

# UNIVERSITY OF SCIENCE, VNU-HCMC

## FACULTY INFORMATION TECHNOLOGY

### CSC14003 - Introduction to Artificial Intelligence



## Project 1

### Searching

**Instructors:** Nguyễn Trần Duy Minh, Nguyễn Ngọc Thảo,  
Nguyễn Thanh Tình, Nguyễn Hải Đăng

**Class:** 22CLC06

**Group:** 06

**Members:**

22127022 – Võ Hoàng Anh

22127154 – Nguyễn Gia Huy

22127192 – Trần Gia Khiêm

22127210 – Phạm Anh Khôi

Ho Chi Minh City – 2024

## Table of contents

Table of contents .....	2
1. Member information .....	4
2. Work assignment.....	4
3. Self-evaluation .....	8
4. Algorithms .....	9
4.1 Level 1: Basic Level .....	9
4.1.1 Breadth-First Search (BFS) .....	9
4.1.2 Depth-First Search (DFS) .....	10
4.1.3 Uniform-Cost Search (UCS).....	11
4.1.4 Greedy Best First Search (GBFS).....	12
4.1.5 A* Search.....	13
4.1.6 Reconstructing the Path .....	14
4.2 Level 2: Time limitation .....	15
4.3 Level 3: Fuel limitation.....	17
4.4 Level 4: Multiple agents .....	19
5. Testing.....	21
5.1 Level 1: Basic Search Algorithms .....	21
5.1.1 Test Case 1.1: Simple Path .....	21
5.1.2 Test Case 1.2: Single Obstacle.....	22
5.1.3 Test Case 1.3: Multiple Obstacles .....	23
5.1.4 Test Case 1.4: No Possible Path .....	24
5.1.5 Test Case 1.5: Complex Maze .....	25
5.2 Level 2: Time Limitation .....	27
5.2.1 Test Case 2.1: Simple Path with Time Constraint .....	27
5.2.2 Test Case 2.2: Path with Toll Booths.....	28
5.2.3 Test Case 2.3: Tight Time Constraint .....	29
5.2.4 Test Case 2.4: Excess Time Constraint.....	30
5.2.5 Test Case 2.5: No Valid Path Within Time .....	31
5.3 Level 3: Fuel Limitation .....	33
5.3.1 Test Case 3.1: Path with Refueling Station .....	33
5.3.2 Test Case 3.2: Insufficient Fuel Without Refueling .....	34

---

5.3.3 Test Case 3.3: Multiple Refueling Stations .....	35
5.3.4 Test Case 3.4: Tight Fuel Constraint .....	36
5.3.5 Test Case 3.5: No Valid Path Within Fuel Limit .....	37
5.4 Level 4: Multiple Agents .....	39
5.4.1 Test Case 4.1: Two Agents with Non-Intersecting Paths .....	39
5.4.2 Test Case 4.2: Two Agents with Intersecting Paths.....	40
5.4.3 Test Case 4.3: Three Agents with narrow path.....	41
5.4.4 Test Case 4.4: Five Agents with Complex Interactions.....	42
5.4.5 Test Case 4.5: Maximum Number of Agents .....	43
6. References .....	45

## 1. Member information

No.	Student ID	Full name	Gmail
1	22127022	Võ Hoàng Anh	vhanh22@clc.fitus.edu.vn
2	22127154	Nguyễn Gia Huy	ngghuy22@clc.fitus.edu.vn
3	22127192	Trần Gia Khiêm	tgkhiem22@clc.fitus.edu.vn
4	22127210	Phạm Anh Khôi	pakhoi22@clc.fitus.edu.vn

## 2. Work assignment

The following table details the work assignments for each group member, specifying the tasks assigned to them and their respective completion rates. The work is divided into different aspects of the project, encompassing all four levels of complexity. The table ensures that all members contribute equally and cover every necessary component of the project.

Task	Description	Assigned Member	Completion Rate (%)
Project Management	Coordination of tasks, meetings, and deadlines	Nguyễn Gia Huy	100%
Level 1 Implementation	Development and testing of basic search algorithms (BFS, DFS, UCS)	Nguyễn Gia Huy	100%
Level 2 Implementation	Modification of algorithms to include time constraints and toll booths	Phạm Anh Khôi	100%

Level 3 Implementation	Extension of algorithms to handle fuel limitations and refueling stations	Trần Gia Khiêm	100%
Level 4 Implementation	Coordination and implementation of multiple agents	Võ Hoàng Anh	100%
Algorithm Documentation	Detailed write-up of the algorithms used in each level, including pseudocode and illustrative images	Nguyễn Gia Huy	100%
Input/Output Handling	Development of functions for reading input files and writing output files	Võ Hoàng Anh	100%
GUI Development	Creating a graphical user interface to visualize the search process	Trần Gia Khiêm	100%
Testing	Creation of test cases and verification of algorithm correctness	Phạm Anh Khôi	100%
Report Writing	Compilation and formatting of the final report	Nguyễn Gia Huy	100%
Video Documentation	Recording and editing demonstration videos of the program in action	Phạm Anh Khôi Trần Gia Khiêm	100%

### Detailed Breakdown of Tasks:

#### 1. Project Management:

- Coordination of tasks, organizing meetings, ensuring deadlines are met.
- Monitoring progress and ensuring all members are contributing equally.
- Overseeing the entire project lifecycle.

#### 2. Level 1 Implementation:

- Implementing Breadth-First Search (BFS), Depth-First Search (DFS), and Uniform-Cost Search (UCS) algorithms.
- Ensuring the algorithms correctly navigate the 2D city map.

- Testing the basic functionality and edge cases.

### 3. **Level 2 Implementation:**

- Modifying search algorithms to account for time constraints and toll booths.
- Ensuring the pathfinding respects the committed delivery time.
- Testing with various scenarios to validate correctness.

### 4. **Level 3 Implementation:**

- Extending algorithms to handle fuel limitations.
- Incorporating refueling stations and ensuring the path is feasible with fuel constraints.
- Rigorous testing to ensure all conditions are met.

### 5. **Level 4 Implementation:**

- Developing a system to handle multiple agents in the city map.
- Implementing turn-based movement and collision avoidance.
- Testing with multiple agents to ensure smooth operation.

### 6. **Algorithm Documentation:**

- Writing detailed descriptions of each algorithm used.
- Including pseudocode and illustrative images to explain the logic.
- Ensuring clarity and comprehensiveness for assessment.

### 7. **Input/Output Handling:**

- Writing functions to read from input files and process the data.
- Implementing output functions to save results in the specified format.
- Ensuring robustness and handling various input cases.

### 8. **GUI Development:**

- Creating a graphical interface to visualize the search process.
- Ensuring real-time display of the vehicle's path and search progress.
- Making the interface user-friendly and informative.

**9. Testing:**

- Developing comprehensive test cases covering all levels and edge cases.
- Running tests to verify the correctness and efficiency of algorithms.
- Documenting test results and any issues found.

**10. Report Writing:**

- Compiling all sections of the report, ensuring proper formatting.
- Including member information, work assignment, self-evaluation, algorithms, and testing.
- Reviewing and editing for clarity and completeness.

**11. Video Documentation:**

- Recording the program's functionality for each test case.
- Editing videos to highlight key features and successful runs.
- Uploading and including URLs in the report.

This work assignment ensures that each member has a clear role and responsibility, contributing to the successful completion of the project.

### 3. Self-evaluation

No.	Requirements	Completion Rate (%)
1	Finish Level 1 successfully.	100%
2	Finish Level 2 successfully.	100%
3	Finish Level 3 successfully.	100%
4	Finish Level 4 successfully.	100%
5	Graphical User Interface (GUI)	100%
6	Generate at least 5 test cases for each level with different attributes. Describe them in the experiment section of your report. Videos to demonstrate each test case.	100%
7	Report your algorithm, and experiment with some reflection or comments.	100%



## 4. Algorithms

### 4.1 Level 1: Basic Level

In Level 1, the objective is to find the path from the starting location of the delivery vehicle (S) to the goal location (G) using various search algorithms. The city map is represented as a 2D array, where each cell has specific values indicating passable spaces, impassable spaces, the starting point, and the goal point. The search algorithms implemented at this level include Breadth-First Search (BFS), Depth-First Search (DFS), Uniform-Cost Search (UCS), Greedy Best First Search, and A\* Search.

#### 4.1.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) explores all the nodes at the present depth level before moving on to the nodes at the next depth level. It is guaranteed to find the shortest path in an unweighted graph.

#### Pseudocode:

```
function BFS(start, goal, grid):
    queue <- empty queue
    visited <- empty set
    queue.enqueue(start)
    visited.add(start)

    while not queue.isEmpty():
        current <- queue.dequeue()
        if current == goal:
            return reconstruct_path(goal)

        for each neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                queue.enqueue(neighbor)
                visited.add(neighbor)
                neighbor.parent <- current

    return "Path does not exist"
```

**Illustration:**

- Initialize the queue with the starting node S.
- Dequeue S and explore its neighbors.
- Enqueue the neighbors that are not visited and mark them as visited.
- Repeat the process until the goal G is reached or the queue is empty.

#### 4.1.2 Depth-First Search (DFS)

Depth-First Search (DFS) explores as far as possible along each branch before backtracking. It is not guaranteed to find the shortest path but is useful for exhaustive searching.

**Pseudocode:**

```
function DFS(start, goal, grid):
    stack <- empty stack
    visited <- empty set
    stack.push(start)
    visited.add(start)

    while not stack.isEmpty():
        current <- stack.pop()
        if current == goal:
            return reconstruct_path(goal)

        for each neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                stack.push(neighbor)
                visited.add(neighbor)
                neighbor.parent <- current


    return "Path does not exist"
```

**Illustration:**

- Initialize the stack with the starting node S.
- Pop S and explore its neighbors.
- Push the neighbors that are not visited and mark them as visited.
- Repeat the process until the goal G is reached or the stack is empty

### 4.1.3 Uniform-Cost Search (UCS)

Uniform-Cost Search (UCS) expands the least cost node first. It is equivalent to Dijkstra's algorithm when all edge costs are equal.

**Pseudocode:**

```
function UCS(start, goal, grid):
    priority_queue <- empty priority queue
    visited <- empty set
    priority_queue.enqueue(start, cost=0)
    visited.add(start)

    while not priority_queue.isEmpty():
        current, cost <- priority_queue.dequeue()
        if current == goal:
            return reconstruct_path(goal)

        for each neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                total_cost <- cost + move_cost(current, neighbor)
                priority_queue.enqueue(neighbor, total_cost)
                visited.add(neighbor)
                neighbor.parent <- current

    return "Path does not exist"
```

**Illustration:**

- Initialize the priority queue with the starting node S with a cost of 0.
- Dequeue the node with the lowest cost and explore its neighbors.
- Enqueue the neighbors with their cumulative cost and mark them as visited.
- Repeat the process until the goal G is reached or the priority queue is empty.

#### 4.1.4 Greedy Best First Search (GBFS)

Greedy Best First Search expands the node that is estimated to be closest to the goal using a heuristic.

**Pseudocode:**

```
function GreedyBFS(start, goal, grid):
    priority_queue <- empty priority queue
    visited <- empty set
    priority_queue.enqueue(start, heuristic(start, goal))
    visited.add(start)

    while not priority_queue.isEmpty():
        current <- priority_queue.dequeue()
        if current == goal:
            return reconstruct_path(goal)

        for each neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                priority_queue.enqueue(neighbor, heuristic(neighbor, goal))
                visited.add(neighbor)
                neighbor.parent <- current

    return "Path does not exist"
```

**Heuristic function:**

```
FUNCTION heuristic(a, b):
    INITIALIZE h AS Manhattan distance between a and b
    RETURN h
```

**Illustration:**

- Initialize the priority queue with the starting node S with its heuristic value.
- Dequeue the node with the lowest heuristic value and explore its neighbors.
- Enqueue the neighbors with their heuristic value and mark them as visited.
- Repeat the process until the goal G is reached or the priority queue is empty.

### 4.1.5 A\* Search

A\* Search uses both the actual cost from the start and a heuristic estimate to the goal to find the optimal path efficiently.

