Coffee-Go-Drone Application

By

Name: Steve and Surya

Student ID: P2308867 - P2100874

Class: DAAA/FT/2A/02

In partial fulfillment of the requirements for the module

ST1507: Data Structure and Algorithm (AI)- DAAA

CA2

Lecturer: Ms. Hubertus Borts Pauwels

Submission Date: 07/08/2024

Introduction :

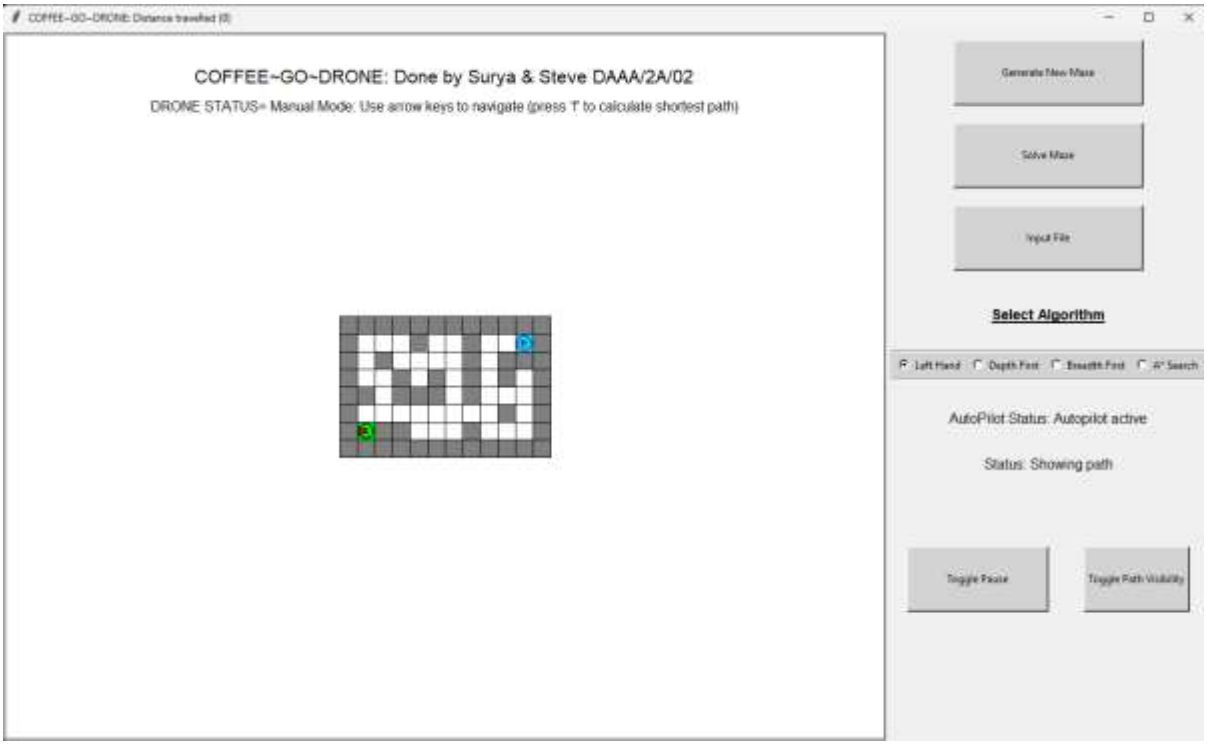
For this assignment you will have an opportunity to apply all that you have learnt with regards to data structures, algorithms, and object-oriented programming as to develop an

application that can help a coffee delivery company study how their drones may be deployed to provide fast and efficient coffee delivery in the busy business district of Heatherthorn County.

