ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN


# BÁO CÁO ĐỒ ÁN THỰC HÀNH 1

## MÔN HỌC: NHẬP MÔN TRÍ TUỆ NHÂN TẠO - 22CLC07

---

# DELIVERY SYSTEM

---

*Sinh viên:*
Trần Đoàn Huy Phước - 22127338
Trần Đức Bình - 22127037
Lê Quang Tân - 22127378
Trần Quốc Việt - 22127454

*Giảng viên:*
Nguyễn Tiến Huy
Nguyễn Trần Duy Minh
Bùi Duy Đăng

# Mục lục


**Mục lục** 1

**1 Thông tin nhóm và bảng phân công báo cáo mức độ hoàn thành** 3
- 1.1 Thông tin sinh viên . . . 3
- 1.2 Bảng phân công và báo cáo mức độ hoàn thành . . . 3

**2 Giới thiệu chung về đồ án** 4
- 2.1 Tổng quan nội dung đồ án . . . 4
- 2.2 Demo đồ án . . . 4
- 2.3 Luồng hoạt động . . . 4
- 2.4 Lớp Board . . . 5
  - 2.4.1 Cấu trúc dữ liệu . . . 5
  - 2.4.2 Các phương thức trong lớp . . . 6
- 2.5 Các hàm quan trọng . . . 7
  - 2.5.1 Hàm `reconstruct_path` . . . 7
  - 2.5.2 Hàm `heuristic` . . . 7

**3 Level 1** 7
- 3.1 Các thuật toán được sử dụng . . . 7
  - 3.1.1 Thuật toán BFS . . . 7
  - 3.1.2 Thuật toán GBFS . . . 8
  - 3.1.3 Thuật toán DFS . . . 8
  - 3.1.4 Thuật toán IDS và DLS . . . 9
  - 3.1.5 Thuật toán Uniform-Cost Search (UCS) . . . 9
  - 3.1.6 Thuật toán A* Search . . . 10
- 3.2 Các test case . . . 11
  - 3.2.1 Kết quả cho map 1 . . . 12
  - 3.2.2 Kết quả cho map 2 . . . 13
  - 3.2.3 Kết quả cho map 3 . . . 14
  - 3.2.4 Kết quả cho map 4 . . . 15
  - 3.2.5 Kết quả cho map 5 . . . 16
- 3.3 Nhận xét . . . 17
  - 3.3.1 Map 1 . . . 17
  - 3.3.2 Map 2 . . . 17
  - 3.3.3 Map 3 . . . 17
  - 3.3.4 Map 4 . . . 17
  - 3.3.5 Map 5 . . . 17

**4 Level 2** 18
- 4.1 Thuật toán . . . 18
- 4.2 Các test case . . . 18
  - 4.2.1 Test case 1 . . . 18
  - 4.2.2 Test case 2 . . . 19
  - 4.2.3 Test case 3 . . . 20
  - 4.2.4 Test case 4 . . . 20
  - 4.2.5 Test case 5 . . . 21

**5 Level 3** 22
- 5.1 Ý tưởng . . . 22
- 5.2 Cấu trúc dữ liệu: . . . 22

5.3 Thuật toán . . . 22
5.4 Các test case . . . 23
5.4.1 Test case 1 . . . 23
5.4.2 Test case 2 . . . 24
5.4.3 Test case 3 . . . 25
5.4.4 Test case 4 . . . 26
5.4.5 Test case 5 . . . 27
5.5 Nhận xét tổng thể . . . 27

**6 Level 4** **28**
6.1 Thuật toán . . . 28
6.1.1 Ý tưởng chung . . . 28
6.1.2 Thuật toán . . . 28
6.2 Các test case . . . 29
6.2.1 Test case 1 . . . 29
6.2.2 Test case 2 . . . 30
6.2.3 Test case 3 . . . 31
6.2.4 Test case 4 . . . 33
6.2.5 Test case 5 . . . 34
6.3 Nhận xét tổng thể . . . 34

**7 Visualize và đọc ghi file** **35**
7.1 Pygame . . . 35
7.1.1 Cách cài đặt . . . 35
7.1.2 Các module chính được sử dụng . . . 35
7.1.3 Những function chính: . . . 35
7.1.4 Những function hỗ trợ . . . 36
7.2 Hàm đọc, ghi file . . . 36

**8 Tham khảo** **36**

# 1 Thông tin nhóm và bảng phân công báo cáo mức độ hoàn thành

## 1.1 Thông tin sinh viên

| THÔNG TIN SINH VIÊN | | | |
|---|---|---|---|
| MSSV | HỌ VÀ TÊN | EMAIL | SĐT |
| 22127037 | Trần Đức Bình | tdbinh22@clc.fitus.edu.vn | 0933873919 |
| 22127338 | Trần Đoàn Huy Phước | tdhphuoc22@clc.fitus.edu.vn | 0852947855 |
| 22127378 | Lê Quang Tân | lqtan22@clc.fitus.edu.vn | 0901931656 |
| 22127454 | Trần Quốc Việt | tqviet22@clc.fitus.edu.vn | 0943791896 |

## 1.2 Bảng phân công và báo cáo mức độ hoàn thành

| BẢNG PHÂN CÔNG VÀ ĐÁNH GIÁ MỨC ĐỘ HOÀN THÀNH | | |
|---|---|---|
| CÔNG VIỆC | SINH VIÊN THỰC HIỆN | MỨC ĐỘ HOÀN THÀNH |
| GUI/Visualize | 22127378 | 100% |
| Cài đặt Class Board | 22127037 | 100% |
| Cài đặt Utils | 22127338 | 100% |
| Level1 BFS và GBFS | 22127454 | 100% |
| Level1 UCS và A*search | 22127037 | 100% |
| Level1 DFS và DLS | 22127338 | 100% |
| Level2 UCS | 22127378 | 100% |
| Level2 A*search | 22127454 | 100% |
| Level3 | 22127037 | 100% |
| Level4 | 22127338 | 90% |

- Mức độ hoàn thành các level:

  - Level 1: 100% (6 thuật toán)
  - Level 2: 100% (2 thuật toán)
  - Level 3: 100% (1 thuật toán)
  - Level 4: 90% (1 thuật toán)

# 2 Giới thiệu chung về đồ án

## 2.1 Tổng quan nội dung đồ án

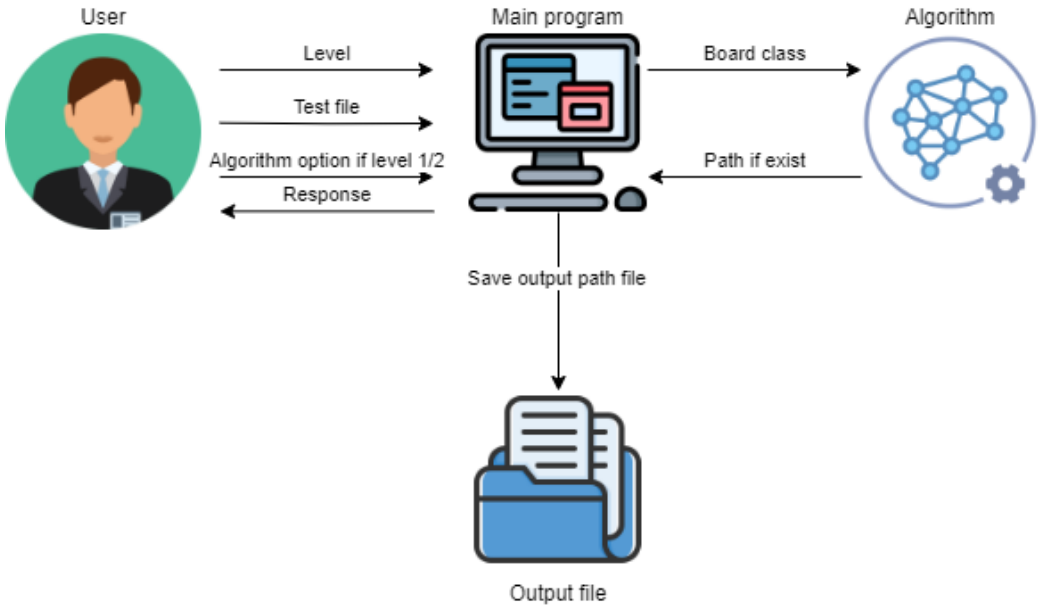
- Dưới sự phát triển của thương mại điện tử, người tiêu dùng có thể dễ dàng mua sắm sản phẩm chỉ với một chiếc điện thoại thông minh ngay tại nhà. Trong bối cảnh này, ngành logistics đóng vai trò quan trọng trong việc đảm bảo vận chuyển hàng hóa hiệu quả từ người bán đến người mua. Chiến lược quan trọng nhất để các nhà cung cấp logistics cạnh tranh hiệu quả là phát triển một hệ thống tìm đường ngắn nhất. Hệ thống này tối ưu hóa các tuyến đường giao hàng, giảm thiểu thời gian di chuyển và tiết kiệm nhiên liệu.
- Chúng ta sẽ chịu trách nhiệm phát triển hệ thống này cho Công ty TNHH Logistics HCMUS, sử dụng các thuật toán tìm kiếm đã học để tìm đường từ xe giao hàng đến khách hàng. Bản đồ thành phố được mô hình hóa trong không gian 2D với n hàng và m cột. Bản đồ bao gồm các đa giác mô tả các công trình như tòa nhà, tường, v.v., và các đối tượng này không thay đổi trên bản đồ. Nhìn chung, bản đồ có cấu trúc như một mê cung và nhiệm vụ của chúng ta là tìm đường đi từ điểm bắt đầu cho tới đích. Hệ thống được chia theo 4 cấp độ (level) với độ khó tăng dần như sau:

- **Level 1:** Không có các yếu tố phụ, đơn giản chỉ là một bài toán tìm đường trong mê cung.
- **Level 2:** Có thêm giới hạn thời gian và các trạm thu phí (Toll Booths), nơi mà khi chúng ta đi qua sẽ mất một khoảng thời gian nhất định, đòi hỏi chúng ta phải chú ý về vấn đề thời gian.
- **Level 3:** Có thêm giới hạn về nhiên liệu và các trạm xăng, đòi hỏi chúng ta phải cân nhắc khả năng xe có thể tới đích mà không bị hết nhiên liêu hay không.
- **Level 4:** Có thêm các xe giao hàng khác đang làm nhiệm vụ, đòi hỏi chúng ta phải cân nhắc khả năng va chạm giữa các xe trên đường.

## 2.2 Demo đồ án

- Link demo1: https://youtu.be/wJlrrRy7XZA
- Link demo2: https://youtu.be/JUQF1u78ZlQ

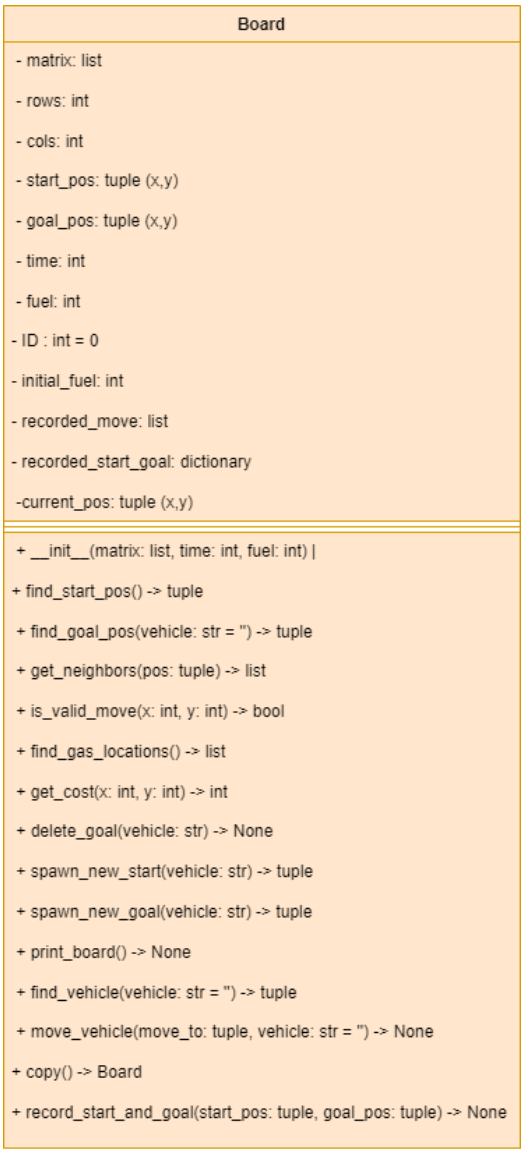
## 2.3 Luồng hoạt động




Hình 1: Luồn hoạt động của chương trình

## 2.4 Lớp Board

- Lớp `Board` đại diện cho một bảng điều khiển trong bài toán giúp việc sử dụng thuật toán ở các level dễ dàng hơn.
- Board chứa các thuộc tính như: ma trận, số dòng/cột, nhiên liệu khởi đầu, thời gian giới hạn, hàm tạo, điểm khởi đầu/kết thúc của xe S, các thuộc tính cần thiết cho level4 như ID, nhiên liệu, danh sách các đỉnh đã đi, danh sách các điểm bắt đầu và kết thúc, vị trí hiện tại.
- Lớp này bao gồm các phương thức để tìm vị trí bắt đầu và kết thúc, tìm các nút lân cận, kiểm tra di chuyển hợp lệ, tìm các trạm nhiên liệu, tính toán chi phí di chuyển, xóa và sinh mới vị trí bắt đầu và kết thúc, in bảng điều khiển, di chuyển xe, sao chép bảng, và ghi lại các vị trí bắt đầu và kết thúc.