**Pseudocode:**

```
function AStar(start, goal, grid):
    open_set <- empty priority queue
    came_from <- empty map
    g_score <- map with default value of infinity
    f_score <- map with default value of infinity

    open_set.enqueue(start, heuristic(start, goal))
    g_score[start] <- 0
    f_score[start] <- heuristic(start, goal)

    while not open_set.isEmpty():
        current <- open_set.dequeue()
        if current == goal:
            return reconstruct_path(came_from, current)

        for each neighbor in get_neighbors(current, grid):
            tentative_g_score <- g_score[current] + move_cost(current, neighbor)
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] <- current
                g_score[neighbor] <- tentative_g_score
                f_score[neighbor] <- g_score[neighbor] + heuristic(neighbor, goal)
                if neighbor not in open_set:
                    open_set.enqueue(neighbor, f_score[neighbor])

    return "Path does not exist"
```

**Illustration:**

- Initialize the priority queue with the starting node S with its f-score.
- Dequeue the node with the lowest f-score and explore its neighbors.
- Update the g-score and f-score of the neighbors if a better path is found and mark them as visited.
- Repeat the process until the goal G is reached or the priority queue is empty.

Each algorithm has its strengths and is suitable for different scenarios. The BFS guarantees the shortest path in an unweighted grid, DFS is useful for exhaustive search, UCS ensures the least cost path, Greedy Best First Search is fast but may not find the optimal path, and A\* combines the benefits of UCS and heuristic search for optimal and efficient pathfinding.

#### 4.1.6 Reconstructing the Path

For all algorithms, once the goal node is reached, the path can be reconstructed by tracing back from the goal node to the start node using the parent references stored during the search process.

**Pseudocode:**



```
function reconstruct_path(came_from, current):  
    total_path <- [current]  
    while current in came_from:  
        current <- came_from[current]  
        total_path.append(current)  
    return total_path[::-1]  # Reverse the path
```

This section provides a comprehensive overview of the search algorithms required for Level 1, including detailed pseudocode and illustrative images to aid understanding.

## 4.2 Level 2: Time limitation

In Level 2, the objective is to find the shortest path from the starting location of the delivery vehicle (S) to the goal location (G) within a committed delivery time  $t_{tt}$ . The map includes toll booths where vehicles must stop for a specified time, increasing the total delivery time. The task is to ensure that the delivery is completed within the committed time.

*Optimal Algorithm: A Search Algorithm\**

The A\* Search Algorithm is chosen for this level due to its efficiency in finding the optimal path while considering both the cost to reach a node and the heuristic estimate to the goal.

### Pseudocode:

```
function AStar(start, goal, grid, tolls, t):
    open_set <- empty priority queue
    came_from <- empty map
    g_score <- map with default value of infinity
    f_score <- map with default value of infinity

    open_set.enqueue(start, heuristic(start, goal))
    g_score[start] <- 0
    f_score[start] <- heuristic(start, goal)

    while not open_set.isEmpty():
        current <- open_set.dequeue()
        if current == goal:
            if g_score[current] <= t:
                return reconstruct_path(came_from, current)
            else:
                return "Path found exceeds committed time"

        for each neighbor in get_neighbors(current, grid):
            move_time <- if neighbor in tolls then tolls[neighbor] else 1
            tentative_g_score <- g_score[current] + move_time
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] <- current
                g_score[neighbor] <- tentative_g_score
                f_score[neighbor] <- g_score[neighbor] + heuristic(neighbor, goal)
                if neighbor not in open_set:
                    open_set.enqueue(neighbor, f_score[neighbor])

    return "Path does not exist within committed time"
```

**Explanation:****1. Initialization:**

- The priority queue (`open_set`) is initialized with the starting node `S`, prioritized by its heuristic value.
- `g_score` tracks the cost from the start node to each node, initialized to infinity.
- `f_score` is the estimated total cost from the start to the goal through each node, initialized to infinity.
- The start node `S` has a `g_score` of 0 and an `f_score` equal to its heuristic value.

**2. Main Loop:**

- The node with the lowest `f_score` is dequeued.
- If the current node is the goal `G` and its `g_score` is within the committed time `t`, the path is reconstructed.
- If the current node is the goal but exceeds the committed time, an appropriate message is returned.
- For each neighbor of the current node, the `tentative_g_score` is calculated, considering the move time (1 minute for regular cells, more for toll booths).
- If the `tentative_g_score` is lower than the existing `g_score`, the path is updated, and the neighbor is added to the priority queue.

**3. Path Reconstruction:**

- The `reconstruct_path` function traces back from the goal node to the start node using the `came_from` map, creating the final path.

This algorithm effectively balances the cost to reach each node with the heuristic estimate to the goal, ensuring the delivery is completed within the committed time if possible. The inclusion of toll booth waiting times makes it well-suited for scenarios with varied move costs.



### 4.3 Level 3: Fuel limitation

In Level 3, the objective is to find the shortest feasible path from the starting location of the delivery vehicle (S) to the goal location (G), considering the fuel tank capacity limitation. Each move consumes 1 liter of fuel, and the vehicle can refuel at gas stations on the map. The map includes fuel stations (yellow squares) where the vehicle can refuel to full capacity.

*Optimal Algorithm: A Search Algorithm with Fuel Constraints\**

The A\* Search Algorithm is chosen for this level due to its efficiency in finding the optimal path while considering both the cost to reach a node, the heuristic estimate to the goal, and the fuel.

**Pseudocode:**

```
function AStarWithFuel(start, goal, grid, fuel_capacity, tolls, fuel_stations):
    open_set <- empty priority queue
    came_from <- empty map
    g_score <- map with default value of infinity
    f_score <- map with default value of infinity
    fuel_remaining <- map with default value of fuel_capacity

    open_set.enqueue(start, heuristic(start, goal))
    g_score[start] <- 0
    f_score[start] <- heuristic(start, goal)
    fuel_remaining[start] <- fuel_capacity

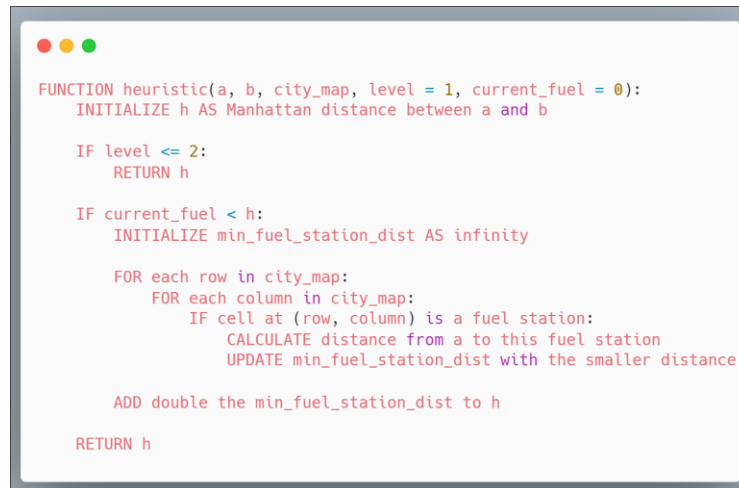
    while not open_set.isEmpty():
        current <- open_set.dequeue()
        if current == goal:
            return reconstruct_path(came_from, current)

        for each neighbor in get_neighbors(current, grid):
            move_cost <- if neighbor in tolls then tolls[neighbor] else 1
            if current in fuel_stations:
                current_fuel <- fuel_capacity
            else:
                current_fuel <- fuel_remaining[current] - 1

            if current_fuel < 0:
                continue # Skip paths that run out of fuel

            tentative_g_score <- g_score[current] + move_cost
            if tentative_g_score < g_score[neighbor] or current_fuel > fuel_remaining[neighbor]:
                came_from[neighbor] <- current
                g_score[neighbor] <- tentative_g_score
                f_score[neighbor] <- g_score[neighbor] + heuristic(neighbor, goal)
                fuel_remaining[neighbor] <- current_fuel
                if neighbor not in open_set:
                    open_set.enqueue(neighbor, f_score[neighbor])

    return "Path does not exist or is not feasible"
```

**More optimal heuristics:**

```
FUNCTION heuristic(a, b, city_map, level = 1, current_fuel = 0):
    INITIALIZE h AS Manhattan distance between a and b

    IF level <= 2:
        RETURN h

    IF current_fuel < h:
        INITIALIZE min_fuel_station_dist AS infinity

        FOR each row in city_map:
            FOR each column in city_map:
                IF cell at (row, column) is a fuel station:
                    CALCULATE distance from a to this fuel station
                    UPDATE min_fuel_station_dist with the smaller distance

        ADD double the min_fuel_station_dist to h

    RETURN h
```

**Explanation:****1. Initialization:**

- The priority queue (`open_set`) is initialized with the starting node  $S$ , prioritized by its heuristic value.
- `g_score` tracks the cost from the start node to each node, initialized to infinity.
- `f_score` is the estimated total cost from the start to the goal through each node, initialized to infinity.
- `fuel_remaining` tracks the fuel left at each node, initialized to the fuel capacity.

**2. Main Loop:**

- The node with the lowest `f_score` is dequeued.
- If the current node is the goal  $G$ , the path is reconstructed.
- For each neighbor of the current node, the `move_cost` is calculated, considering tolls.
- The fuel remaining is updated based on whether the current node is a fuel station.
- If moving to a neighbor leaves the vehicle without fuel, that path is skipped.
- If the path to the neighbor is better (lower `g_score`) or has more fuel, it is updated, and the neighbor is added to the priority queue.

**3. Path Reconstruction:**

- The `reconstruct_path` function traces back from the goal node to the start node using the `came_from` map, creating the final path.

This algorithm effectively balances the cost to reach each node, the heuristic estimate to the goal, and the fuel constraints, ensuring the delivery is completed within the fuel limits if possible. The inclusion of fuel stations and toll booths makes it well-suited for scenarios with varied move costs and fuel requirements.

## 4.4 Level 4: Multiple agents

In Level 4, the objective is to manage multiple delivery vehicles in the city, each with its own starting and goal locations. The interactions between vehicles are limited to competing for movement space, and the movements are turn-based. Each vehicle aims to optimize its path effectively while avoiding collisions with other vehicles. The process continues until the main delivery from  $s$  to  $g$  is completed or determined to be infeasible.

*Optimal Algorithm: Conflict-Based Search (CBS) for Multi-Agent Path Finding*

The Conflict-Based Search (CBS) algorithm is designed to handle multiple agents by planning paths for each agent individually and resolving conflicts as they arise. It ensures that agents do not occupy the same cell simultaneously and provides an optimal solution for multi-agent pathfinding problems.

### Pseudocode:

```
function CBS(agents, start_positions, goal_positions):
    constraints = []
    root = create_node(paths={}, constraints=constraints, cost=0)

    for agent in agents:
        root.paths[agent] = find_path(agent, start_positions[agent], goal_positions[agent],
constraints)
        if root.paths[agent] == None:
            return None

    open_set = priority_queue()
    open_set.push(root)

    while not open_set.is_empty():
        current = open_set.pop()

        conflict = find_conflict(current.paths)
        if conflict == None:
            return current.paths

        for agent in conflict.agents:
            new_constraints = current.constraints.copy()
            new_constraints.append(create_constraint(conflict, agent))
            new_node = create_node(paths=current.paths.copy(), constraints=new_constraints, cost=0)

            new_path = find_path(agent, start_positions[agent], goal_positions[agent], new_constraints)
            if new_path != None:
                new_node.paths[agent] = new_path
                new_node.cost = calculate_cost(new_node.paths)
                open_set.push(new_node)

    return None
```

**Explanation:****1. Initialization:**

- The CBS function initializes the root node with empty paths and constraints.
- For each agent, it computes an initial path from the start to the goal using a search algorithm, taking into account any constraints.
- If any agent's path is not found, it returns None, indicating infeasibility.

**2. Main Loop:**

- The CBS function uses a priority queue (`open_set`) to explore nodes, starting with the root node.
- It dequeues the node with the lowest cost and checks for conflicts in the paths of all agents.
- If no conflict is found, it returns the current paths as the solution.
- If a conflict is found, it generates new nodes with additional constraints to resolve the conflict, and these nodes are added to the priority queue for further exploration.

**3. Conflict Resolution:**

- The `find_conflict` function identifies conflicts between the paths of agents.
- The `create_constraint` function generates new constraints to avoid the identified conflict for the specified agent.
- The `find_path` function re-computes the path for the agent considering the new constraints.

