VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

PROJECT 1: SEARCHING REPORT

Course: CSC14003 - Introduction to Artificial Intelligence

Group: 10Cent
22127213 - Võ Minh Khôi
22127158 - Nhâm Đức Huy
22127129 - Nguyễn Tấn Hoàng
22127042 - Lê Nguyễn Minh Châu

Instructor:
Mr. Nguyễn Thanh Tình

Ho Chi Minh City, July 27th, 2024

Contents

Project 1 Report: Searching

1. Introduction

This Introduction to Artificial Intelligence (CSC14003) project report focuses on implementing and comparing various search algorithms for optimizing delivery routes in a 2D city map. The algorithms include Breadth-First Search (BFS), Depth-First Search (DFS), Uniform-Cost Search (UCS), Greedy Best-First Search (GBFS), and A* Search. Additionally, the project explores advanced levels involving time limitations, fuel constraints, and multiple agents.

The goal is to evaluate each algorithm based on pathfinding efficiency and runtime to determine their suitability for practical applications in logistics optimization. The results will provide insights into the strengths and weaknesses of these algorithms, aiding in the development of more efficient route-planning systems.

2. Group Information

No.	Full Name	Student ID	Email
1	Lê Nguyễn Minh Châu	22127042	Lnmchau22@clc.fitus.edu.vn
2	Nguyễn Tấn Hoàng	22127129	Nthoang22@clc.fitus.edu.vn
3	Nhâm Đức Huy	22127158	Ndhuy22@clc.fitus.edu.vn
4	Võ Minh Khôi	22127213	Vmkhoi22@clc.fitus.edu.vn

Demo video: Demo Project 1 - Searching (Introduction to AI) (youtube.com)
(https://www.youtube.com/watch?v=1ds5QyIgyrs)

3. Work Assignment

No.	Full Name	Tasks	Completion Rate
1	Lê Nguyễn Minh Châu	Implement Level 4	100%
		Implement UI	100%
		Testing	100%
2	Nguyễn Tấn Hoàng	Implement Level 3, 4	100%
		Testing	100%
3	Nhâm Đức Huy	Implement Level 1	100%
		Design/Implement UI	100%
		Write Report	100%
		Testing	100%
4	Võ Minh Khôi	Implement Level 2	100%
		Design/Implement UI	100%
		Write Report	100%
		Testing	100%

4. Self-Evaluation

No.	Details		Score	Completion Rate
1	Finish Level 1		15%	15%
2	Finish Level 2		15%	15%
3	Finish Level 3		15%	15%
4	Finish Level 4		10%	10%
5	Implement Graphical User Interface		10%	10%
6	Generate test cases with different attributes (5 tests for each level)		15%	15%
7	Report	Detailed algorithm description.	20%	20%
		Test cases description		
	Total		**100%**	**100%**

5. Detailed Algorithm Descriptions

1. Level 1 (Basic Level)

1.1 Breadth-First Search

1.1.1 Idea

The goal of this algorithm is to find the shortest path from a starting position to an ending position in a grid. The algorithm uses a standard Breadth-First Search (BFS) approach to explore all possible paths efficiently, ensuring that the first time it reaches the goal, the path is guaranteed to be the shortest. It handles obstacles in the grid by skipping over them and uses a queue to manage the nodes to be explored.

1.1.2 Step-by-step Description

- The BFS algorithm starts by initializing a visited set and a parent dictionary, and adds the starting node to a queue, marking it as visited. The algorithm then processes nodes from the queue, exploring their adjacent nodes (up, down, left, right). For each valid and unvisited adjacent node, it records its parent, checks if it's the goal, and if so, reconstructs and returns the path. Otherwise, it adds the node to the queue and marks it as visited.
- The process continues until the queue is empty or the goal node is found. If the goal is reached, the path is returned. If the queue is exhausted without finding the goal, None is returned, indicating no path exists. BFS guarantees the shortest path due to its level-by-level exploration.

1.1.3 Pseudocode

```
CLASS BFS INHERITS SearchAlgorithm:
    FUNCTION __init__(matrix, start, end):
        CALL super().__init__(matrix, start, end)

    FUNCTION Try():
    visited <- set()
    parent <- dictionary()
    queue <- deque([start])

    visited.add(start)

    WHILE queue is not empty DO:
        current <- queue.popleft()

        FOR each (moveX, moveY) in directions DO:
            newRow <- current[0] + moveX
            newCol <- current[1] + moveY
            neighbor <- (newRow, newCol)

            IF 0 <= newRow < len(matrix) AND 0 <= newCol < len(matrix[0]) AND 
matrix[newRow][newCol] != -1 AND neighbor NOT IN visited THEN:
                parent[neighbor] <- current

                IF neighbor == end THEN:
                    RETURN create_path(parent)

                queue.append(neighbor)
                visited.add(neighbor)

    RETURN None
```

1.1.4 Complexity

Time complexity**:** O(**V**), where **V** is the number of vertices (cells) in the grid. Each cell is visited at most once.
Space complexity**:** O(**V**), where **V** is the number of vertices (cells) in the grid. The space is used for the `visited` set, `parent` dictionary, and the `queue`.

1.1.5 Correctness

The BFS algorithm is a fundamental algorithm used to find the shortest path in an unweighted graph, which is exactly the graph (the 2D matrix) being considered. The algorithm makes sure it explores all paths with the distance k before exploring any path with distance k + 1, which guarantees that the algorithm can find the shortest path before any longer, more inefficient paths.

1.2 Depth-First Search

1.2.1 Idea

The main idea of DFS (Depth-First Search) is to explore as deeply as possible along each branch of a graph or grid before backtracking. It uses a stack (or recursion) to keep track of nodes to be explored, diving into one path until it reaches a dead end, then backtracks and explores other paths. This approach ensures that every possible path is explored but may not always find the shortest path.

1.2.2 Step-by-step Description

- The DFS algorithm initializes a `visited` set, a `parent` dictionary, and a stack with the starting node. It marks the starting node as visited and then processes nodes from the stack. For each node, it explores its valid and unvisited adjacent nodes, records their parent, and adds them to the stack.
- The algorithm continues until the stack is empty or the goal node is found. If the goal is reached, it reconstructs and returns the path. If not, it returns None, indicating no path exists. DFS explores deeply along each branch before backtracking, which may lead to suboptimal paths.

1.2.3 Pseudocode

```
CLASS DFS INHERITS SearchAlgorithm:
    FUNCTION __init__(matrix, start, end):
        CALL super().__init__(matrix, start, end)

    FUNCTION Try():
        visited <- set()
        parent <- dictionary()
        stack <- [start]

        visited.add(start)

        WHILE stack is not empty DO:
            current <- stack.pop()

            FOR each (moveX, moveY) in directions DO:
                newRow <- current[0] + moveX
                newCol <- current[1] + moveY
                neighbor <- (newRow, newCol)

                IF 0 <= newRow < len(matrix) AND 0 <= newCol < len(matrix[0]) AND 
matrix[newRow][newCol] != -1 AND neighbor NOT IN visited THEN:
                    parent[neighbor] <- current

                    IF neighbor == end THEN:
                        RETURN create_path(parent)

                    stack.append(neighbor)
                    visited.add(neighbor)

        RETURN None
```

1.2.4 Complexity

Time complexity**:** O(**V**+**E**) where **V** is the number of vertices (cells) and **E** is the number of edges (connections). In the worst case, DFS explores each vertex and edge once.

Space complexity: O(**V**) this includes the space for the `visited` set, `parent` dictionary, and the stack used for recursion or explicit stack management. In the worst case, the stack or recursion depth can grow up to the number of vertices.

1.2.5 Correctness

The DFS algorithm can always find a path to a target, given that the graph is finite. This is because this algorithm finds all possible paths from the start. On the other hand, this way of navigating through the graph cannot guarantee the most efficient way to reach the target, as the algorithm tries to travel as deeply as possible into the graph, which may find longer, suboptimal paths.

1.3 Uniform-Cost Search

1.3.1 Idea

Uniform-Cost Search (UCS) is a search algorithm that finds the shortest path in a grid by exploring nodes based on their cumulative cost from the start node. It uses a priority queue to expand nodes with the lowest path cost first, ensuring that the first time it reaches the goal node, it has found the shortest path.

1.3.2 Step-by-step Description

- Uniform-Cost Search (UCS) begins by initializing a priority queue with the starting node, assigned a cost of 0 and an empty path. A `visited` set is also created to track nodes that have been explored. The algorithm then enters a loop where it processes nodes from the queue based on their cumulative cost, ensuring the node with the lowest cost is expanded first.
- As nodes are dequeued, UCS checks if the current node is the goal. If so, it returns the path to the goal. For each adjacent node, UCS calculates the new path cost and, if the node has not been visited, adds it to the queue for further exploration. Note that a node may be added to the queue multiple times; however, UCS ensures that the node with the minimum cost is chosen for expansion. This approach not only guarantees finding the shortest path but also improves efficiency by eliminating the need to store and update the cost of nodes separately, thereby reducing space complexity. The process continues until the goal node is reached or the queue is emptied, in which case it returns `None` if no path was found.

1.3.3 Pseudocode

```
CLASS UCS INHERITS SearchAlgorithm:
    FUNCTION __init__(matrix, start, end):
        CALL super().__init__(matrix, start, end)

    FUNCTION Try():
    queue <- [(0, start, [])]  # priority queue with cost, node, and path
    visited <- set()

    WHILE queue is not empty DO:
        (cost, current, path) <- queue.pop(0)

        IF current is end THEN:
            RETURN path + [end]

        IF current NOT IN visited THEN:
            visited.add(current)
            newPath <- path + [current]

            FOR each (moveX, moveY) in directions DO:
                newRow <- current[0] + moveX
                newCol <- current[1] + moveY
                neighbor <- (newRow, newCol)

                IF 0 <= newRow < len(matrix) AND 0 <= newCol < len(matrix[0]) AND
matrix[newRow][newCol] != -1 AND neighbor NOT IN visited THEN:
                    queue.append((cost + 1, neighbor, newPath))

    RETURN None
```

1.3.4 Complexity

Time complexity**:** O($b^{1+\lfloor C*/\epsilon \rfloor}$) where b is the branching factor (i.e. the number of available actions in each state), $C*$ is the cost of the optimal solution, and ϵ>0 is a lower bound of the cost of each action.

Space complexity**:** O($b^{1+\lfloor C*/\epsilon \rfloor}$) the space complexity is the same as the time complexity.

1.3.5 Correctness:

The UCS algorithm is a variant of Dijkstra's algorithm, which can find the shortest path from a starting point to a target within a weighted or unweighted graph. There are various proofs that had been made, including the proof using mathematical induction. (Proof)

1.4 Greedy Best First Search

1.4.1 Idea

Greedy Best-First Search (GBFS) is a search algorithm designed to find a path from a starting node to a goal node by exploring nodes based on their heuristic estimates of the cost to reach the goal. The algorithm prioritizes nodes that are estimated to be closer to the goal, using only the heuristic function to guide the search.

1.4.2 Step-by-step Description

- GBFS begins by initializing a priority queue with the starting node, using the heuristic value of 0. A visited set is created to track explored nodes, and a parent dictionary records the path. The algorithm processes nodes from the priority queue, selecting nodes with the lowest heuristic cost first.
- During each iteration, GBFS checks if the current node is the goal. If it is, the algorithm reconstructs and returns the path. For each valid adjacent node, it calculates the heuristic cost to the goal and adds the node to the priority queue if it hasn't been visited. The process continues until the goal node is reached or the queue is empty, returning None if no path is found. GBFS does not consider the path cost but only the heuristic, which may lead to non-optimal paths.

1.4.3 Pseudocode

```
CLASS UCS INHERITS SearchAlgorithm:
    FUNCTION __init__(matrix, start, end):
        CALL super().__init__(matrix, start, end)

    FUNCTION Try:
    visited <- set()
    parent <- dictionary()
    priority_queue <- []

    heapq.heappush(priority_queue, (0, start))

    WHILE priority_queue is not empty DO:
        (_, current) <- priority_queue.pop(0)

        IF current is end THEN:
            RETURN create_path(parent)

        IF current NOT IN visited THEN:
            visited.add(current)

            FOR each (moveX, moveY) in directions DO:
                newRow <- current[0] + moveX
                newCol <- current[1] + moveY
                neighbor <- (newRow, newCol)

                IF 0 <= newRow < len(matrix) AND 0 <= newCol < len(matrix[0]) AND
matrix[newRow][newCol] != -1 AND neighbor NOT IN visited THEN:
                    parent[neighbor] <- current
                    heuristic <- manhattan_distance(neighbor, end)
                    heapq.heappush(priority_queue, (heuristic, neighbor))

    RETURN None
```

1.4.4 Complexity

Time complexity: worst-case time complexity of O$(\boldsymbol{b^m})$, where **b** is the branching factor (average number of successors per node) and **m** is the maximum depth of the search space.

Space complexity: O**(b * m)**, where **b** is the branching factor (average number of successors per node) and **m** is the maximum depth of the search space.

1.4.5 Correctness

The GBFS algorithm does not take the cost of the path into consideration, rather it uses heuristic functions. This can cause the algorithm to get stuck in an infinite path or follow suboptimal paths.

1.5 A* (A Star)

1.5.1 Idea

The A* search algorithm is an informed search algorithm used to find the shortest path between a starting node and a goal node in a graph. It combines the efficiency of UCS's algorithm with the heuristic information provided by the heuristic function (in this case, Manhattan distance) to guide the search towards the goal.

1.5.2 Step-by-step Description

- A priority queue queue isinitialized with a tuple containing the initial cost (0), the current cost (0), the start node, and an empty path.
- A set `visited` is initialized to keep track of nodes that have already been visited.
- An empty list `path` is initialized to store the path.

