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

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.

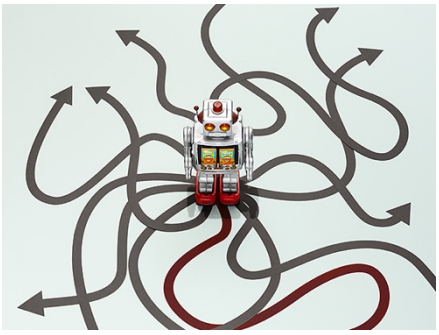

Figure. Illustration for the robot navigation context.

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.

```
(base) khoale@khoas-MacBook-Pro-2 Main Code % python searchmain.py RobotNav-test.txt DFS
```

Figure. Example of an input command using DFS method.

Note: The GUI display will automatically run without additional input command.

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.

```
RobotNav-test.txt DFS
< Node (7, 0)> 30
['down', 'down', 'down', 'right', 'up', 'up', 'right', 'right', 'down', 'right', 'up', 'up', 'up', 'right', 'down', 'down', 'down', 'right', 'down', 'right', 'up', 'up', 'left', 'up', 'up', 'ri
ght']
```

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

- If the goal cannot be reached, the output will be:

[filename] [method]
No goal is reachable; [number_of_nodes]

```
RobotNav-test.txt AS
No goal is reachable; 40
```

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

D. 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.

```
class Grid:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.grid = [[0] * cols for _ in range(rows)]  # Initialize map with all cells as 0 (empty)

    def add_wall(self, x, y, w, h):
        for i in range(x, min(x + w, self.rows)):
            for j in range(y, min(y + h, self.cols)):
                self.grid[i][j] = 1  # Add wall cell as 1 (occupied)

    def is_valid(self, x, y): # Valid moves at empty cell and within the map's boundary
        return 0 <= x < self.rows and 0 <= y < self.cols and self.grid[x][y] != 1
```

Figure. Code handling the Grid initialization section

Components:

__init__(self, rows, cols): Initializes rows, cols and 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 cell by 1).
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.

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.

```python
class Node:
    def __init__(self, x, y, parent=None, path_cost=0):
        self.x = x
        self.y = y
        self.parent = parent
        self.path_cost = path_cost
        self.depth = 0

    # If sum of x and y should be prioritized
    def __lt__(self, other):
        return self.x + self.y < other.x + other.y

    # If value of x and y should be prioritized
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

def reconstruct_path(node):
    path = []
    while node:
        path.append(node)
        node = node.parent
    return path[::-1]
```

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

__eq__(self, other): Defines equality based on x and y coordinates.

reconstruct_path(node): Reconstructs the path from the goal node to the start node for invalid path.

3. Method Obtaining Path

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

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.

```python
# Return direction taken from action
def get_direction(current, next):
    if current.y < next.y:
        return 'down'
    elif current.y > next.y:
        return 'up'
    elif current.x < next.x:
        return 'right'
    elif current.x > next.x:
        return 'left'
    else:
        return 'stay' # This scenario will not happen!
```

Figure. Code handling the path (direction) determination.

4. Reading Map Configuration

Description: Reads the map configuration from a file, including dimensions (first line 'lines[0]'), start (second line 'lines[1]'), goal coordinates (third line 'lines[2]'), and wall positions (rest of the lines 'lines[3:]'), then add and construct these data to prepare for the methods' executions.

Components:

Reads map dimensions, start coordinates, and goal coordinates from the file with the specified line configuration.

Breakdown reading map function, for instance:

```python
re.match(r'\[(\d+),\s*(\d+)\]', lines[0])
```

-> It look for a string that starts with '[', followed by one or more digits '\d+', followed by a comma ',' followed by whitespace '\s*', followed by one or more digits '\d+', and ends with ']'. lines[0] indicates it reads the first line from the file.

Parses wall coordinates and initializes walls in the grid.

Initializes the grid with provided dimensions and walls.

```python
# Read map configuration from file
with open(filename, 'r') as file:
    lines = [line.strip() for line in file.readlines()]
    # Read map's dimensions
    dimension_match = re.match(r'\[(\d+),\s*(\d+)\]', lines[0])
    cols, rows = map(int, (dimension_match.group(1), dimension_match.group(2)))
    # Match's start coordinates
    start_match = re.match(r'\((\d+),\s*(\d+)\)', lines[1])
    start = (int(start_match.group(1)), int(start_match.group(2)))
    # Extract goal coordinates
    goal_coords = re.findall(r'\((\d+),\s*(\d+)\)', lines[2])
    goal = []
    for x_str, y_str in goal_coords:
        try:
            x, y = map(int, (x_str, y_str))
            goal.append((x, y))
        except ValueError:
            print(f"Invalid coordinate format: ({x_str}, {y_str}).")
    if not goal:
        print("No valid goal node found.")
        sys.exit(1)

    # Initialize the wall
    walls = []
    for wall in lines[3:]:
        wall_coords = re.findall(r'\d+', wall)
        if len(wall_coords) == 4:  # Ensure correct wall's coordinates as template
            walls.append(tuple(map(int, wall_coords)))

    # Initialize the grid
    grid = Grid(rows, cols)
    for wall in walls:
        grid.add_wall(*wall)
```

Figure. Code handling map file reader and constructor.

