowest heuristic value.

+ GBFS may not always find the optimal solution but tends to be more efficient than BFS and DFS in most scenarios, especially with the application of heuristic values.

+ The code implementation maintains a priority queue of nodes based on their heuristic values, allowing efficient selection and expansion of nodes.

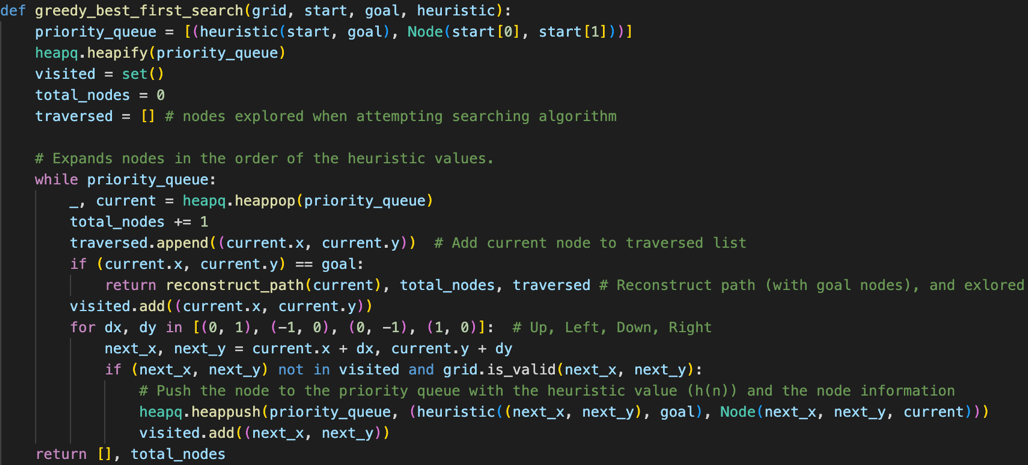
+ Return with total\_nodes explored (integer) and the set of traversed nodes’ coordination.

Figure. Code handling GBFS search method’s algorithm.

### b. A Star (AS):

**Description:** AS evaluates nodes by combining the actual cost to reach them from the start and the estimated cost to reach the goal. It uses a priority queue to prioritize nodes based on their total cost.

**Function:**

+ The a\_star\_search function takes the grid, start and goal coordinates, and a heuristic function as input.

+ A\* evaluates nodes by combining the actual cost to reach them from the start and the estimated cost to reach the goal.

+ It uses a priority queue to prioritize nodes based on their total cost (actual cost + heuristic value).

+ The algorithm guarantees finding the shortest path to the goal in weighted graphs if the heuristic is admissible and consistent.

+ A\* intelligently balances between the cost to reach a node and the estimated cost to reach the goal, efficiently exploring promising paths.

+ It push the node to the priority queue with the total estimated cost: f(n) = g(n) + h(n)

* f(n): The total estimated cost of reaching the goal node from the current node n.
* g(n): The actual cost of reaching node n from the start node.
* h(n): The heuristic estimate of the cost from node n to the goal node.

A computer screen shot of a program

Description automatically generated+ The code implementation maintains a priority queue of nodes based on their total cost, enabling efficient selection and expansion of nodes.

+ Return with total\_nodes explored (integer) and the set of traversed nodes’ coordination.

Figure. Code handling AS search method’s algorithm.

### c. Custom Search 2 (CUS2):

**Description:** CUS2 is a combination of Greedy Best First Search and A\* search methods.

**Function:**

+ The custom\_search\_2 function takes the grid, start and goal coordinates, and a heuristic function as input.

+ As a combining elements of GBFS and A\* search methods, CUS2 aims to leverage their respective strengths.

+ It maintains a priority queue based on both the total cost (f(n) = g(n) + h(n), in similar to A\* method) and the path cost.

+ The algorithm explores nodes based on their total cost and path cost, prioritizing nodes that are both close to the goal and have low path costs.