- The algorithm processes nodes from the priority queue until it is empty.
- For each node, it checks if the node has been visited. If not, it adds the node to the `visited` set.
- A new path is created by appending the current node to the existing path.
- If the current node is the goal node, the path is returned.
- For each possible move `{(i+1, j), (i-1, j), (i, j+1), (i, j-1),` and `(i, j)}`, the algorithm calculates the new row and column for the neighbor node.
- If the neighbor node is within the matrix bounds and is not an obstacle (-1), and has not been visited, the new cost (`newGCost`) and the total estimated cost (`fCost`) are calculated.
- The neighbor node and the updated costs are pushed onto the priority queue.

1.5.3 Pseudocode

```
CLASS AStar INHERITS SearchAlgorithm:
    FUNCTION __init__(matrix, start, end):
        CALL super().__init__(matrix, start, end)

    FUNCTION Try():
        queue <- [(0, 0, start, empty list)]  # priority queue with cost, current
node, and path
        visited <- set()
        path <- empty list

        WHILE queue is not empty DO
            (fcost, cost, current, path) <- heapq.heappop(queue)

            IF current NOT IN visited THEN
                visited.add(current)
                newPath <- path + [current]

                IF current == end THEN
                    RETURN newPath

                FOR each (moveX, moveY) IN directions DO
                    newRow <- current[0] + moveX
                    newCol <- current[1] + moveY
                    neighbor <- (newRow, newCol)

                    IF 0 <= newRow < len(matrix) AND 0 <= newCol < len(matrix[0]) THEN
                        IF matrix[newRow][newCol] != -1 AND neighbor NOT IN visited
THEN
                            newGCost <- cost + 1
                            fCost <- cost + 1 + manhattan_distance(neighbor, end)
                            heapq.heappush(queue, (fCost, newGCost, neighbor,
newPath))

        RETURN None  # no path found
```

1.5.4 Complexity

Time complexity: O(b^d) where b is the branching factor and d is the depth of the optimal solution.

Space complexity: O(b^d) due to storing all generated nodes in the priority queue and visited set.

1.5.5 Correctness

The correctness of the A* algorithm depends on the heuristic function being admissible and consistent or not. The heuristic used in this implementation (Manhattan distance) is admissible and consistent, which makes the A* algorithm complete and optimal – it can always find the shortest path from the start node to the goal node. The following is the proof by contradiction:

- Assume that A* does not find the shortest path. Let P be the path found by A* and P' be the actual shortest path. Let $g(P)$ be the cost of path P, $g(P')$ be the cost of path P'.
- By assumption, $g(P') < g(P)$.
- Since A* uses $f(n) = g(n) + h(n)$ and expands nodes in order of increasing cost of function f, the node n on the optimal path P' would have $f(n) \leq g(P')$ (because heuristic function h is admissible and consistent).
- If $f(n) \leq g(P')$ and $g(P') < g(P)$, then $f(n) < g(P)$. Then A* would have expanded node n before expanding any node on path P (since $f(n) < f(P)$).
- Therefore, A* would have found and expanded the nodes on the optimal path P' before completing path P, contradicting the assumption that A* did not find the shortest path.
- Hence, our initial assumption is incorrect, and A* must find the shortest path.

1.6 Class **SearchAlgorithm**:

```
CLASS SearchAlgorithm:
    FUNCTION __init__(matrix, start, end):
        self.matrix <- matrix
        self.start <- start
        self.end <- end
        self.directions <- [(-1, 0), (1, 0), (0, -1), (0, 1)]

    FUNCTION create_path(parent):
        path <- empty list
        current <- end

        WHILE current is NOT None DO:
            path.append(current)
            next_node <- parent.get(current)
            IF next_node is None THEN:
                BREAK
            current <- next_node

        path.reverse()

        IF path[0] != start THEN:
            RETURN None

        RETURN path

    FUNCTION Try():
        PASS
```

- Support functions:

```
FUNCTION manhattan_distance(point1, point2):
    x1, y1 <- point1
    x2, y2 <- point2
    RETURN abs(x1 - x2) + abs(y1 - y2)

FUNCTION level1(algorithm):
    adjacency_matrix <- mat
    start_node <- agent['S']
    goal_node <- goal['G']

    algorithms <- Dictionary{
            "Depth First Search": DFS,
            "Uniform Cost Search": UCS,
            "A*": AStar,
            "Breadth First Search": BFS,
            "Greedy Best First Search": GBFS
        }
        TRY:
            RETURN algorithms[algorithm](adjacency_matrix, start_node,
goal_node).Try()
        EXCEPT:
            RAISE ValueError("Invalid algorithm name")
```

2. Level 2 (Time Limitation)

2.1 Idea

The goal of this algorithm is to find the shortest path from a starting position to an ending positio under a specified time limit. The grid can contain cells with different traversal times, and the algorithm must account for these when calculating the shortest path. The algorithm uses a modified Breadth-First Search (BFS) approach with a priority queue to handle the time constraints effectively.

2.2 Step-by-step Description

Algorithm: Modifed BFS on 3-dimensional map

- Consider the map as a 3-dimensional map with the third dimension is the elapsed time.
- Each cell will be divided into multiple cells corresponding to each moment it is traversed.
- Start at cell `(Si, Sj, 0)` and end at cell `(Gi, Gj, t)` with `t` within the acceptable range.
- From cell `(i, j, t)`, we can move to four adjacent cells `(i+1, j, nextTime)`, `(i-1, j, nextTime)`, `(i, j+1, nextTime)`, `(i, j-1, nextTime)`, and `(i, j, nextTime)`. Except when that cell is a wall, a waiting cell, or `nextTime > time_limit` (time limit exceeded).
    - `nextTime = t + 1` if that cell is not a waiting cell.
    - `nextTime = t + 1 + waiting_time` if that cell is a waiting cell.
    - If that cell is a wall or `nextTime > time_limit`, that cell is skipped.
- When reaching cell `(Gi, Gj, t)` with any `t <= time_limit`, we stop and return the shortest path from `(Si, Sj, 0)` to `(Gi, Gj, t)`.
- The path is guaranteed to be the shortest among the valid path because BFS is used in the searching process.

2.3 Pseudocode

```
FUNCTION level2(mat, time, start, end)
    distance_matrix <- array of size [len(mat)][len(mat[0])][time + 10] filled with
infinity
    distance_matrix[start[0]][start[1]][0] <- 0
    queue <- [(start[0], start[1], 0)]

    WHILE queue is not empty DO
        (row, col, curr_time) <- queue.pop(0)
        cur_dis <- distance_matrix[row][col][curr_time]

        IF curr_time > time THEN
            CONTINUE

        IF (row, col) == end THEN
            path <- []
            tmp_time <- curr_time
            WHILE (row, col) != start DO
                path.append((row, col))
                extra_time <- 1
                IF mat[row][col] is an integer THEN
                    extra_time <- mat[row][col] + 1
                FOR each (dr, dc) in [(-1, 0), (1, 0), (0, -1), (0, 1)] DO
                    nr <- row + dr
                    nc <- col + dc
                    prv_time <- tmp_time - extra_time
                    IF 0 <= nr < len(mat) AND 0 <= nc < len(mat[0]) AND
distance_matrix[nr][nc][prv_time] < distance_matrix[row][col][tmp_time] THEN

                        IF mat[nr][nc] == -1 THEN
                            CONTINUE
                        row, col <- nr, nc
                        tmp_time <- prv_time
                        BREAK
            RETURN path[::-1]

        FOR each (dr, dc) in [(-1, 0), (1, 0), (0, -1), (0, 1)] DO
            nr <- row + dr
            nc <- col + dc
            IF 0 <= nr < len(mat) AND 0 <= nc < len(mat[0]) AND mat[nr][nc] != -1 THEN
                cell <- mat[nr][nc]
                new_time <- curr_time + (0 IF cell == 0 OR cell is a string ELSE cell)
+ 1
                IF cur_dis + 1 < distance_matrix[nr][nc][new_time] THEN
                    distance_matrix[nr][nc][new_time] <- cur_dis + 1
                    queue.append((nr, nc, new_time))

    RETURN []
```

2.4 Complexity

Time complexity: O(V * T) where V is the number of vertices (cells) in the grid, and T is the maximum time limit.

Space complexity: O(V * T) due to the *distance_matrix* which stores distances for each cell at each time step. The queue also contributes to the space complexity but is bounded by the number of cells and the maximum time limit.

2.5 Correctness

This implementation is a modified version of the BFS algorithm, only with an extra dimension in the distance array to store the time traveled. This does not change the fact that the algorithm must find all paths with the distance k from the start before finding paths with distance $k + 1$ from the start node, which guarantees that the algorithm will find the shortest possible path.

3. Level 3 (Fuel Limitation)

3.1 Idea

The goal of this algorithm is to find the shortest path under both a specified time limit and fuel constraints. The grid can contain cells with different traversal times, fuel stations for refueling, and obstacles that cannot be crossed. The algorithm uses a modified Breadth-First Search (BFS) approach on a 4-dimensional grid, with 2 extra dimensions represent the elapsed time and fuel levels. This method ensures that all possible states (combinations of time and fuel at each cell) are explored to find the shortest path that adheres to the constraints.

3.2 Step-by-step Description

- Consider the map as a 4-dimensional map with the third and fourth dimensions being the elapsed time and remaining fuel of the vehicle.
- Each cell will be divided into each moment it is traversed and the remaining fuel of the vehicle.
- Start at cell `(Si, Sj, 0, fuel_capacity)` and end at cell `(Gi, Gj, t, fuel)` with `t` and `fuel` being any value.
- From cell `(i, j, t, fuel)`, we can move to four adjacent cells `(i+1, j, nextTime, nextFuel)`, `(i-1, j, nextTime, nextFuel)`, `(i, j+1, nextTime, nextFuel)`, `(i, j-1, nextTime, nextFuel)`, and `(i, j, nextTime, nextFuel)`. Except when that cell is a wall, a waiting cell, a gas station, `nextTime > time_limit` (time limit exceeded), or `nextFuel < 0`.
  - `nextTime = t+1` if that cell is not a waiting cell.
  - `nextTime = t+1+waiting_time` if that cell is a waiting cell.
  - `nextFuel = fuel_capacity` if that cell is a gas station.
  - `nextFuel = fuel-1` if that cell is not a gas station.
  - If that cell is a wall, `nextTime > time_limit`, or `nextFuel < 0`, that cell is skipped.
- When reaching cell `(Gi, Gj, t, fuel)` with any `t <= time_limit` and `fuel >= 0`, stop and return the shortest path from `(Si, Sj, 0, fuel_capacity)` to `(Gi, Gj, t, fuel)`.
- The path is guaranteed to be the shortest among the valid path because BFS is used in the searching process.

3.3 Pseudocode

```
FUNCTION level3(mat, time, fuel, start, end)
    distance_matrix <- array of size [len(mat)][len(mat[0])][time + 10][fuel + 10]
filled with infinity
    distance_matrix[start[0]][start[1]][0][fuel] <- 0
    queue <- [(start[0], start[1], 0, fuel)]

    WHILE queue is not empty DO
        (i, j, t, f) <- queue.pop(0)
        cur_dis <- distance_matrix[i][j][t][f]

        IF f < 0 OR t > time THEN
            CONTINUE

        IF (i, j) == end THEN
            path <- []
            tmp_time <- t
            tmp_fuel <- f
            WHILE (i, j) != start DO
                path.append((i, j))
                extra_time <- 1
                IF mat[i][j] is an integer THEN
                    extra_time <- mat[i][j] + 1
                ELSE IF mat[i][j][0] == 'F' THEN
                    extra_time <- int(mat[i][j][1:]) + 1
                FOR each (dr, dc) in [(-1, 0), (1, 0), (0, -1), (0, 1)] DO
                    ni <- i + dr
                    nj <- j + dc
                    IF 0 <= ni < len(mat) AND 0 <= nj < len(mat[0]) THEN
                        prv_time <- tmp_time - extra_time
                        IF distance_matrix[ni][nj][prv_time][tmp_fuel + 1] <
distance_matrix[i][j][tmp_time][tmp_fuel] THEN
                            i, j <- ni, nj
                            tmp_fuel <- tmp_fuel + 1
                            tmp_time <- prv_time
                            BREAK
                        ELSE IF tmp_fuel == fuel THEN
                            found_path <- FALSE
                            FOR pre_fuel in range(fuel) DO
                                IF distance_matrix[ni][nj][prv_time][pre_fuel] <
distance_matrix[i][j][tmp_time][tmp_fuel] THEN
                                    found_path <- TRUE
                                    tmp_fuel <- pre_fuel
                                    i, j <- ni, nj
                                    tmp_time <- prv_time
                                    BREAK
                            IF found_path == TRUE THEN
                                BREAK
            path.append(start)
            RETURN path[::-1]
```

```
        FOR each (dr, dc) in [(-1, 0), (1, 0), (0, -1), (0, 1)] DO
            ni <- i + dr
            nj <- j + dc
            IF ni < 0 OR nj < 0 OR ni >= len(mat) OR nj >= len(mat[0]) THEN
                CONTINUE
            IF mat[ni][nj] == -1 THEN
                CONTINUE
            next_fuel <- f - 1
            refuel_time <- 0
            stay_time <- 1
            IF mat[ni][nj] is a string AND mat[ni][nj][0] == 'F' THEN
                next_fuel <- fuel
                refuel_time <- int(mat[ni][nj][1:])
            ELSE
                stay_time <- max(stay_time, mat[ni][nj] + 1)
            total_time <- t + stay_time + refuel_time
            IF cur_dis + 1 < distance_matrix[ni][nj][total_time][next_fuel] THEN
                queue.append((ni, nj, total_time, next_fuel))
                distance_matrix[ni][nj][total_time][next_fuel] <- cur_dis + 1
    RETURN []
```

3.4 Complexity

Given:

- R as the number of rows in the map
- C as the number of columns in the map
- T as the time limit
- F as the fuel capacity

Time complexity: O(R*C*T*F)

Space complexity: O(R*C*T*F)

3.5 Correctness

As this implementation is again a modification of the BFS algorithm, the algorithm guarantees to find the shortest path possible.