**4. Path Calculation:**

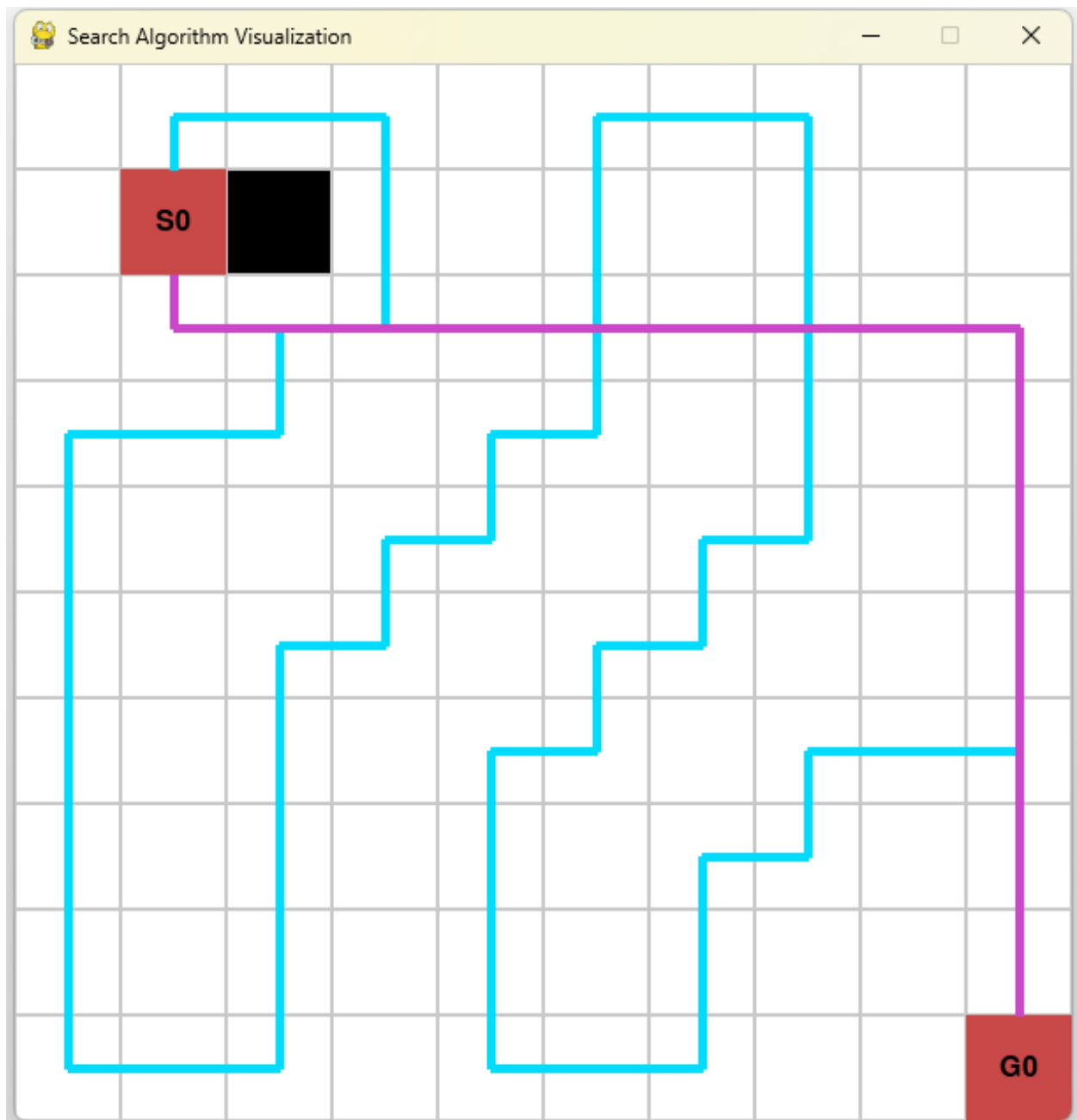
- The `calculate_cost` function computes the total cost of the paths, which is used to prioritize nodes in the priority queue.

This algorithm manages the paths of multiple agents effectively by ensuring they do not occupy the same cell simultaneously. It uses a conflict-based search strategy to coordinate movements and optimize paths for each vehicle, considering both their own goals and the main delivery goal. This makes it well-suited for complex, multi-agent environments.



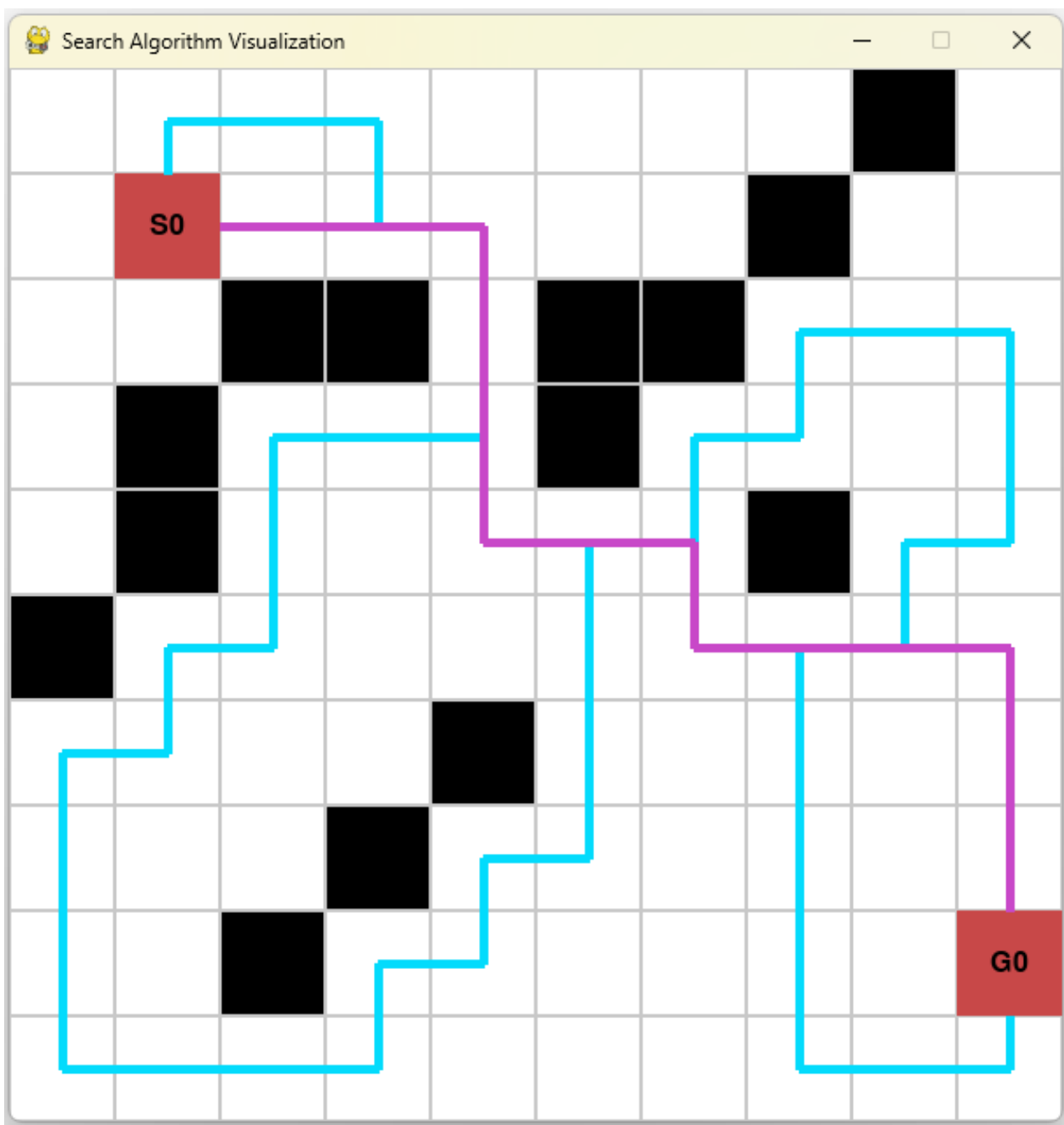
### 5.1.2 Test Case 1.2: Single Obstacle

- **Description:** Path from S to G with one obstacle blocking the direct path.
- **Input File:** input2\_level1.txt
- **Expected Output:** Path navigating around the obstacle.
- **Results:** The path is exactly as expected.



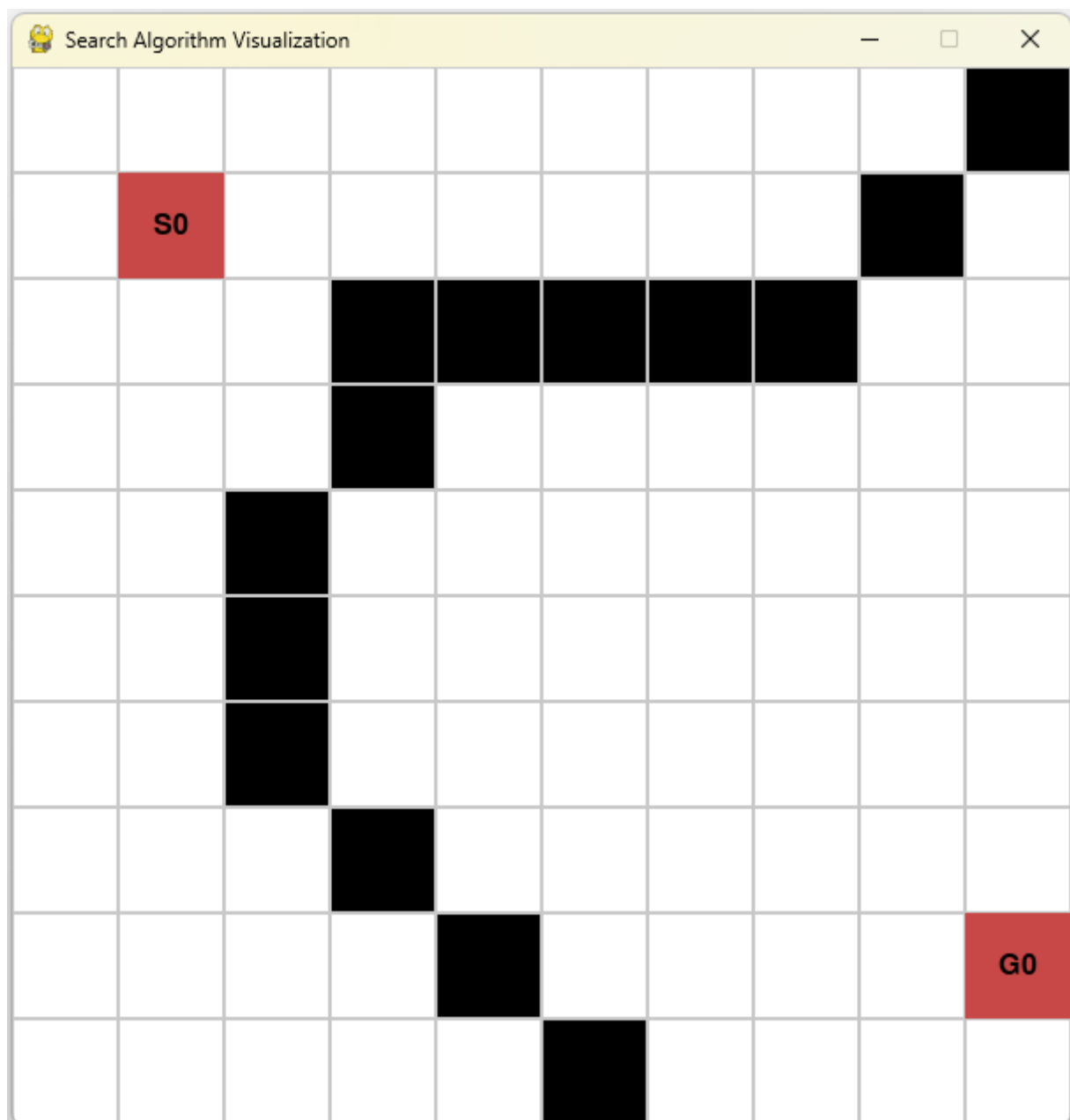
### 5.1.3 Test Case 1.3: Multiple Obstacles

- **Description:** Path from S to G with several obstacles.
- **Input File:** input3\_level1.txt
- **Expected Output:** Path finding the optimal way around multiple obstacles.
- **Results:** The path is exactly as expected.