5. Search Method Invocation

Description: Invokes the specified search method (DFS, BFS, CUS1, GBFS, AS, CUS2) based on user

```
# Perform search by relevant method, obtain path and total_nodes variable
if method == "DFS":
    path, total_nodes, traversed = depth_first_search(grid, start, goal[0])
elif method == "BFS":
    path, total_nodes, traversed = breadth_first_search(grid, start, goal[0])
elif method == "CUS1":
    path, total_nodes, traversed = custom_search_1(grid, start, goal[0])
elif method == "GBFS":
    path, total_nodes, traversed = greedy_best_first_search(grid, start, goal[0], heuristic)
elif method == "AS":
    path, total_nodes, traversed = a_star_search(grid, start, goal[0], heuristic)
elif method == "CUS2":
    path, total_nodes, traversed = custom_search_2(grid, start, goal[0], heuristic)

else:
    print("Invalid search method. Please choose among: DFS, BFS, CUS1 (uninformed) and GBFS, AS, CUS2 (informed)")
    sys.exit(1)

# Print result (including the total nodes and path)
if path:
    print(f"{filename} {method}")
    print(f"< Node ({goal[0][0]}, {goal[0][1]})> {total_nodes}") # add {len(path)} component for printing total_moves
    print([get_direction(path[i], path[i+1]) for i in range(len(path)-1)])
else:
    print(f"{filename} {method}")
    print(f"No goal is reachable; {total_nodes}")
```

Figure. Code handling DFS search method's algorithm.

input and executes the search algorithm on the initialized grid.

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.
+ For a valid search method command, it returns the path (coordination), total_nodes explored (number), traversed nodes (coordination). Then, it prints the result output to the terminal console, including the filename and method as request, goal node reached, total number of nodes explored, and the path (get_direction by 'up', 'down', 'left', 'right' from path's coordination that neglect the node of the starting point) if a solution is found, else returns as unreachable (with message).

Notice: Code explanation can be found furthermore in the actual code file.

E. Search 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 (number) 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.

```python
def depth_first_search(grid, start, goal): # LIFO approach
    stack = [(Node(start[0], start[1]), [])]  # Stack (path history)
    visited = set()
    total_nodes = 0 # number of nodes expanded during search
    traversed = [] # nodes explored when attempting searching algorithm

    # Evaluates nodes based on depth of the path and explores deeper levels before branching out.
    while stack:
        current, path = stack.pop() # popping a tuple (current, path) from the stack, retrieves both the current node and the path leading to it.
        total_nodes += 1
        traversed.append((current.x, current.y))  # Add current node to traversed list
        if (current.x, current.y) == goal:
            return path + [current], total_nodes, traversed  # Return current path (with goal node), and explored
        visited.add((current.x, current.y)) # Add visited node, avoid duplication
        unvisited_neighbors = []
        for dx, dy in [(0, -1), (-1, 0), (0, 1), (1, 0)]:  # Up, Left, Down, Right
            next_x, next_y = current.x + dx, current.y + dy # attempt moves in Up-Left-Down-Right
            if grid.is_valid(next_x, next_y) and (next_x, next_y) not in visited: # Check valid next cell and visited state
                child_node = Node(next_x, next_y, current, current.path_cost + 1) # Create Node object child_node with neighboring, current node,
                unvisited_neighbors.append((child_node, path + [current]))  # Append the current node to the path
        stack.extend(unvisited_neighbors[::-1])  # Add unvisited neighbors in reverse order
    return None, total_nodes # Return an empty path and the total number of nodes explored if no path is found
```

Figure. Code handling method search execution.

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 (number) 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.

```python
def breadth_first_search(grid, start, goal, max_iterations=None): # FIFO approach
    queue = deque([[Node(start[0], start[1])]])  # Queue of paths
    visited = set()
    total_nodes = 0 # number of nodes expanded during search
    traversed = [] # nodes explored when attempting searching algorithm

    # Evaluates nodes based on breadth of the search tree, exploring all neighbors before moving to the next level.
    while queue:
        path = queue.popleft()
        total_nodes += 1
        current = path[-1]  # Last node in the path
        traversed.append((current.x, current.y))  # Add current node to traversed list
        if (current.x, current.y) == goal:
            return path, total_nodes, traversed  # Return path (with goal node), and explored
        if max_iterations is not None and len(path) >= max_iterations:
            continue
        visited.add((current.x, current.y)) # Add visited node, avoid duplication
        for dx, dy in [(0, -1), (-1, 0), (0, 1), (1, 0)]:  # Up, Left, Down, Right
            next_x, next_y = current.x + dx, current.y + dy # attempt moves in Up-Left-Down-Right
            if (next_x, next_y) not in visited and grid.is_valid(next_x, next_y): # Check valid next cell and visited state
                child_node = Node(next_x, next_y) # Create Node object child_node with neighboring node.
                queue.append(path + [child_node])  # Append the new node to the path
                visited.add((next_x, next_y)) # Add visited node
    return None, total_nodes # Return an empty path and the total number of nodes explored if no path is found
```

Figure. Code handling BFS search method's algorithm.