4. Level 4 (Multiple Agents)

4.1 Idea

The goal of this algorithm is to find a path for multiple agents from their starting positions to their respective goals under time and fuel constraints. The grid contains fuel stations that allow agents to refuel. The algorithm uses a combination of BFS and heuristics to find the shortest path while managing the fuel consumption. It iteratively attempts to move the agents towards their goals, refueling at stations as necessary and dynamically adjusting the goals if an agent reaches its destination.

4.2 Step-by-step Description

1. Initialize Fuel Distance Map:
   - Call `getFuelDistance()` to create a heuristic map of fuel distances.
2. Check for Wall Blocking Goal:
   - Execute `level3()`. If it returns -1, terminate the process as the goal is blocked by a wall.
3. Initialize Variables:
   - Initialize `path` as an empty list.
   - Initialize `fuel` list with the fuel capacity for each agent (considering up to 5 additional agents).
   - Set a counter `cnt` to 0.
4. Main Loop: Repeat until the goal is reached or fuel is exhausted:
   - Increment `cnt` by 1.
   - If the fuel for the first agent is 0, set `path` to -1 and break the loop.
   - If the first agent has reached the goal, break the loop.
   - Add the current position of the first agent to `path`.
   - For each agent:
     - Determine `start` and `goal` positions.
     - If the agent is at its goal, skip to the next agent.
     - Call `findPath(start, goal, fuel[idx])` to find the path from `start` to `goal` with the current fuel level.
     - If the path is -1, continue to the next agent.
     - Move the agent to the next cell in the path.
     - Decrease the agent's fuel by 1.
     - If the agent is at a fuel station, refill its fuel to full capacity.
     - If the agent is the first agent, add the next cell in the path to `path`.
     - If the agent reaches its goal, generate a new goal for the agent if it's not the first agent.
5. Return Path:
   - Return the `path` list which contains the sequence of cells the first agent traversed to reach the goal.

4.3 Pseudocode

- Main function:

```
FUNCTION level4(mat, time, fuel, start, end, agent):

    IF level3() == -1 THEN
        RETURN (-1, -1)

    path <- 2D array of empty lists, size [len(agent)][]
    fuel <- array of size len(agent) + 5 filled with initial fuel value
    goalList <- []

    FOR i FROM 0 TO len(agent) - 1 DO:
        pos <- agent['S']
        IF i > 0 THEN
            pos <- agent['S' + str(i)]
        path[i].append(pos)

    cnt <- 0
    goalIdx <- 0

    WHILE TRUE DO
        IF fuel[0] == 0 THEN
            path <- -1
            goalList <- -1
            BREAK

        IF isGoal(0) THEN
            BREAK

        FOR idx FROM 0 TO len(agent) - 1 DO

            cnt <- cnt + 1
            start, goal <- (-1, -1), (-1, -1)

            IF idx == 0 THEN
                start <- findPos('S')
                goal <- findGoal('G')
            ELSE
                start <- findPos('S' + str(idx))
                goal <- findGoal('G' + str(idx))

            IF fuel[idx] == 0 THEN
                path[idx].append(path[idx][LAST])
                CONTINUE

            curPath <- findPath(start, goal, fuel[idx])

            IF curPath == -1 THEN
                path[idx].append(path[idx][LAST])
                goalList.append([])
                FOR j FROM 0 TO len(agent) - 1 DO
                    IF j > 0 THEN
                        goalList[goalIdx].append((j, findGoal('G' + str(j))))
                    ELSE
                        goalList[goalIdx].append((j, findGoal('G')))
                goalIdx <- goalIdx + 1
                CONTINUE
```

```
            goToCell(curPath[0], curPath[1])
            fuel[idx] <- fuel[idx] - 1

            IF isStation(curPath[1][0], curPath[1][1]) THEN
                fuel[idx] <- initial fuel value

            path[idx].append(curPath[1])

            IF isGoal(idx) THEN
                IF idx == 0 THEN
                    BREAK
                generateNewGoal(idx)

            goalList.append([])
            FOR j FROM 0 TO len(agent) - 1 DO
                IF j > 0 THEN
                    goalList[goalIdx].append((j, findGoal('G' + str(j))))
                ELSE
                    goalList[goalIdx].append((j, findGoal('G')))
            goalIdx <- goalIdx + 1

    RETURN (path, goalList)
```

- Support functions:

```
FUNCTION level3(mat, time, fuel, start, end)
    distance_matrix <- array of size [len(mat)][len(mat[0])][time + 10][fuel + 10]
filled with infinity
    distance_matrix[start[0]][start[1]][0][fuel] <- 0
    queue <- [(start[0], start[1], 0, fuel)]

    WHILE queue is not empty DO
        (i, j, t, f) <- queue.pop(0)
        cur_dis <- distance_matrix[i][j][t][f]

        IF f < 0 OR t > time THEN
            CONTINUE

        IF (i, j) == end THEN
            path <- []
            tmp_time <- t
            tmp_fuel <- f
            WHILE (i, j) != start DO
                path.append((i, j))
                extra_time <- 1
                IF mat[i][j] is an integer THEN
                    extra_time <- mat[i][j] + 1
                ELSE IF mat[i][j][0] == 'F' THEN
                    extra_time <- int(mat[i][j][1:]) + 1
                FOR each (dr, dc) in [(-1, 0), (1, 0), (0, -1), (0, 1)] DO
                    ni <- i + dr
                    nj <- j + dc
                    IF 0 <= ni < len(mat) AND 0 <= nj < len(mat[0]) THEN
                        prv_time <- tmp_time - extra_time
                        IF distance_matrix[ni][nj][prv_time][tmp_fuel + 1] <
distance_matrix[i][j][tmp_time][tmp_fuel] THEN
                            i, j <- ni, nj
                            tmp_fuel <- tmp_fuel + 1
                            tmp_time <- prv_time
                            BREAK
```

```
                        ELSE IF tmp_fuel == fuel THEN
                            found_path <- FALSE
                            FOR pre_fuel in range(fuel) DO
                                IF distance_matrix[ni][nj][prv_time][pre_fuel] <
distance_matrix[i][j][tmp_time][tmp_fuel] THEN
                                    found_path <- TRUE
                                    tmp_fuel <- pre_fuel
                                    i, j <- ni, nj
                                    tmp_time <- prv_time
                                    BREAK
                            IF found_path == TRUE THEN
                                BREAK
            path.append(start)
            RETURN path[::-1]

        FOR each (dr, dc) in [(-1, 0), (1, 0), (0, -1), (0, 1)] DO
            ni <- i + dr
            nj <- j + dc
            IF ni < 0 OR nj < 0 OR ni >= len(mat) OR nj >= len(mat[0]) THEN
                CONTINUE
            IF mat[ni][nj] == -1 THEN
                CONTINUE
            next_fuel <- f - 1
            refuel_time <- 0
            stay_time <- 1
            IF mat[ni][nj] is a string AND mat[ni][nj][0] == 'F' THEN
                next_fuel <- fuel
                refuel_time <- int(mat[ni][nj][1:])
            ELSE
                stay_time <- max(stay_time, mat[ni][nj] + 1)
            total_time <- t + stay_time + refuel_time
            IF cur_dis + 1 < distance_matrix[ni][nj][total_time][next_fuel] THEN
                queue.append((ni, nj, total_time, next_fuel))
                distance_matrix[ni][nj][total_time][next_fuel] <- cur_dis + 1
    RETURN []

FUNCTION initIntrMap()
    FOR i FROM 0 TO len(intrMap) - 1 DO
        FOR j FROM 0 TO len(intrMap[0]) - 1 DO
            IF intrMap[i][j] is an integer THEN
                IF intrMap[i][j] != 0 AND intrMap[i][j] != -1 THEN
                    intrMap[i][j] <- 0
            ELSE
                IF intrMap[i][j][0] != 'S' THEN
                    intrMap[i][j] <- 0
    RETURN

FUNCTION revert(revList)
    FOR each info in revList DO
        (i, j, cell) <- info
        intrMap[i][j] <- cell
    RETURN
```

```
FUNCTION findPath(start, end, fuel)
    dis <- 4D array of size len(intrMap) x len(intrMap[0]) x (time + 10) x (fuel + 10)
filled with infinity
    dis[start[0]][start[1]][0][fuel] <- 0
    queue <- [(start[0], start[1], 0, fuel)]

    dx <- [0, 1, 0, -1]
    dy <- [1, 0, -1, 0]
    tmpWall <- empty list

    FOR k FROM 0 TO 3 DO
        IF start[0] + dx[k] < 0 OR start[1] + dy[k] < 0 OR start[0] + dx[k] >=
len(intrMap) OR start[1] + dy[k] >= len(intrMap[0]) THEN
            CONTINUE
        IF intrMap[start[0] + dx[k]][start[1] + dy[k]] is a string AND
intrMap[start[0] + dx[k]][start[1] + dy[k]][0] == 'S' THEN
            tmpWall.append((start[0] + dx[k], start[1] + dy[k], intrMap[start[0] +
dx[k]][start[1] + dy[k]]))
            intrMap[start[0] + dx[k]][start[1] + dy[k]] <- -1

    WHILE queue is not empty DO
        (row, col, cur_time, cur_fuel) <- queue.pop(0)
        cur_dis <- dis[row][col][cur_time][cur_fuel]

        IF cur_fuel < 0 OR cur_time > time THEN
            CONTINUE

        IF (row, col) == end THEN
            path <- empty list
            tmp_fuel <- cur_fuel
            tmp_time <- cur_time

            WHILE (row, col) != start DO
                path.append((row, col))
                extra_time <- 1

                IF mat[row][col] is an integer THEN
                    extra_time <- mat[row][col] + 1
                ELSE IF mat[row][col][0] == 'F' THEN
                    extra_time <- int(mat[row][col][1:]) + 1

                FOR each (dr, dc) in [(-1, 0), (1, 0), (0, -1), (0, 1)] DO
                    ni <- row + dr
                    nj <- col + dc
                    IF 0 <= ni < len(intrMap) AND 0 <= nj < len(intrMap[0]) THEN
                        prv_time <- tmp_time - extra_time
                        IF dis[ni][nj][prv_time][tmp_fuel + 1] <
dis[row][col][tmp_time][tmp_fuel] THEN
                            row, col <- ni, nj
                            tmp_fuel <- tmp_fuel + 1
                            tmp_time <- prv_time
                            BREAK
```

```
                        ELSE IF tmp_fuel == fuel THEN
                            found_path <- FALSE
                            FOR pre_fuel FROM 0 TO fuel - 1 DO
                                IF dis[ni][nj][prv_time][pre_fuel] <
dis[row][col][tmp_time][tmp_fuel] THEN
                                    found_path <- TRUE
                                    tmp_fuel <- pre_fuel
                                    row, col <- ni, nj
                                    tmp_time <- prv_time
                                    BREAK
                            IF found_path == TRUE THEN
                                BREAK

            path.append(start)
            revert(tmpWall)
            RETURN path[::-1]

        FOR k FROM 0 TO 3 DO
            next_row <- row + dx[k]
            next_col <- col + dy[k]

            IF next_row < 0 OR next_col < 0 OR next_row >= len(intrMap) OR next_col >=
len(intrMap[0]) THEN
                CONTINUE
            IF intrMap[next_row][next_col] == -1 THEN
                CONTINUE

            next_fuel <- cur_fuel - 1
            refuel_time <- 0
            stay_time <- 1

            IF mat[next_row][next_col] is a string AND mat[next_row][next_col][0] ==
'F' THEN
                next_fuel <- fuel
                refuel_time <- int(mat[next_row][next_col][1:])
            ELSE
                stay_time <- max(stay_time, mat[next_row][next_col] + 1)

            total_time <- cur_time + stay_time + refuel_time

            IF cur_dis + 1 < dis[next_row][next_col][total_time][next_fuel] THEN
                queue.append((next_row, next_col, total_time, next_fuel))
                dis[next_row][next_col][total_time][next_fuel] <- cur_dis + 1

    revert(tmpWall)

    IF isStation(start[0], start[1]) THEN
        FOR k FROM 0 TO 3 DO
            next_row <- start[0] + dx[k]
            next_col <- start[1] + dy[k]
            IF intrMap[next_row][next_col] == 0 THEN
                path <- [(start[0], start[1]), (next_row, next_col)]
                RETURN path
    RETURN -1

FUNCTION findPos(cell)
    FOR i FROM 0 TO len(intrMap) - 1 DO
        FOR j FROM 0 TO len(intrMap[0]) - 1 DO
            IF intrMap[i][j] == cell THEN
                RETURN (i, j)
    RETURN -1
```

```
FUNCTION findGoal(cell)
    FOR i FROM 0 TO len(mat) - 1 DO
        FOR j FROM 0 TO len(mat[0]) - 1 DO
            IF mat[i][j] == cell THEN
                RETURN (i, j)
    RETURN -1

FUNCTION isGoal(idx)
    start, goal <- 'S', 'G'
    IF idx > 0 THEN
        start <- 'S' + str(idx)
        goal <- 'G' + str(idx)

    FOR i FROM 0 TO len(intrMap) - 1 DO
        FOR j FROM 0 TO len(intrMap[0]) - 1 DO
            IF intrMap[i][j] == start AND mat[i][j] == goal THEN
                RETURN TRUE
    RETURN FALSE

FUNCTION isStation(row, col)
    IF mat[row][col] is a string AND mat[row][col][0] == 'F' THEN
        RETURN TRUE
    RETURN FALSE

FUNCTION goToCell(start, goal)
    intrMap[start[0]][start[1]], intrMap[goal[0]][goal[1]] <-
intrMap[goal[0]][goal[1]], intrMap[start[0]][start[1]]
    RETURN

FUNCTION generateNewGoal(idx)
    cellList <- empty list
    goalLabel <- 'G' + str(idx)

    FOR i FROM 0 TO len(mat) - 1 DO
        FOR j FROM 0 TO len(mat[0]) - 1 DO
            IF mat[i][j] == goalLabel THEN
                RETURN (i, j)
    RETURN -1
```

4.4 Complexity

Given:

- S: number of stations
- R: number of rows
- C: number of columns
- T: time limit
- F: fuel capacity

Time complexity: O(S*R*C*T*F)

Space complexity: O(R*C*T*F)

6. Implementation

1. Programming Language

The program was implemented using Python (3.11.4), a high-level programming language known for its simplicity and readability. Python was chosen due to its extensive libraries and community support which were a great help in the development process

2. Supporting Libraries

Several Python libraries were utilized to aid in the implementation:

- *heapq*: Implement priority queues required for algorithms like UCS, GBFS, and A*.
- *time*: Measure the runtime of each algorithm.
- *deque (from collections):* Used to implement efficient queues for BFS, which requires fast append and pop operations from both ends of the queue.
- *tkinter:* Provides a graphical user interface to visualize the grid and the paths found by the algorithms, making it easier to understand the algorithm's performance.
- *copy:* Utilized for deep copying complex data structures, ensuring that modifications in one part of the program do not unintentionally affect other parts.
- *random:* Generates random test cases and grids to evaluate the performance and robustness of the implemented algorithms.