+ CUS2 may offer a good balance between optimality and efficiency, depending on the problem characteristics.

+ The code implementation combines the priority queue approach of GBFS with the total cost calculation of A\*, allowing for effective exploration of the search space.

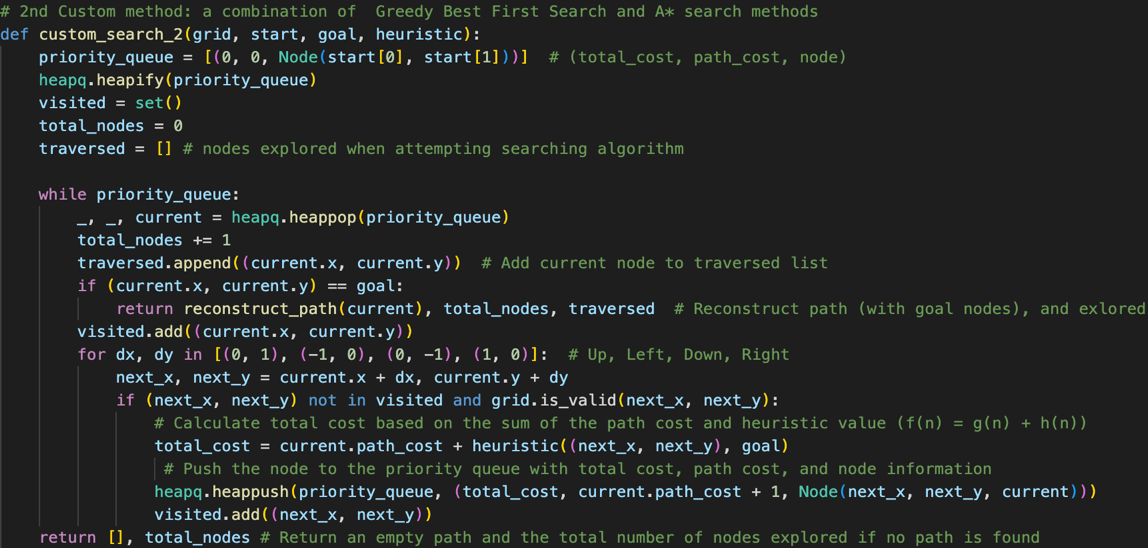
+ Return with total\_nodes explored (integer) and the set of traversed nodes’ coordination.

Figure. Code handling CUS2 - custom search method’s algorithm.

# G. Heuristic

Heuristic distance measures provide estimates of the cost to reach the goal from a given state. In the context of robot navigation, heuristic functions such as Manhattan distance and Euclidean distance are used to estimate the distance between the current state and the goal state.

1. Manhattan Distance

Manhattan Distance: Measures the sum of the horizontal and vertical distances between two points on a grid.

Mathematic function: |x1 - x2|, |y1 - y2|

1. Euclidean Distance

Euclidean Distance: Represents the straight-line distance between two points in Euclidean space.