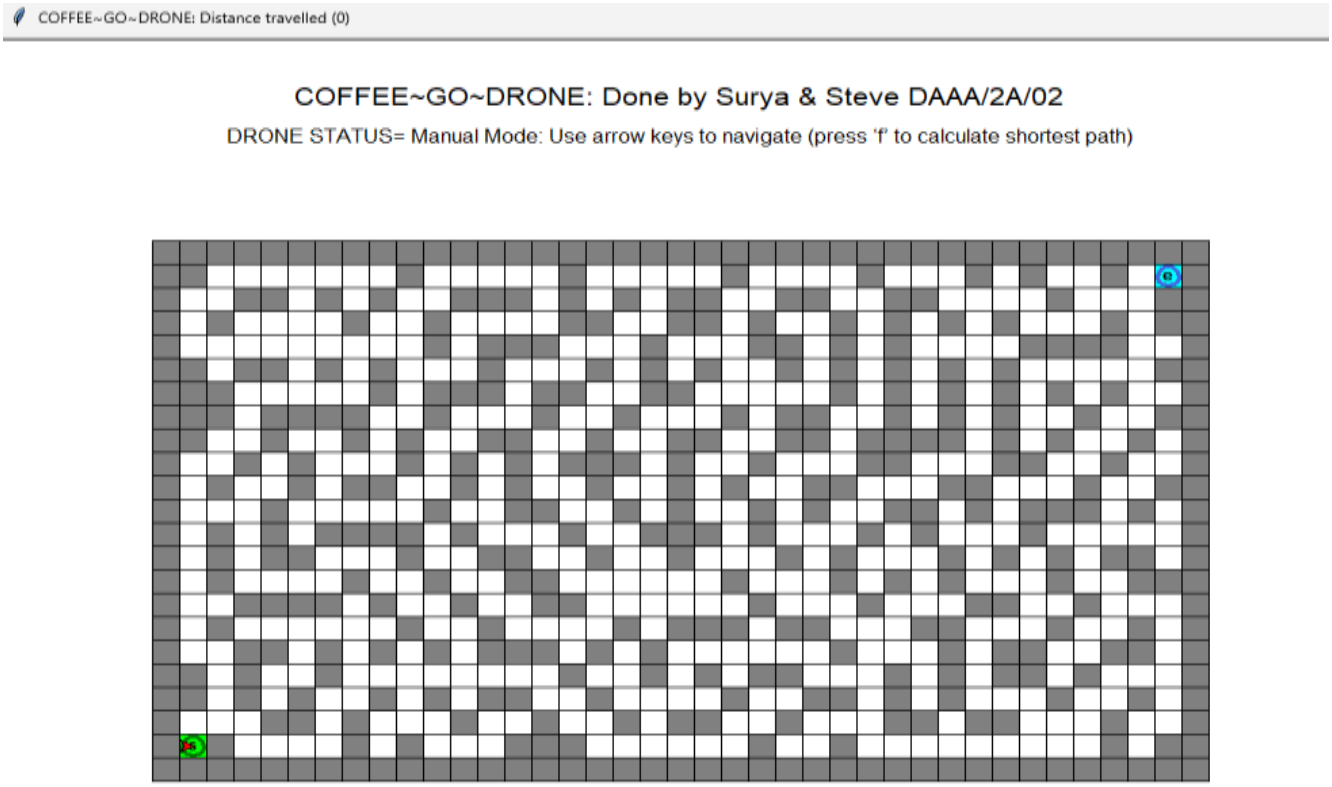
Project Overview:

User Interface and Introduction:

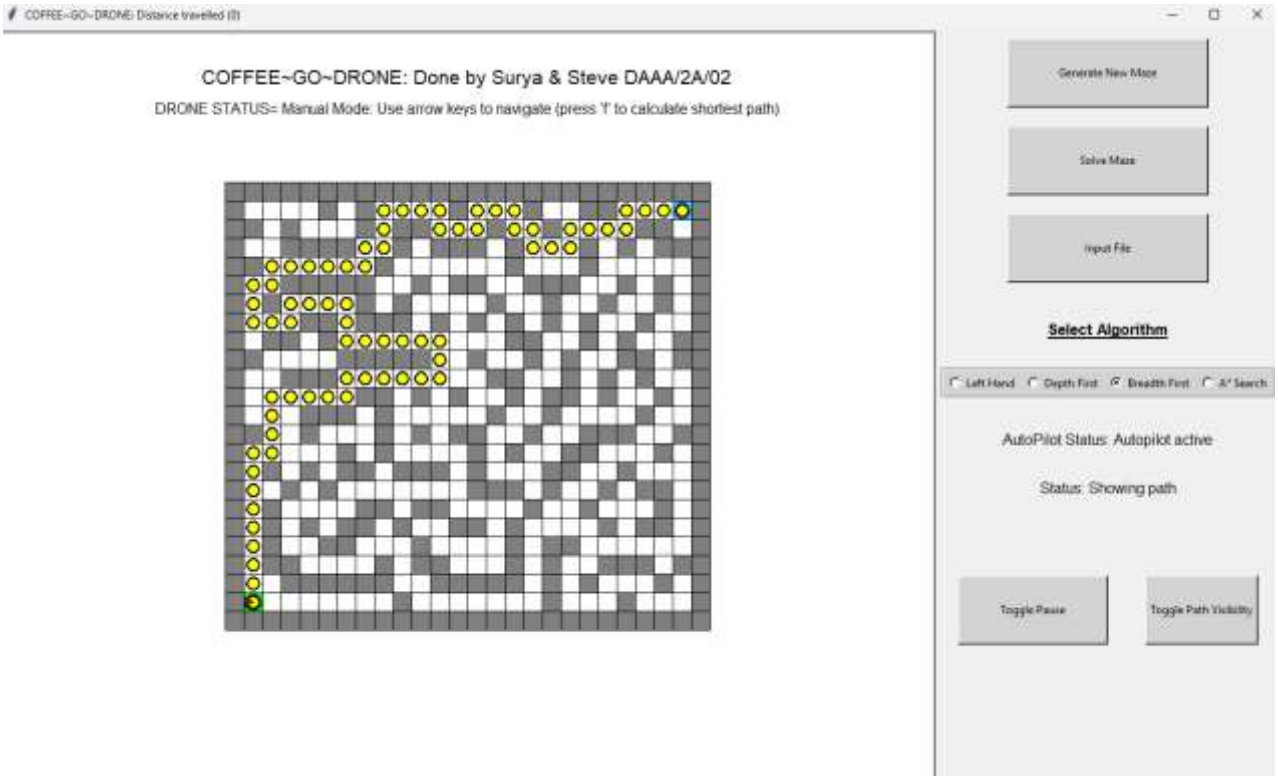
When run python main.py default.txt in which default txt is the given drone in our CA2 specifications. Users will be brought to this page



1. The number of steps the drone has moved is recorded on the top left corner of the page
2. Drone status will be constantly updated and recorded on top and to the side of the table
3. The main maze will the drone will be centered in the middle of User Interface.
4. When click Generate New Maze a new random maze will be generated