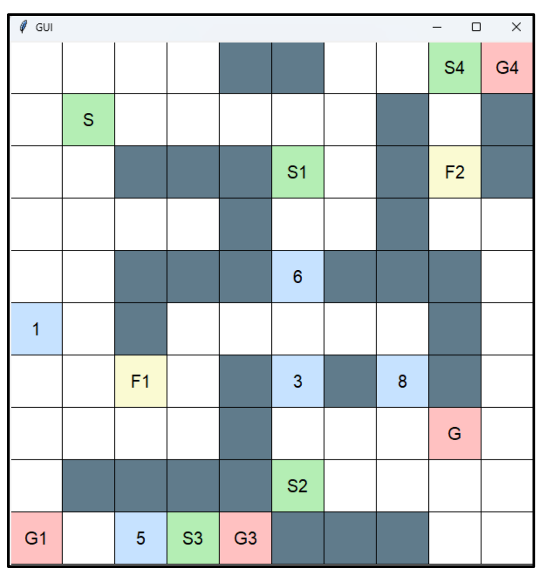
3. Input File Format

The input file is numbered according to the convention *input1_level1.txt, input1_level2.txt*, etc. The input file format is described as follows:

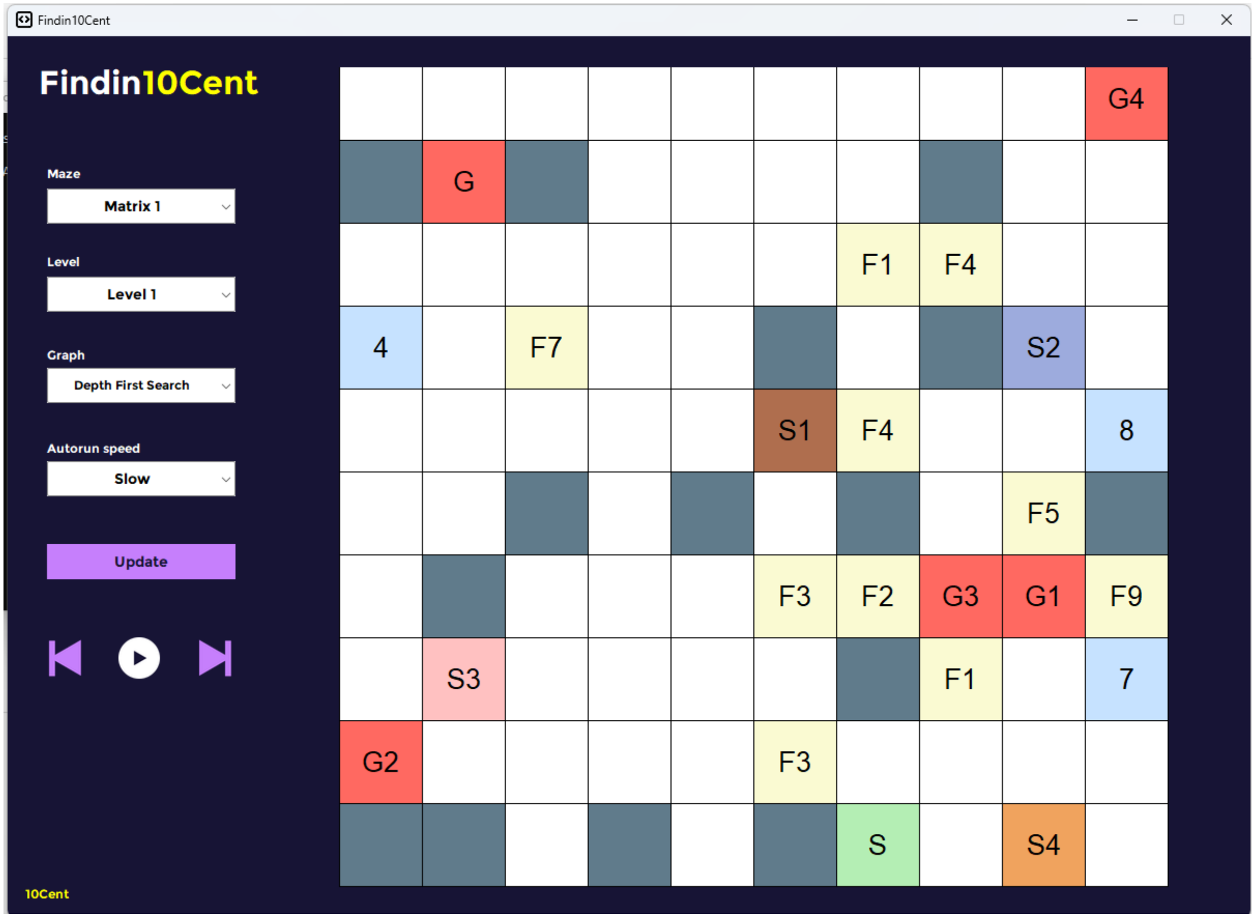
- The first line contains 4 positive integers, corresponding to the number of rows and columns of the map, committed delivery time, and fuel tank capacity.
- The following lines represent the information on the map.

Input file example: *input.txt*

```
10 10 100 10
0 0 0 0 -1 -1 0 0 S4 G4
0 S 0 0 0 0 0 -1 0 -1
0 0 -1 -1 -1 S1 0 -1 F2 -1
0 0 0 0 -1 0 0 -1 0 0
0 0 -1 -1 -1 6 -1 -1 -1 0
1 0 -1 0 0 0 0 0 -1 0
0 0 F1 0 -1 3 -1 8 -1 0
0 0 0 0 -1 0 0 0 G 0
0 -1 -1 -1 -1 S2 0 0 0 0
G1 0 5 S3 G3 -1 -1 -1 0 0
```



4. Graphical User Interface (GUI)



The GUI for our delivery system visualization includes the following features:

- Map Display: The 2D map is displayed, showing entities llike walls, paths, starting locations, goal locations, toll booths and gas stations.
- Path Visualization: The calculated path from the starting location to the goal is highlighted by the name and color of the entity (S, S1, S2, etc.).
- Step-by-Step Execution: Users can visualize the search process step-by-step.
- Control Buttons: The GUI includes buttons to visualize the next step, previous step and can automate the visualization with customizable speed or interval between each steps.
- Algorithm Selection: A dropdown menu allows users to select which level to visualize and which algorithm to perform (for level 1)
- Map selection: Each level comes with a set of 5 default maps with different attributes for the users to choose.

7. Test Cases and Results

Level 1

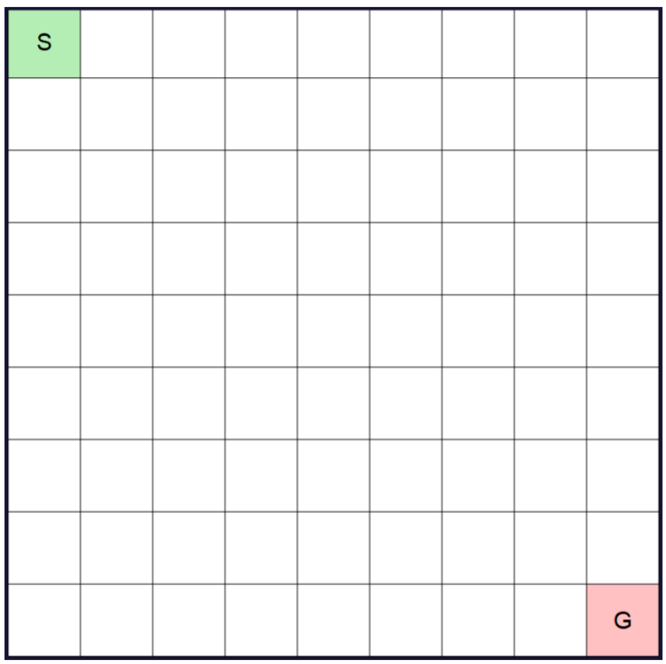
1.1 Test 1: Heuristics Favoring

1.1.1 Description

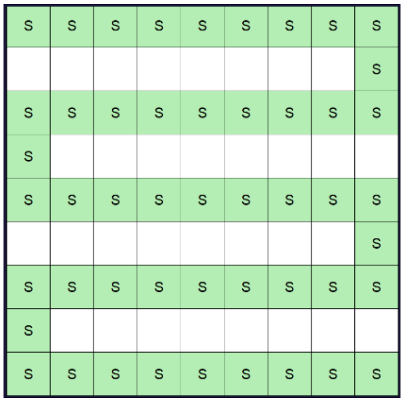
For algorithms using heuristics, the shortest path can be found without having to explore unnecessary paths, which is more advantageous than algorithms that do not use heuristics.

For uninformed search algorithms, the shortest path can also be found but it requires visiting unnecessary paths and cells.

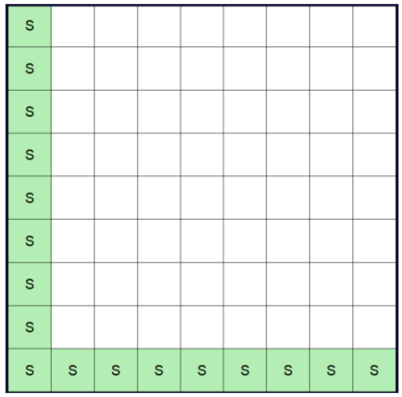
1.1.2 Visualization:



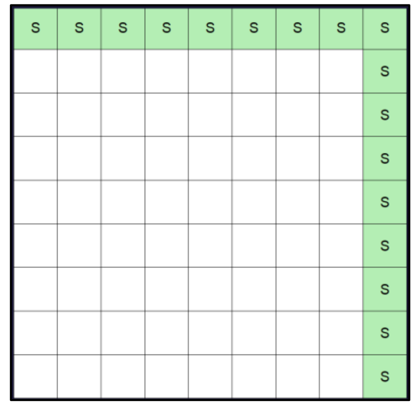
1.1.3 Result



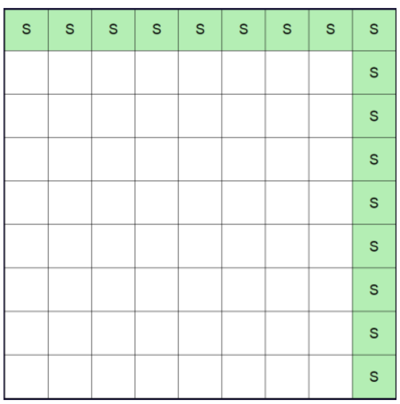
DFS



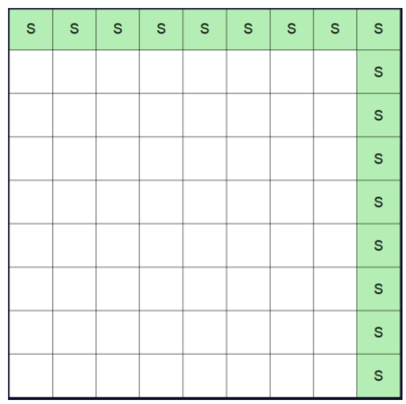
BFS



GBFS



UCS



*A**

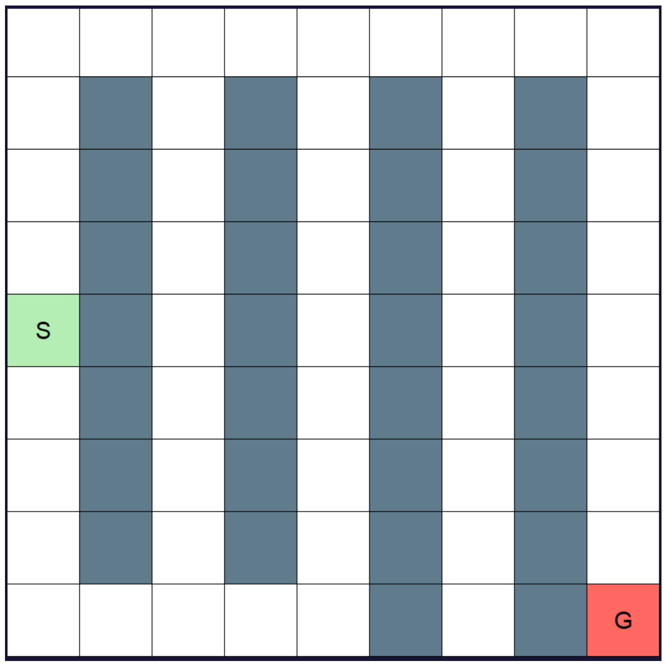
1.2 Test 2: Uninformed Search Favoring

1.2.1 Description

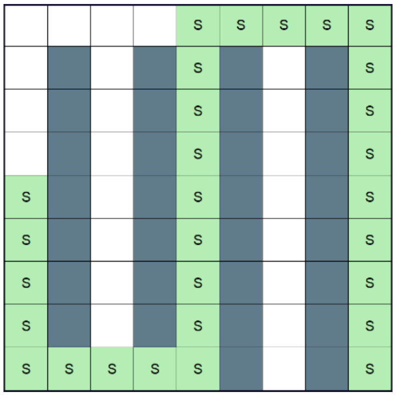
For heuristic-based algorithms, the path found may not be optimal.