Hình 2: Class Board

### 2.4.1 Cấu trúc dữ liệu

- **Ma trận (matrix):** Ma trận lưu trữ thông tin về các ô trên bảng.
- **Hàm tạo (Constructor):** Khởi tạo bảng với ma trận(matrix), thời gian(time), và nhiên liệu(fuel).

- **Vị trí bắt đầu(start_pos) và kết thúc(goal_pos):** Có dạng tuple (x,y), được tìm thấy bằng cách duyệt qua ma trận để xác định các ô có chứa 'S' và 'G'.
- **Các thuộc tính khác cho lv4:** `ID` (dùng để nhận biết số hiệu xe), `initial_fuel` (dùng để lưu lượng xăng ban đầu), `fuel` (dùng để lưu lượng xăng hiện tại), `recorded_move` (dùng để ghi nhận hết tất cả đường đi của các xe), `recorded_start_goal` (dùng để ghi nhận tất cả cặp điểm bắt đầu và kết thúc của các xe phụ nếu có random), `current_pos` (ghi nhận vị trí hiện tại của xe).

#### 2.4.2 Các phương thức trong lớp

- **Tìm vị trí bắt đầu và kết thúc:**
  - `find_start_pos(vehicle='')`: Tìm vị trí bắt đầu của xe.
  - `find_goal_pos(vehicle='')`: Tìm vị trí kết thúc của xe.
  - Biến vehicle có giá trị mặc định none thì nó sẽ tìm S và G. Nếu gán vehicle = "1"thì nó sẽ tìm xe 1 tương tụ như các xe khác.
- **Lấy các nút lân cận:**
  - `get_neighbors(pos)`: Lấy danh sách các ô lân cận của một ô cho trước theo thứ tự lên, xuống, trái, phải.
- **Kiểm tra di chuyển hợp lệ:**
  - `is_valid_move(x, y)`: Kiểm tra xem di chuyển đến ô (x, y) có hợp lệ hay không.
  - Ô đó hợp lệ nếu nằm trong kích thước ma trận và giá trị khác -1.
- **Tìm trạm nhiên liệu:**
  - `find_gas_locations()`: Tìm và trả về list các vị trí trạm nhiên liệu trong bảng.
- **Tính toán chi phí di chuyển:**
  - `get_cost(x, y)`: Tính toán chi phí di chuyển đến ô (x, y).
  - Thời gian di chuyển đến mọi ô cơ bản sẽ là 1 nếu ô đó là trạm dừng có giá trị là số hoặc là trạm xăng thì sẽ cộng thêm giá trị vào 1. Ví dụ di chuyển đến ô có giá trị F2 sẽ tốn 1 + 2(thời gian dừng đổ xăng) = 3 đơn vị thời gian.
- **Xóa và sinh mới vị trí bắt đầu và kết thúc:**
  - `delete_goal(vehicle)`: Xóa vị trí kết thúc của xe thay giá trị ô bằng 0.
  - `spawn_new_start(vehicle)`: Ghi nhận vị trí bắt đầu mới của xe là vị trí kết thúc cũ.
  - `spawn_new_goal(vehicle)`: Sinh mới vị trí kết thúc của xe.
- **In ma trận:**
  - `print_board()`: In ma trận chứa trong lớp Board.
- **Di chuyển xe:**
  - `move_vehicle(move_to, vehicle='')`: Di chuyển xe đến ô cho trước.
- **Sao chép bảng:**
  - `copy()`: Tạo một bản sao của bảng. Sử dụng cho chuyển trạng thái giữa các agents ở lv4
- **Ghi lại vị trí bắt đầu và kết thúc:**
  - `record_start_and_goal(start_pos=None, goal_pos=None)`: Ghi lại vị trí bắt đầu và kết thúc của xe.

## 2.5 Các hàm quan trọng

### 2.5.1 Hàm `reconstruct_path`

- **Mục đích:** hàm `reconstruct_path` được sử dụng để tái tạo lại đường đi từ điểm bắt đầu (`start`) đến điểm đích (`goal`) cho các thuật toán search sau khi đã hoàn thành nhiệm vụ. Đây là hàm trọng yếu được sử dụng xuyên suốt level1 tới level4.
- **Tham số:**

  - `came_from` (dict): Một từ điển trong đó các khóa là các điểm (nodes) và các giá trị là các điểm từ đó chúng được đến. Nó ánh xạ mỗi điểm đến điểm trước đó.
  - `start` (tuple): Điểm bắt đầu của đường đi.
  - `goal` (tuple): Điểm đích của đường đi.

- **Trả về:** Trả về một danh sách các điểm tạo thành đường đi từ điểm bắt đầu đến điểm đích. Nếu điểm đích không nằm trong từ điển `came_from`, hàm sẽ trả về `None`.

### 2.5.2 Hàm `heuristic`

- **Mục đích:** Hàm `heuristic` được sử dụng để tính toán khoảng cách ước lượng giữa hai điểm trong không gian hai chiều. Nó thường được sử dụng trong thuật toán A* để ước lượng chi phí từ một điểm hiện tại đến điểm đích. Hàm này được tính bằng công thức khoảng cách Manhattan.

$$\text{heuristic}(a, b) = |a_x - b_x| + |a_y - b_y|$$

- **Tham số:**

  - `a` (tuple): Tọa độ của điểm đầu tiên (thường là vị trí hiện tại).
  - `b` (tuple): Tọa độ của điểm thứ hai (thường là goal của Board).

- **Trả về:** khoảng cách Manhattan giữa hai điểm, được tính bằng tổng các giá trị tuyệt đối của sự chênh lệch tọa độ theo trục x và trục y.

# 3 Level 1

## 3.1 Các thuật toán được sử dụng

### 3.1.1 Thuật toán BFS

- Thuật toán Breadth-First Search (BFS) là một thuật toán tìm kiếm rộng được sử dụng phổ biến để tìm đường đi ngắn nhất trong các bài toán mê cung hoặc đồ thị không trọng số. Ở level 1, các trọng số giữa các ô là như nhau (đều là 1) nên chúng ta hoàn toàn có thể áp dụng nó để tìm ra được đường đi ngắn nhất.
- Hàng Đợi (Queue): BFS sử dụng một hàng đợi để lưu trữ các ô cần được khám phá. Hàng đợi này tuân theo nguyên tắc FIFO (First In, First Out), nghĩa là ô được thêm vào đầu tiên sẽ được xử lý trước.
- Các bước của thuật toán:

  - Tạo một hàng đợi (queue) để lưu các ô cần được kiểm tra. Ban đầu, hàng đợi này chỉ chứa điểm bắt đầu, đại diện cho vị trí bắt đầu của người giao hàng.
  - Tạo một hàng đợi (queue) để lưu các ô cần được kiểm tra. Ban đầu, hàng đợi này chỉ chứa điểm bắt đầu, đại diện cho vị trí bắt đầu của người giao hàng.
  - Tạo một tập hợp (visited set) để lưu các ô đã được kiểm tra qua nhằm tránh việc phải kiểm tra lại chúng.
  - Đặt ô bắt đầu vào hàng đợi và đánh dấu là đã xét.
  - Lấy ô đầu tiên ra khỏi hàng đợi.

- Nếu ô đó là điểm đích, kết thúc và trả về đường đi.
- Kiểm tra các ô láng giềng của ô đang xét. Nếu ô láng giềng đó chưa được xét và không phải là vật cản: Đánh dấu ô đó là đã đi qua và thêm vào hàng đợi
- Ghi lại ô hiện tại là ô cha của ô láng giềng để có thể xây dựng đường đi sau khi tìm thấy điểm đích.
- Nếu tìm thấy điểm đích, xây dựng đường đi từ điểm đích quay ngược lại điểm bắt đầu bằng cách theo dõi các ô cha.
- Nếu hàng đợi rỗng mà không tìm thấy điểm đích, báo rằng không có đường đi.

### 3.1.2 Thuật toán GBFS

- Greedy Best-First Search (GBFS) là một thuật toán tìm kiếm dựa trên một hàm heuristic được xây dựng sẵn, sử dụng để tìm đường đi trong các bài toán không gian trạng thái như mê cung. Thuật toán này luôn chọn mở rộng nút có giá trị heuristic thấp nhất, cố gắng tiếp cận đích nhanh nhất có thể.
- Nguyên lý hoạt động:

- Heuristic: ta sẽ sử dụng một hàm heuristic để ước lượng chi phí từ một ô cho đến đích. Ở đây, chúng ta sẽ sử dụng khoảng cách Manhattan để tính tổng khoảng cách theo trục x và y giữa hai điểm.
- Hàng đợi ưu Tiên: GBFS duy trì một hàng đợi ưu tiên (priority queue) để lưu trữ các ô cần mở rộng, sắp xếp chúng dựa trên giá trị heuristic. - Các bước của thuật toán
- Đặt ô bắt đầu vào trong hàng đợi ưu tiên với giá trị heuristic của nó.
- Đặt tất cả các ô khác là chưa được xét (unvisited).
- Lấy ô có giá trị heuristic thấp nhất từ hàng đợi ưu tiên (nút hiện tại). Nếu ô đó là đích, trả về đường đi từ điểm bắt đầu đến đích.
- Đánh dấu ô hiện tại là đã được xét (visited).
- Thêm các ô láng giềng của ô hiện tại vào hàng đợi ưu tiên nếu chúng chưa được xét và cập nhật giá trị heuristic của chúng.
- Nếu hàng đợi ưu tiên rỗng và không tìm thấy đích, kết luận rằng không có đường đi từ điểm bắt đầu đến đích.

### 3.1.3 Thuật toán DFS

- Depth first search (DFS) là thuật toán tìm kiếm theo chiều sâu, nghĩa là nó duyệt qua mỗi nhánh của đồ thị trước khi quay lại và thử các nhánh khác. DFS không đảm bảo tìm được đường đi ngắn nhất nhưng thường sẽ tiết kiệm bộ nhớ hơn BFS (vì DFS dùng stack chỉ để lưu đường đi hiện tại và nhánh chờ xử lý).
- Cấu trúc dữ liệu

- Ngăn xếp (Stack): Để lưu trữ các nút đang chờ để được khám phá tiếp.
- Tập hợp (Set): Để đánh dấu các nút đã được duyệt nhằm tránh việc duyệt lại các nút này và gây ra vòng lặp.

- Các bước của thuật toán:

- Thuật toán bắt đầu từ hàm DFS: Lấy vị trí bắt đầu (start) và vị trí mục tiêu (goal) từ bản đồ. Nếu không có vị trí bắt đầu hoặc đích, thuật toán trả về None.
- Khởi tạo một tập hợp visited để lưu trữ các nút đã duyệt và một danh sách path để lưu trữ đường đi từ vị trí bắt đầu đến mục tiêu.

- Lập trình hàm dfsSearch nhận một nút làm đầu vào. Nếu nút hiện tại là đích, thêm nó vào danh sách path và trả về True. Nếu không, đánh dấu nút hiện tại là đã thăm bằng cách thêm nó vào tập hợp visited. Sau đó, lấy các nút hàng xóm của nút hiện tại và duyệt qua từng nút hàng xóm. Nếu nút hàng xóm chưa được duyệt, gọi đệ quy dfsSearch. Nếu tìm thấy đường đi đến đích qua nút hàng xóm, thêm nút hiện tại vào danh sách path và trả về True. Nếu không tìm thấy đường đi từ nút hiện tại, trả về False.

- Cuối cùng, hàm DFS sẽ gọi dfsSearch để bắt đầu đi từ nút xuất phát. Nếu tìm thấy đường đi, đảo ngược danh sách path để có đường đi từ bắt đầu đến đích và trả về path. Nếu không tìm thấy đường đi, trả về None.

#### 3.1.4 Thuật toán IDS và DLS

- Depth-Limited Search (DLS) là một thuật toán tìm kiếm được phát triển từ Depth-First Search (DFS), ta quy định một giới hạn độ sâu (depth limit) để ngăn không cho thuật toán đi quá sâu trong cây tìm kiếm. Điều này giúp DLS tránh được vấn đề tràn bộ nhớ trong trường hợp các cây tìm kiếm quá lớn và đảm bảo rằng thuật toán không chạy mãi mãi trong các trường hợp không có đường đi đến mục tiêu.
-Iterative Deepening Search (IDS) là một thuật toán tìm kiếm nâng cấp của DFS và DLS. IDS thực hiện nhiều lần tìm kiếm theo chiều sâu với độ sâu ngày càng tăng, bắt đầu từ độ sâu 0 và tăng dần cho đến khi tìm thấy đích.
- Trong đồ án này, ta sẽ thực hiện hàm IDS dựa trên nền của DLS.
- Cấu trúc dữ liệu: Tương tự DFS, ta dùng Stack và Set.
- Các bước của thuật toán DLS:

- Khởi tạo một tập hợp visited để lưu trữ các nút đã duyệt.