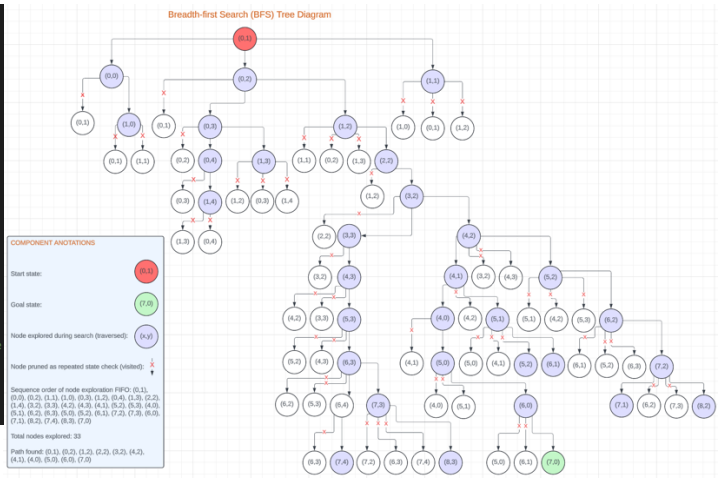

Figure. Tree based diagram for BFS method's node expansion (including notations). RobotNav-test.txt

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 (number) 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.

```python
def custom_search_1(grid, start, goal):
    depth_limit = 0

    while True:
        result, total_nodes, traversed = depth_limited_search(grid, start, goal, depth_limit)
        if result:
            return result, total_nodes, traversed
        depth_limit += 1

# The method combine the features of DFS and BFS altogether.
def depth_limited_search(grid, start, goal, depth_limit):
    stack = [(Node(start[0], start[1]), [])]  # Stack (path history)
    visited = set()
    total_nodes = 0 # number of nodes expanded during search
    traversed = [] # nodes explored when attempting searching algorithm

    while stack:
        current, path = stack.pop()
        total_nodes += 1
        traversed.append((current.x, current.y))  # Add current node to traversed list
        if (current.x, current.y) == goal:
            return path + [current], total_nodes, traversed  # Return the current path (with goal node), and explored
        # Evaluates nodes based on depth limit. If it reaches the depth limit, it will stop. Otherwise, keep going deeper.
        if current.depth < depth_limit:
            visited.add((current.x, current.y)) # Add visited node, avoid duplication
            unvisited_neighbors = [] # set list of unvisited neighboring cells
            for dx, dy in [(0, -1), (-1, 0), (0, 1), (1, 0)]:  # Up, Left, Down, Right
                next_x, next_y = current.x + dx, current.y + dy # attempt moves in Up-Left-Down-Right
                if grid.is_valid(next_x, next_y) and (next_x, next_y) not in visited: # Check valid next cell and visited state
                    child_node = Node(next_x, next_y, current, current.path_cost + 1) # Create Node object child_node with neighboring, current node
                    unvisited_neighbors.append((child_node, path + [current]))  # Append the current node to the path
            stack.extend(unvisited_neighbors[::-1])  # Add unvisited neighbors in reverse order
    return None, total_nodes, traversed # Return an empty path, total number of nodes explored and traversed if no path is found
```

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.

```python
def greedy_best_first_search(grid, start, goal, heuristic):
    priority_queue = [(heuristic(start, goal), Node(start[0], start[1]))]
    heapq.heapify(priority_queue)
    visited = set()
    total_nodes = 0 # number of nodes expanded during search
    traversed = [] # nodes explored when attempting searching algorithm

    # Expands nodes in the order of the heuristic values.
    while priority_queue:
        _, current = heapq.heappop(priority_queue)
        total_nodes += 1
        traversed.append((current.x, current.y))  # Add current node to traversed list
        if (current.x, current.y) == goal:
            return reconstruct_path(current), total_nodes, traversed # Reconstruct path (with goal nodes), and exlored
        visited.add((current.x, current.y)) # Add visited node, avoid duplication
        for dx, dy in [(0, -1), (-1, 0), (0, 1), (1, 0)]:  # Up, Left, Down, Right
            next_x, next_y = current.x + dx, current.y + dy # attempt moves in Up-Left-Down-Right
            if (next_x, next_y) not in visited and grid.is_valid(next_x, next_y): # Check valid next cell and visited state
                # Push the node to the priority queue with the heuristic value (h(n)) and the node information
                heapq.heappush(priority_queue, (heuristic((next_x, next_y), goal), Node(next_x, next_y, current)))
                visited.add((next_x, next_y))
    return [], total_nodes # Return an empty path and the total number of nodes explored if no path is found
```

Figure. Code handling GBFS search method's algorithm.

Function:

+ The greedy_best_first_search function takes the grid, start and goal coordinates, and a heuristic function as input.
+ 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 f(n) = h(n); h(n) is the heuristic cost estimate from node n to the goal node. heuristic((next_x, next_y), 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 (number) and the set of traversed nodes' coordination.

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.

```python
def a_star_search(grid, start, goal, heuristic):
    priority_queue = [(0, heuristic(start, goal), Node(start[0], start[1]))]
    heapq.heapify(priority_queue)
    visited = set()
    total_nodes = 0 # number of nodes expanded during search
    traversed = [] # nodes explored when attempting searching algorithm

    # Evaluates nodes based on both cost to reach the node and the heuristic estimate of the cost from current to goal node.
    while priority_queue:
        _, _, current = heapq.heappop(priority_queue)
        total_nodes += 1
        traversed.append((current.x, current.y))  # Add current node to traversed list
        if (current.x, current.y) == goal:
            return reconstruct_path(current), total_nodes, traversed # Reconstruct path (with goal nodes), and explored
        visited.add((current.x, current.y)) # Add visited node, avoid duplication
        for dx, dy in [(0, -1), (-1, 0), (0, 1), (1, 0)]:  # Up, Left, Down, Right
            next_x, next_y = current.x + dx, current.y + dy # attempt moves in Up-Left-Down-Right
            if (next_x, next_y) not in visited and grid.is_valid(next_x, next_y): # Check valid next cell and visited state
                # Push the node to the priority queue with the total estimated cost (f(n) = g(n) + h(n))
                heapq.heappush(priority_queue, (heuristic(start, goal) + heuristic((next_x, next_y), goal),
                                                heuristic((next_x, next_y), goal), Node(next_x, next_y, current)))
                visited.add((next_x, next_y))
    return [], total_nodes, traversed  # Return an empty path, total nodes explored and traversed if no path is found
```

Figure. Code handling AS search method's algorithm.