Uninformed search algorithms will explore as much as possible, ensuring that the optimal path is found

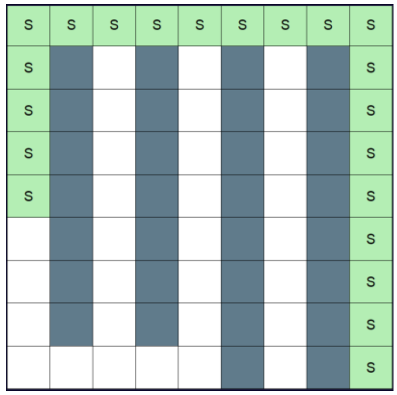
1.2.2 Visualization:



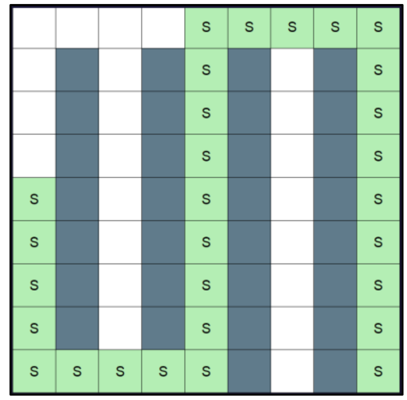
1.2.3 Result



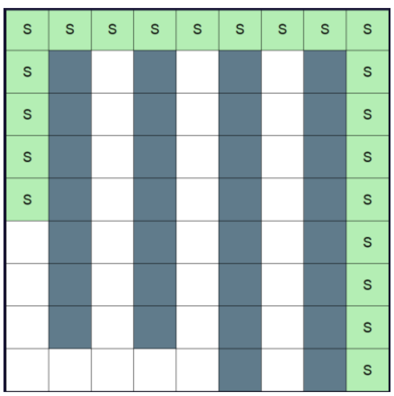
DFS



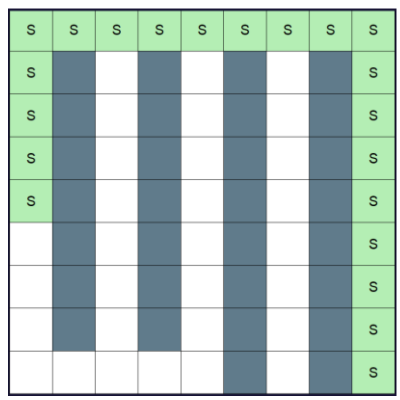
BFS



GBFS



UCS



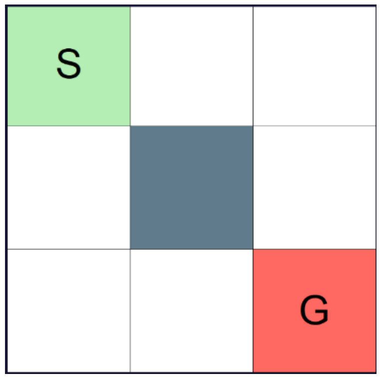
*A**

1.3 Test 3: 3x3 Map

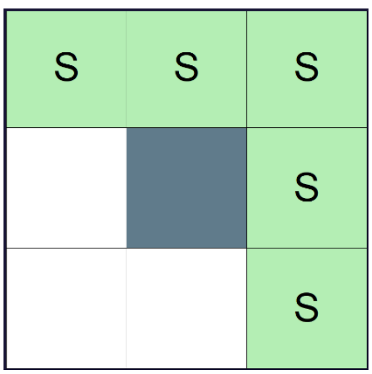
1.3.1 Description

A simple 3x3 maps with random attributes.

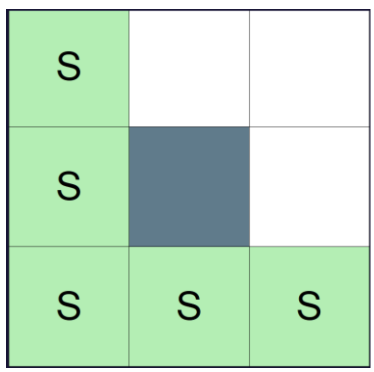
1.3.2 Visualization:



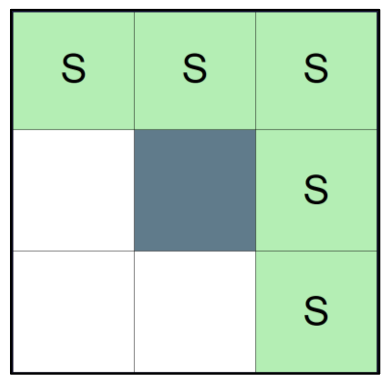
1.3.3 Result



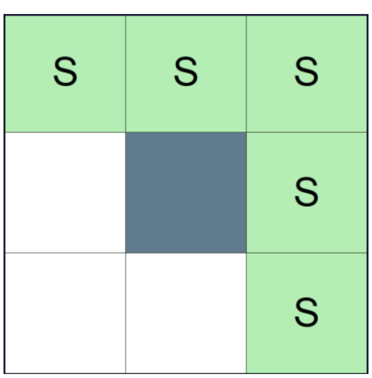
DFS



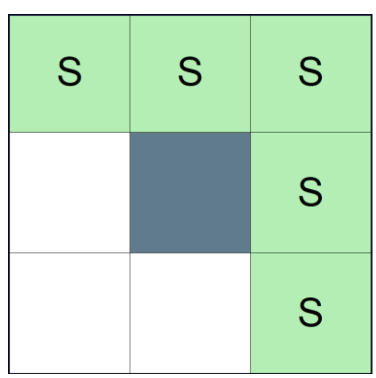
BFS



GBFS



UCS



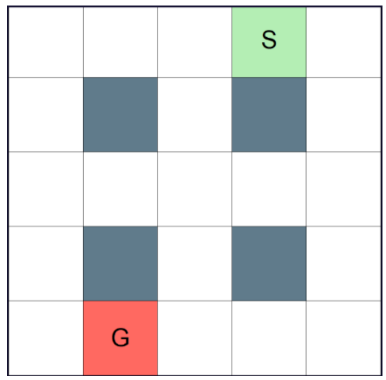
*A**

1.4 Test 4: 5x5 Map

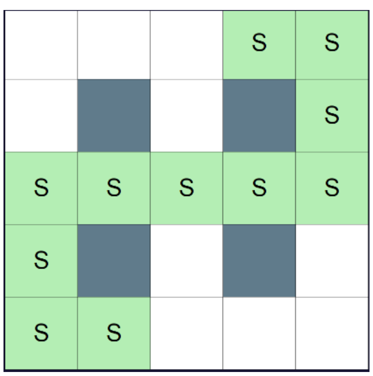
1.4.1 Description

A simple 5x5 maps with random attributes, suitable for testing searching algorithms in small-sized maps.

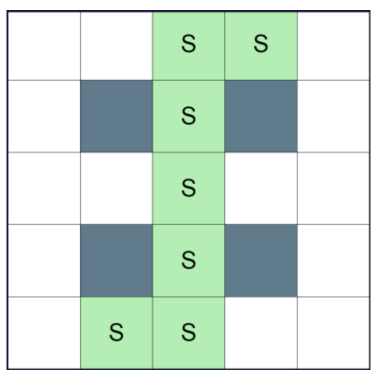
1.4.2 Visualization:



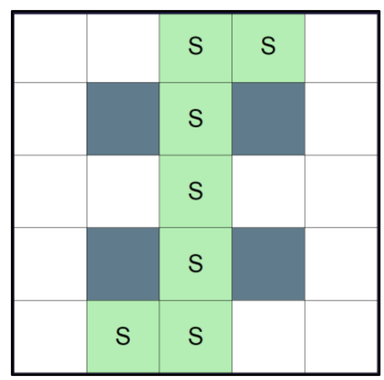
1.4.3 Result



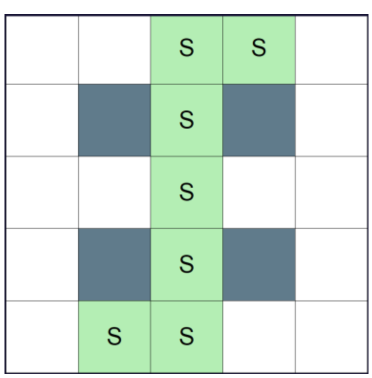
DFS



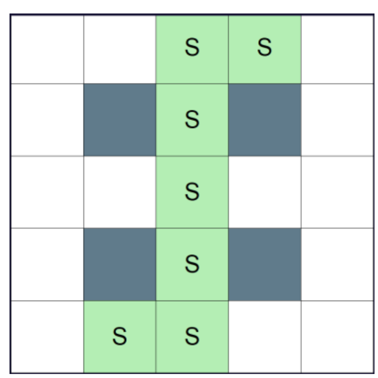
BFS



GBFS



UCS



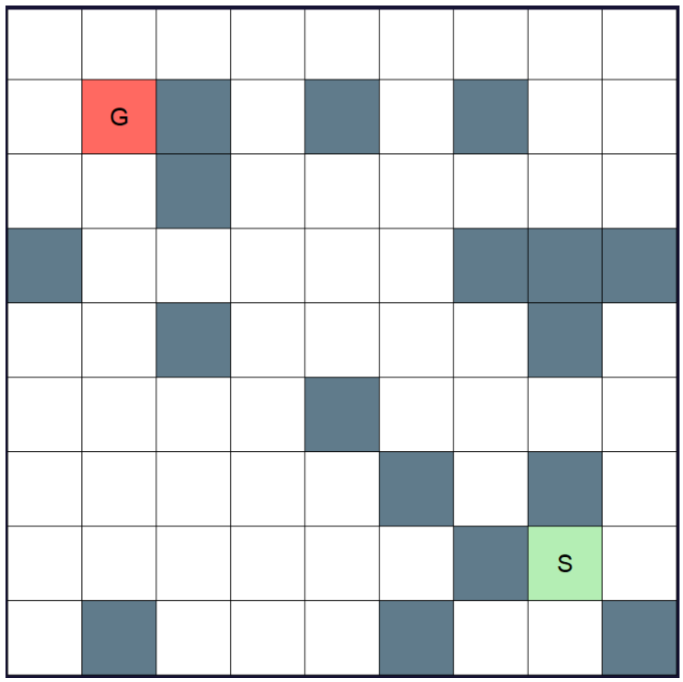
*A**

1.5 Test 5: 5x5 Map

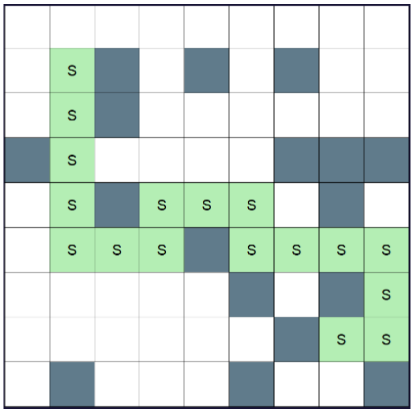
1.5.1 Description

A 9x9 map with random attributes, suitable for testing search algorithms in middle-sized maps.