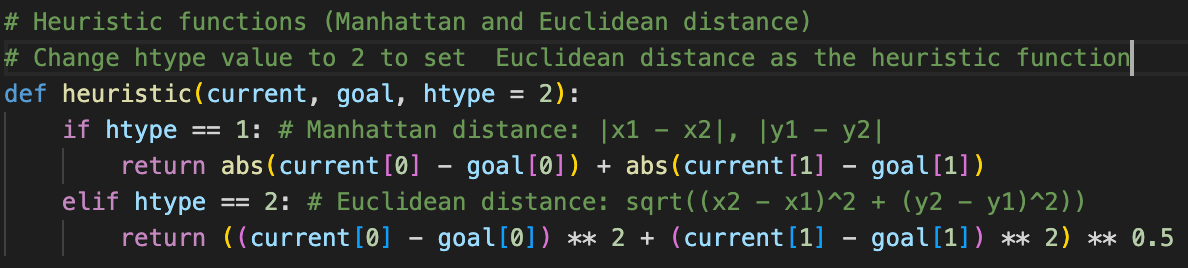
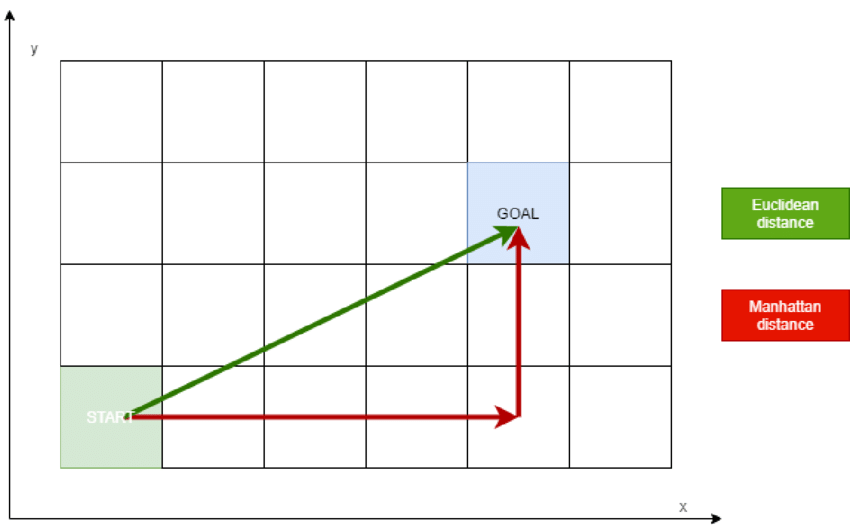
Mathematic function:

Figure. Manhattan and Euclidean Heuristic illustration

Figure. Code handling the Heuristic distance definition.

1. Code functions

**Description:** Provides heuristic functions to estimate the cost from a node to the goal node, used in informed search algorithms (GBFS, AS, CUS2).

**Component:**

+ heuristic(current, goal, htype): Calculates the heuristic value based on the chosen heuristic type (Manhattan or Euclidean distance).

### a. Manhattan Distance Heuristic (htype == 1):

**Formula:** ∣​−​∣+∣​−​∣

**Explanation:**

+abs(current[0] - goal[0]) Absolute difference between the current node's x-coordinate and the goal node's x-coordinate.

+abs(current[1] - goal[1]) Absolute difference between the current node's y-coordinate and the goal node's y-coordinate.

### b. Euclidean Distance Heuristic (htype == 2):

**Formula:**

**Explanation:**((current[0] - goal[0]) \*\* 2 + (current[1] - goal[1]) \*\* 2) \*\* 0.5

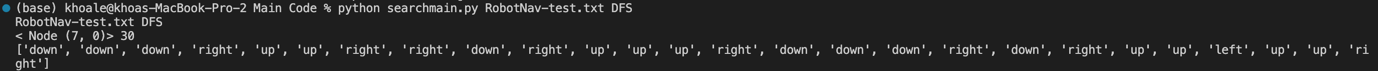
Square root of the sum between ‘Square of the difference between the current node's x-coordinate to the goal node's x-coordinate’ and ‘Square of the difference between the current node's y-coordinate and the goal node's y-coordinate’.

**Hence:** These heuristic functions guide the search algorithms by providing information about the potential cost of reaching the goal, allowing them to make informed decisions about which nodes to explore next.

-> While applying these heuristic formula, perhaps it may takes a sight difference while varying these two heuristic distances to the informed methods?

# H. Testing

The results obtained from running the search algorithms on the "RobotNav-test.txt" map are as the follows:

* **Uninformed methods** (DFS, BFS, CUS1):

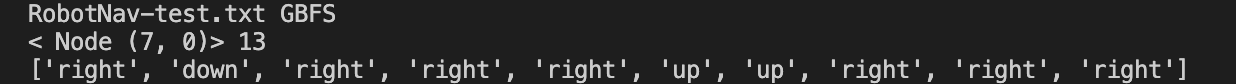
DFS: Reached the goal (7,0) in 30 nodes, traversing a path of length 26.