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. heuristic(start, (next_x, next_y))
  - h(n): The heuristic cost estimate from node n to the goal node. heuristic((next_x, next_y), goal)
+ 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 (number) and the set of traversed nodes' coordination.

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 at default, however, when g(n) value is greater than h(n), it will switch to GBFS approach as f(n) = h(n) to avoid redundant node exploration. Hence, set take_cost correspondingly.
+ The algorithm explores nodes based on the take_cost (valued as g(n) + h(n) or h(n) only), 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 (number) and the set of traversed nodes' coordination.

```python
def custom_search_2(grid, start, goal, heuristic):
    priority_queue = [(0, 0, Node(start[0], start[1]))]  # (total_cost, path_cost, node)
    heapq.heapify(priority_queue)
    visited = set()
    total_nodes = 0 # number of nodes expanded during search
    traversed = [] # nodes explored when attempting searching algorithm

    while priority_queue:
        _, _, current = heapq.heappop(priority_queue)
        total_nodes += 1
        traversed.append((current.x, current.y))  # Add current node to traversed list
        if (current.x, current.y) == goal:
            return reconstruct_path(current), total_nodes, traversed  # Reconstruct path (with goal nodes), and exlored
        visited.add((current.x, current.y)) # Add visited node, avoid duplication
        for dx, dy in [(0, -1), (-1, 0), (0, 1), (1, 0)]:  # Up, Left, Down, Right
            next_x, next_y = current.x + dx, current.y + dy # attempt moves in Up-Left-Down-Right
            if (next_x, next_y) not in visited and grid.is_valid(next_x, next_y): # Check valid next cell and visited state
                # Calculate total cost based on the sum of the path cost and heuristic value (f(n) = g(n) + h(n))
                total_cost = current.path_cost + heuristic((next_x, next_y), goal)
                if heuristic(start, (next_x, next_y)) > heuristic((next_x, next_y), goal): # If g(n) > h(n)
                    take_cost = heuristic((next_x, next_y), goal) # If f(n) = h(n): Same as GBFS
                else:
                    take_cost = total_cost # Else, (f(n) = g(n) + h(n)): Same as AS
                # Push the node to the priority queue with the take_cost, path cost, and node information
                heapq.heappush(priority_queue, (take_cost, current.path_cost, Node(next_x, next_y, current)))
                visited.add((next_x, next_y))
    return [], total_nodes, traversed  # Return an empty path and the total number of nodes explored if no path is found
```

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

F. 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 equation: |x1 - x2|, |y1 - y2|

2. Euclidean Distance

Euclidean Distance: Represents the straight-line distance between two points in Euclidean space.
Mathematic equation: $\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$

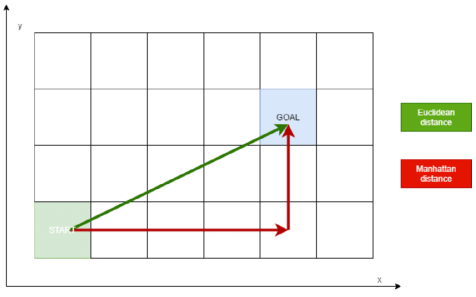

Figure. Manhattan and Euclidean Heuristic illustration

```python
# Heuristic functions (Manhattan and Euclidean distance)
# Change htype value to 2 to set  Euclidean distance as the heuristic function
def heuristic(current, goal, htype = 2):
    if htype == 1: # Manhattan distance: |x1 - x2|, |y1 - y2|
        return abs(current[0] - goal[0]) + abs(current[1] - goal[1])
    elif htype == 2: # Euclidean distance: sqrt((x2 - x1)^2 + (y2 - y1)^2))
        return ((current[0] - goal[0]) ** 2 + (current[1] - goal[1]) ** 2) ** 0.5
```

Figure. Code handling the Heuristic distance definition.

3. 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: $|x_{current} - x_{goal}| + |y_{current} - y_{goal}|$

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: $\sqrt{(x_{goal} - x_{current})^2 + (y_{goal} - y_{current})^2}$

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?

G. Testing

Below are 10 experimental test cases (with the sample RobotNav-test.txt map), which are 10 custom map files generated to test the agent's algorithms under different environmental conditions, providing output result with estimated processing time (from GUI display) by each. Hence, conclude.