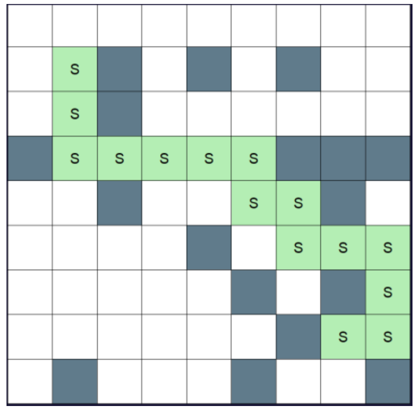
1.5.2 Visualization:



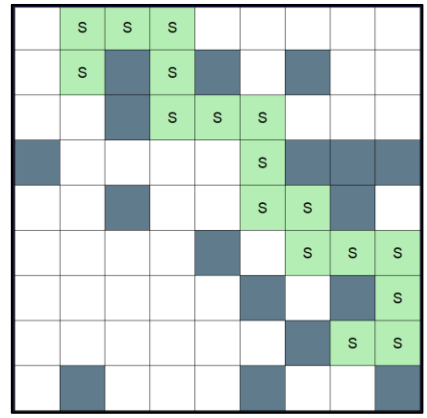
1.5.3 Result



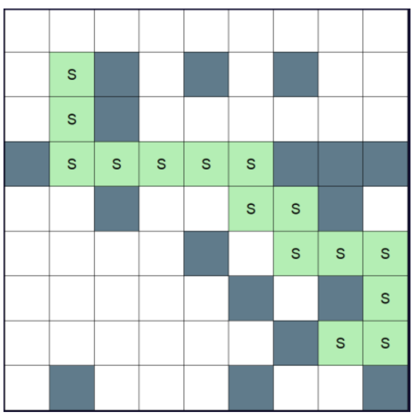
DFS



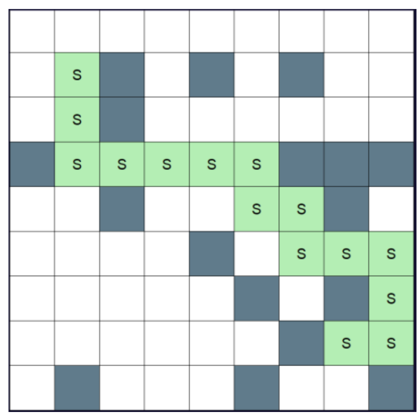
BFS



GBFS



UCS



*A**

Level 2

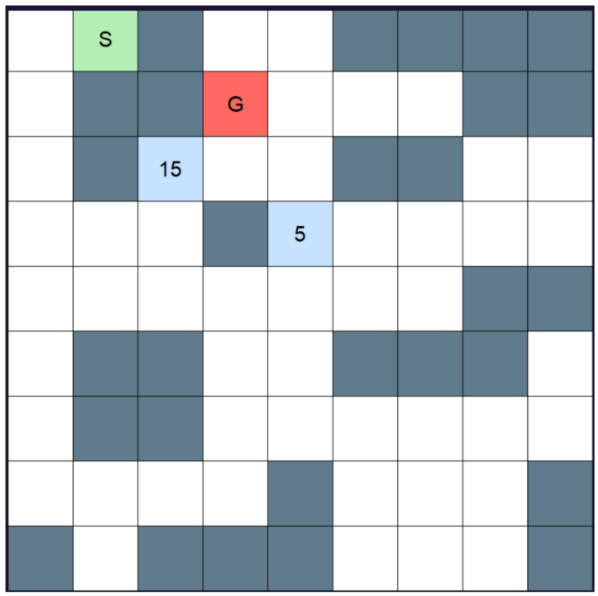
2.1 Test 1: 9x9 Map

2.1.1 Description

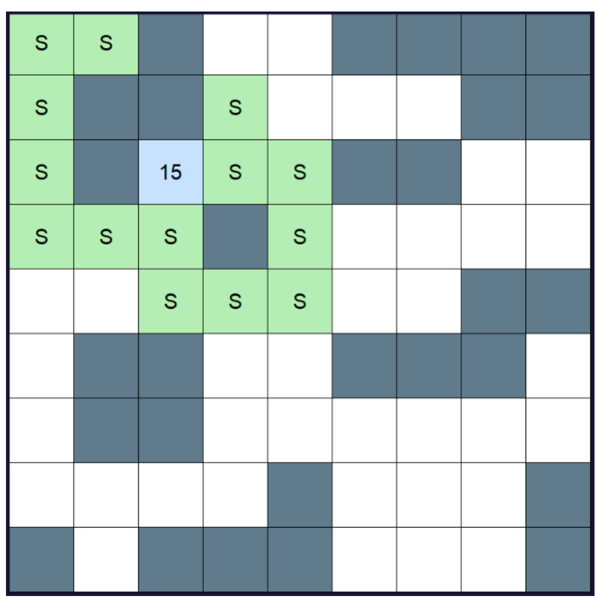
A 9x9 map with toll booths. In this test case, the vehicle has to go a longer path in order to satisfy the time limit condition.

- Time limit: 20

2.1.2 Visualization:



2.1.3 Result



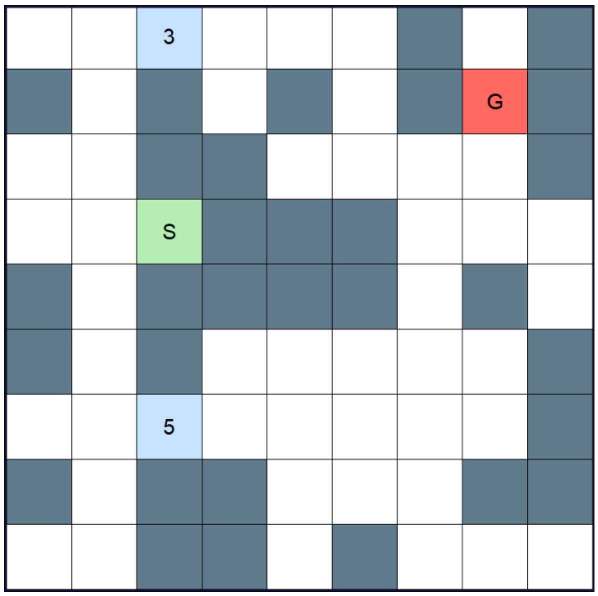
2.2 Test 2: 9x9 Map

2.2.1 Description

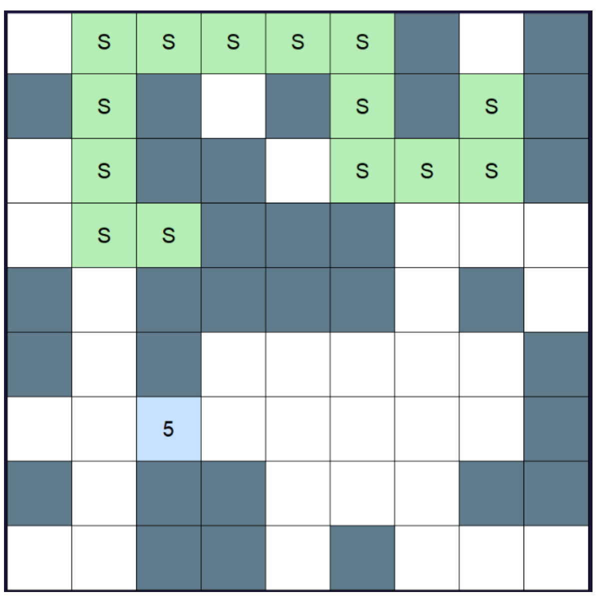
A 9x9 map with toll booths.

- Time limit: 100

2.2.2 Visualization:



2.2.3 Result



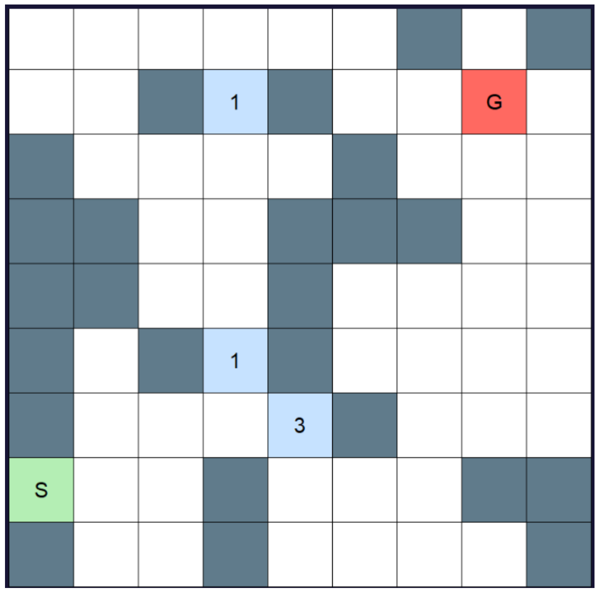
2.3 Test 3: 9x9 Map

2.3.1 Description

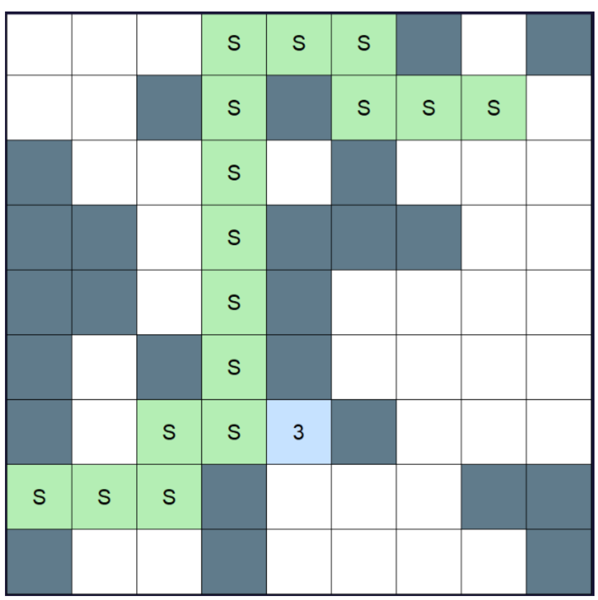
A 9x9 map with toll booths. In this example, there are 2 valid paths with acceptable travel time, but the vehicle will prioritize the path shorter in terms of length.

- Time limit: 100

2.3.2 Visualization:



2.3.3 Result



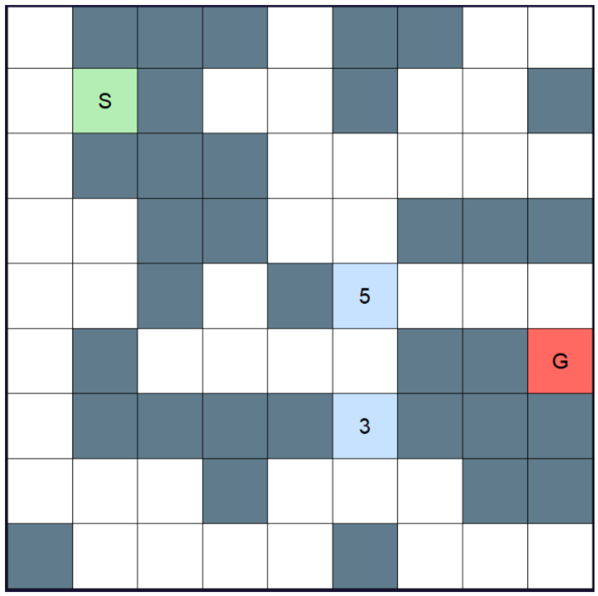
2.4 Test 4: 9x9 Map

2.4.1 Description

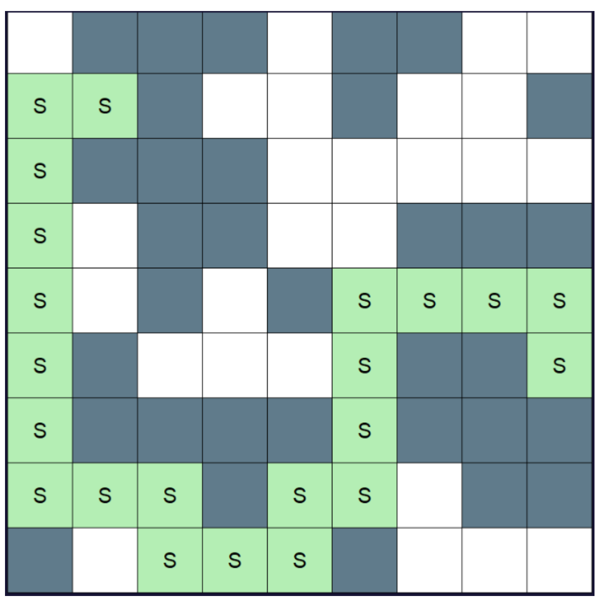
A 9x9 map with toll booths.

- Time limit: 100

2.4.2 Visualization:



2.4.3 Result



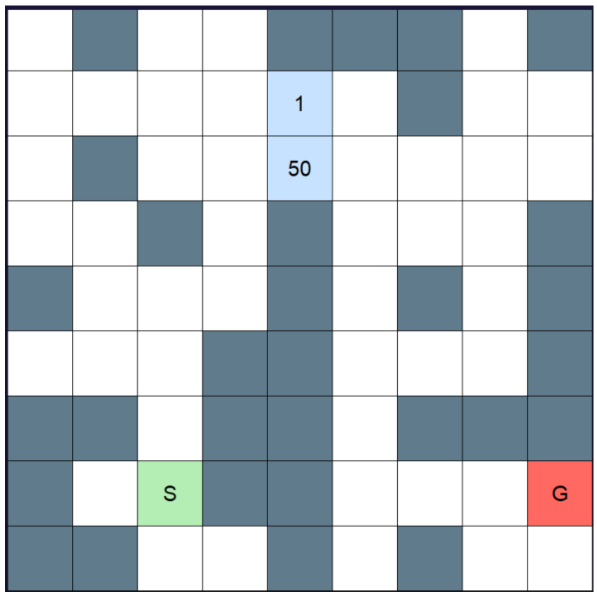
2.5 Test 5: 9x9 Map

2.5.1 Description

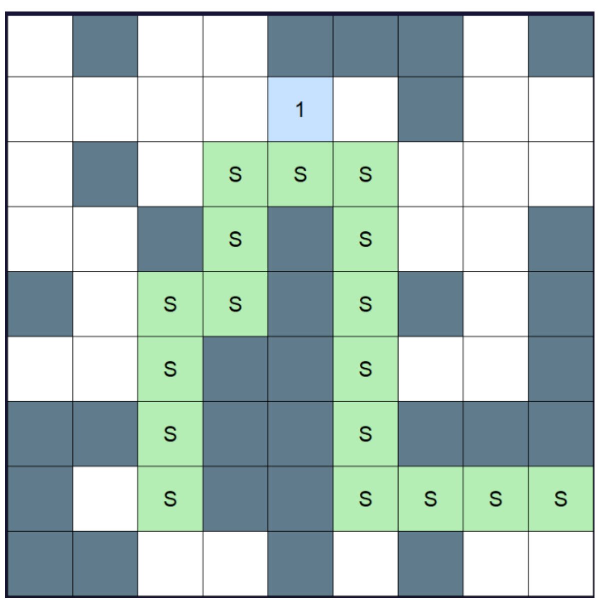
A 9x9 map with toll booths. In this example, there is another path with significantly shorter travel time but it is not optimal in path length.