#### 5.1.4 Test Case 1.4: No Possible Path

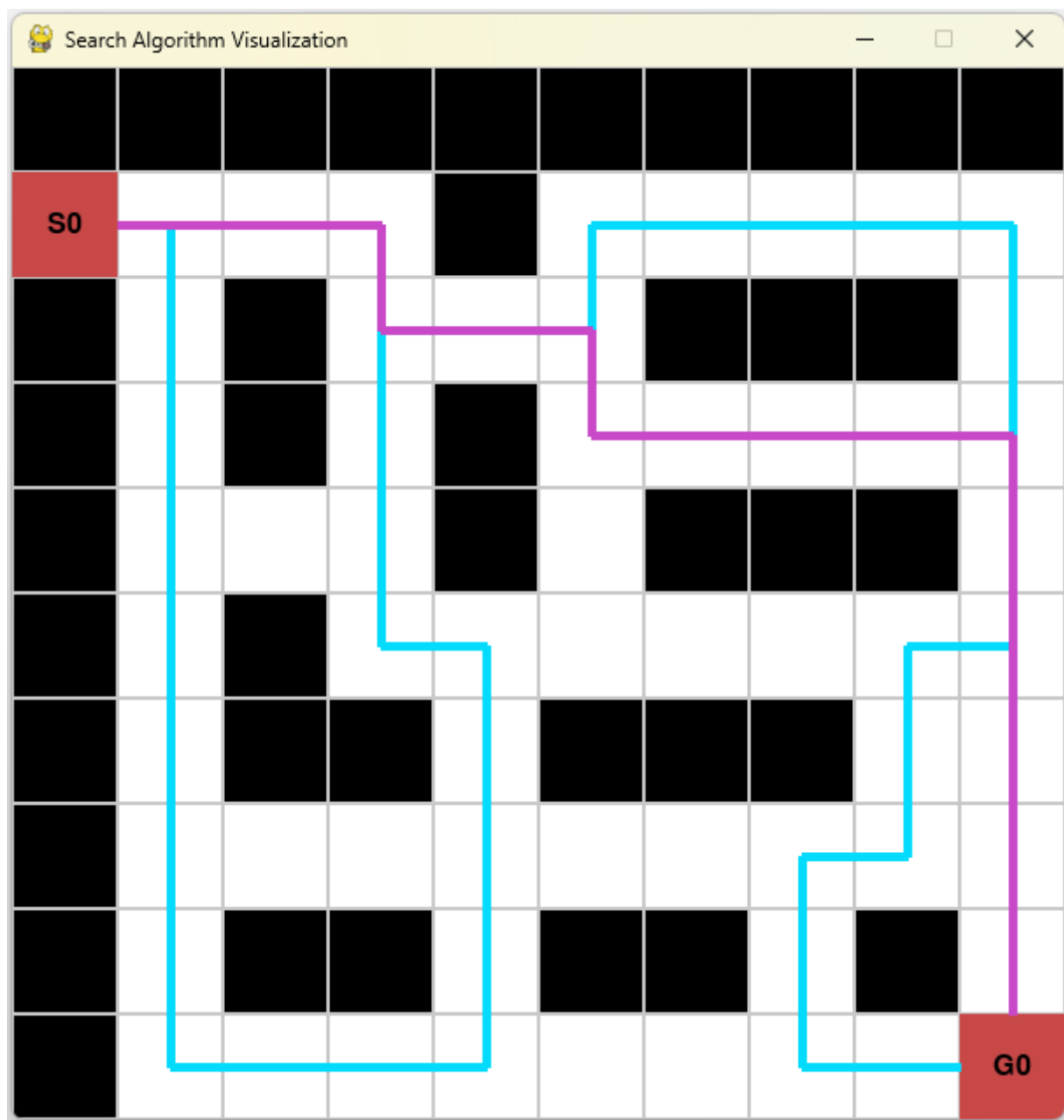
- **Description:** S and G separated by an impassable barrier.
- **Input File:** input4\_level1.txt
- **Expected Output:** No path exists.
- **Results:** No path exists.





### 5.1.5 Test Case 1.5: Complex Maze

- **Description:** A maze-like configuration with a path from S to G.
- **Input File:** input5\_level1.txt
- **Expected Output:** Correct path through the maze.
- **Results:** The path is exactly as expected.



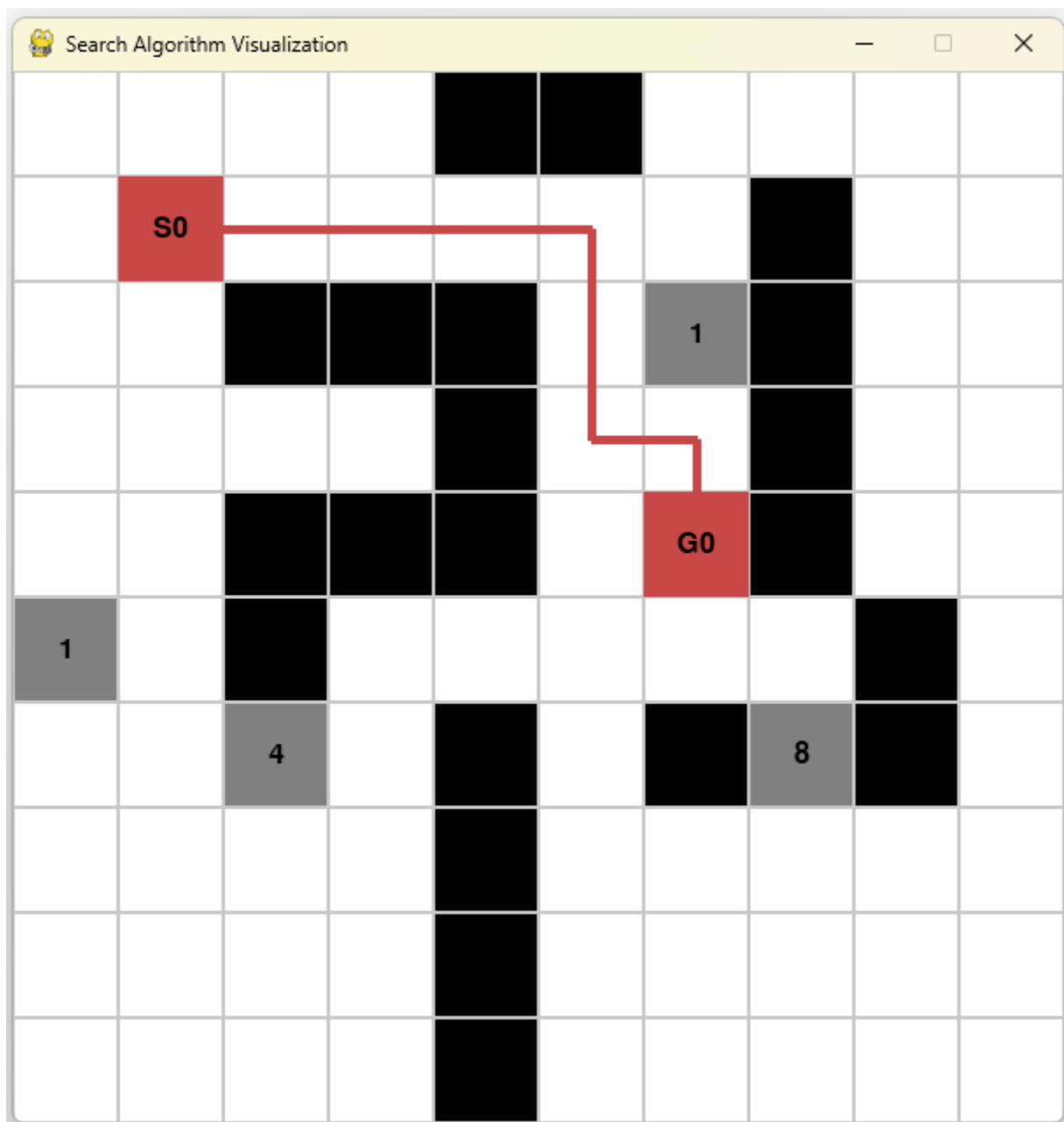
For simplicity without affecting the logic, all test cases are tested on a 10 x 10 map. Below is a statistical table of the path length of the algorithms in each test case.

Test case	Algorithms				
	BFS	DFS	UCS	GBFS	A*
1.1	16	54	16	16	16
1.2	16	54	16	16	16
1.3	15	45	15	15	15
1.4	-1	-1	-1	-1	-1
1.5	17	39	17	17	17

## 5.2 Level 2: Time Limitation

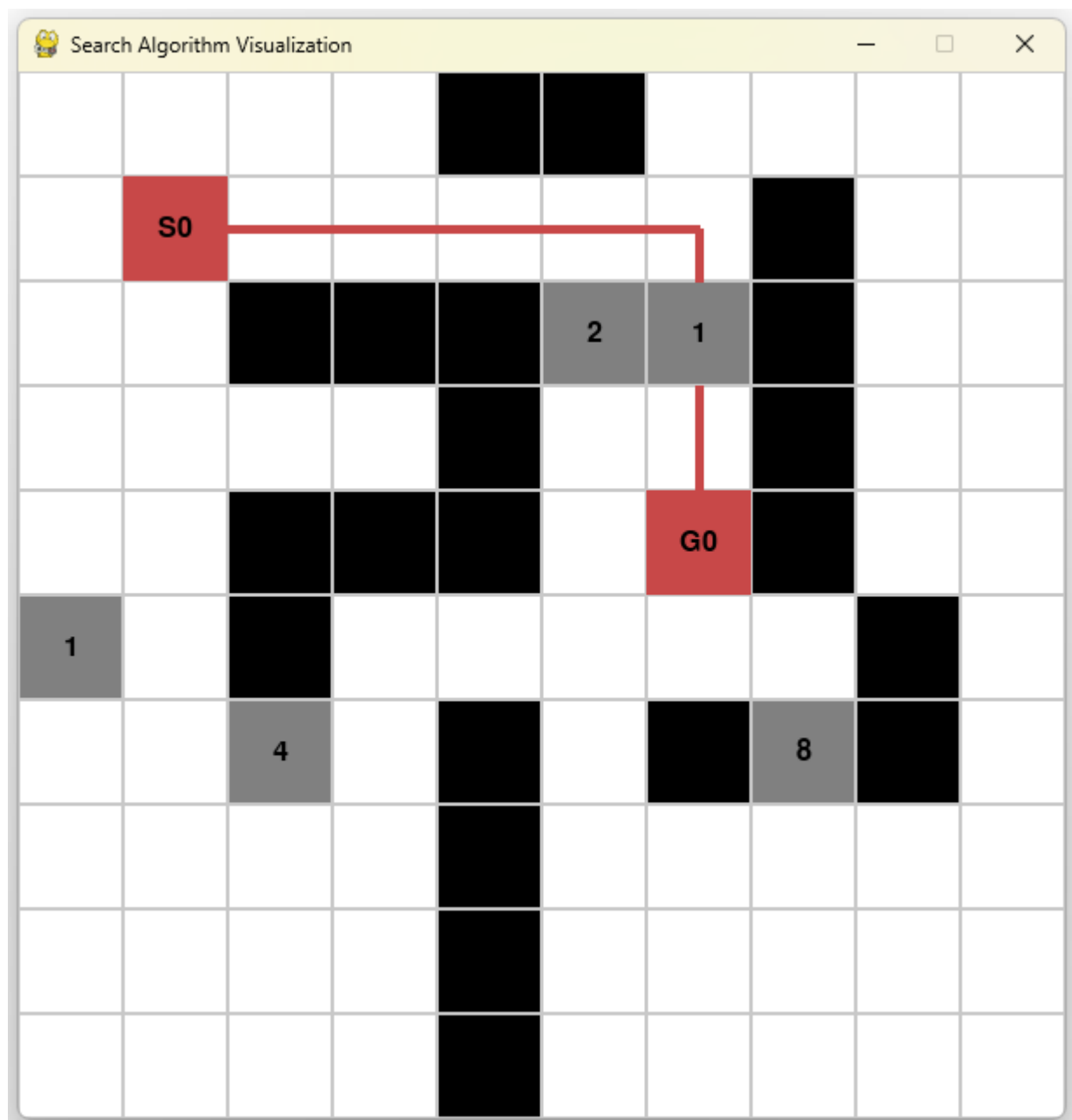
### 5.2.1 Test Case 2.1: Simple Path with Time Constraint

- **Description:** Direct path from S to G with a time constraint.
- **Input File:** input1\_level2.txt
- **Expected Output:** Path meeting the time constraint.
- **Results:** The path is exactly as expected.