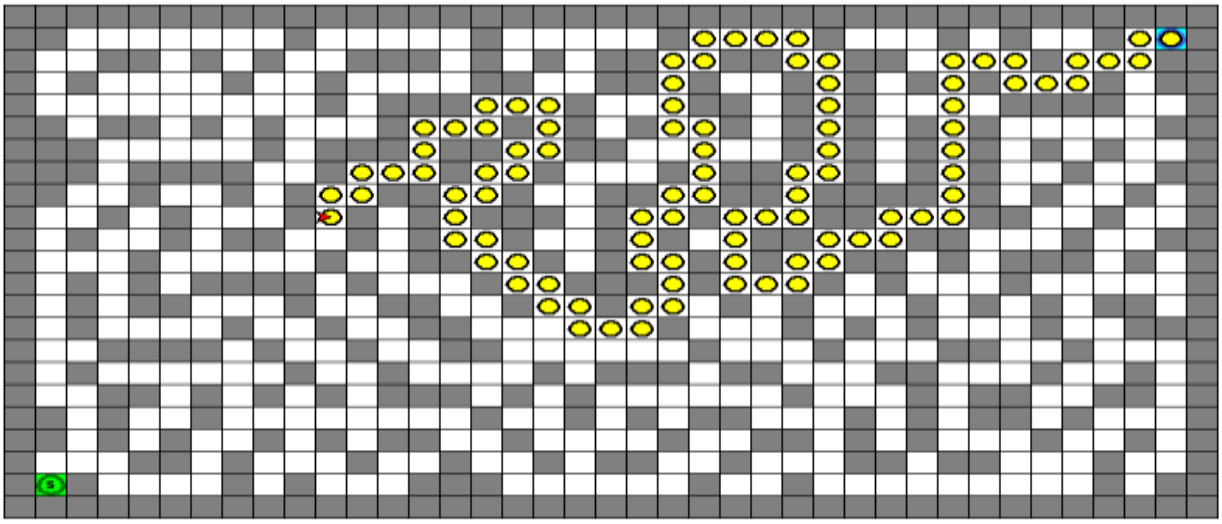
5. User will be able to choose from 4 different algorithm to find the best path from the start to the end position. The 4 Algorithms are Left Hand A Star Dept First and Breadth First. By clicking on one of the 4 tick boxes on the right and the path found by this algorithm will be displayed. Alternatively, user can choose the algorithm of choice and press F to display the path



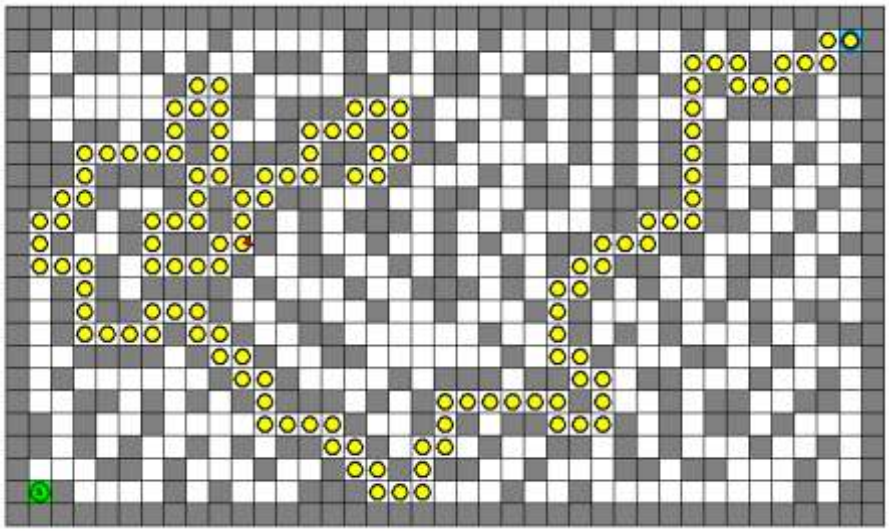
6. The user will be able to manually move across the maze and when repeating step 4 the path from their current position to the end point will be displayed

COFFEE~GO~DRONE: Done by Surya & Steve DAAA/2A/02

DRONE STATUS= Manual Mode: Use arrow keys to navigate (press 'f' to calculate shortest path)



7. To hide the and re-show the predefined path, user can toggle with H key.
8. To reset the whole Process and restart from the start position, user can press R
9. To Automatically let the drone follow the predefined path to the destination, User can press G and the drone will start moving in the predefined direction



10. Drone movement can be paused with P key. Noted when paused, the drone cannot be manually operated unless it is unpaused.
11. To allow the drone to automatically move a distance then switch to manual Mode, press G and pause at where you want to start manual operation and C to enter Manual Mode if you want to keep going automatically simply press P again.
12. When end point is reached after automatic movement of the drone user can press C to manually Move it around again

13. When done the operation can be quit by press Q key on the keyboard

Code and Features Highlights

OOP Concept Applied in All Algorithms Class to search for directions.
The program supports four maze-solving algorithms:

- **Left Hand:** A simple algorithm that follows the left wall of the maze, suitable for simpler mazes.
- **Depth First:** An exhaustive search strategy that explores paths deeply, ensuring complete exploration.
- **Breadth First**: A systematic search that explores paths uniformly, ideal for finding shortest paths.
- **A* Search:** An efficient heuristic-based algorithm that balances exploration and path cost to find optimal solutions.