- Time limit: 100

2.5.2 Visualization:



2.5.3 Result



Level 3

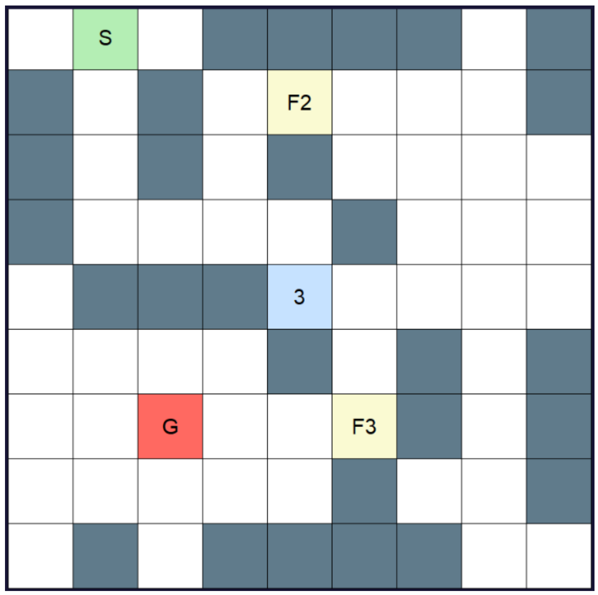
3.1 <u>Test 1</u>: 9x9 Map

3.1.1 Description

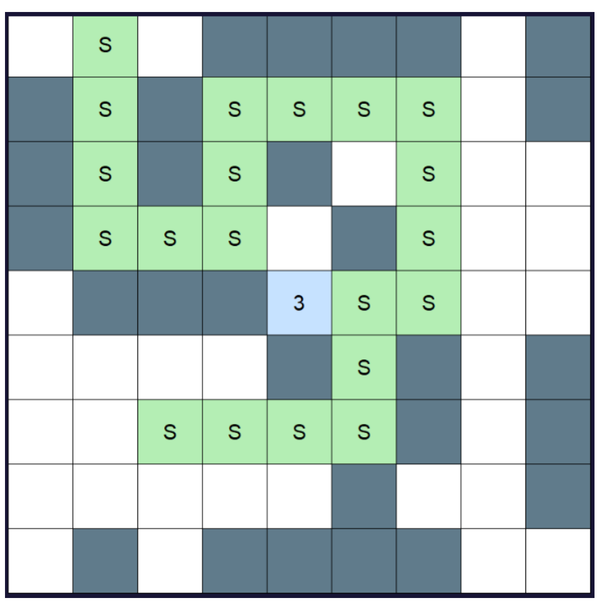
A 9x9 map with toll booths and fuel stations.

- Time limit: 100
- Fuel: 8

3.1.2 Visualization:



3.1.3 Result



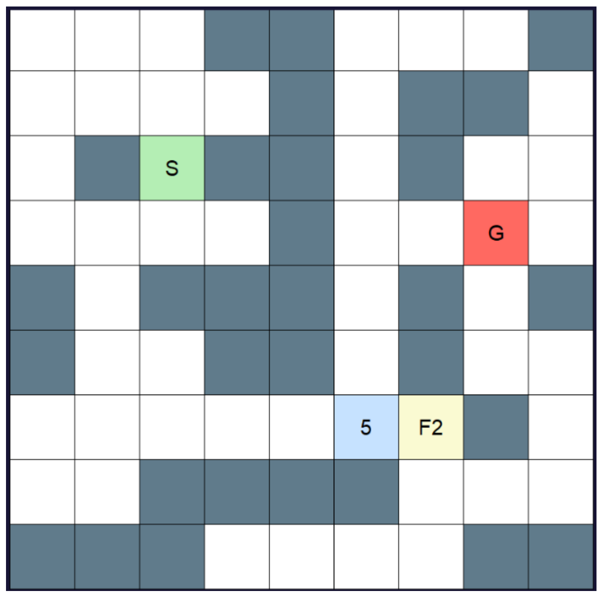
3.2 Test 2: 9x9 Map

3.2.1 Description

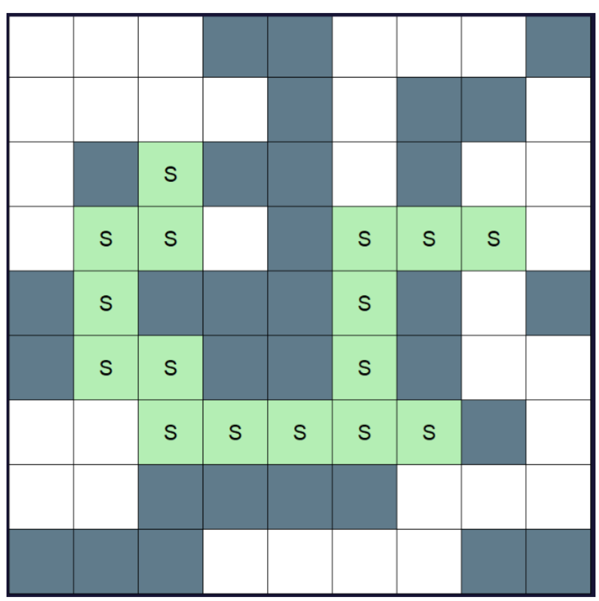
A 9x9 map with toll booths and fuel stations. In this testcase, the vehicle will have to go to a cell not located in the path to refuel, only then can it get to the goal.

- Time limit: 100
- Fuel: 12

3.2.2 Visualization:



3.2.3 Result



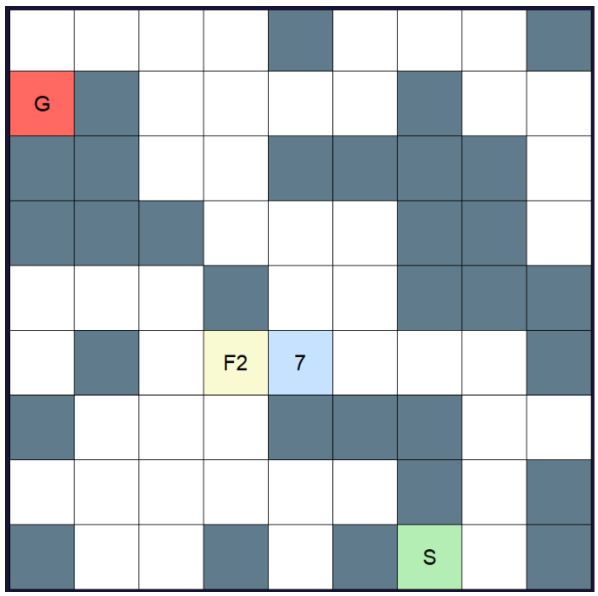
3.3 Test 3: 9x9 Map

3.3.1 Description

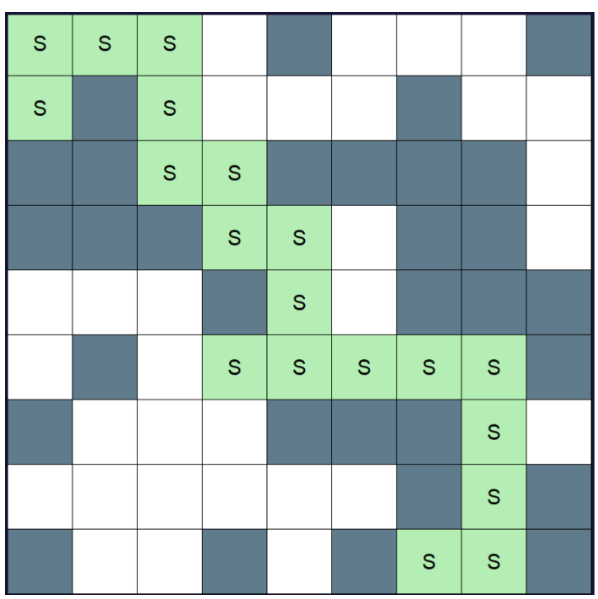
A 9x9 map with toll booths and fuel stations. In this testcase, the vehicle will have to go to a cell not located in the path to refuel, only then can it get to the goal.

- Time limit: 100
- Fuel: 12

3.3.2 Visualization:



3.3.3 Result



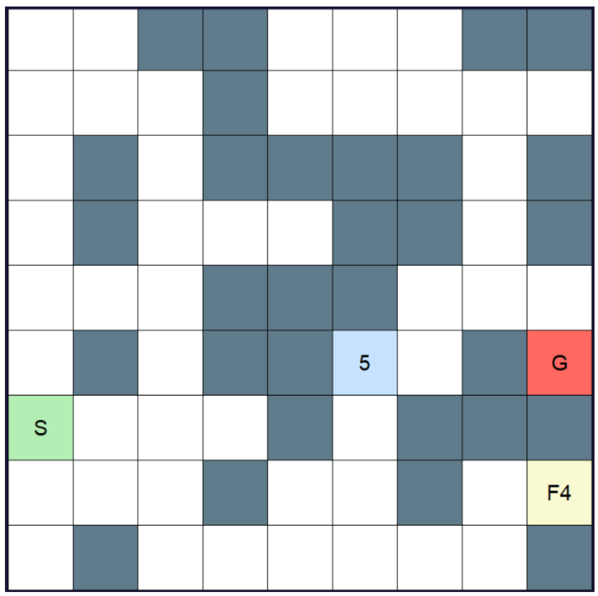
3.4 Test 4: 9x9 Map

3.4.1 Description

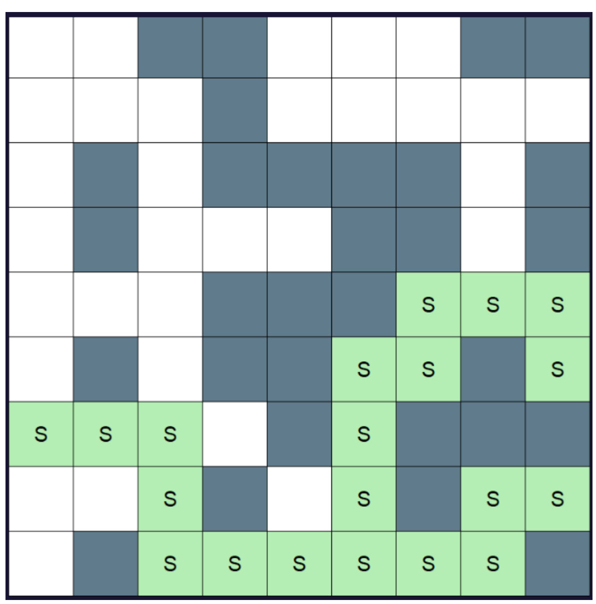
A 9x9 map with toll booths and fuel stations. In this testcase, the vehicle will have to go to a cell not located in the path to refuel, only then can it get to the goal.

- Time limit: 100
- Fuel: 14

3.4.2 Visualization:



3.4.3 Result



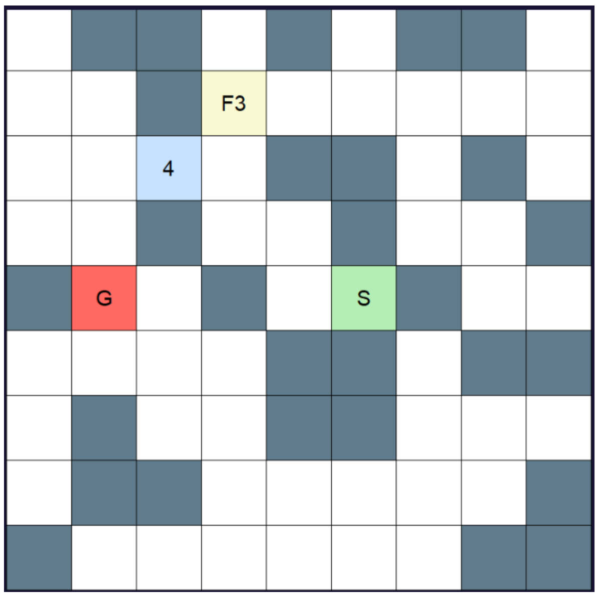
3.5 Test 5: 9x9 Map

3.5.1 Description

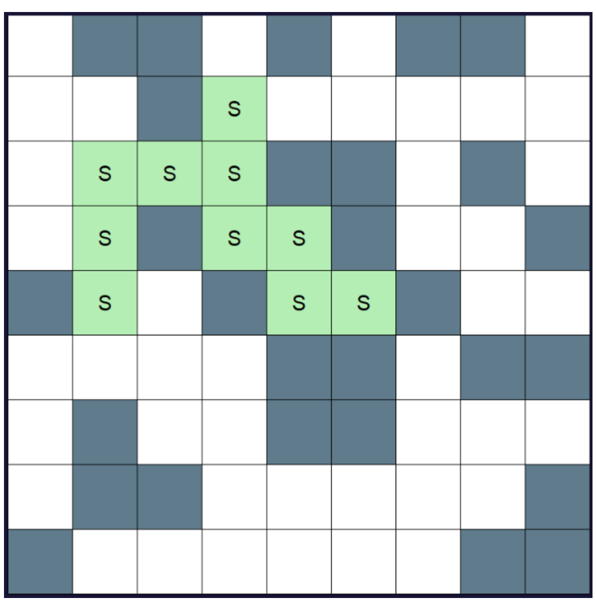
A 9x9 map with toll booths and fuel stations. In this testcase, the vehicle will have to go to a cell not located in the path to refuel, only then can it get to the goal.