- Gọi hàm dlssearch từ vị trí bắt đầu. Trong hàm dlssearch: Kiểm tra xem độ sâu hiện tại có vượt quá depthlimit không. Nếu có, trả về None. Hoặc không, kiểm tra xem nút hiện tại có phải là đích không. Nếu có, trả về đường đi từ nút bắt đầu đến nút đích. Hoặc không, đánh dấu nút hiện tại là đã duyệt. Lấy các nút hàng xóm của nút hiện tại và duyệt qua từng nút hàng xóm. Nếu nút hàng xóm chưa được duyệt, gọi đệ quy dlssearch trên nút hàng xóm với độ sâu tăng thêm 1. Nếu tìm thấy đường đi, trả về kết quả.

- Nếu không tìm thấy đường đi tại độ sâu hiện tại, trả về None.

- Các bước của thuật toán IDS:

- Bắt đầu với một biến depthlimit được khởi tạo bằng 0.

- Trong mỗi vòng lặp, gọi hàm DLS với depthlimit hiện tại để tìm kiếm đường đi đến mục tiêu.

- Nếu tìm thấy đường đi, trả về kết quả.

- Nếu không, tăng depthlimit và lặp lại cho đến khi tìm thấy mục tiêu.

- Tăng đến 50 nếu không tìm ra được đich thì sẽ return None, thoát vòng lặp

#### 3.1.5 Thuật toán Uniform-Cost Search (UCS)

- Uniform-Cost Search (UCS) là thuật toán tìm kiếm theo chi phí đồng đều, nghĩa là nó duyệt qua các nút theo thứ tự tăng dần của chi phí từ vị trí bắt đầu. UCS đảm bảo tìm được đường đi ngắn nhất từ vị trí bắt đầu đến vị trí đích nếu chi phí cho mỗi bước đi là không âm.
- **Cấu trúc dữ liệu:**

- Hàng đợi ưu tiên (Priority Queue): Để lưu trữ các nút đang chờ được khám phá, được sắp xếp theo chi phí.

- Từ điển (Dictionary): Để lưu trữ chi phí nhỏ nhất đến từng nút và nút cha của chúng để có thể tái tạo lại đường đi.

- **Các bước của thuật toán UCS:**

- Thuật toán bắt đầu từ hàm UCS: Lấy vị trí bắt đầu (start) và vị trí mục tiêu (goal) từ bản đồ. Nếu không có vị trí bắt đầu hoặc đích, thuật toán trả về None.
- Khởi tạo một hàng đợi ưu tiên `frontier` với phần tử đầu tiên là vị trí bắt đầu và chi phí là 0. Khởi tạo từ điển `came_from` để lưu trữ các nút cha và từ điển `cost_so_far` để lưu trữ chi phí nhỏ nhất đến từng nút.
- Trong khi hàng đợi `frontier` không trống, lấy nút có chi phí nhỏ nhất ra khỏi hàng đợi.
- Nếu nút hiện tại là mục tiêu, tái tạo đường đi từ `came_from` và trả về đường đi.
- Nếu không, duyệt qua các nút hàng xóm của nút hiện tại. Nếu chi phí đến nút hàng xóm nhỏ hơn chi phí đã lưu trong `cost_so_far` hoặc nút hàng xóm chưa được duyệt, cập nhật chi phí và nút cha, sau đó thêm nút hàng xóm vào hàng đợi `frontier`.

#### 3.1.6 Thuật toán A* Search

- A* Search là thuật toán tìm kiếm theo chi phí thấp nhất với hàm đánh giá được kết hợp giữa chi phí từ nút bắt đầu đến nút hiện tại và ước lượng chi phí từ nút hiện tại đến đích. A* đảm bảo tìm được đường đi ngắn nhất từ vị trí bắt đầu đến vị trí đích nếu hàm heuristic không đánh giá quá cao chi phí thực tế.
- **Cấu trúc dữ liệu:**
  - **Hàng đợi ưu tiên (Priority Queue):** Để lưu trữ các nút đang chờ được khám phá, được sắp xếp theo hàm đánh giá.
  - **Từ điển (Dictionary):** Để lưu trữ chi phí nhỏ nhất đến từng nút và nút cha của chúng để có thể tái tạo lại đường đi.
- Trong A* Search, hàm đánh giá được tính toán bằng công thức:

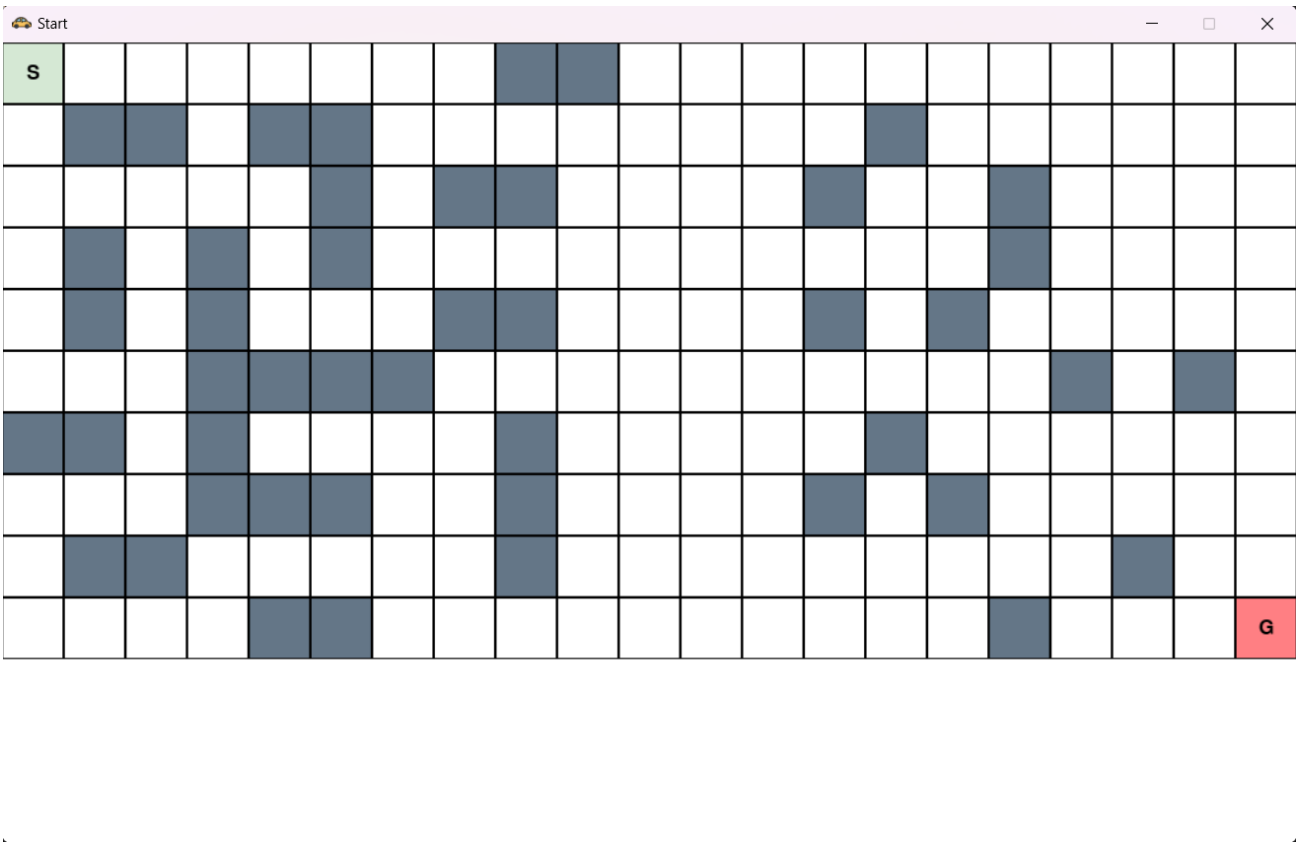
$$f(n) = g(n) + h(n)$$

  - $g(n)$ là chi phí thực tế từ nút bắt đầu đến nút hiện tại.
  - $h(n)$ là chi phí ước lượng từ nút hiện tại đến nút đích, được tính dựa trên hàm heuristic.

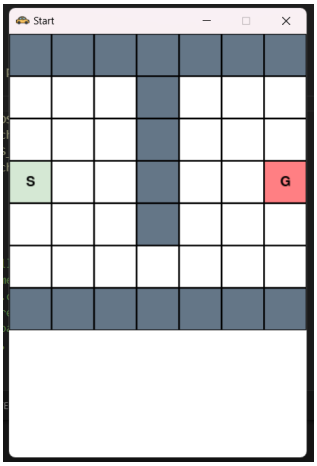
- **Các bước của thuật toán A*:**
  - Thuật toán bắt đầu từ hàm A_search: Lấy vị trí bắt đầu (start) và vị trí mục tiêu (goal) từ bản đồ. Nếu không có vị trí bắt đầu hoặc đích, thuật toán trả về None.
  - Khởi tạo một hàng đợi ưu tiên `frontier` với phần tử đầu tiên là vị trí bắt đầu và chi phí là 0. Khởi tạo từ điển `came_from` để lưu trữ các nút cha và từ điển `cost_so_far` để lưu trữ chi phí nhỏ nhất đến từng nút.
  - Trong khi hàng đợi `frontier` không trống, lấy nút có hàm đánh giá nhỏ nhất ra khỏi hàng đợi.
  - Nếu nút hiện tại là mục tiêu, tái tạo đường đi từ `came_from` và trả về đường đi.
  - Nếu không, duyệt qua các nút hàng xóm của nút hiện tại. Nếu chi phí đến nút hàng xóm nhỏ hơn chi phí đã lưu trong `cost_so_far` hoặc nút hàng xóm chưa được duyệt, cập nhật chi phí và nút cha, sau đó thêm nút hàng xóm vào hàng đợi `frontier` với hàm đánh giá là tổng chi phí và ước lượng chi phí đến đích(heuristic).

### 3.2 Các test case

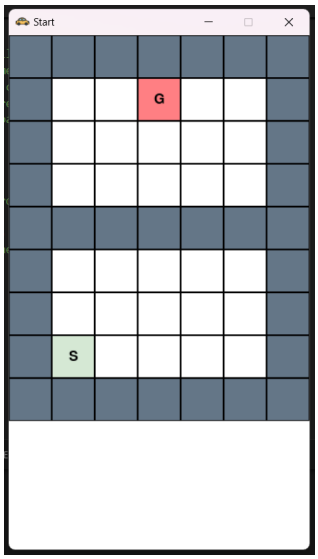
- Ta có lần lượt các map cho level 1 như sau: (gồm 5 map)