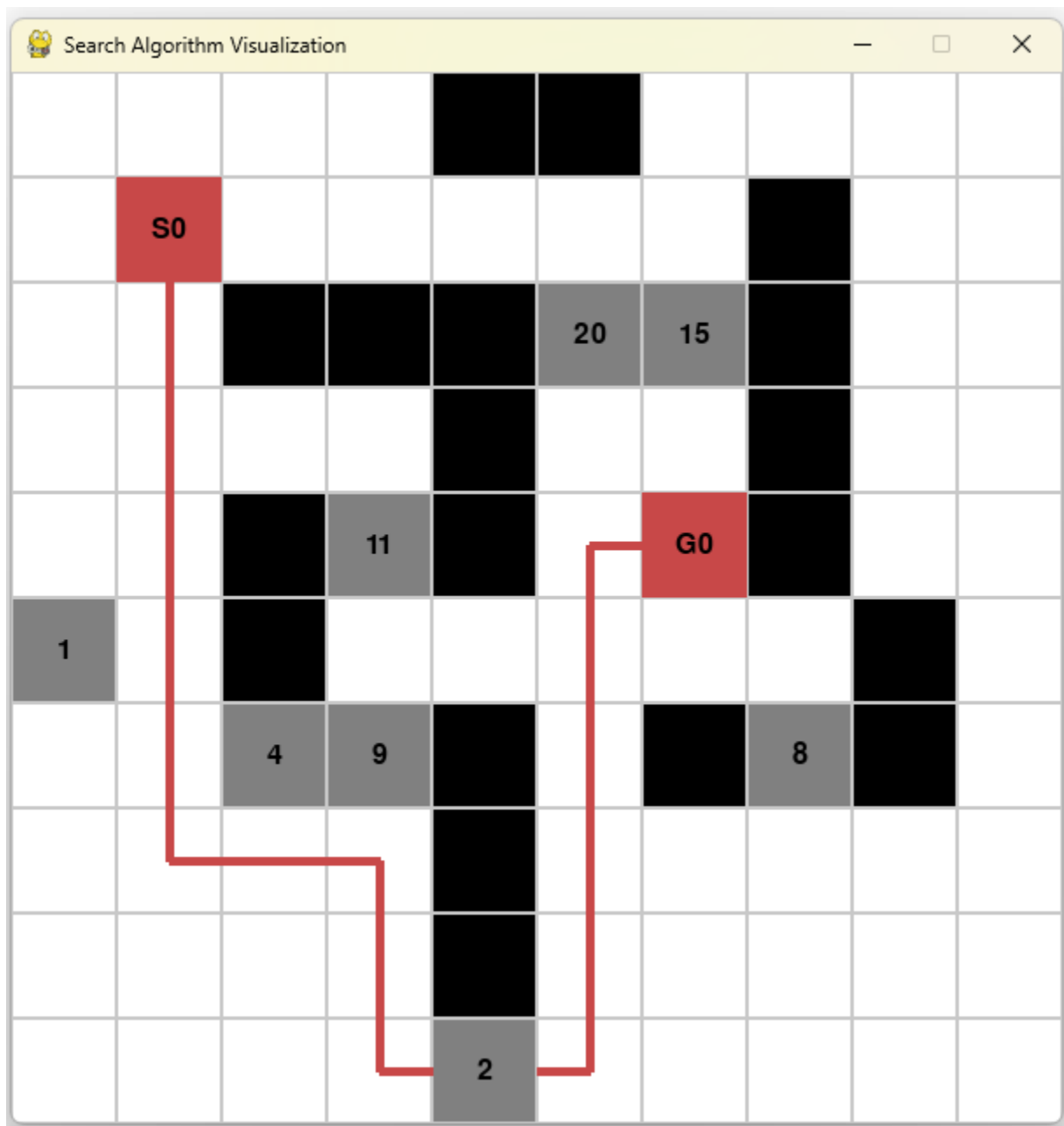
### 5.2.2 Test Case 2.2: Path with Toll Booths

- **Description:** Path from S to G with toll booths affecting travel time.
- **Input File:** input2\_level2.txt
- **Expected Output:** Path accounting for toll booth delays.
- **Results:** The path is exactly as expected.



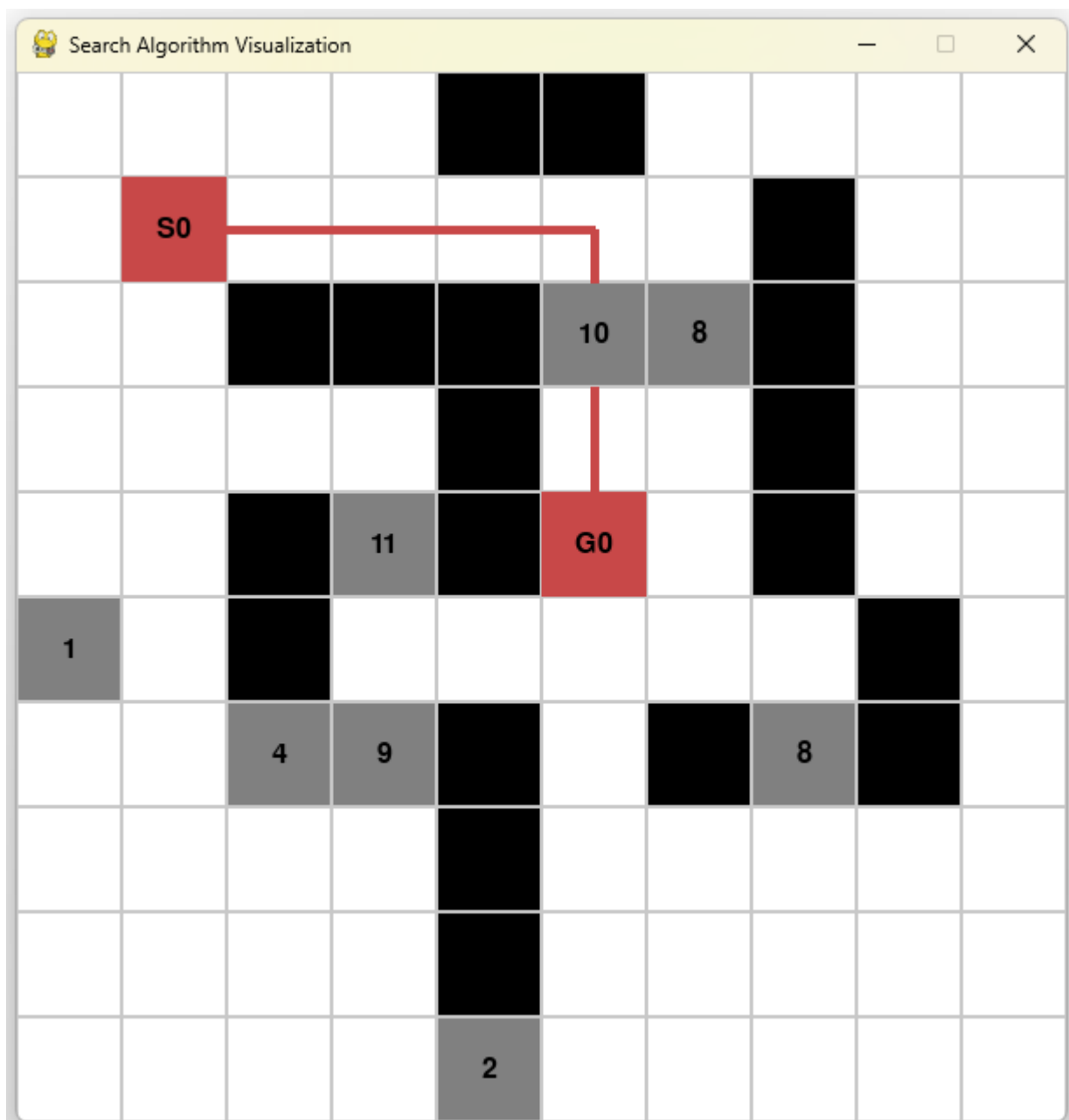
### 5.2.3 Test Case 2.3: Tight Time Constraint

- **Description:** A tight time constraint making the optimal path challenging.
- **Input File:** input3\_level2.txt
- **Expected Output:** Path within the tight time limit.
- **Results:** The path is exactly as expected.



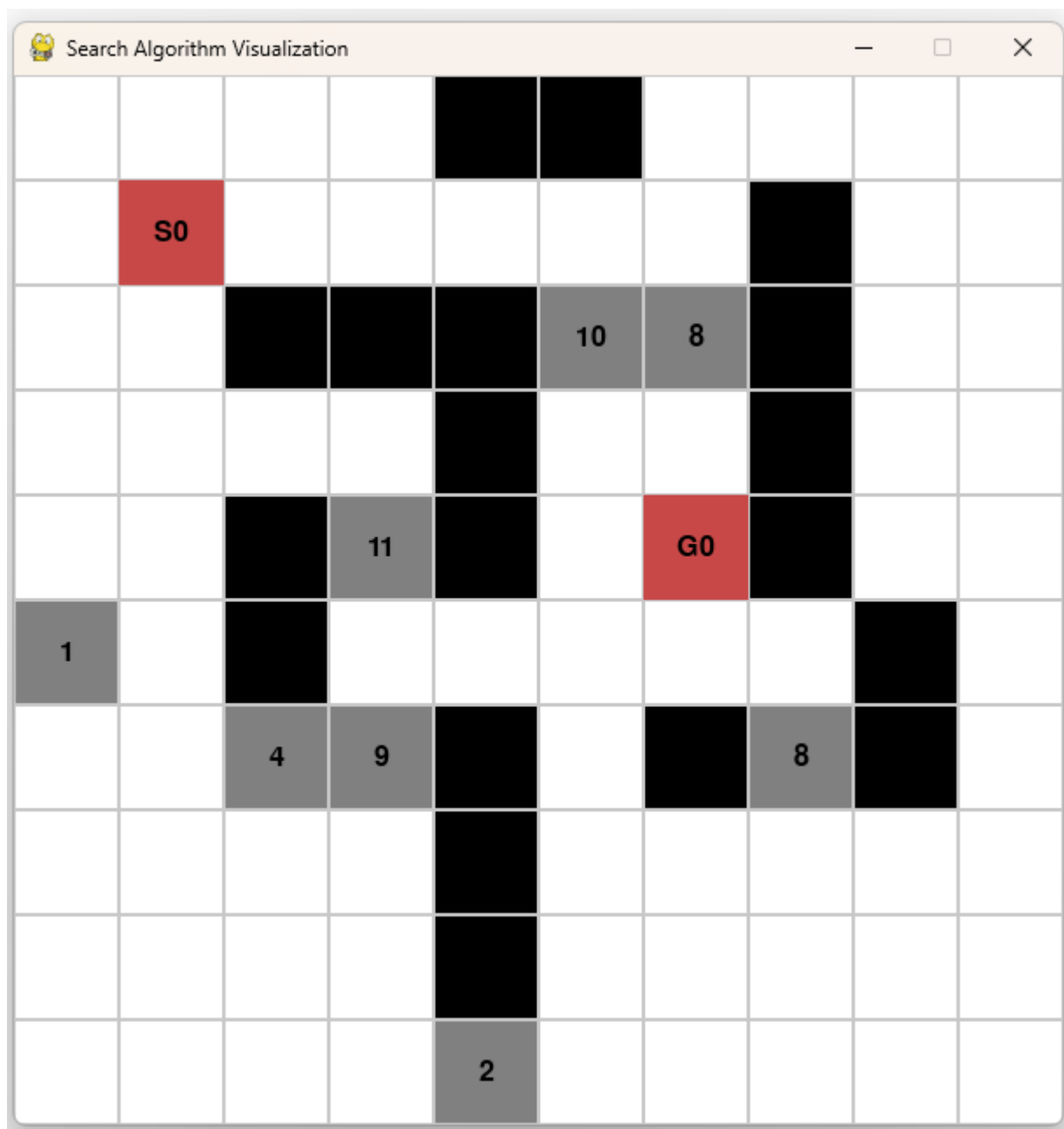
### 5.2.4 Test Case 2.4: Excess Time Constraint

- **Description:** Generous time constraint allowing multiple valid paths.
- **Input File:** input4\_level2.txt
- **Expected Output:** Path that meets the constraint with flexibility.
- **Results:** The path is exactly as expected.



### 5.2.5 Test Case 2.5: No Valid Path Within Time

- **Description:** Path not possible within the given time constraint.
- **Input File:** input5\_level2.txt
- **Expected Output:** Indication that no valid path exists.
- **Results:** No path exists.



For simplicity without affecting the logic, all test cases are tested on a 10 x 10 map with different distributions of toll booths and time constraints. Below is a statistical table of path lengths of the agent in each test case.