BFS: Reached the goal (7,0) in 33 nodes, traversing a path of length 9.

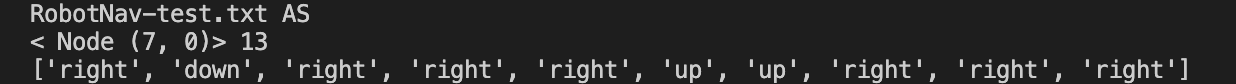
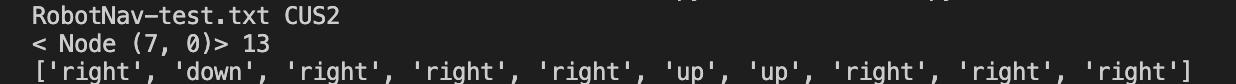
CUS1: Reached the goal (7,0) in 30 nodes, traversing a path of length 26.

**Notice:** Both Manhattan and Euclidean distance approaches make no change to these uninformed methods.

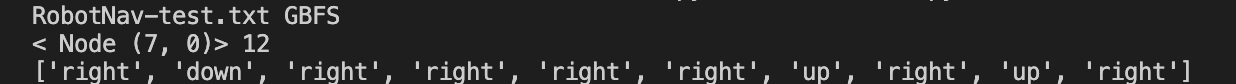
* **Informed methods** (GBFS, AS, CUS2):

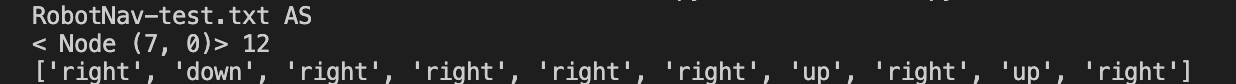
a. Using Manhattan distance:

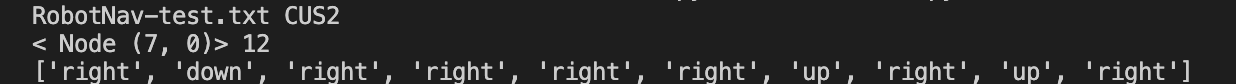
GBFS: Reached the goal (7,0) in 13 nodes, traversing a path of length 8.

AS: Reached the goal (7,0) in 13 nodes, traversing a path of length 8.

CUS2: Reached the goal (7,0) in 13 nodes, traversing a path of length 8.

b. Using Euclidean distance:

GBFS: Reached the goal (7,0) in 12 nodes, traversing a path of length 8.

AS: Reached the goal (7,0) in 12 nodes, traversing a path of length 8.

CUS2: Reached the goal (7,0) in 12 nodes, traversing a path of length 8.

1. Analysis:

DFS (Deep First Search):

**Advantage** - Memory efficient;

**Disadvantage** - Not guaranteed to find the shortest path, can get stuck in infinite loops.

BFS (Breadth First Search):

**Advantage** - Amongst the shortest path;

**Disadvantage** - Memory intensive, may explore unnecessary nodes.

CUS1 (Inspired by Depth Limited method):

**Advantage** - Balances memory usage and path efficiency;

**Disadvantage** - May explore unnecessary nodes in certain scenarios.

GBFS (Greedy Best First Search):

**Advantage** - Efficient memory usage, finds a reasonably short path;

**Disadvantage** - Not guaranteed to find the shortest path, susceptible to local optima.

AS (A Star or A\*):

**Advantage** - Guarantees the shortest path, balances memory usage and path efficiency;

**Disadvantage** - Requires admissible heuristic, may be computationally expensive in large search spaces.

Custom Search 2 (CUS2):

**Advantages** - Optimized combination of GBFS and A\* methods, leveraging strengths of both algorithms. Similar to GBFS, CUS2 utilizes memory efficiently, prioritizing the expansion of nodes based on heuristic values. Like AS, CUS2 guarantees finding the shortest path by considering both the actual cost to reach a node from the start and the estimated cost to reach the goal

**Disadvantages** – Complexity, computationally intensive - may still be computationally expensive in large search spaces as require heuristic evaluation and node prioritization.