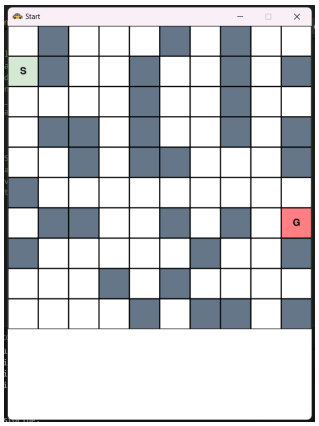

Hình 3: Map 1



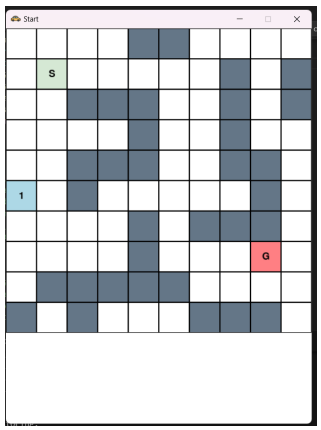

Hình 4: Map 2




Hình 5: Map 3

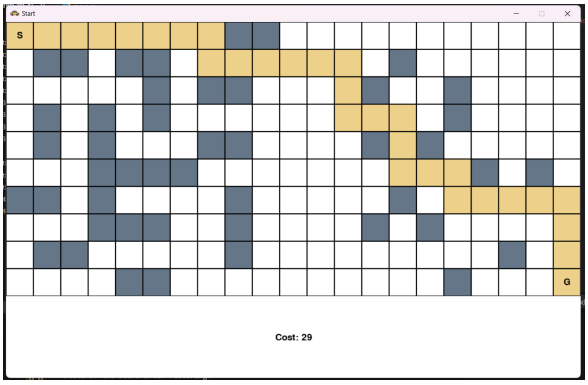



Hình 6: Map 4

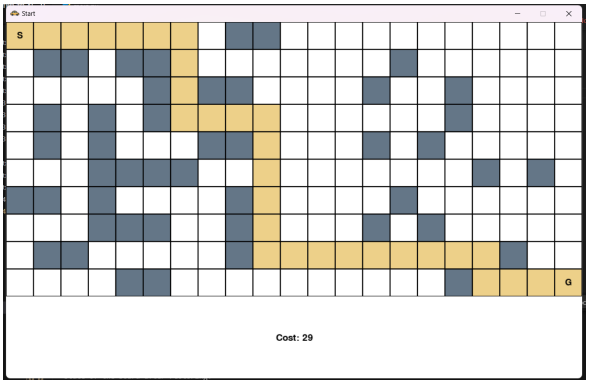



Hình 7: Map 5

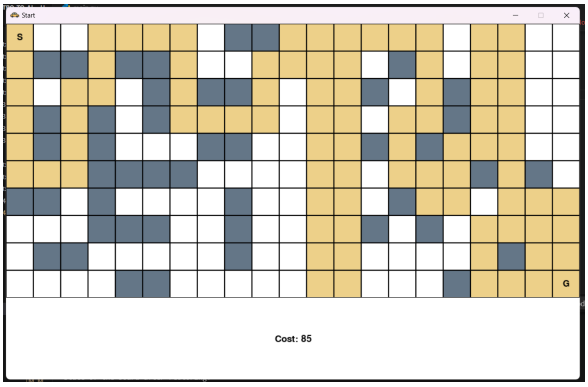
### 3.2.1 Kết quả cho map 1



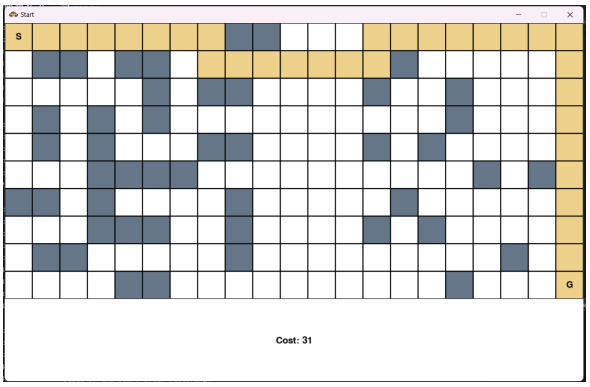

Hình 8: Map 1 - A*



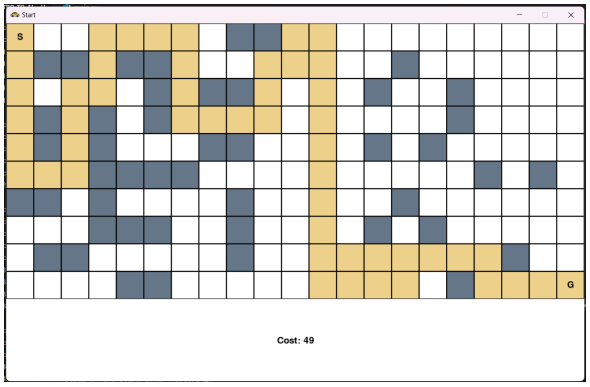

Hình 9: Map 1 - BFS




Hình 10: Map 1 - DFS

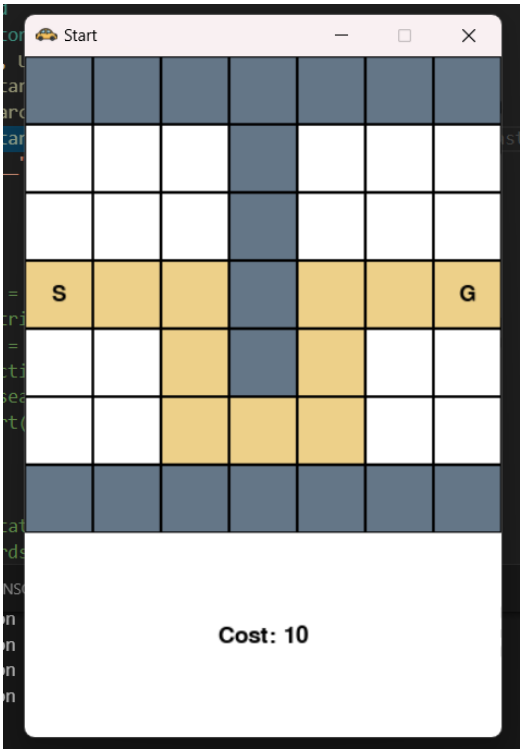



Hình 11: Map 1 - GBFS

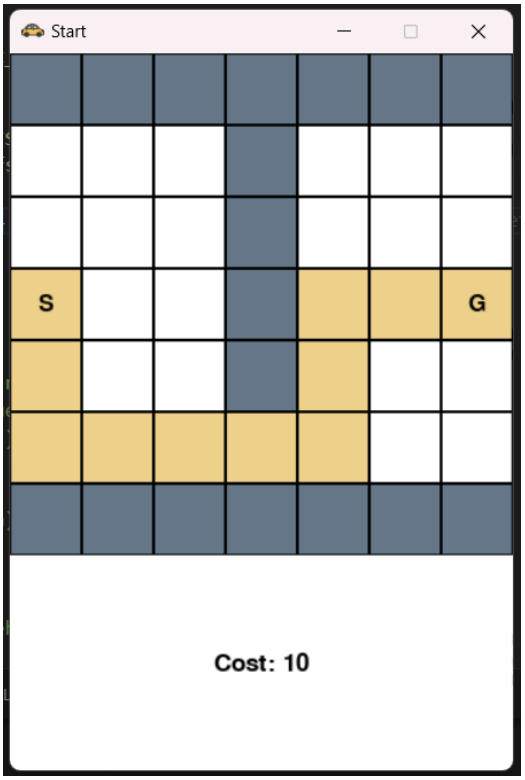



Hình 12: Map 1 - IDS

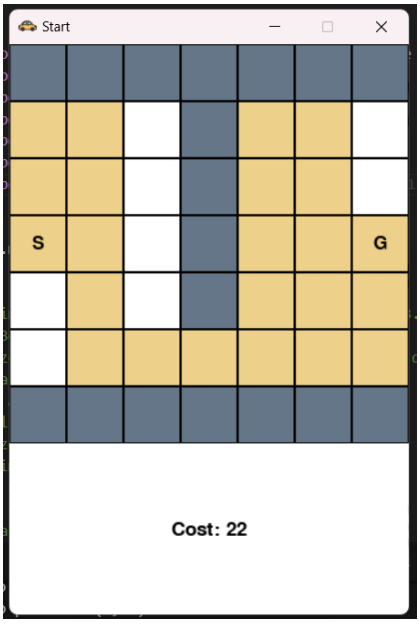
### 3.2.2 Kết quả cho map 2



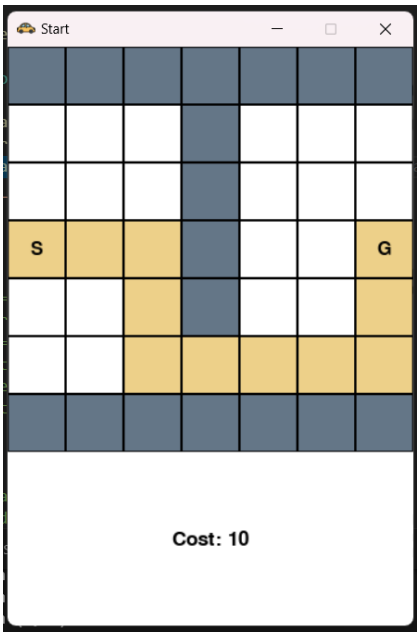

Hình 13: Map 2 - A*



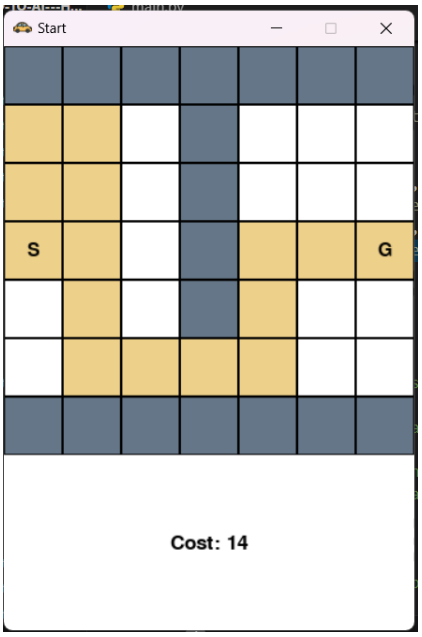

Hình 14: Map 2 - BFS




Hình 15: Map 2 - DFS

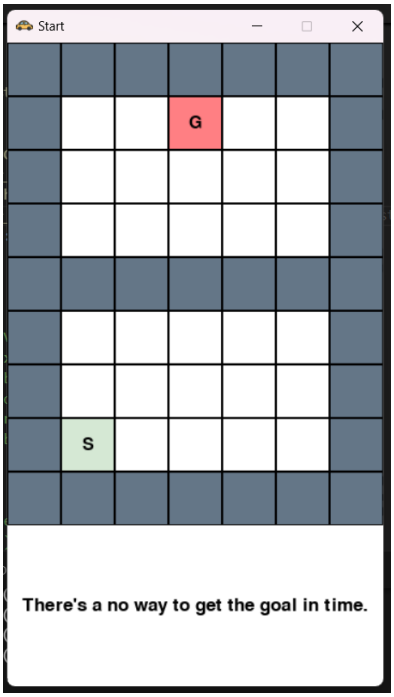



Hình 16: Map 2 - GBFS

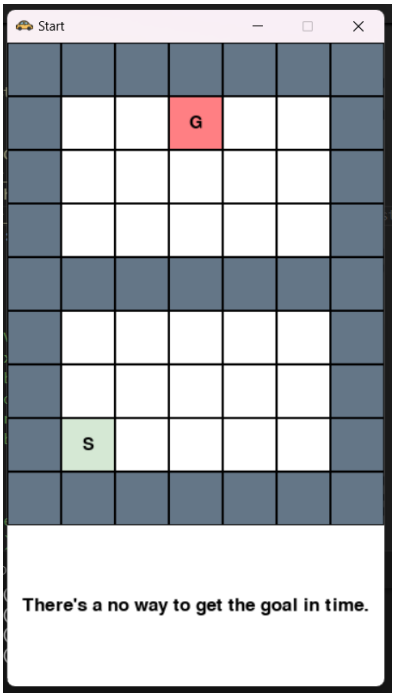



Hình 17: Map 2 - IDS

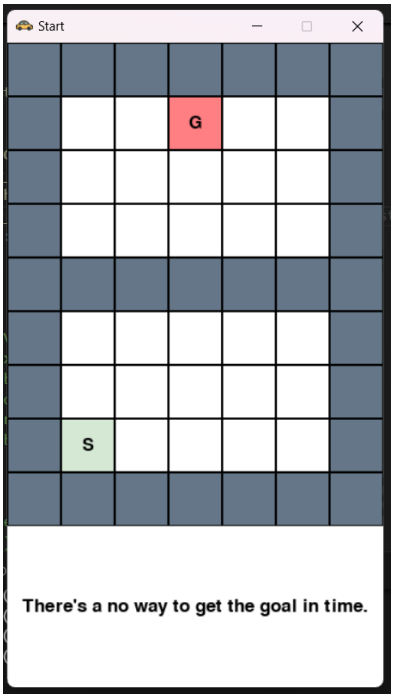
### 3.2.3 Kết quả cho map 3



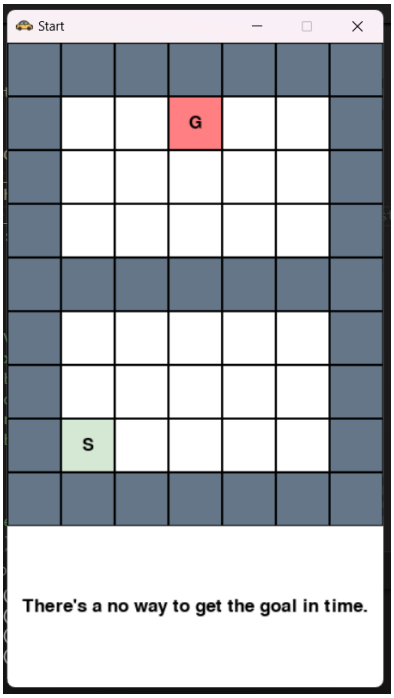

Hình 18: Map 3 - A*



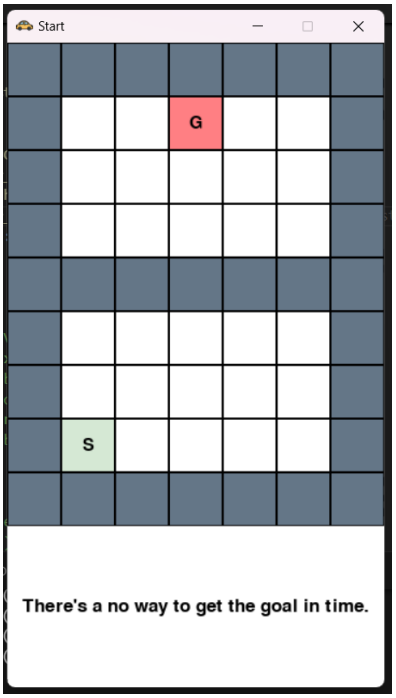

Hình 19: Map 3 - BFS




Hình 20: Map 3 - DFS

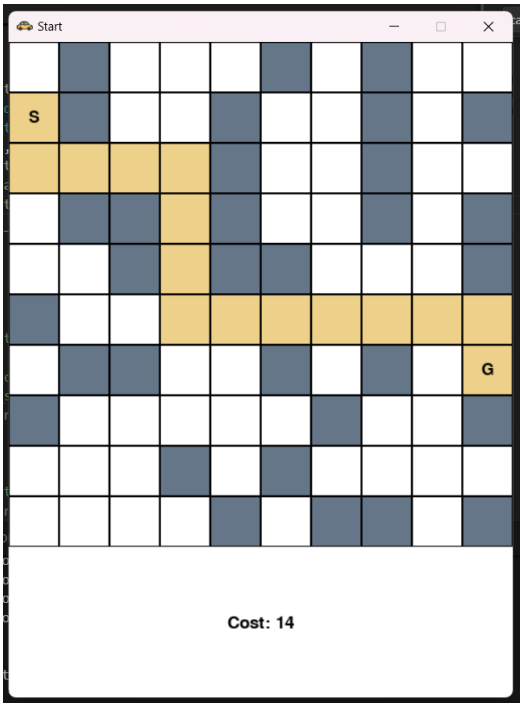



Hình 21: Map 3 - GBFS

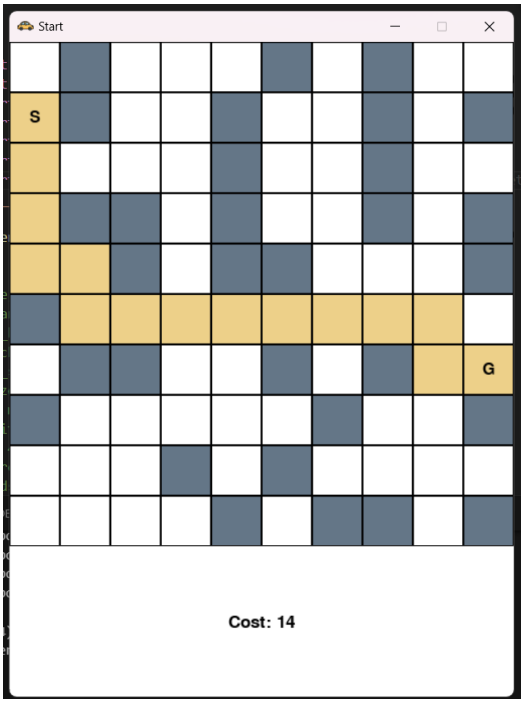



Hình 22: Map 3 - IDS

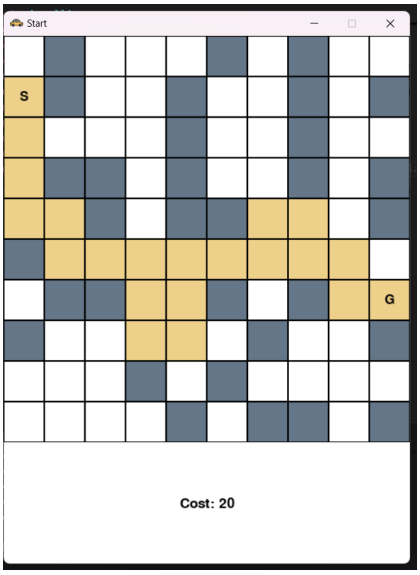
### 3.2.4 Kết quả cho map 4



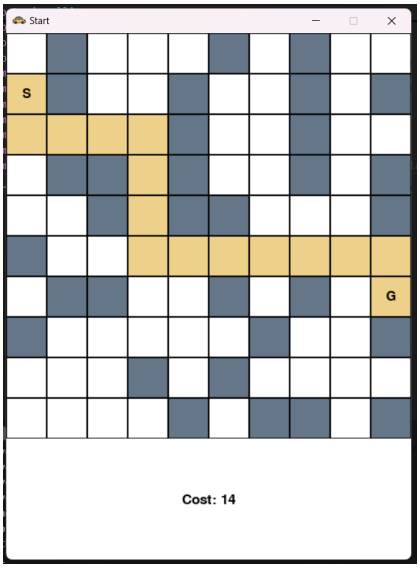

Hình 23: Map 4 - A*



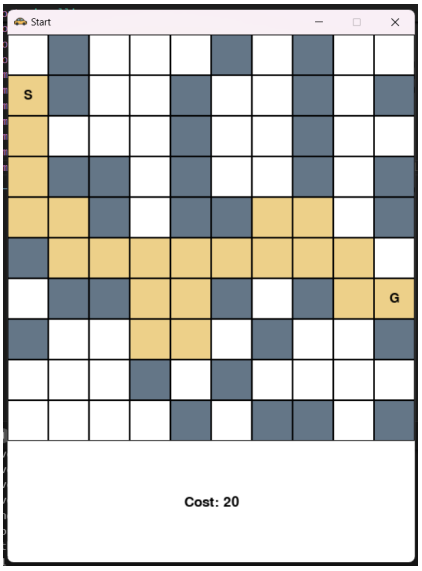

Hình 24: Map 4 - BFS




Hình 25: Map 4 - DFS

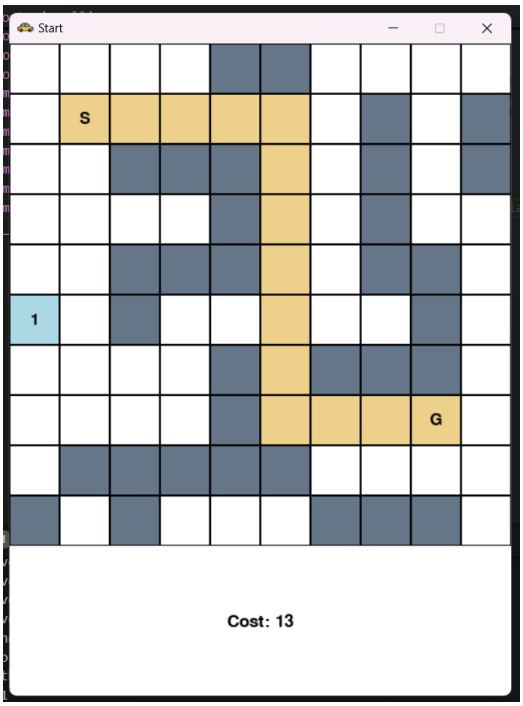



Hình 26: Map 4 - GBFS

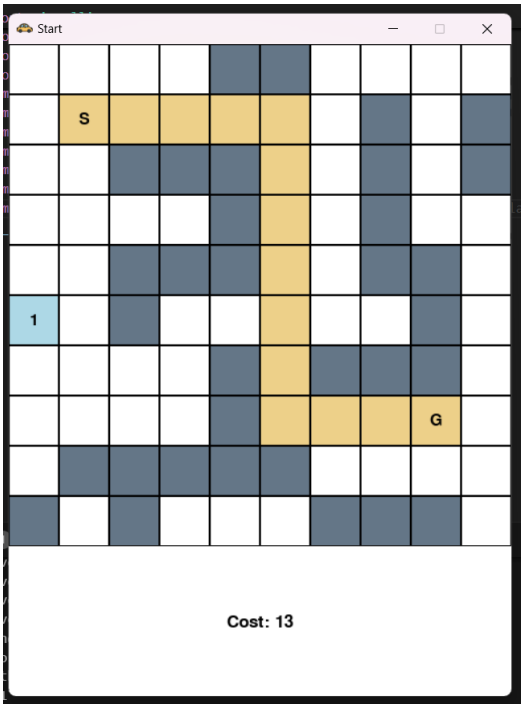



Hình 27: Map 4 - IDS

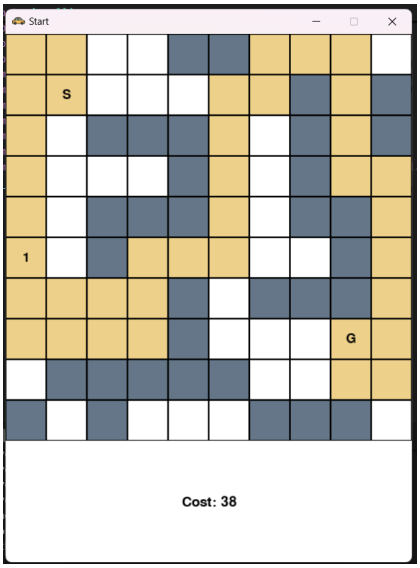
### 3.2.5 Kết quả cho map 5



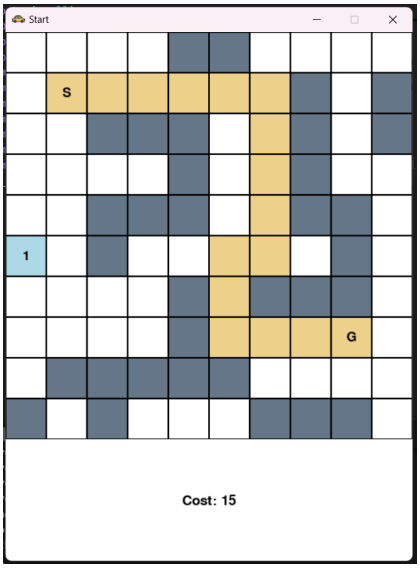

Hình 28: Map 5 - A*



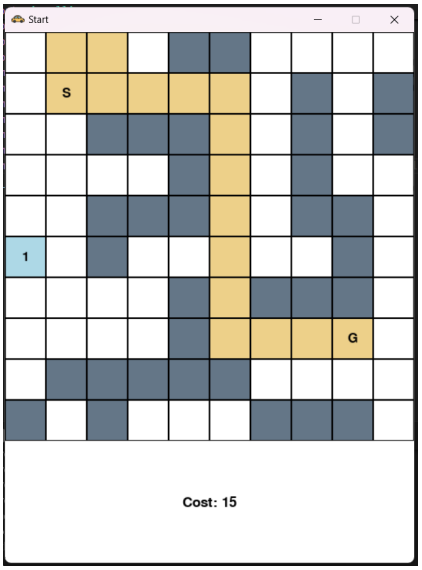

Hình 29: Map 5 - BFS




Hình 30: Map 5 - DFS




Hình 31: Map 5 - GBFS




Hình 32: Map 5 - IDS

## 3.3 Nhận xét

### 3.3.1 Map 1

- Đây là bản đồ có điểm S và G cách nhau xa nhất, nằm ở 2 cực của bản đồ. Vì vậy các thuật toán tìm đường đi với chi phí cao. Đặc biệt là DFS, khi nó duyệt gần hết bản đồ, đúng bản chất của thuật toán này là duyệt theo chiều sâu.
- Đối với đồ thị không trọng số như ma trận lv1, thuật toán BFS và A* là thuật toán chạy nhanh và ít tốn chi phí nhất