- Time limit: 100
- Fuel: 7

3.5.2 Visualization:



3.5.3 Result



Level 4

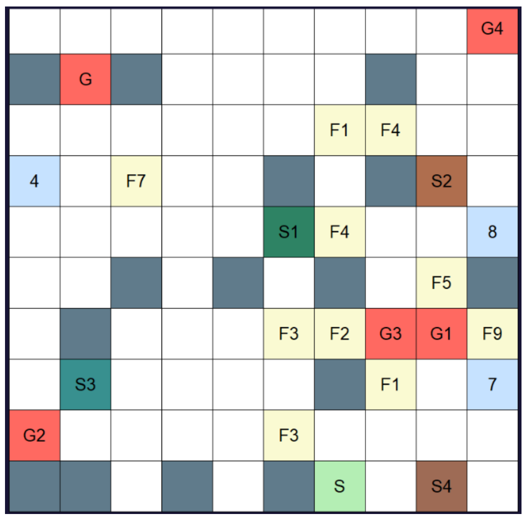
4.1 <u>Test 1</u>: 9x9 Map

4.1.1 Description

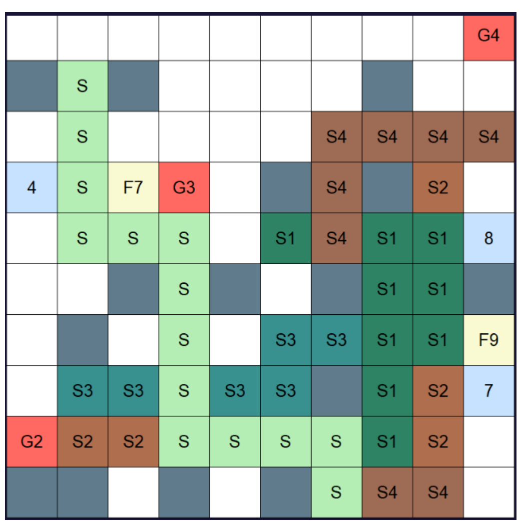
A 9x9 map with toll booths, fuel stations and multiagents with random paths are involved in the searching process so the vehicle has to precisely calculate the path to avoid colliding with other agents in the process and get to the goal in the given time and fuel constraints.

- Time limit: 100
- Fuel: 20
- Agents: 4

4.1.2 Visualization:



4.1.3 Result



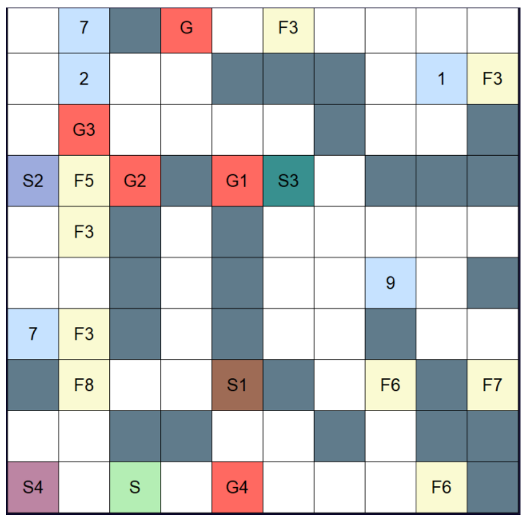
4.2 <u>Test 2</u>: 9x9 Map

4.2.1 Description

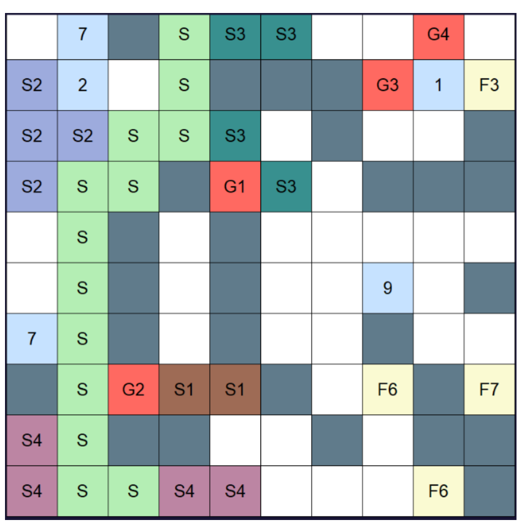
A 9x9 map with toll booths, fuel stations and multiagents with random paths are involved in the searching process so the vehicle has to precisely calculate the path to avoid colliding with other agents in the process and get to the goal in the given time and fuel constraints.

- Time limit: 100
- Fuel: 20
- Agents: 4

4.2.2 Visualization:



4.2.3 Result



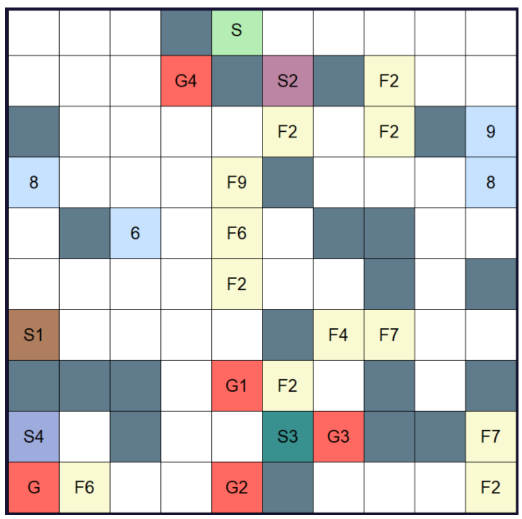
4.3 <u>Test 3</u>: 9x9 Map

4.3.1 Description

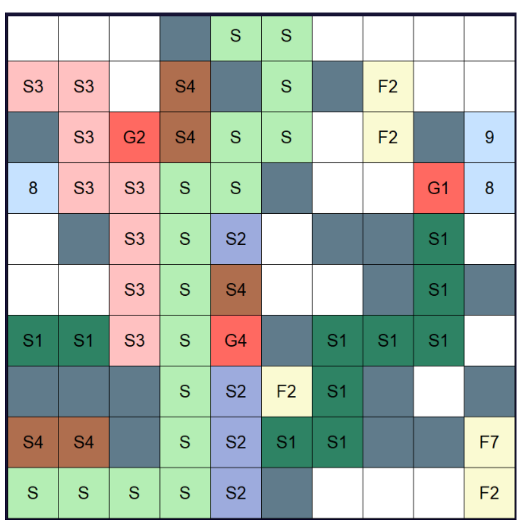
A 9x9 map with toll booths, fuel stations and multiagents with random paths are involved in the searching process so the vehicle has to precisely calculate the path to avoid colliding with other agents in the process and get to the goal in the given time and fuel constraints.

- Time limit: 100
- Fuel: 20
- Agents: 4

4.3.2 Visualization:



4.3.3 Result



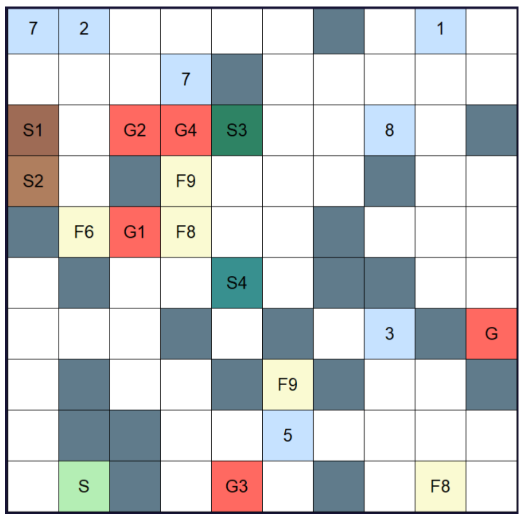
4.4 Test 4: 9x9 Map

4.4.1 Description

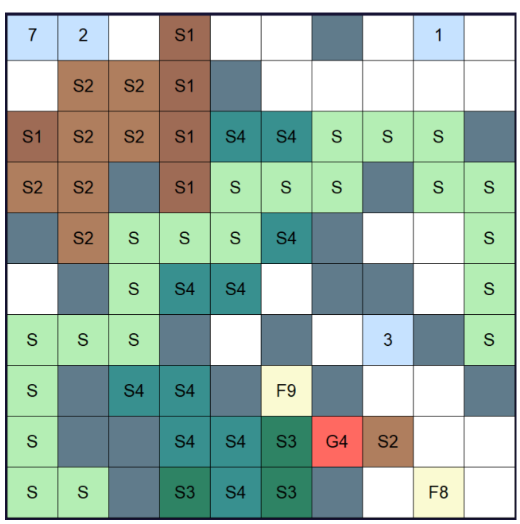
A 9x9 map with toll booths, fuel stations and multiagents with random paths are involved in the searching process so the vehicle has to precisely calculate the path to avoid colliding with other agents in the process and get to the goal in the given time and fuel constraints.

- Time limit: 100
- Fuel: 20
- Agents: 4

4.4.2 Visualization:



4.4.3 Result



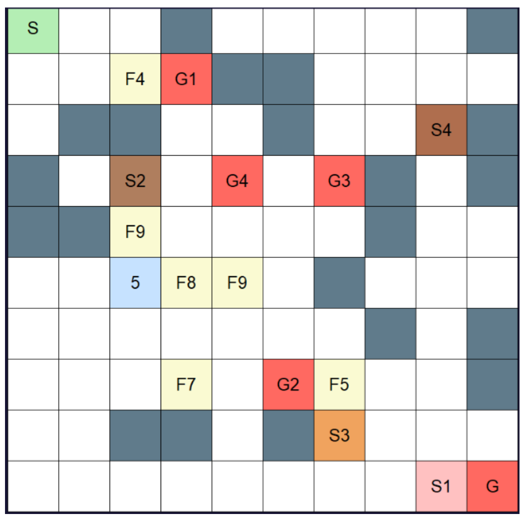
4.5 <u>Test 3</u>: 9x9 Map

4.5.1 Description

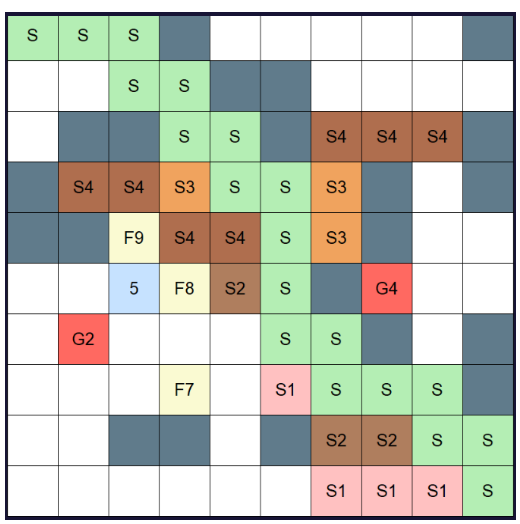
A 9x9 map with toll booths, fuel stations and multiagents with random paths are involved in the searching process so the vehicle has to precisely calculate the path to avoid colliding with other agents in the process and get to the goal in the given time and fuel constraints.

- Time limit: 100
- Fuel: 20
- Agents: 4

4.5.2 Visualization:



4.5.3 Result



8. Conclusion

Report Summary

This project report details the implementation and evaluation of various search algorithms used for pathfinding in a 2D grid-based city map. We focused on solving the delivery problem with additional complexities like toll booths, fuel limitations, and multiple delivery agents using algorithms such as Breadth-First Search (BFS), Depth-First Search (DFS), Uniform-Cost Search (UCS), Greedy Best First Search, and A*. The project involved comprehensive testing and detailed graphical user interface (GUI) for visualization.

The Importance of Pathfinding in Real-World Applications

Pathfinding algorithms play a crucial role in various real-world applications. In logistics and transportation, efficient route planning can lead to significant cost savings and timely deliveries. In robotics, pathfinding is essential for navigation and task execution. Additionally, these algorithms are used in video games for character movement, in network routing to find optimal data paths, and in geographic information systems (GIS) for mapping and navigation. The ability to find the shortest or most cost-effective path in different environments directly impacts the efficiency and effectiveness of these applications.

The Significance of Each Algorithm in Real-World Applications

- **Breadth-First Search (BFS):** BFS is particularly useful in scenarios where the shortest path in terms of the number of steps is required. It is effective in unweighted grids and ensures that the first path found to the goal is the shortest.
- **Depth-First Search (DFS):** While not ideal for shortest pathfinding, DFS is useful in applications where all possible paths need to be explored, such as puzzle solving or exploring large search spaces.
- **Uniform-Cost Search (UCS):** UCS is ideal for finding the least cost path in weighted graphs, making it useful in applications where different paths have different costs, such as transportation networks with varying tolls.
- **Greedy Best First Search:** This algorithm is efficient in finding a path quickly by prioritizing nodes that are closest to the goal, making it useful in applications requiring fast, not always optimal solutions.
- **A* Search**: A* combines the strengths of BFS and UCS, providing the shortest path efficiently by considering both the cost to reach the node and the estimated cost to the goal. It is widely used in applications requiring optimal pathfinding, such as GPS navigation and game development.

Reflections and Potential Improvements

Reflecting on the project, we encountered several challenges, particularly in implementing algorithms that account for tolls, fuel limitations, and multiple agents. These challenges required problem-solving and understanding of algorithmic principles.

Potential improvements:

- **Enhanced GUI:** Further development of the GUI to include real-time updates and more interactive features.
- **Optimization:** Implementing optimization techniques to improve the efficiency of the algorithms, particularly for large and complex maps.
- **Advanced Algorithms:** Utilizing advanced pathfinding algorithms, such as Dijkstra's algorithm or genetic algorithms, to handle more complex scenarios.

9. References

GeeksforGeeks - https://www.geeksforgeeks.org/

research.ncl.ac.uk; ; Newcastle University - https://research.ncl.ac.uk/

https://www.geeksforgeeks.org/heuristic-function-in-ai/

Baeldung on CS - https://www.baeldung.com/

Viblo - https://viblo.asia/

Stanford CS Theory - https://theory.stanford.edu/

Stack Overflow - https://stackoverflow.com/

Ariel Procaccia - https://procaccia.info/

Wikipedia - https://en.wikipedia.org/

Google Scholar - https://scholar.google.com

Graphable - https://www.graphable.ai/

ResearchGate - https://www.researchgate.net/

SpringerLink - https://link.springer.com/