Method (with heuristic)	Output Result	Approx Recorded Time (from GUI motion)
Test map: RobotNav-test.txt (Sample test)		
DFS (Depth-first search)	< Node (7, 0)> 28 ['up', 'right', 'down', 'down', 'right', 'right', 'down', 'right', 'up', 'up', 'up', 'right', 'down', 'down', 'down', 'right', 'up', 'up', 'up', 'right']	7.6 seconds
BFS (Breadth-first search)	< Node (7, 0)> 33 ['down', 'right', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']	7.6 seconds
CUS1 (Depth-limited search)	< Node (7, 0)> 28 ['up', 'right', 'down', 'down', 'right', 'right', 'down', 'right', 'up', 'up', 'up', 'right', 'down', 'down', 'down', 'right', 'up', 'up', 'up', 'right']	7.6 seconds
GBFS - Manhattan (Greedy-best-first search)	< Node (7, 0)> 13 ['right', 'down', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']	3.3 seconds
GBFS - Euclidean (Greedy-best-first search)	< Node (7, 0)> 12 ['right', 'down', 'right', 'right', 'right', 'right', 'up', 'right', 'up', 'right']	3.1 seconds
AS - Manhattan (A-star search)	< Node (7, 0)> 13 ['right', 'down', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']	3.3 seconds
AS - Euclidean (A-star search)	< Node (7, 0)> 12 ['right', 'down', 'right', 'right', 'right', 'right', 'up', 'right', 'up', 'right']	3.1 seconds
CUS2 - Manhattan	< Node (7, 0)> 13 ['right', 'down', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']	3.3 seconds
CUS2 - Euclidean	< Node (7, 0)> 12 ['right', 'down', 'right', 'right', 'right', 'right', 'up', 'right', 'up', 'right']	3.1 seconds
Test map 3: +shape.txt		
DFS (Depth-first search)	< Node (4, 4)> 21 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'up', 'right', 'down']	5.2 seconds
BFS (Breadth-first search)	< Node (4, 4)> 20 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'right']	4.8 seconds
CUS1 (Depth-limited search)	< Node (4, 4)> 21 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'up', 'right', 'down']	5.2 seconds
GBFS - Manhattan (Greedy-best-first search)	< Node (4, 4)> 9 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'right']	2.6 seconds
GBFS - Euclidean (Greedy-best-first search)	< Node (4, 4)> 10 ['down', 'down', 'down', 'right', 'down', 'right', 'right', 'right']	2.8 seconds
AS - Manhattan (A-star search)	< Node (4, 4)> 9 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'right']	2.6 seconds
AS - Euclidean (A-star search)	< Node (4, 4)> 10 ['down', 'down', 'down', 'right', 'down', 'right', 'right', 'right']	2.8 seconds
CUS2 - Manhattan	< Node (4, 4)> 9 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'right']	2.6 seconds
CUS2 - Euclidean	< Node (4, 4)> 10 ['down', 'down', 'down', 'right', 'down', 'right', 'right', 'right']	2.8 seconds

Test map 1: complexmap.txt		
DFS (Depth-first search)	< Node (19, 19)> 61 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'up', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'left', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'right', 'down', 'right']	17.9 seconds
BFS (Breadth-first search)	< Node (19, 19)> 168 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'down', 'down', 'left', 'left', 'down', 'down', 'left', 'left', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	38.3 seconds
CUS1 (Depth-limited search)	< Node (19, 19)> 61 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'up', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'left', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'right', 'down', 'right']	17.9 seconds
GBFS - Manhattan (Greedy-best-first search)	< Node (19, 19)> 98 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'down', 'down', 'right', 'right', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	26.0 seconds
GBFS - Euclidean (Greedy-best-first search)	< Node (19, 19)> 110 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	28.1 seconds
AS - Manhattan (A-star search)	< Node (19, 19)> 98 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'down', 'down', 'right', 'right', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	25.9 seconds
AS - Euclidean (A-star search)	< Node (19, 19)> 110 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	28.2 seconds
CUS2 - Manhattan	< Node (19, 19)> 98 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'down', 'down', 'right', 'right', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	25.9 seconds
CUS2 - Euclidean	< Node (19, 19)> 110 ['down', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	28.1 seconds