- Thuật toán IDS ở bản đồ này là sự cải tiến của DFS, duyệt sâu khi tăng độ sâu, nên nó tốt hơn DFS với chi phí là 49. Tuy nhiên, nó vẫn tốn chi phí hơn A* và BFS rất nhiều - GBFS không quan tâm đến tổng chi phí từ điểm bắt đầu S đến điểm đang xét, mà chỉ tập trung vào việc tiến gần điểm đích G nhanh nhất có thể. Trong bản đồ này thuật toán cứ liên tục di chuyển qua phải rồi xuống dưới sao cho khoảng cách Mattahan gần nhất với điểm G nên đã bỏ qua đường đi ít tốn hơn (tiến tới ô mà xung quanh nó là vật cản, nhưng miễn là gần hơn với đích thì thuật toán sẽ chọn ô đó), vì vậy cost lớn hơn A* một chút: 31.

### 3.3.2 Map 2

- Đây là bản đồ nhỏ hơn bản đồ 1, điểm S và G cách nhau 6 ô theo đường ngang và giữa đường ngang đó có 1 ô cản. Các thuật toán đều tìm với chi phí thấp.
- BFS và A* và GBFS tìm đường ngắn nhất với cost là 10. Ở bản đồ này địa hình khá thuận lợi, khi đi theo hướng ưu tiên khoảng cách Mattahan gần nhất với điểm G thì hướng đi đó không có sự khác biệt (không có vật cản), nên A* và GBFS đều tìm đường đi bằng nhau.
- Với DFS và IDS, nó luôn quét gần hết cả bản đồ nên chi phí tìm kiếm cao (lần lượt 22 và 14).

### 3.3.3 Map 3

- Đây là một trường hợp đặc biệt khi không tồn tại một đường đi nào khả dụng giữa điểm bắt đầu và điểm kết. Vì vậy mà tất cả các kết quả đều giống nhau khi không thể tìm thấy con đường khả dụng.

### 3.3.4 Map 4

- Bản đồ có một số lượng vật cản đáng kể, tạo ra nhiều đường cụt và các khu vực khó tiếp cận. Các vật cản được phân bố rải rác và không theo một quy luật cụ thể nào, tạo ra nhiều thách thức trong việc tìm đường đi tối ưu.
- A* và BFS, GBFS đã tìm được đường đi tối ưu trong điều kiện này, khẳng định sự hiệu quả của chúng trong bản đồ với nhiều vật cản. Lưu ý : GBFS Dù có thể nhanh nhưng không đảm bảo tối ưu nếu không có heuristic tốt.
- DFS và IDS: Đã tìm được đường đi nhưng với chi phí cao hơn, cho thấy các thuật toán này không phù hợp với bản đồ có nhiều vật cản ngẫu nhiên.

### 3.3.5 Map 5

- Vật cản trong bản đồ này cũng được phân bố rải rác, tạo ra nhiều ngõ cụt và các khu vực khó tiếp cận. Tuy nhiên, số lượng vật cản có vẻ ít hơn và các lối đi thông thoáng hơn so với bản đồ trước.
- A* và BFS: Tìm được đường đi tối ưu với chi phí 13.
- DFS: Không hiệu quả với chi phí cao nhất là 38.
- GBFS và IDS: Hiệu quả hơn DFS nhưng vẫn không tối ưu với chi phí 15.