1. Application:

Based on the results, BFS is the most optimal uninformed method as it guarantees the shortest path with a reasonable number of nodes.

All informed methods, including GBFS, AS and CUS2 perform equally well in terms of finding the shortest path with minimal nodes.

Furthermore, we may take notice that Euclidean distance is more effective when applying into all informed methods, while only taking 12 nodes to accomplish the task (Manhattan distance takes 13 nodes).

# I. Conclusion

Selecting the most suitable search algorithm for robot navigation depends on factors like environment complexity and available heuristic information.

+ Uninformed methods like BFS are effective when finding any feasible path is the goal, guaranteeing the shortest path but being memory-intensive.

+ Informed methods like Greedy Best First Search (GBFS), A\* (AS) and Custom Search 2 (CUS2) excel when heuristic information is available, offering optimal solutions efficiently.

+ The choice between Manhattan and Euclidean distance heuristics impacts performance, with Euclidean distance often outperforming Manhattan distance.

-> To improve performance, strategies like refining heuristic functions, exploring hybrid approaches, parallelization, and dynamic path planning can be employed. Continued refinement and innovation in algorithm design can further enhance performance and adaptability.

J. Research Initiatives

For the Research Initiatives, a GUI display interface was created, implementing the GUI class in the gui.py file. The GUI display window visualizes the navigation of a robot agent through obstacles (walls) gradually step-by-step to reach the goal node on the map and the total nodes it has explored before locating goad node (traversed).

The GUI class initializes the graphical user interface with essential components such as the grid display, start and goal positions, obstacles (walls), traversed and the path taken by the robot agent to the goal target. Here's a breakdown of the analysis:

## A screen shot of a computer code Description automatically generated 1. Instantiate GUI class:

Figure. Code publish GUI class.

In searchmain.py, GUI class is published (from gui.py) with these components:

+ grid\_instance = Grid(rows, cols): Initializes an instance of the Grid class with only the specified number of rows and columns. By using this instance, it is easier to separate these inheritance data to the main Grid class, which will be used to scale the window in regard to the map’s dimension.

+ if path: This condition checks if the path variable is not None, implying that a path from the start to the goal has been found, only this case it would create the GUI display.

+ app = GUI(grid\_instance, grid, start, goal[0], [tuple((node.x, node.y)) for node in path], traversed): This line instantiates the GUI class with the relative parameters ‘grid\_instance’, ‘grid’ (the actual grid), ‘start’ and ‘goal[0]’ (coordinations), ‘[tuple((node.x, node.y)) for node in path]’ (convert path from list of Node objects to a list of tuples containing the (x, y) coordinates), ‘traversed’ (all explored nodes’ coordination).

+ app.mainloop(): GUI event loop, allowing the window to be displayed and interacted.

## 2. Initialization:

The \_\_init\_\_ method initializes the GUI window with a title and size based on the dimensions of the grid provided using tkinter extension.

It initializes attributes to store information about the grid, start and goal positions, the path, traversed cells and the cell size for drawing and visualization.

It set first step in the path to be displayed.

A canvas widget is created to draw the grid and visualize the map elements.

Figure. Code initialise GUI display.

## 3. Draw Grid and update steps:

The draw\_grid method iterates over each cell in the grid to visualize it based on its content, distinguished by different colours:

+ Start position is represented in red.

+ Goal position is represented in lime.

+ Explored nodes represented in purple.

+ Paths (by each steps) represented in yellow.

+ Walls are represented in dark grey.

+ Empty cells are represented in white.