Each algorithm is implemented with consideration for flexibility and performance, allowing users to select the desired method through the interface.

```python
class BreadthFirst_MazeSolver:
    def __init__(self, pos_x, pos_y, maze):
        self.maze = maze
        self.pos_x = pos_x
        self.pos_y = pos_y
        self.queue = Deque()  # Use the custom Deque class
        self.queue.append((self.pos_x, self.pos_y))
        self.visited = set([(self.pos_x, self.pos_y)])
        self.numbered_path_nodes = []
        self.parent = {}  # To keep track of the path
        self.dx = [0, 1, 0, -1]
        self.dy = [-1, 0, 1, 0]

    def solve_maze_bfs(self):
```

```python
class DepthFirst_MazeSolver:
    def __init__(self, pos_x, pos_y, maze):
        self.maze = maze
        self.pos_x = pos_x
        self.pos_y = pos_y
        self.stack = [(self.pos_x, self.pos_y)]
        self.visited = set(self.stack)
        self.numbered_path_nodes = []
        self.dx = [0, 1, 0, -1]
        self.dy = [-1, 0, 1, 0]

    def solve_maze_dfs(self):
        """Solve the maze using Depth First Search (DFS)."""
        start_x, start_y = None, None
        for my, row in enumerate(self.maze):
            for mx, cell in enumerate(row):
                if cell == 's':
                    start_x, start_y = mx, my
```

draw_maze Aa ab .* No results

```python
import heapq

class AStar_MazeSolver:
    def __init__(self, pos_x, pos_y, maze):
        self.maze = maze
        self.pos_x = pos_x
        self.pos_y = pos_y
        self.start = (self.pos_x, self.pos_y)
        self.goal = self.find_goal_position()
        self.dx = [0, 1, 0, -1]
        self.dy = [-1, 0, 1, 0]
        self.numbered_path_nodes = []
        self.open_set = []
        self.came_from = {}
        self.cost_so_far = {self.start: 0}
```

```python
def solve_maze(t, tile_drawer):
    global initial_start_position, solved_path_coordinates
    global gx, gy, x, y
    solve_button.config(state=tk.DISABLED)  # Disable the button
    generate_button.config(state=tk.DISABLED)  # Disable the button
    disable_key_controls()  # Disable key controls
    # Move the turtle back to the initial starting position before solving the maze
    gx,gy,x,y = find_position()
    t.setheading(0)  # Set the turtle to face up
    maze_str = read_file(FILE_IN_PLAY)
    maze = string_to_maze(maze_str)

## Using Left Hand
    if selected_algorithm.get() == "left_hand":
        left_hand_solver = LeftHand_MazeSolver(gx, gy, maze)
        solved_maze, solved_path = left_hand_solver.solve()

## Using Depth First
    elif selected_algorithm.get() == "depth_first":
        depth_first_solver = DepthFirst_MazeSolver(gx, gy, maze)
        solved_maze, solved_path = depth_first_solver.solve_maze_dfs()

## Using Breadth First
    elif selected_algorithm.get()== "breadth_first":
        breadth_first_solver = BreadthFirst_MazeSolver(gx, gy, maze)
        solved_maze, solved_path = breadth_first_solver.solve_maze_bfs()

## Using A Star
    elif selected_algorithm.get() == "a_star":
        astar_solver = AStar_MazeSolver(gx, gy, maze)
        solved_maze, solved_path = astar_solver.solve_maze_astar()



    # Convert the solved maze list back to a string
    solved_maze_str = '\n'.join([''.join(row) for row in solved_maze])
    solved_path_coordinates = solved_path
    initial_start_position = draw_maze(solved_maze_str, t, tile_drawer)  # Save the new starting position
    Move_Turtle().move_turtle_to_start(t,(x,y))
    screen.update()
```