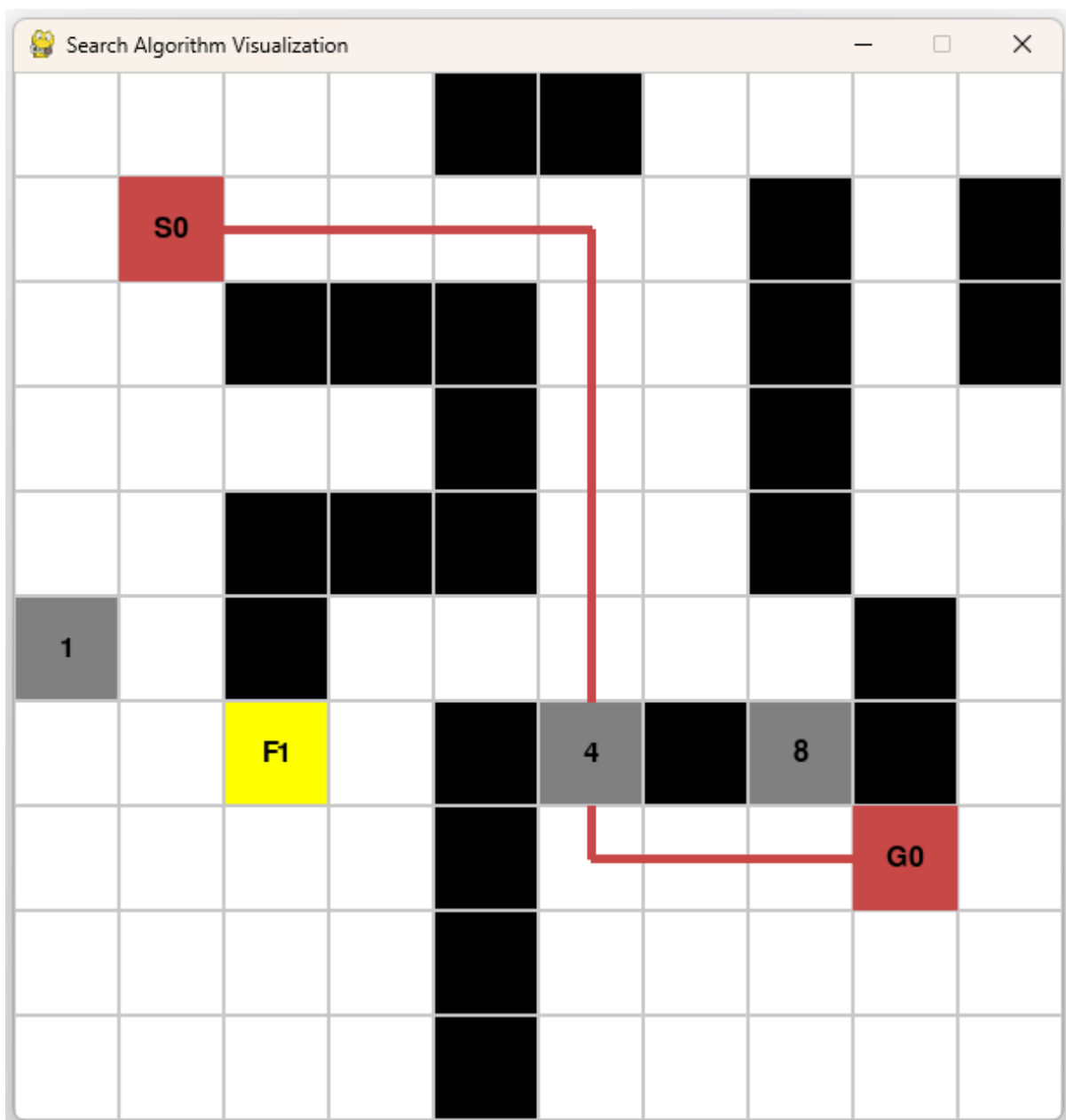
Test case	Time constraint	Time cost	Path length
2.1	20	8	8
2.2	20	9	8
2.3	20	20	18
2.4	20	17	7
2.5	15	16	-1



## 5.3 Level 3: Fuel Limitation

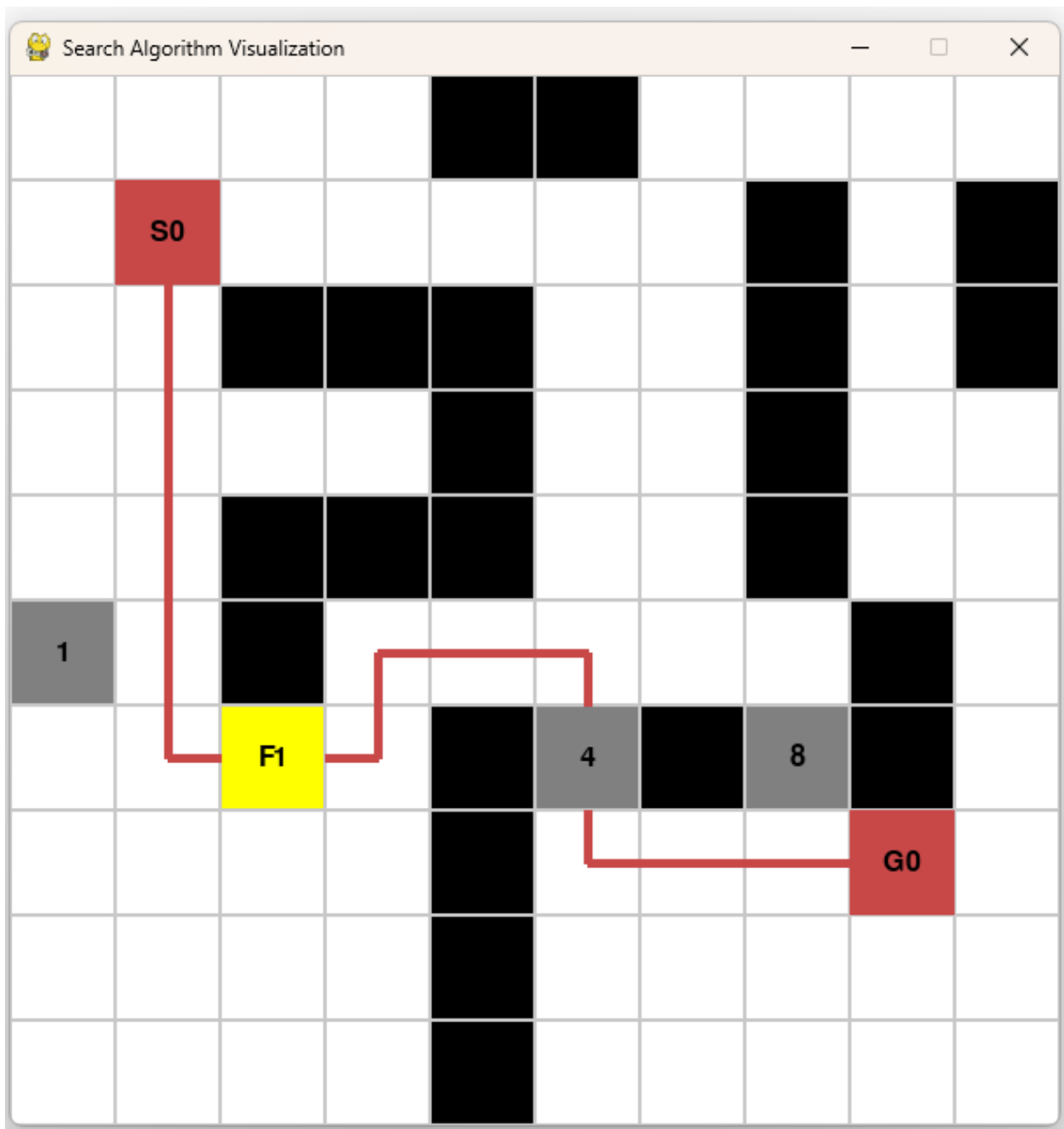
### 5.3.1 Test Case 3.1: Path with Refueling Station

- **Description:** Path from S to G requiring a stop at a refueling station.
- **Input File:** input1\_level3.txt
- **Expected Output:** Path including a stop at the refueling station.
- **Results:** The path is exactly as expected.



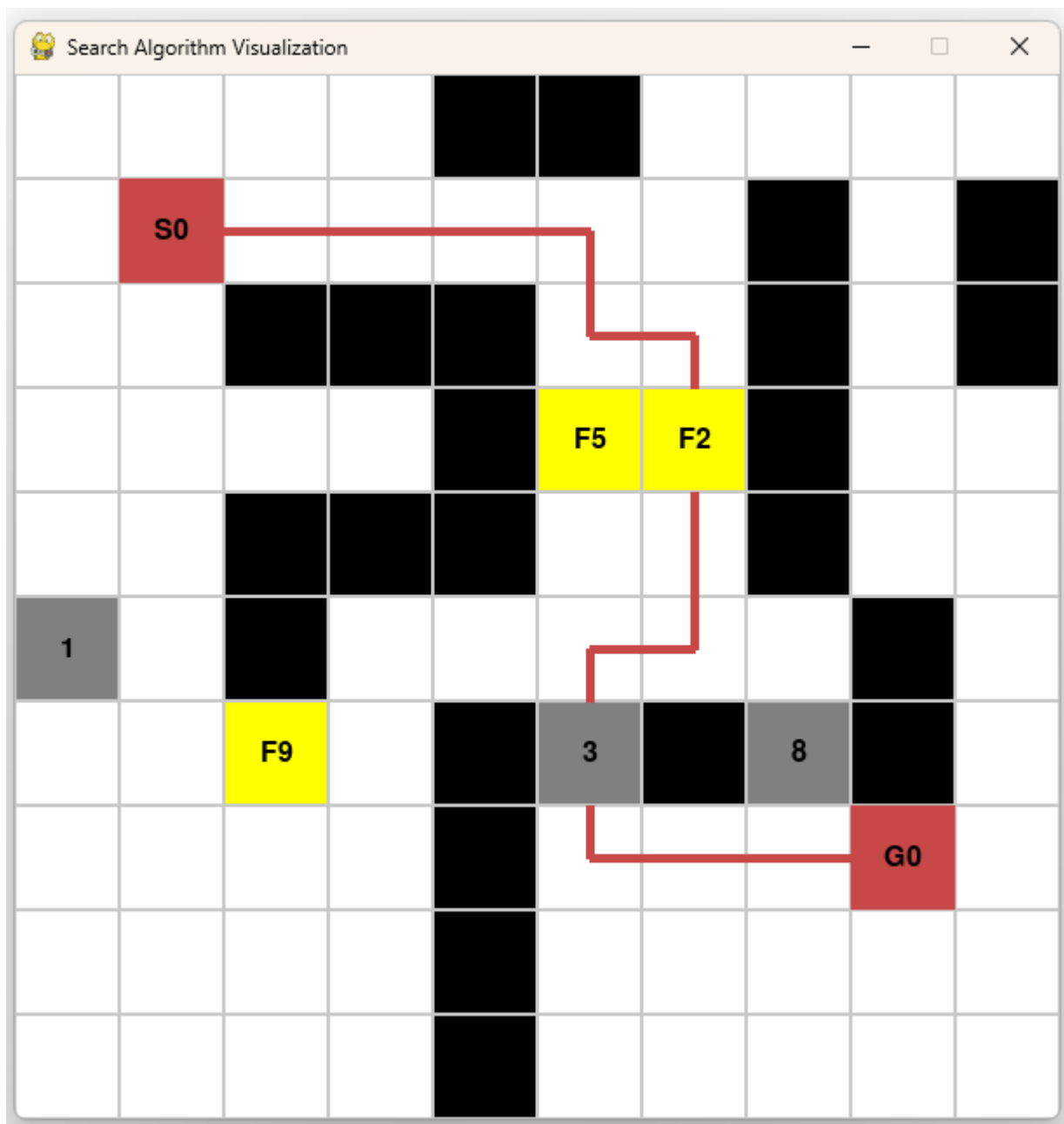
### 5.3.2 Test Case 3.2: Insufficient Fuel Without Refueling

- **Description:** Direct path from S to G without enough fuel.
- **Input File:** input2\_level3.txt
- **Expected Output:** Path demonstrating the need for refueling.
- **Results:** The path is exactly as expected.