Test map 2: map.txt		
DFS	< Node (5, 9)> 60	17.1 seconds
(Depth-first search)	['up', 'right', 'down', 'down', 'left', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
BFS	< Node (5, 9)> 83	21.2 seconds
(Breadth-first search)	['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
CUS1	< Node (5, 9)> 60	17.1 seconds
(Depth-limited search)	['up', 'right', 'down', 'down', 'left', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
GBFS - Manhattan	< Node (5, 9)> 74	19.5 seconds
(Greedy-best-first search)	['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
GBFS - Euclidean	< Node (5, 9)> 74	19.5 seconds
(Greedy-best-first search)	['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
AS - Manhattan	< Node (5, 9)> 74	19.5 seconds
(A-star search)	['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
AS - Euclidean	< Node (5, 9)> 74	19.5 seconds
(A-star search)	['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
CUS2 - Manhattan	< Node (5, 9)> 74	19.5 seconds
	['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
CUS2 - Euclidean	< Node (5, 9)> 74	19.5 seconds
	['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'down', 'down', 'down', 'right', 'down', 'down', 'down', 'down', 'down', 'down']	
Test map 8: invalid.txt (goal node hidden underneath a wall)		
DFS	No goal is reachable; 37	0.0 seconds
(Depth-first search)		(GUI display not operated)
BFS	No goal is reachable; 24	0.0 seconds
(Breadth-first search)		(GUI display not operated)
CUS1	Not Applicable	0.0 seconds
(Depth-limited search)		(GUI display not operated)
GBFS - Manhattan	No goal is reachable; 24	0.0 seconds
(Greedy-best-first search)		(GUI display not operated)
GBFS - Euclidean	No goal is reachable; 24	0.0 seconds
(Greedy-best-first search)		(GUI display not operated)
AS - Manhattan	No goal is reachable; 24	0.0 seconds
(A-star search)		(GUI display not operated)
AS - Euclidean	No goal is reachable; 24	0.0 seconds
(A-star search)		(GUI display not operated)
CUS2 - Manhattan	No goal is reachable; 24	0.0 seconds
		(GUI display not operated)
CUS2 - Euclidean	No goal is reachable; 24	0.0 seconds
		(GUI display not operated)
Test map 9: array.txt		
DFS	< Node (8, 4)> 53	15.1 seconds
(Depth-first search)	['up', 'up', 'up', 'up', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up']	
BFS	< Node (8, 4)> 56	12.9 seconds
(Breadth-first search)	['up', 'up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down']	
CUS1	< Node (8, 4)> 53	15.1 seconds
(Depth-limited search)	['up', 'up', 'up', 'up', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up']	
GBFS - Manhattan	< Node (8, 4)> 38	9.3 seconds
(Greedy-best-first search)	['up', 'up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down']	
GBFS - Euclidean	< Node (8, 4)> 33	8.3 seconds
(Greedy-best-first search)	['up', 'up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down']	
AS - Manhattan	< Node (8, 4)> 38	9.3 seconds
(A-star search)	['up', 'up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down']	
AS - Euclidean	< Node (8, 4)> 33	8.3 seconds
(A-star search)	['up', 'up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down']	
CUS2 - Manhattan	< Node (8, 4)> 38	9.3 seconds
	['up', 'up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down']	
CUS2 - Euclidean	< Node (8, 4)> 33	8.3 seconds
	['up', 'up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down']	
Test map 10: dotdot.txt		
DFS	< Node (4, 1)> 11	2.9 seconds
(Depth-first search)	['up', 'right', 'right', 'down', 'down', 'right', 'right', 'up']	
BFS	< Node (4, 1)> 13	3.3 seconds
(Breadth-first search)	['up', 'right', 'right', 'right', 'right', 'down']	
CUS1	< Node (4, 1)> 11	2.9 seconds
(Depth-limited search)	['up', 'right', 'right', 'down', 'down', 'right', 'right', 'up']	
GBFS - Manhattan	< Node (4, 1)> 8	2.3 seconds
(Greedy-best-first search)	['up', 'right', 'right', 'right', 'right', 'down']	
GBFS - Euclidean	< Node (4, 1)> 7	2.1 seconds
(Greedy-best-first search)	['up', 'right', 'right', 'right', 'right', 'down']	
AS - Manhattan	< Node (4, 1)> 8	2.3 seconds
(A-star search)	['up', 'right', 'right', 'right', 'right', 'down']	
AS - Euclidean	< Node (4, 1)> 7	2.1 seconds
(A-star search)	['up', 'right', 'right', 'right', 'right', 'down']	
CUS2 - Manhattan	< Node (4, 1)> 8	2.3 seconds
	['up', 'right', 'right', 'right', 'right', 'down']	
CUS2 - Euclidean	< Node (4, 1)> 7	2.1 seconds
	['up', 'right', 'right', 'right', 'right', 'down']	

Test map 4: nogoal.txt (this map's goal node has been blocked)		
DFS	No goal is reachable; 23	0.0 seconds
(Depth-first search)		(GUI display not operated)
BFS	No goal is reachable; 16	0.0 seconds
(Breadth-first search)		(GUI display not operated)
CUS1	Not Applicable	0.0 seconds
(Depth-limited search)		(GUI display not operated)
GBFS - Manhattan	No goal is reachable; 16	0.0 seconds
(Greedy-best-first search)		(GUI display not operated)
GBFS - Euclidean	No goal is reachable; 16	0.0 seconds
(Greedy-best-first search)		(GUI display not operated)
AS - Manhattan	No goal is reachable; 16	0.0 seconds
(A-star search)		(GUI display not operated)
AS - Euclidean	No goal is reachable; 16	0.0 seconds
(A-star search)		(GUI display not operated)
CUS2 - Manhattan	No goal is reachable; 16	0.0 seconds
		(GUI display not operated)
CUS2 - Euclidean	No goal is reachable; 16	0.0 seconds
		(GUI display not operated)
Test map 5: halfcomplex.txt		
DFS	< Node (0, 0)> 39	10.8 seconds
(Depth-first search)	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'up']	
BFS	< Node (0, 0)> 80	19.0 seconds
(Breadth-first search)	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'up']	
CUS1	< Node (0, 0)> 39	10.8 seconds
(Depth-limited search)	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'up']	
GBFS - Manhattan	< Node (0, 0)> 36	10.2 seconds
(Greedy-best-first search)	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'up']	
GBFS - Euclidean	< Node (0, 0)> 31	9.2 seconds
(Greedy-best-first search)	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'up']	
AS - Manhattan	< Node (0, 0)> 36	10.2 seconds
(A-star search)	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'up']	
AS - Euclidean	< Node (0, 0)> 31	9.2 seconds
(A-star search)	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'up']	
CUS2 - Manhattan	< Node (0, 0)> 36	10.2 seconds
	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'up']	
CUS2 - Euclidean	< Node (0, 0)> 31	9.2 seconds
	['right', 'right', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'left', 'up', 'left', 'left', 'left', 'left', 'left', 'up']	
Test map 6: xshape.txt		
DFS	< Node (0, 0)> 9	2.7 seconds
(Depth-first search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
BFS	< Node (0, 0)> 20	4.8 seconds
(Breadth-first search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
CUS1	< Node (0, 0)> 9	2.7 seconds
(Depth-limited search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
GBFS - Manhattan	< Node (0, 0)> 9	2.7 seconds
(Greedy-best-first search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
GBFS - Euclidean	< Node (0, 0)> 10	2.9 seconds
(Greedy-best-first search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
AS - Manhattan	< Node (0, 0)> 9	2.7 seconds
(A-star search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
AS - Euclidean	< Node (0, 0)> 10	2.9 seconds
(A-star search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
CUS2 - Manhattan	< Node (0, 0)> 9	2.7 seconds
	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
CUS2 - Euclidean	< Node (0, 0)> 10	2.9 seconds
	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
Test map 7: dot.txt		
DFS	< Node (0, 0)> 9	2.7 seconds
(Depth-first search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
BFS	< Node (0, 0)> 24	5.7 seconds
(Breadth-first search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
CUS1	< Node (0, 0)> 9	2.7 seconds
(Depth-limited search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
GBFS - Manhattan	< Node (0, 0)> 9	2.7 seconds
(Greedy-best-first search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
GBFS - Euclidean	< Node (0, 0)> 9	2.7 seconds
(Greedy-best-first search)	['up', 'left', 'up', 'up', 'left', 'left', 'up', 'left']	
AS - Manhattan	< Node (0, 0)> 9	2.7 seconds
(A-star search)	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
AS - Euclidean	< Node (0, 0)> 9	2.7 seconds
(A-star search)	['up', 'left', 'up', 'up', 'left', 'left', 'up', 'left']	
CUS2 - Manhattan	< Node (0, 0)> 9	2.7 seconds
	['up', 'up', 'up', 'up', 'left', 'left', 'left', 'left']	
CUS2 - Euclidean	< Node (0, 0)> 9	2.7 seconds
	['up', 'left', 'up', 'up', 'left', 'left', 'up', 'left']	