# 4 Level 2

## 4.1 Thuật toán

- Chúng ta sẽ sử dụng lại thuật toán A* search cho level. Cơ bản thuật toán có phần giống với thuật toán A* search ở level 1, vẫn sử dụng heuristic function h(n) cũ, xác định khoảng cách từ một ô tới điểm đích bằng công thức Manhattan và sử dụng công thức tính chi phí f(n)=g(n)+h(n). Nhưng có một điểm khác là ở level 2, chúng ta sẽ có thêm các Toll Booth và giới hạn thời gian. Do khi đi qua mỗi ô đều sẽ mất 1 phút nên ta sẽ sử dụng thời gian cho đơn vị tính chi phí. Còn các ô có Toll Booth, chúng ta sẽ cộng thêm số phút ta phải đợi ở Toll Booth đó vào chi phí. Điều đó có nghĩa là giá trị f(n) không phải lúc nào cũng bằng 1 nữa. Do thuật toán sẽ luôn trả về đường đi có chi phí ít nhất đồng nghĩa với thời gian đi ít nhất nên ta sẽ dùng kết quả đó để so sánh với thời gian cho phép. Nếu thời gian đi lớn hơn thời gian quy định, ta sẽ hủy kết quả và báo không tìm được đường đi.
**Cấu trúc dữ liệu:**

- **Hàng đợi ưu tiên (Priority Queue):** Để lưu trữ các nút đang chờ được khám phá, được sắp xếp theo hàm đánh giá.
- **Từ điển (Dictionary):** Để lưu trữ chi phí nhỏ nhất đến từng nút và nút cha của chúng để có thể tái tạo lại đường đi.

- Trong A* Search, hàm đánh giá được tính toán bằng công thức:

$$f(n) = g(n) + h(n)$$

- $g(n)$ là chi phí (thời gian) thực tế từ nút bắt đầu đến nút hiện tại.
- $h(n)$ là chi phí (thời gian) ước lượng từ nút hiện tại đến nút đích, được tính dựa trên hàm heuristic.

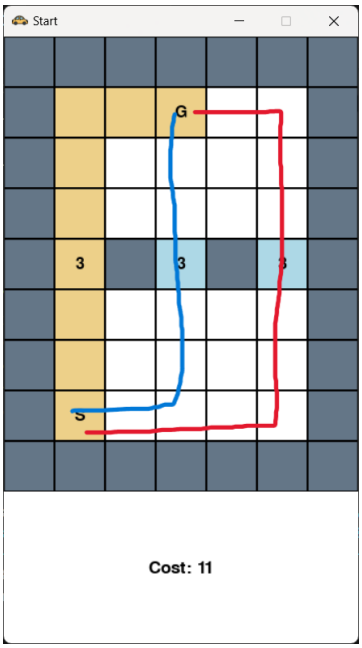
**Các bước của thuật toán A*:**

- Thuật toán bắt đầu từ hàm A_search: Lấy vị trí bắt đầu (start) và vị trí mục tiêu (goal) từ bản đồ. Nếu không có vị trí bắt đầu hoặc đích, thuật toán trả về None.
- Khởi tạo một hàng đợi ưu tiên `frontier` với phần tử đầu tiên là vị trí bắt đầu và chi phí là 0. Khởi tạo danh sách `came_from` để lưu trữ các ô cha và danh sách `cost_so_far` để lưu trữ chi phí nhỏ nhất đến từng ô.
- Trong khi hàng đợi `frontier` không trống, lấy ô có hàm đánh giá nhỏ nhất ra khỏi hàng đợi.
- Nếu ô hiện tại là mục tiêu, tái tạo đường đi từ `came_from` và trả về đường đi.
- Nếu không, duyệt qua các ô hàng xóm của nút hiện tại. Nếu chi phí đến ô hàng xóm nhỏ hơn chi phí đã lưu trong `cost_so_far` hoặc ô hàng xóm chưa được duyệt, cập nhật chi phí và nút cha, sau đó thêm nút hàng xóm vào hàng đợi `frontier` với hàm đánh giá là tổng chi phí và ước lượng chi phí đến đích(heuristic).
- Kiểm tra xem `cost_so_far` của đường đi hiện tại có vượt quá thời gian yêu cầu hay không, nếu có thì không có đường đi phù hợp, trả về None

## 4.2 Các test case

### 4.2.1 Test case 1

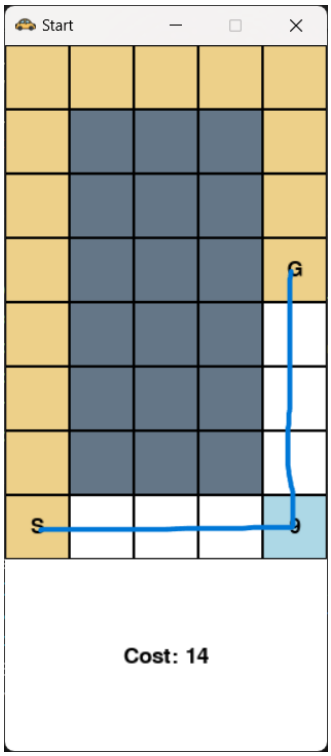
- Map có kích thước 9x7, giới hạn thời gian là 15 phút. Có tổng cộng 3 trạm thu phí với thời gian chờ bằng nhau. Để đến được đích bắt buộc ta phải đi qua ít nhất một trạm thu phí. Khi chọn đi qua 2 trạm thu phí đầu, thời gian và số ô đi qua sẽ bằng nhau. Việc quyết định đi qua trạm thu phí nào phụ thuộc vào ô đầu tiên ta đi qua sau khi xuất phát. Ở trường hợp này, hệ thống sẽ chọn đường qua trạm thu phí số 1 vì thứ tự ưu tiên luôn là đi lên ô phía trên trước.




Hình 33: Map của test case 1

### 4.2.2 Test case 2

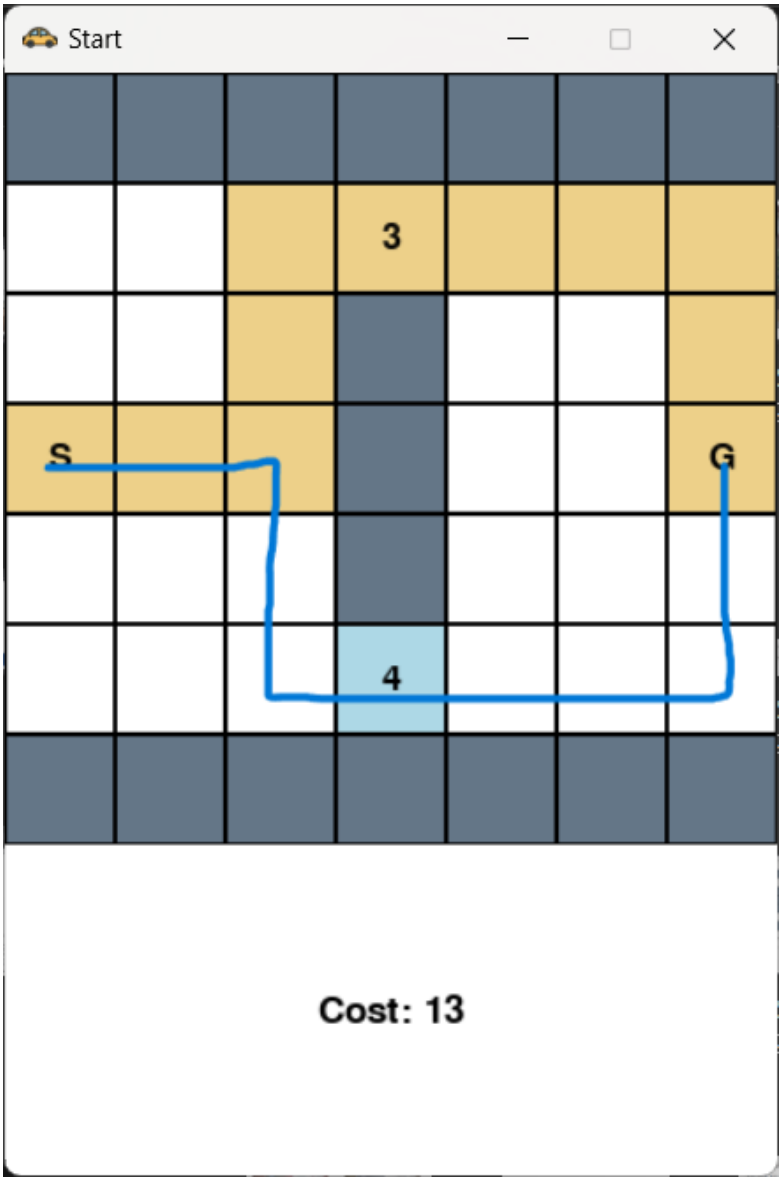
- Map có kích thước 8x5, giới hạn thời gian là 20 phút. Hệ thống sẽ chọn đường lên trên dù đi qua nhiều ô hơn vì nếu qua trạm thu phí sẽ phải 9 phút dẫn đến di chuyển lâu hơn.




Hình 34: Map của test case 2

#### 4.2.3 Test case 3

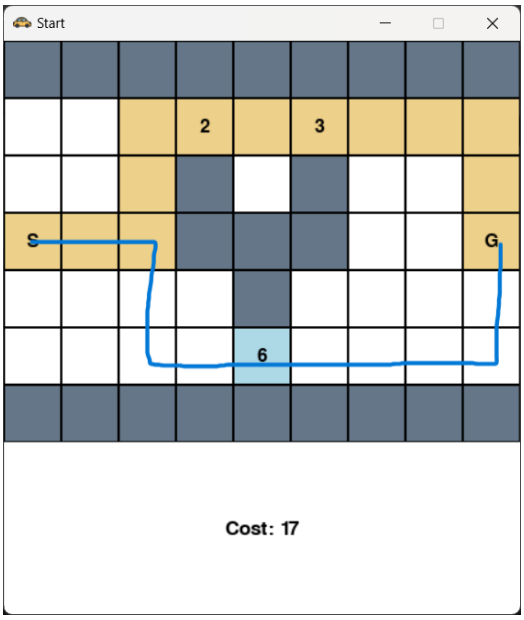
- Map có kích thước 7x7, giới hạn thời gian 15 phút. Có tổng cộng 2 trạm thu phí và muốn đi đến đích sẽ phải qua ít nhất 1 trạm. Đường qua 2 trạm thu phí đều có số ô bằng nhau nhưng hệ thống sẽ chọn đi qua trạm thu phí ở trên vì chỉ phải chờ có 3 phút.




Hình 35: Map của test case 3

#### 4.2.4 Test case 4

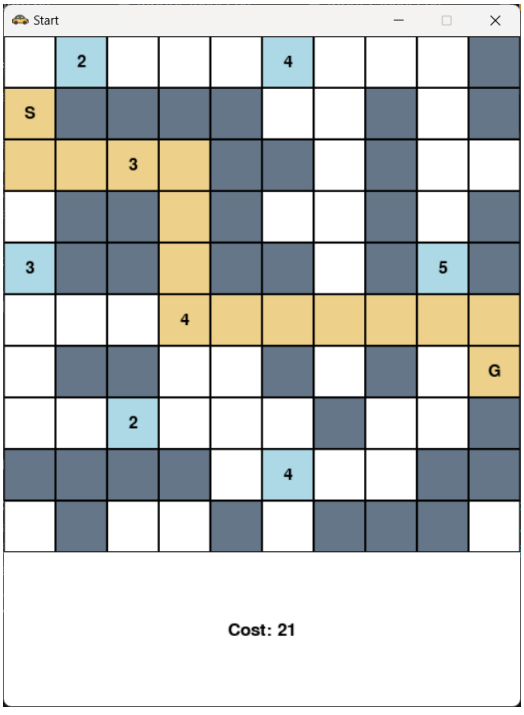
- Map có kích thước 7x9, giới hạn thời gian 20 phút. Có tổng cộng 3 trạm thu phí, trong đó có 2 trạm thu phí nằm trên cùng một con đường đến đích. Có 2 con đường đi đến đích và đều có số ô bằng nhau, khác ở việc đường đi phía trên bắt buộc phải qua 2 trạm thu phí. Dù vậy hệ thống vẫn sẽ chọn đường phía trên vì tổng thời gian chờ ở hai trạm thu phí trên vẫn bé hơn thời gian chờ ở trạm thu phí dưới ($2 + 3 < 6$)




Hình 36: Map của test case 4

### 4.2.5 Test case 5

- Map có kích thước 10x10, giới hạn thời gian 21 phút. Map có nhiều đường đi được tới đích nhưng chỉ có một con đường thỏa thời gian cho phép.




Hình 37: Map của test case 5

# 5 Level 3

## 5.1 Ý tưởng

- **Thuật toán A* (A-star)** là một thuật toán tìm kiếm trên đồ thị được sử dụng rộng rãi để tìm đường đi ngắn nhất từ một điểm bắt đầu đến một điểm đích. Thuật toán này kết hợp giữa chi phí đi từ điểm bắt đầu đến điểm hiện tại và ước lượng chi phí từ điểm hiện tại đến đích (thông qua hàm heuristic).

- Ở level 3 với sự xuất hiện của fuel theo như đề bài thì chúng em phải thay đổi so với các hàm A* ở level 1,2 để xử lý trường hợp không đủ xăng.
- Với ý tưởng là đầu tiên em sẽ cho nó chạy A* như bình thường tìm đường ngắn nhất từ điểm bắt đầu đến đích để tìm đường ngắn nhất. Nếu đường này vượt time limit hoặc không đủ xăng thì sẽ không trả về.
- Nếu như thực hiện tìm đường chạy thẳng từ điểm bắt đầu đến đích không thành công thì sẽ bắt đầu tìm theo kiểu khác đó chính là tìm đến các trạm xăng gần nhất để tiếp nhiên liệu và tìm đường ngắn nhất từ trạm xăng đó đến đích. Trường hợp vẫn không đủ xăng thì vẫn sẽ tìm đến trạm xăng gần và khả thi nhất để tiếp tục tiếp nhiên liệu cho đến khi đến được đích.