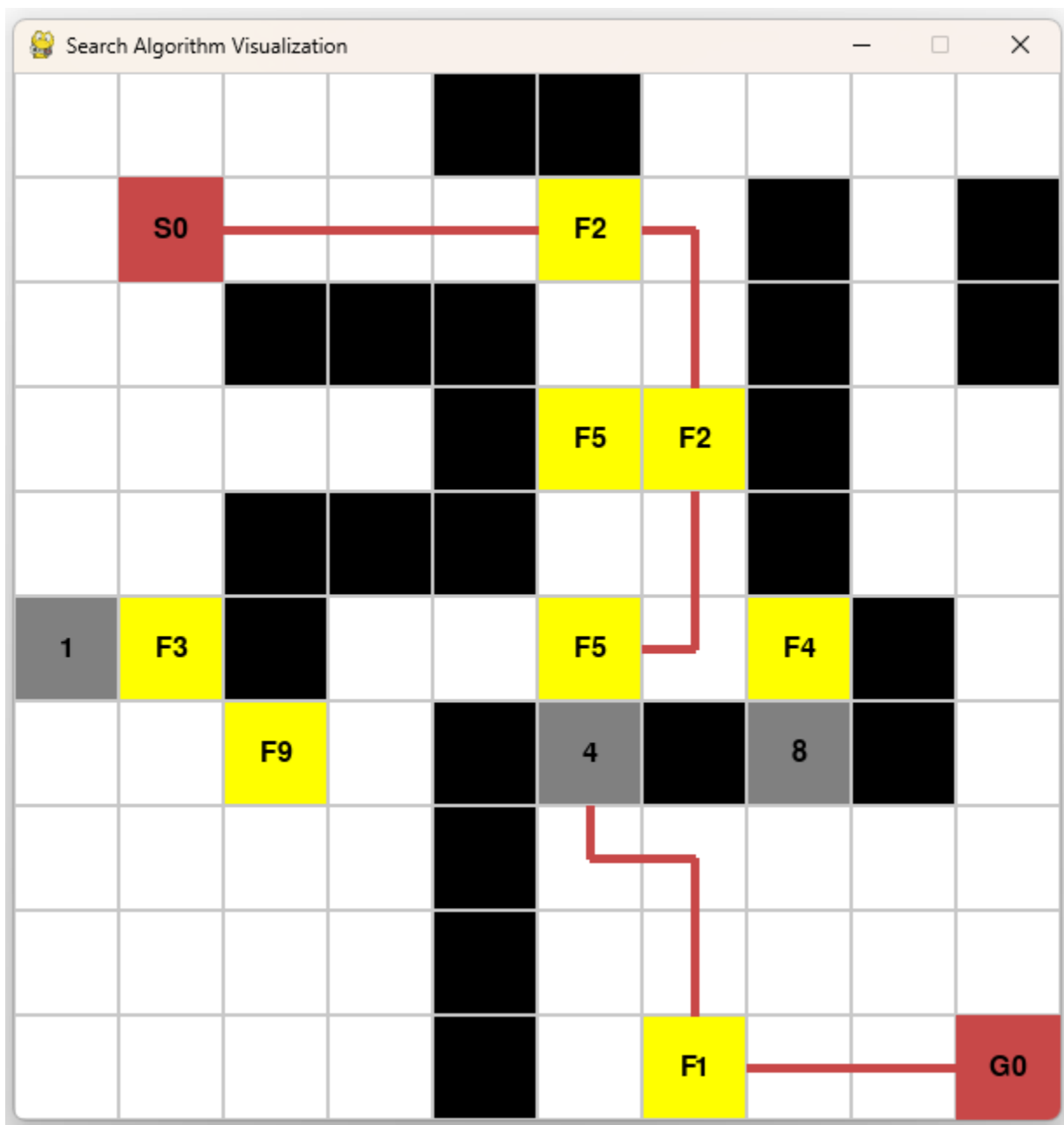
### 5.3.3 Test Case 3.3: Multiple Refueling Stations

- **Description:** Path from S to G with multiple refueling options.
- **Input File:** input3\_level3.txt
- **Expected Output:** Optimal path using the best refueling option.
- **Results:** The path is exactly as expected.



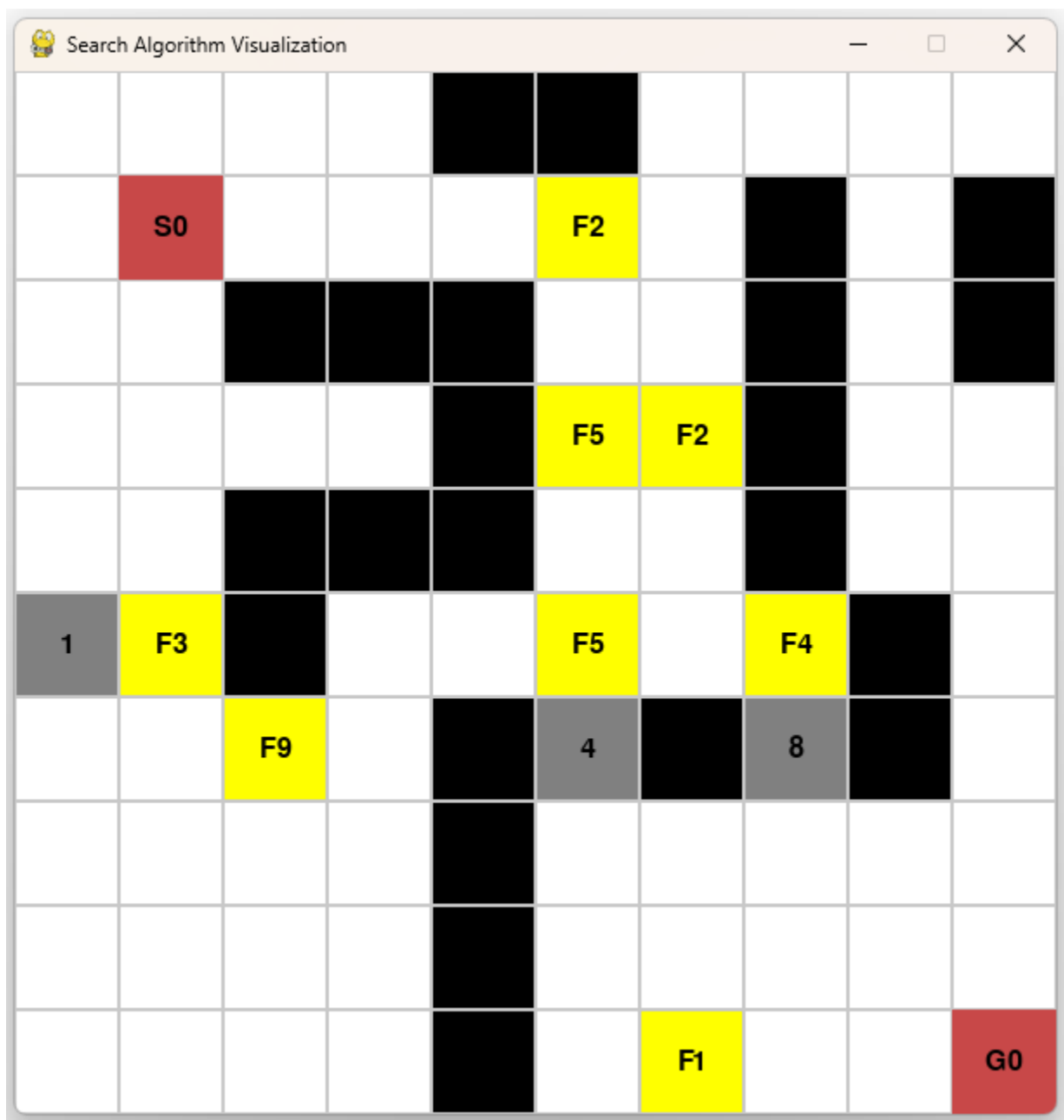
### 5.3.4 Test Case 3.4: Tight Fuel Constraint

- **Description:** Minimal fuel available, requiring precise pathfinding.
- **Input File:** input4\_level3.txt
- **Expected Output:** Path within the tight fuel limit.
- **Results:** The path is exactly as expected.



### 5.3.5 Test Case 3.5: No Valid Path Within Fuel Limit

- **Description:** Path not possible with the given fuel constraint.
- **Input File:** input5\_level3.txt
- **Expected Output:** Indication that no valid path exists.
- **Results:** No path exists.



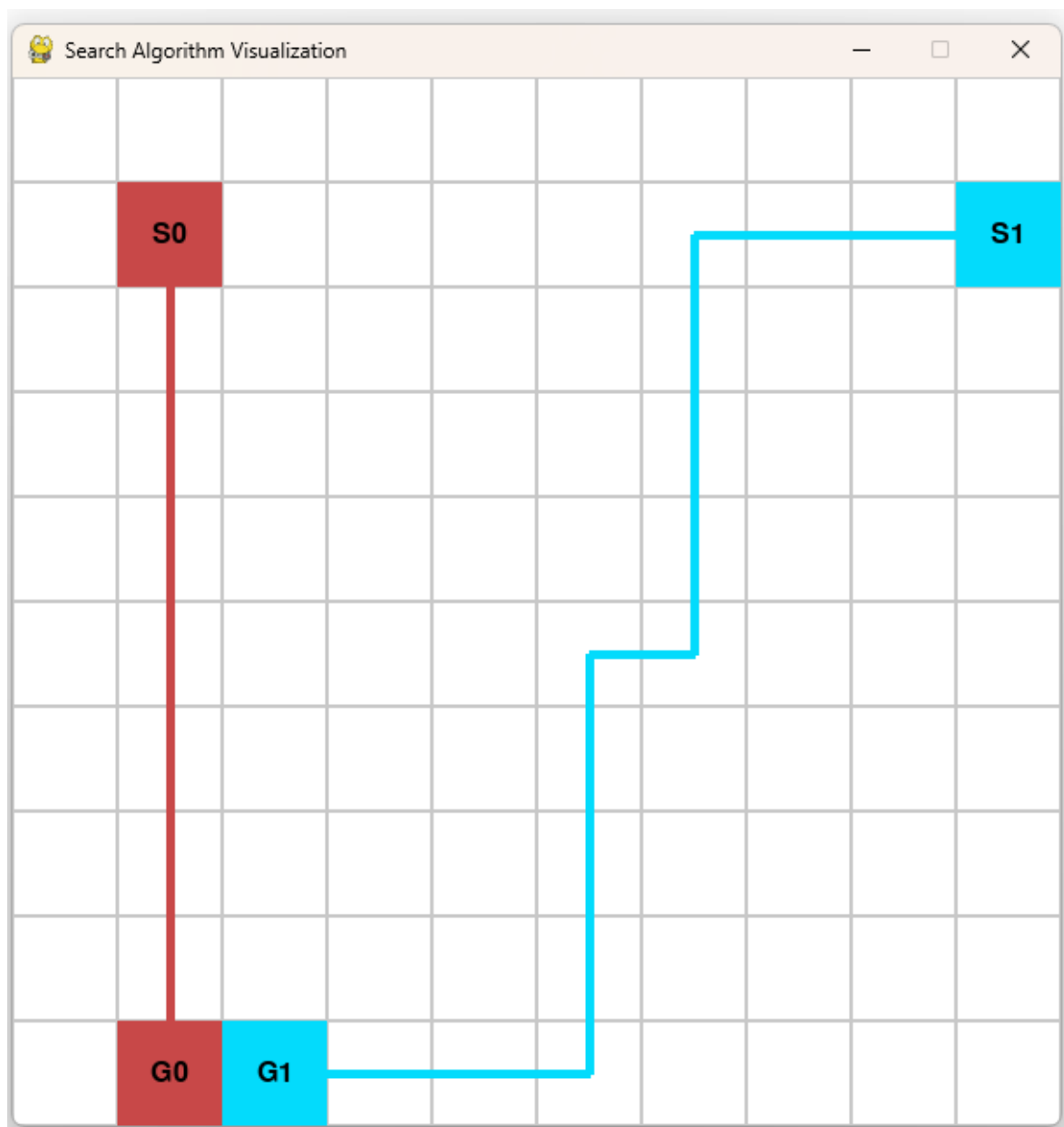
For simplicity without affecting the logic, all test cases are tested on a 10 x 10 map with different distributions of toll booths, fuel stations and time constraints. Below is a statistical table of path lengths of the agent and input parameters in each test case.