Figure. Set of tables showing test cases with Method-Result-Time measurement values recorded.

1. Analysis:

DFS (Deep First Search):

Advantage - Memory efficient;

Disadvantage - Not guaranteed to find the shortest path. Can get stuck in tremendous loops.

BFS (Breadth First Search):

Advantage - Amongst the shortest path;

Disadvantage - Memory intensive, may explore unnecessary nodes which consumes more time.

CUS1 (Inspired by Depth Limited method):

Advantage - Memory efficient, may leads to path efficiency, yet yield similar result to DFS;

Disadvantage - Not guaranteed to find the shortest path. Can get stuck in infinite loops, especially when goal node can't be found.

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. Utilizes memory, 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.

2. Application:

Based on the results, BFS is the most effective uninformed method in term of path finding, nonetheless, it is not an optimal solution as it usually explore extensive nodes that causes time consumption and memory inefficiency.
All informed methods, including GBFS, AS and CUS2 perform equally well in terms of finding the shortest path with minimal nodes.
Euclidean distance tends to be more effective when applying into all informed methods, within these cases scenarios, compared to Manhattan distance. Yet, it is noticeably that these 2 heuristics algorithms cannot be determined whether which are more optimal as it relies and depends strictly on the environment scenario provided.

H. 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 and time consuming.
+ 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.

I. Features/Bugs/Missing

1. Features:

a. Code Execution and Map Parsing
+ The code adeptly reads map files and constructs the map with accurate definitions of dimensions, walls, start, and goal coordinates.
+ It parses the map configuration from a file, including dimensions, start and goal coordinates, and wall positions, preparing for the execution of search methods.
+ 10 additional custom maps are created alongside with the sample map file.

b. Tree-Based Search Algorithms

+ Rigorously defines tree-based search algorithms such as Depth-first search (DFS), Breadth-first search (BFS) for uninformed searches, and Greedy-best-first search (GBFS), A star (AS) for informed searches, allowing the robot agent to locate the goal and display its path with all exploration details.
+ Implements two custom search methods, CUS1 utilizing Depth-limited search and CUS2 combining GBFS and A* methods, which can also locate the goal and display exploration details appealing to either uninformed and informed approaches.

c. Graphical User Interface (GUI)

+ Constructs a GUI display interface that accurately visualizes how the robot agent navigates to find its path to the goal target throughout obstacles (wall).
+ The node expansion (traversed variable) and path taken to the goal target is displayed dynamically with gradual changes (per 0.2s and 0.1s) to show how the agent navigating as a sequence.
+ The GUI visualise node exploration (traversed) and the path taken by the robot agent.

2. Bugs: CUS1 (depth-limited search) algorithms is not applicable for map environment that goal state is unreachable, by which the algorithms will get stuck in an infinite loop for depth_level incrementation.

3. Missing: The project only initialize the GUI display as a Research Initiatives while cannot fulfill the design concept for the 2 others.

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 it explored node expansion (traversed), all by dynamic visualization by each states.

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:

1. Instantiate GUI class:

```
# Create an instance containing only rows and column for Grid class, used to scale the GUI display window
grid_instance = Grid(rows, cols)

# Create GUI window if path is not None
if path:
    from gui import GUI
    app = GUI(grid_instance, grid, start, goal[0], [tuple((node.x, node.y)) for node in path], traversed)
    app.mainloop()
```

Figure. Code publish GUI class.

If path was found, from 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 while inheriting data from searchmain.py file's Node class.
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 (current_step).

```
class GUI(tk.Tk):
    # Draw the dmap display
    def __init__(self, grid_instance, grid, start, goal, path, traversed):
        super().__init__()
        self.title("GUI Display")
        window_width = 30 * grid_instance.rows # Cell size * num_of_rows
        window_height = 30 * grid_instance.cols  # Cell size * num_of_cols
        self.geometry(f"{window_width}x{window_height}") # Size of the GUI display window
        self.grid = grid
        self.start = start
        self.goal = [goal] # List of goal nodes
        self.path = [Node(x, y) for x, y in path]
        self.traversed = [Node(x, y) for x, y in traversed]
        self.cell_size = 30
        self.canvas = tk.Canvas(self, width=window_width, height=window_height)
        self.canvas.pack()
        self.draw_grid()
        self.current_step = 1 # Started from the first step
        self.current_traversed = 1 # Started from the first step
        self.display_next_step() # Show paths as a sequent of motion by gradual changess.
```

Figure. Code initialise GUI display.