## 5.2 Cấu trúc dữ liệu:

- **Priority Queue (Heapq):** Được sử dụng để lưu trữ và truy xuất các điểm nút với chi phí ưu tiên (chi phí thấp nhất). Sử dụng heapq thay vì stack hay queue bởi vì heapq cho phép truy xuất điểm nút với chi phí thấp nhất một cách hiệu quả (O(log n)), trong khi stack (LIFO) và queue (FIFO) không hỗ trợ sắp xếp theo chi phí ưu tiên.
- **Dictionary (came_from, cost_so_far, fuel_at_node):** Được sử dụng để lưu trữ thông tin về các điểm nút đã được khám phá, bao gồm điểm nút trước đó, chi phí tích lũy và nhiên liệu còn lại.
- **Frontier:** Biến `frontier` là một priority queue (hàng đợi ưu tiên) lưu trữ các điểm nút cần khám phá tiếp theo cùng với chi phí ước tính từ điểm bắt đầu đến điểm đích. Mỗi phần tử trong `frontier` bao gồm: (chi phí ước tính, vị trí hiện tại, nhiên liệu còn lại).

## 5.3 Thuật toán

**Hàm a_star_search(board, start, goal, initial_fuel):** Đây là hàm search phụ bổ trợ cho level 3.

- Khởi tạo `frontier` (hàng đợi ưu tiên) với chi phí bằng 0 tại điểm bắt đầu.
- Khởi tạo các dictionary để lưu trữ điểm nút trước đó (came_from), chi phí tích lũy (cost_so_far) và nhiên liệu còn lại tại mỗi điểm nút (fuel_at_node).
- Lặp lại cho đến khi `frontier` trống:
  - Lấy điểm nút có chi phí thấp nhất từ `frontier`.
  - Nếu điểm nút này là đích, trả về đường đi và chi phí tổng cộng.
  - Với mỗi điểm nút lân cận, tính toán chi phí mới và nhiên liệu mới.
  - Nếu nhiên liệu mới = 0 nhưng điểm neighbour này lại là goal thì sẽ lập tức cập nhật dictionary và thêm điểm nút vào và bỏ qua điều kiện dưới.
  - Nếu nhiên liệu mới ít hơn hơn hoặc bằng 0, bỏ qua điểm nút này.
  - Nếu điểm nút chưa được khám phá hoặc chi phí mới thấp hơn chi phí đã lưu, cập nhật dictionary và thêm điểm nút vào `frontier` với ưu tiên mới.

**Hàm A_star_search(board):** Đây là hàm chính của level 3.

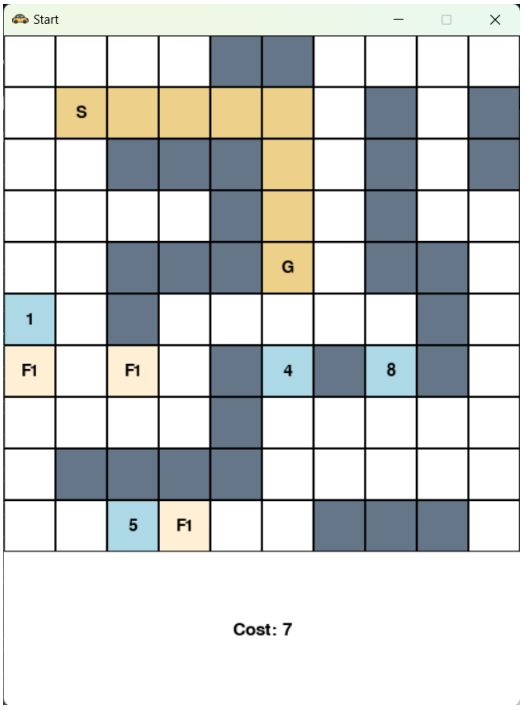
- Thử tìm đường trực tiếp từ điểm bắt đầu đến điểm đích với nhiên liệu ban đầu bằng hàm **hàm a_star_search(board, start, goal, initial_fuel)**.
- Nếu tìm được đường đi và chi phí nhỏ hơn hoặc bằng thời gian cho phép, trả về đường đi.

- Nếu không đủ nhiên liệu, thử tìm đường đi qua các trạm xăng.
- Với mỗi trạm xăng, tìm đường đi từ điểm bắt đầu đến trạm xăng và từ trạm xăng đến điểm đích bằng hàm **hàm a_star_search(board, start, goal, initial_fuel)**.
  - Đầu tiên, tìm đường từ điểm bắt đầu đến trạm xăng (`gas_station`).
  - Nếu tìm được đường đi đến trạm xăng, tiếp tục tìm đường từ trạm xăng đến điểm đích sau khi đổ đầy nhiên liệu.
  - Nếu không tìm được đường đến trạm xăng hoặc từ trạm xăng đến điểm đích, bỏ qua trạm xăng này.
  - Kết hợp các đường đi từ điểm bắt đầu đến trạm xăng và từ trạm xăng đến điểm đích thành một đường đi hoàn chỉnh, tránh trùng lặp trạm xăng.
  - Tính toán tổng chi phí của đường đi hoàn chỉnh.
  - Nếu tổng chi phí của đường đi hoàn chỉnh nhỏ hơn chi phí ngắn nhất đã tìm thấy, cập nhật đường đi ngắn nhất và chi phí ngắn nhất.
- Trả về đường đi ngắn nhất nếu chi phí nhỏ hơn hoặc bằng thời gian cho phép, ngược lại trả về None.

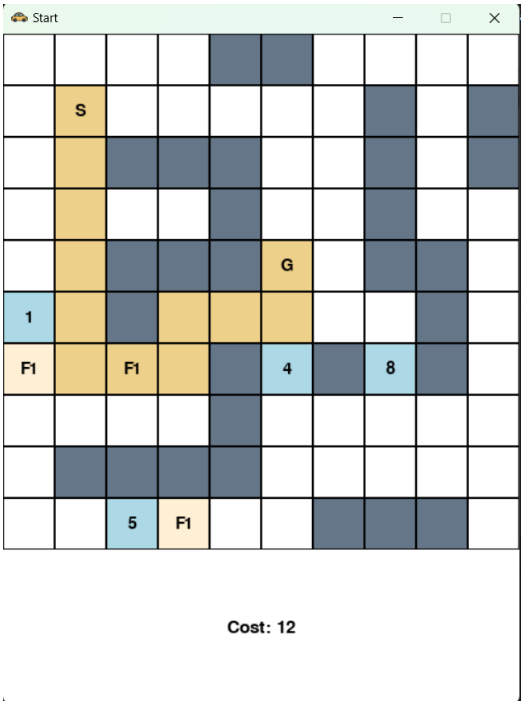
## 5.4 Các test case

### 5.4.1 Test case 1

- Ma trận: 10x10, Time limit: 18, fuel: 7. Vừa đủ fuel về đích.




(a) fuel = 7



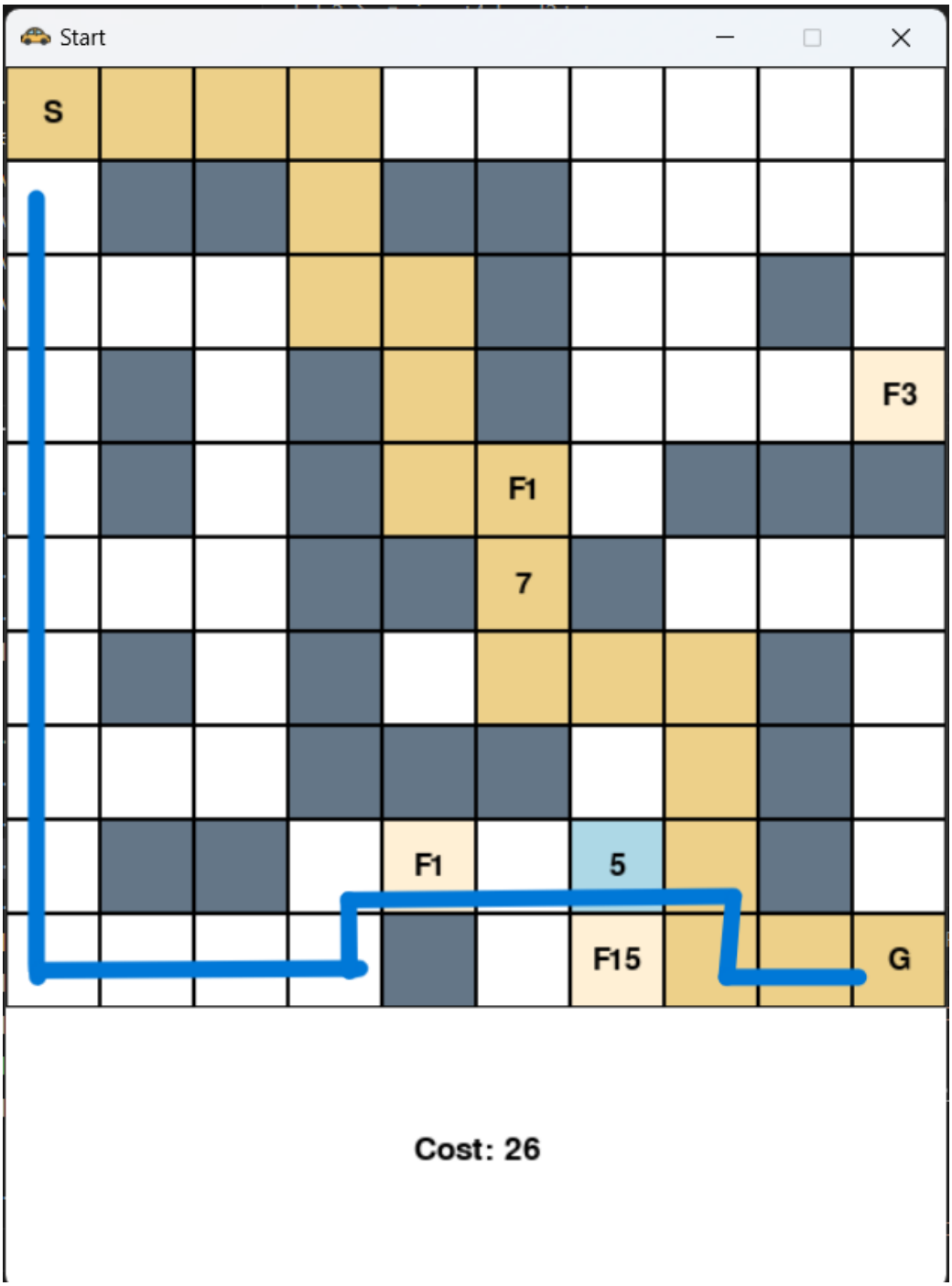

(b) fuel = 6

Hình 38: Kết quả test case 1

- Test case 1 mặc định sẽ là fuel = 7 nhưng em có chạy thêm trường hợp nếu fuel = 6 để so sánh.
- Có thể thấy ở hình (a) với việc fuel = 7 thì xe hoàn toàn có thể chạy đến đích và hoàn thành. Hình (b) với việc thay đổi tham số fuel = 6 thì không nên nó phải chạy xuống để tìm trạm xăng r mới tìm về đích.

### 5.4.2 Test case 2

- Ma trận: 10x10, Time limit: 26, Fuel: 15.Tồn tại2 đường cùng cost nhưng chọn ít ô hơn.