Figure. Code draw the GUI display.

display\_next\_step method executes gradual updates, presents step-by-step display after 500 ms from the first to the last n-1 (n number of step) of the goal path’s set of steps.

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generatedThe GUI class is instantiated and integrated into the main application (searchmain.py) to display the navigation process. These examples illustrate of how traversed and path nodes differed by search methods and heuristic applications (not the main GUI display).

Figure. GUI display for GBFS, AS, CUS2 methods

Euclidean heuristic.

Figure. GUI display for GBFS, AS, CUS2 methods

Manhattan heuristic.

Figure. GUI display for BFS method.

Figure. GUI display for DFS and CUS1 methods.

# K. Acknowledgements/Resources:

[1] Stuart Russell and Peter Norvig (1995). "Artificial Intelligence: A Modern Approach": Foundational textbook providing insights into search algorithms.

[2] AI Master. "Classic Search": Clear explanations of classic search algorithms.

[3] Python Documentation. "Data Structures and Algorithms": Reference for Python data structures and algorithms.

[4] Amit Patel. “Amit’s Thoughts on Pathfinding: Heuristics”: Insightful articles on pathfinding algorithms and heuristics.

[5] Warren, D. H. D. (1969). “An improved program for tree search. Simon Fraser University”: Presents an improved program for tree-based search, offering valuable insights into search algorithms.

[6] Richard E. Korf (1996). "Artificial Intelligence Search Algorithms": Comprehensive resource delving into various AI search algorithms and their applications.

[7] ] Irfan, M., & Basalamah, S. (2018). "Comparative Analysis of Pathfinding Algorithms”: Research paper comparing A\*, Dijkstra, and BFS algorithms.

[8] GitHub. NMT (2017). “Tkinter 8.5 reference: a GUI for Python”: Method to draw grid and shapes using tkinter extension.

[9] GeeksforGeeks (2021). “Python | after method in Tkinter”: Method to update function after a timeframe.

# L. References:

[1] Stuart Russell and Peter Norvig (1995). "Artificial Intelligence: A Modern Approach". Retrieved from http://repo.darmajaya.ac.id/4836/1/Stuart%20Russell%2C%20Peter%20Norvig-Artificial%20Intelligence\_%20A%20Modern%20Approach-Prentice%20Hall%20%28%20PDFDrive%20%29.pdf

[2] AI Master. "Classic Search". Retrieved from https://ai-master.gitbooks.io/classic-search/content/what-is-depth-limited-search.html

[3] Python Documentation. "Data Structures and Algorithms". Retrieved from https://docs.python.org/3/tutorial/datastructures.html

[4] Amit Patel. “Amit’s Thoughts on Pathfinding: Heuristics”. Retrieved from https://theory.stanford.edu/~amitp/GameProgramming/

[5] Warren, D. H. D. (1969). “An improved program for tree search. Simon Fraser University”. Retrieved from http://www.sfu.ca/~arashr/warren.pdf?fbclid=IwAR2D0y8Xgn1F0Sjk2NwrrErAG5tlgorQZXLHIN57C3ZkyTppua\_BCRrjvCU

[6] Richard E. Korf (1996). "Artificial Intelligence Search Algorithms". Retrieved from <https://dl.acm.org/doi/10.5555/1882723.1882745>

[7] Irfan, M., & Basalamah, S. (2018). “Comparative Analysis of Pathfinding Algorithms: A. Dijkstra and B. BFS on Maze Runner Game. ResearchGate”. Retrieved from

<https://www.researchgate.net/publication/325368698_Comparative_Analysis_of_Pathfinding_Algorithms_A_Dijkstra_and_BFS_on_Maze_Runner_Game>

[8] GitHub. NMT (2017). “Tkinter 8.5.” Retrieved from: https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/create\_polygon.html

[9] GeeksforGeeks (2021). “Python | after method in Tkinter”. Retrieved from: <https://www.geeksforgeeks.org/python-after-method-in-tkinter/>