It set first step in traversed (node explored) to be displayed (current_traversed).
A canvas widget (draw_grid method) is called to draw the grid and visualize the map elements.
Call function (display_next_step) to display traversed and path by dynamic motion.

3. Draw Grid and update steps:

The draw_grid method iterates over each cell in the grid as rectangular polygons to visualize it based on its content, distinguished by different colours:
+ Start position is represented in red.
+ Goal position is represented in lime.
+ Walls are represented in dark grey.
+ Empty cells are represented in white.

```
def draw_grid(self):
    for i in range(self.grid.rows): # x0, y0, x1, y1 is positional data of a polygon
        for j in range(self.grid.cols):
            x0, y0 = i * self.cell_size, j * self.cell_size
            x1, y1 = x0 + self.cell_size, y0 + self.cell_size
            if (i, j) in self.goal: # Goal cell(s) as lime color
                self.canvas.create_rectangle(x0, y0, x1, y1, fill="lime")
            elif (i, j) == self.start: # Match starting cell as red color
                self.canvas.create_rectangle(x0, y0, x1, y1, fill="red")
            elif self.grid.grid[i][j] == 1: # Wall cell(s) as gray color
                self.canvas.create_rectangle(x0, y0, x1, y1, fill="dark gray")
            elif (i, j) in self.traversed: # All explored cells as purple color
                self.canvas.create_rectangle(x0, y0, x1, y1, fill="purple")
            else: # Empty cell(s) as white color
                self.canvas.create_rectangle(x0, y0, x1, y1, fill="white")
```

Figure. Code draws the GUI interface.

Dynamic Motion:

```
# For n steps, display from 1 to (n-1)th step, for either node expansion (traversed) and path finding.
# Node expansion are displayed gradually by each 0.2s, until it find the goal node, the path will be shown by each 0.1s
def display_next_step(self):
    if self.current_traversed < (len(self.traversed)-1):
        node = self.traversed[self.current_traversed]
        x0, y0 = node.x * self.cell_size, node.y * self.cell_size
        x1, y1 = x0 + self.cell_size, y0 + self.cell_size
        self.canvas.create_rectangle(x0, y0, x1, y1, fill="purple") # Display explored node as purple color
        self.current_traversed += 1
        self.after(200, self.display_next_step)  # Update traversed expansion every 0.2 seconds

    elif self.current_traversed == (len(self.traversed)-1): # Once the node expansion reach the goal node, start displaying path
        if self.current_step < (len(self.path)-1):
            node = self.path[self.current_step]
            x0, y0 = node.x * self.cell_size, node.y * self.cell_size
            x1, y1 = x0 + self.cell_size, y0 + self.cell_size
            self.canvas.create_rectangle(x0, y0, x1, y1, fill="yellow") # Path displayed as yellow color.
            self.current_step += 1 # Add step variable  by 1 after each function call
            self.after(100, self.display_next_step)  # After 0.1 seconds, update next steps.
```

Figure. Code updates traversed and path as dynamic motion.

display_next_step method executes gradual update progression of the node expansion and path from the starting state to the goal state. Initially, it will gradually show traversed in self, expanded from current to next node in the array as purple cell, which the method will be call each 0.2 second to visualise motion, until the current_traversed variable reach the goal state (n-1, while n is the time node expanded). Once the agent locate the node (current_traversed match the traversed length), GUI interface starts to show gradual motion of path in self as yellow, from current to next node, which display_next_step method will be called each 0.1 second.

The GUI class is instantiated and integrated into the main application (searchmain.py) to display the navigation process. These images illustrate how traversed and path differed by search methods and heuristic applications (main GUI display motions are sent with this report), using sample map file.

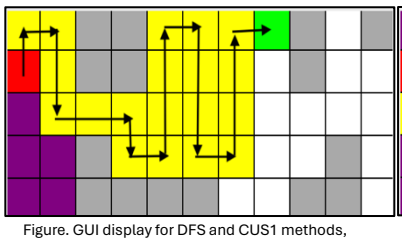

Figure. GUI display for DFS and CUS1 methods, with arrows labelling the path direction.

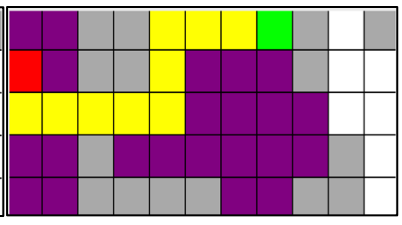

Figure. GUI display for BFS method.

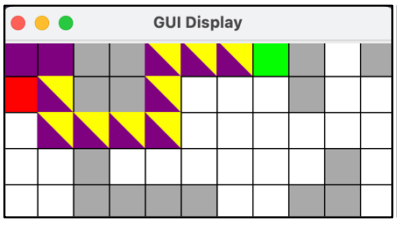

Figure. GUI display for GBFS, AS, CUS2 methods Manhattan heuristic. Set as triangles for visualization

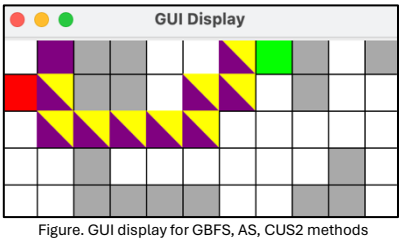

Figure. GUI display for GBFS, AS, CUS2 methods Euclidean heuristic. Set as triangles for visualization

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/