Using Global Variable Making variables accessible across all functions in main file

```python
def main():
    global selected_algorithm
    global generate_button
    global solve_button
    global screen
    global t
    global tile_drawer
    global root
    global steps
    global content
    global label_turtle
    global statuses, status_pause_label, status_path_label, error, log_text
```

Using Tkinter for User-Friendly Interface

```
tile_drawer = Tile(t)
generate_button = tk.Button(root, text="Generate New Maze", command=lambda: generate_new_maze(t, tile_drawer), bg='#d3d3
solve_button = tk.Button(root, text="Solve Maze", command=lambda: solve_maze(t, tile_drawer), bg='#d3d3d3')

generate_button.grid(padx=75, pady=10, row=0, column=11, sticky='nsew')
solve_button.grid(padx=75, pady=10, row=1, column=11, sticky='nsew')

algorithm_frame = tk.Frame(root)
algorithm_frame.grid(padx=2, pady=2, row=3, column=11, sticky='nsew')
algorithm_frame.grid_rowconfigure(0, weight=1)
algorithm_frame.grid_columnconfigure(0, weight=1)
#add text to the frame make the font bold

file_input_button = Button(root, text="Input File", command=open_file_input_window , bg='#d3d3d3')
file_input_button.grid(padx=75, pady=10, row=2, column=11, sticky='nsew')


algorithm_label = tk.Label(algorithm_frame, text="Select Algorithm", font='Helvetica 13 bold underline' )
algorithm_label.grid(padx=2, pady=2, row=0, column=0, columnspan=4, sticky='nsew')

radio_frame = tk.Frame(algorithm_frame, bg='#d3d3d3', bd=2, relief='groove')
radio_frame.grid(padx=2, pady=2, row=1, column=0, columnspan=4, sticky='nsew')
```

Summary of Data Structure Use

Data Structure	Code	Reasons for implementation
List: lists are used extensively in this program to represent the maze's structure, handle paths, and manage algorithm selections. Lists provide a dynamic and flexible way to store ordered collections of items, which is ideal for the varying sizes and structures of mazes.		The maze is stored as a list of lists (2D list), where each sub-list represents a row, and each element within the sub-list represents a tile in the maze (e.g., walls, paths, start, and end points). This allows for easy access and modification of specific tiles when drawing or solving the maze.
Dictionaries: In this program, dictionaries are leveraged for managing mappings and configuration settings.	From line 87 Main.py file def enable_key_controls(): global maze_str screen.onkey(lambda: move('U'), 'Up') screen.onkey(lambda: move('D'), 'Down') screen.onkey(lambda: move('L'), 'Left') screen.onkey(lambda: move('R'), 'Right')	Facilitate fast lookup and management of configurations, command mappings, and algorithm parameters, streamlining program logic.

	screen.onkey(lambda: solve_maze(t, tile_drawer), 'f') screen.onkey(lambda: follow_path(t), 'g') screen.onkey(lambda: reset_position(), 'r') screen.onkey(lambda: toggle_path_visibility(), 'h')	

Heaps (in A* Alogorithm):
The heap,is used to implement a priority queue for the open set of nodes. Nodes are prioritized based on their estimated total cost (f(n) = g(n) + h(n)), where g(n) is the cost to reach the node, and h(n) is the heuristic estimate to the goal.
	From line 29: A* Class file
	def solve_maze_astar(self):
	"""Solve the maze using A* Search Algorithm."""
	
	heapq.heappush(self.open_set, (0 + self.heuristic(self.start, self.goal), 0, self.start))
	
	while self.open_set:
	_, current_cost, current = heapq.heappop(self.open_set)
	
	if current == self.goal:
	break
		Efficient Priority Management: Provides O(log n) time complexity for insertion and extraction, making it ideal for dynamic, ordered access to nodes.
		□ Optimal Pathfinding: Ensures that the algorithm always explores the most promising paths first, contributing to the efficiency of A*.