Test case	Time constraint	Time cost	Fuel capacity	Refill time	Path length
3.1	20	17	15	0	13
3.2	20	20	10	1	15
3.3	20	20	10	1	15
3.4	35	32	5	4	18
3.5	35	-1	4	-1	-1

## 5.4 Level 4: Multiple Agents

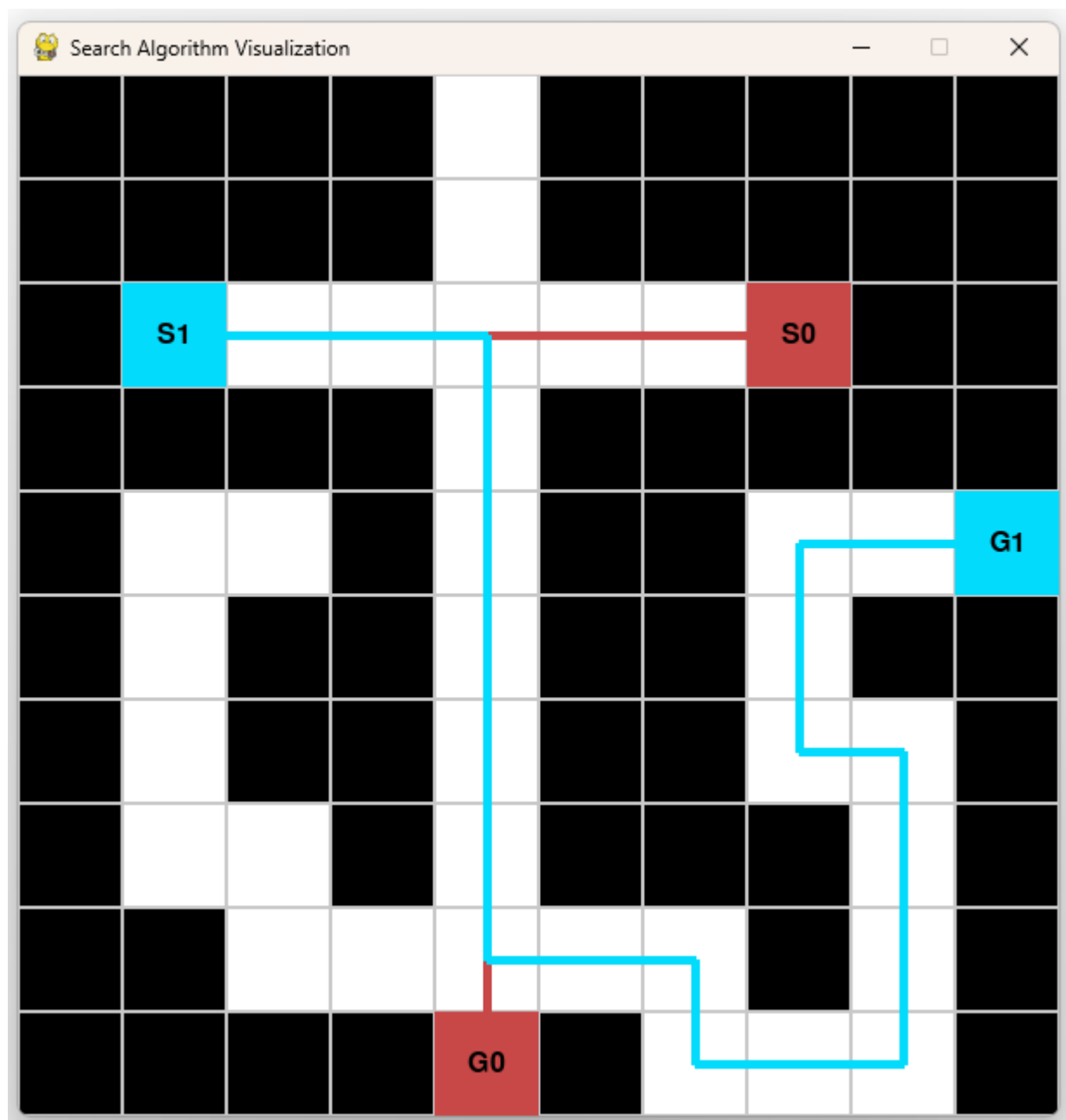
### 5.4.1 Test Case 4.1: Two Agents with Non-Intersecting Paths

- **Description:** Two agents with separate paths from S to G.
- **Input File:** input1\_level4.txt
- **Expected Output:** Independent paths for both agents.
- **Results:** The path is coordinated between agents.



#### 5.4.2 Test Case 4.2: Two Agents with Intersecting Paths

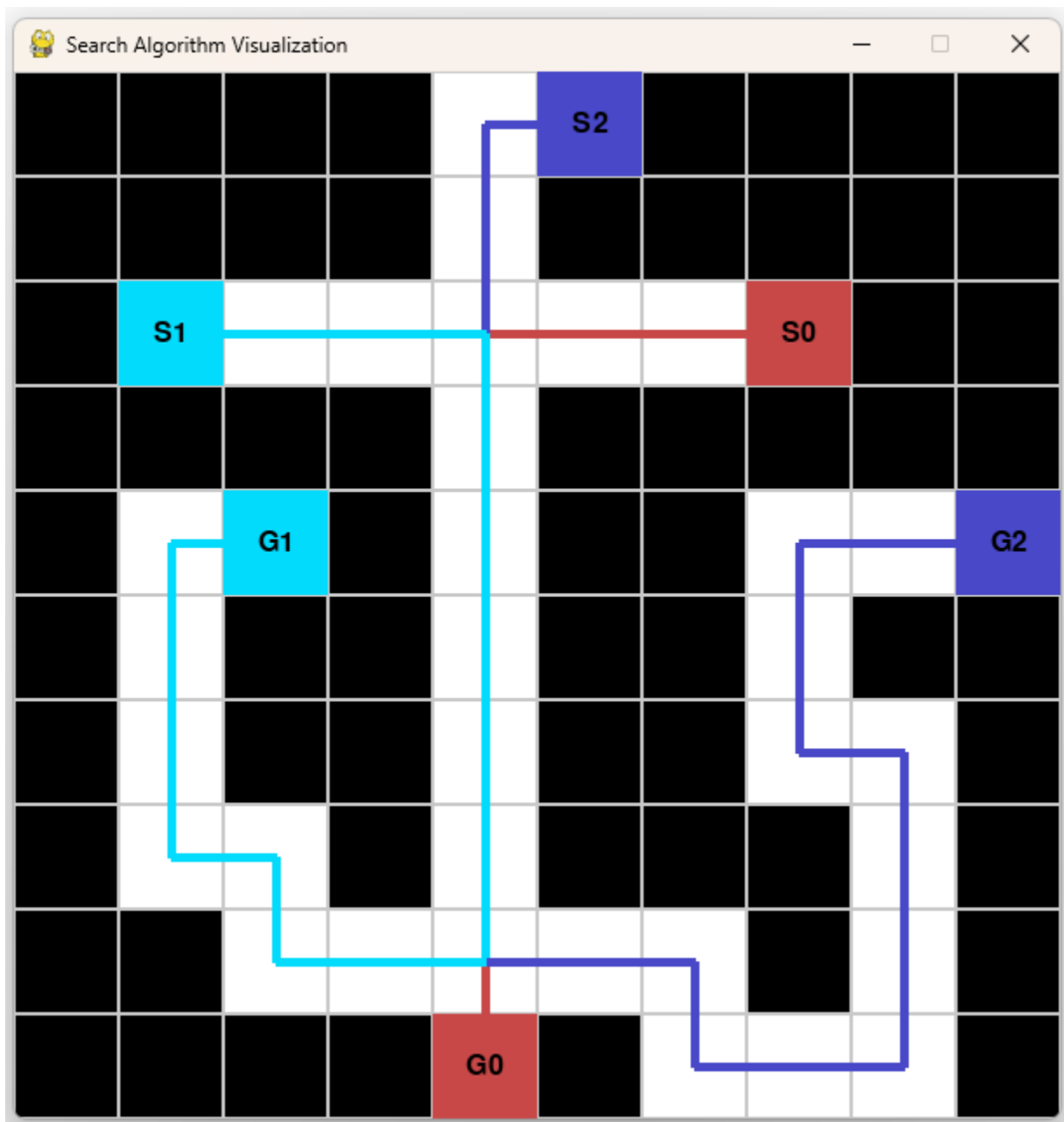
- **Description:** Two agents with paths that intersect.
- **Input File:** input2\_level4.txt
- **Expected Output:** Coordinated paths avoiding collision.
- **Results:** The path is coordinated between agents.





#### 5.4.3 Test Case 4.3: Three Agents with narrow path

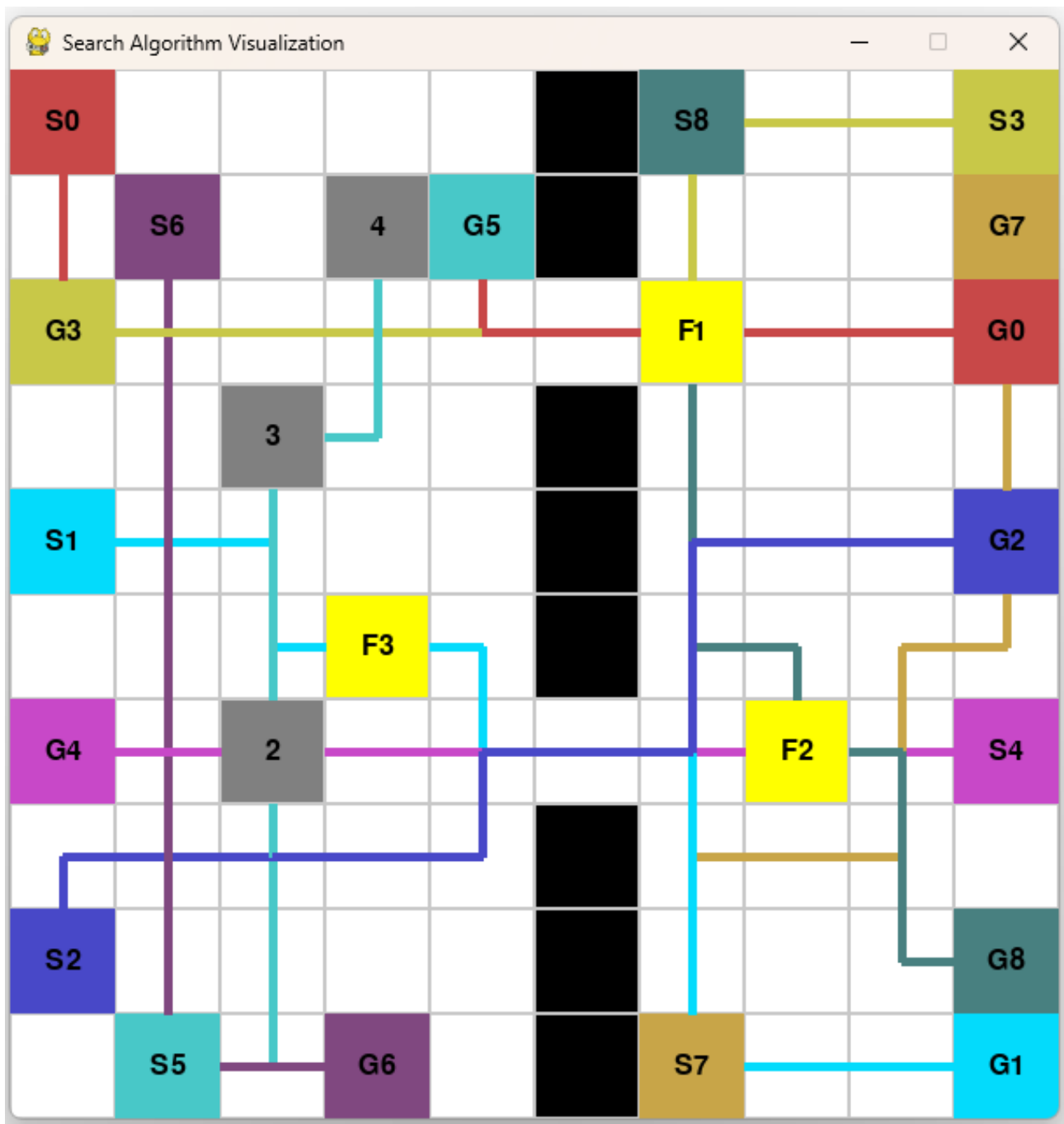
- **Description:** Three agents with narrow paths, need to wait
- **Input File:** input3\_level4.txt
- **Expected Output:** Coordinated paths for all three agents.
- **Results:** The path is coordinated between agents.





#### 5.4.5 Test Case 4.5: Maximum Number of Agents

- **Description:** Maximum of 9 agents with complex paths.
- **Input File:** input5\_level4.txt
- **Expected Output:** Coordinated paths for all agents.
- **Results:** The path is coordinated between agents.



For simplicity without affecting the logic, all test cases are tested on a 10 x 10 map with different distributions of toll booths, fuel stations, agents and time constraints. Below is a statistical table of path lengths of the agents and input parameters in each test case.

*Note:* In test cases 2 and 3, the total path part means that the first term is the actual path, the next term is the waiting time for another agent. This is also true for test cases 4 and 5 for larger numbers of agents.

Test case	Number of agents	Agent								
		1	2	3	4	5	6	7	8	9
4.1	2	9	16							
4.2	2	11	23 + 1							
4.3	3	11	18 + 2	23 + 1						
4.4	5	14	16	14	12	10				
4.5	9	14	15	15	12	10	12	11	12	12

These test cases cover a range of scenarios for each level, ensuring that the algorithms are robust and capable of handling various complexities. The results section will be filled in with the actual performance metrics and success/failure status after running the tests.

## 6. References

1. GeeksforGeeks. (2023b, March 22). *Search algorithms in AI*. GeeksforGeeks.  
<https://www.geeksforgeeks.org/search-algorithms-in-ai/>
2. Lim, S., & Kuby, M. (2010). Heuristic algorithms for siting alternative-fuel stations using the Flow-Refueling Location Model. *European Journal of Operational Research*, 204(1), 51–61.  
<https://doi.org/10.1016/j.ejor.2009.09.032>
3. *Pygame Front Page — pygame v2.6.0 documentation*. (n.d.). <https://www.pygame.org/docs/>
4. Sharon, G., Stern, R., Felner, A., & Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219, 40–66.  
<https://doi.org/10.1016/j.artint.2014.11.006>