Custom Deque:
used to manage the queue of nodes to be explored in the BFS algorithm. It ensures nodes are processed in a First-In-First-Out (FIFO) order, which is critical for BFS.
	class Deque:
	def __init__(self):
	self.items = []
	
	def append(self, item):
	self.items.append(item)
	
	def popleft(self):
	if not self.is_empty():
	return self.items.pop(0)
	else:
	raise IndexError("pop from an empty deque")
	
	def is_empty(self):
	return len(self.items) == 0
	
	def __len__(self):
	return len(self.items)
		A deque efficiently supports operations from both ends, and while the custom implementation here has an O(n) pop operation from the front, the choice provides an intuitive structure for managing nodes. Using the built-in collections.deque would optimize this further.

6. Challenges and Limitations

The maze-solving program is a sophisticated tool that effectively demonstrates the use of various algorithms in navigating and solving mazes. However, like any complex software, it has its challenges and limitations. These can be categorized into algorithm limitations, technical challenges, and user interface constraints.

Algorithm Limitations

1. **Left Hand Algorithm**:
    - **Simplicity**: The Left Hand algorithm is straightforward and may not handle complex mazes efficiently. It primarily relies on hugging the left wall, which can lead to inefficient paths and increased solving time.
    - **Complex Mazes**: In mazes with open areas or multiple paths, the Left Hand method might not find the shortest path, especially if there are loops or multiple exits.
    - **Non-Standard Layouts**: Mazes that do not follow traditional structures or have irregular designs may cause the algorithm to fail or get stuck.
2. **Depth First Search (DFS)**:
    - **Exhaustive Exploration**: DFS explores all possible paths before finding a solution, which can be time-consuming in large mazes.
    - **Memory Usage**: The algorithm may require significant memory resources for deep paths, as it stores nodes in a stack until a solution is found.
3. **Breadth First Search (BFS)**:
    - **Uniform Exploration**: While BFS is excellent for finding the shortest path, it may be less efficient in terms of speed and performance, especially in large mazes with many nodes.
    - **Queue Management**: Managing the queue can be challenging, particularly in mazes with a high branching factor.
4. *A Search**:
    - **Heuristic Dependence**: A* relies on heuristic functions to estimate the cost to the goal. Poor heuristics can lead to suboptimal paths or increased solving time.
    - **Complexity**: Implementing A* requires careful consideration of heuristic accuracy and efficiency, which can be challenging for beginners.

Technical Challenges

1. **Integration of Algorithms**:
    - **Complex Design**: Integrating multiple algorithms required a modular design to maintain performance and ensure seamless switching between methods.
    - **Algorithm Compatibility**: Ensuring each algorithm works correctly with the maze structure and user interface posed challenges in handling edge cases and exceptions.
2. **Handling Large Mazes**:
    - **Performance Bottlenecks**: Large or intricate mazes can slow down the program, particularly with algorithms that explore many nodes.

- **Optimization Needs**: Balancing the trade-off between performance and accuracy required optimizations in code execution and memory usage.

3. **User Interface and Visualization**:
   - **Turtle Graphics Limitations**: While Turtle graphics provide visual appeal, they may not handle high-speed drawing efficiently, affecting responsiveness in complex mazes.
   - **User Feedback**: Incorporating real-time feedback and control mechanisms added complexity to the user interface design.

- **Performance Optimization**

1. **Code Efficiency**:
   - **Refactoring**: Refactoring code to improve efficiency and readability could enhance overall performance and maintainability.
   - **Parallel Processing**: Utilizing parallel processing techniques could speed up solving times, particularly in large mazes.
2. **Resource Management**:
   - **Memory Optimization**: Optimizing memory usage, especially for depth-first and breadth-first algorithms, could reduce resource consumption.
   - **Speed Improvements**: Implementing caching or other techniques to improve speed and responsiveness in maze-solving.
3. **Scalability**:
   - **Handling Larger Mazes**: Enhancing scalability to handle larger and more complex mazes efficiently could increase the program's applicability.