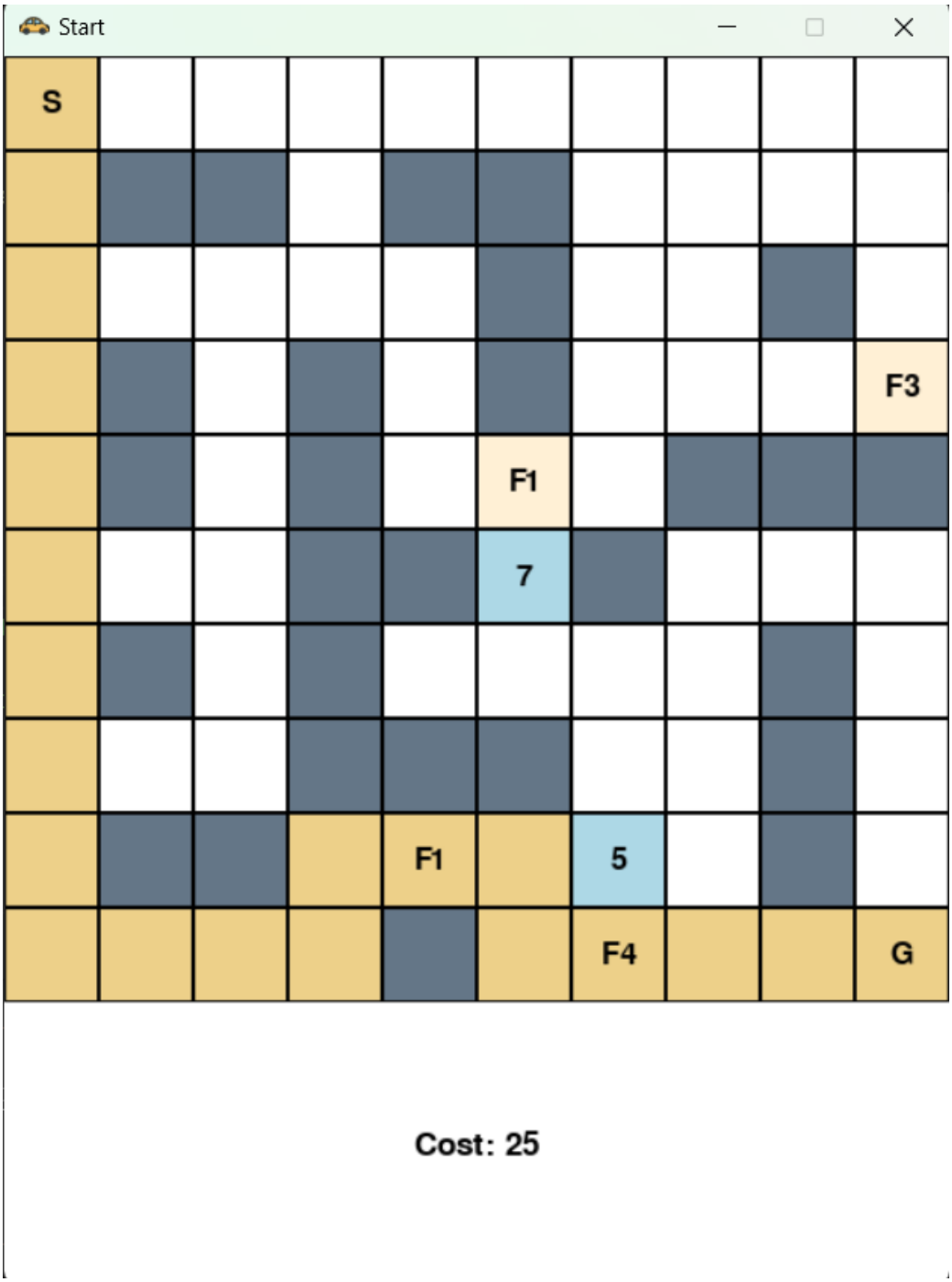



Hình 39: Kết quả test case 2

- Ngoài đường màu vàng có cost là 26 thì đường màu xanh cũng tương tự nhưng đường màu vàng 18 ô và đường màu xanh 20 ô nên thuật toán search ra đường vàng nhanh hơn và trả về.

### 5.4.3 Test case 3

- Ma trận:10x10, Time limit: 26, Fuel: 15. Tối ưu thời gian giữa trạm xăng và toll booth.

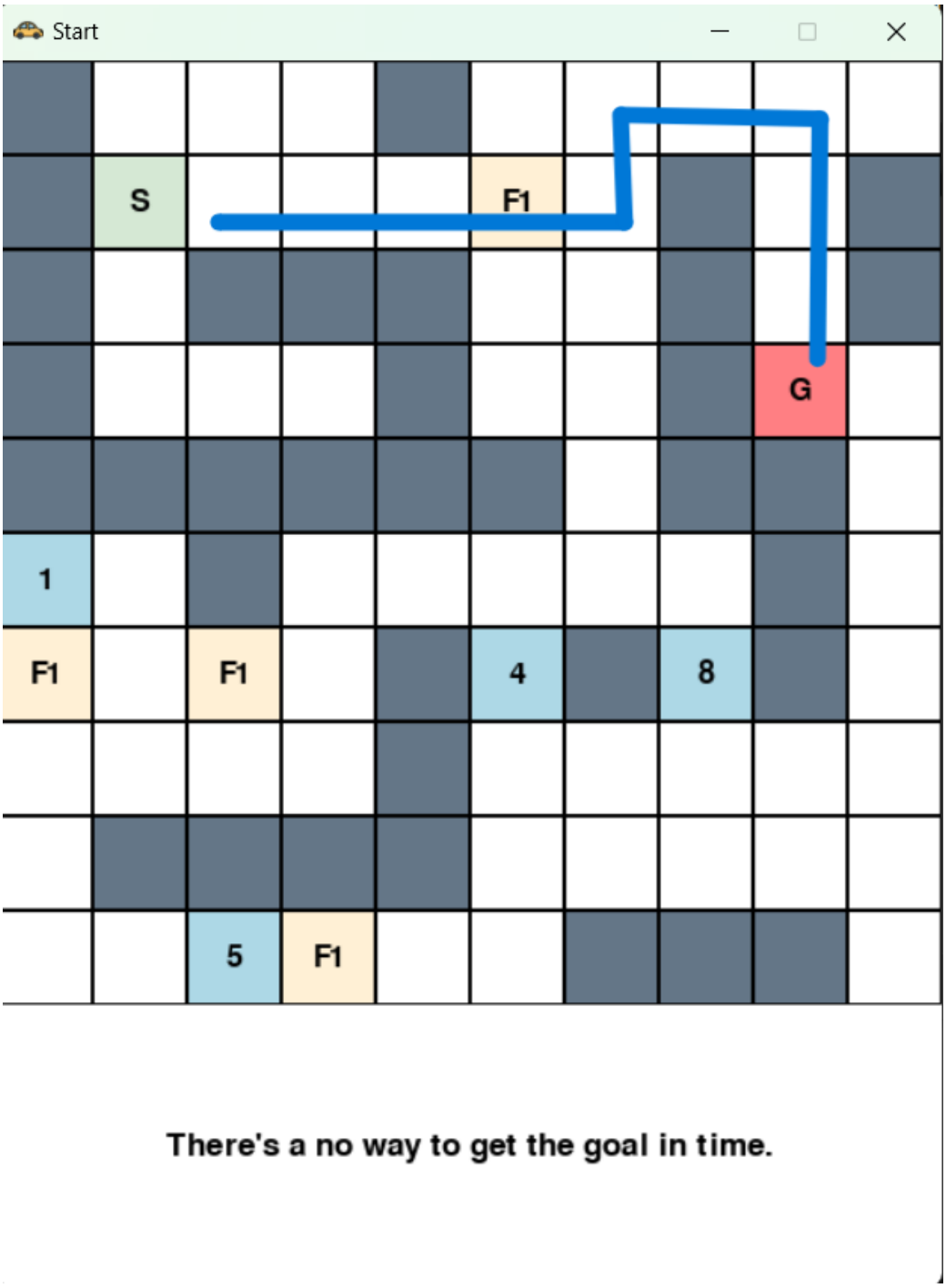



Hình 40: Kết quả test case 3

- Ta có tổng cost nếu như chọn đi qua ô toll booth 5 là 26 nhưng đi qua trạm xăng F4 thì chỉ là 25 nên chương trình chọn trạm xăng thay vì ô 5 mặc dù vẫn đủ xăng.

### 5.4.4 Test case 4

- Ma trận: 10x10, Time limit: 10 ,Fuel: 10. Không đủ thời gian để đến đích.

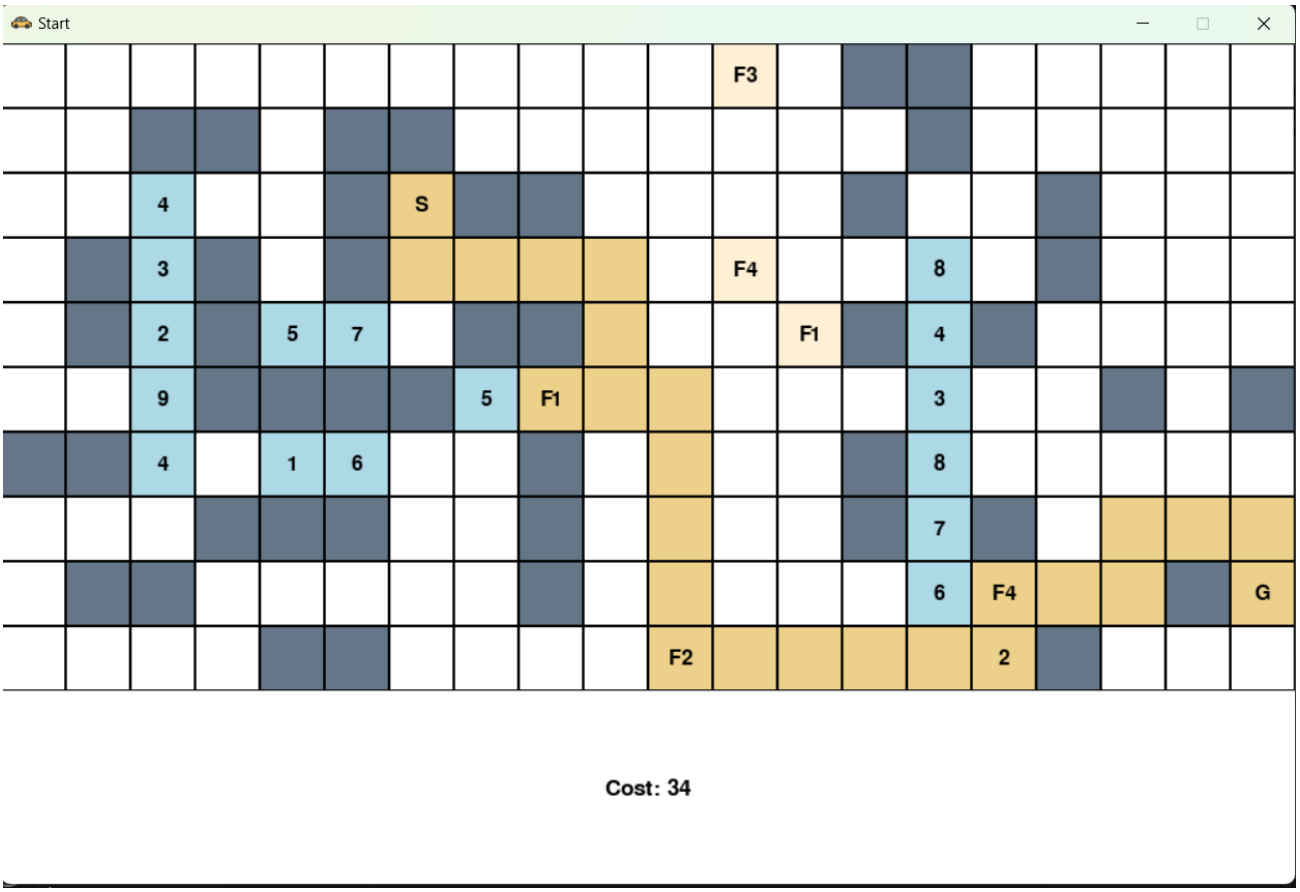



Hình 41: Kết quả test case 4

- Ta có thể thấy chỉ với time limit là 10 thì xe không thể về đích mặc dù ta có đường đi đó là đường màu xanh(vẽ thêm, chương trình không hỗ trợ in)nên thuật toán trả về none chương trình in ra kết quả như vậy.

### 5.4.5 Test case 5

- Ma trận: 10x20, Time limit: 50, Fuel: 7. Vượt qua nhiều trạm xăng để tiếp xăng tới đích và có nhiều đường đi nhưng chọn đường ngắn nhất.




Hình 42: Kết quả test case 5

- Với đường đi màu vàng ta có thể thấy cần phải đi qua 3 trạm xăng thì mới có thể đủ fuel về đến đích và đường này có tổng cost là 34 do nó sẽ đi trùng lại với ô bên phải F1 ở trường hợp này là hợp lý vì bên trái là toll booth 5 sẽ mấy thêm thời gian.
- Đây không phải là đường đi duy nhất nhưng nó là ngắn nhất. Ví dụ đường đi khác là chạy lên F3 rồi về F4 dưới F3 sau đó qua F2 tiếp theo là F4 gần goal và cuối cùng là về goal nhưng đường này rất không hiệu quả và chi phí cao nên chương trình lựa chọn đường có cost thấp nhất là màu vàng để trả về.

## 5.5 Nhận xét tổng thể

- Nhìn chung ta có thể thấy thuật toán ở level 3 xử lý khá tốt các test case và đều trả về đường đi tối ưu nhất khi có nhiều đường về đích cũng như nhưng trường hợp liên quan tới quản lý xăng.
- Điều này giúp tạo vững nền tảng vững chắc để tiếp tục xây dựng chương trình ở level 4 khi mà có nhiều xe hơn và việc xử lý phức tạp hơn rất nhiều.

# 6 Level 4

## 6.1 Thuật toán

### 6.1.1 Ý tưởng chung

- Chuyển hóa bài toán bằng cách liệt kê tất cả trạng thái của các xe và theo dõi xem liệu rằng các trạng thái đó nếu xe chính tới đích thì coi như đã giải được bài toán. Các xe phải thích ứng với sự thay đổi của môi trường. Ví dụ: Có 4 xe, xe 0 di chuyển một ô, thì môi trường các xe 1,2,3 phải biết rằng rằng xe 0 đã di chuyển và đang đứng ở ô nào, cũng như là biết luôn vị trí của các xe khác. Tới lượt xe 1 di chuyển cũng vậy, xe số 0,2,3 phải nhìn được các xe còn lại ở đâu.
- Kế thừa hàm A search từ lv3 để áp dụng cho thuật toán tìm đường đi ở lv4: Đi kiếm đường đi tối ưu nhất mà đảm bảo lượng xăng khi tới đích vẫn còn.
- Phải xác định các tiêu chí trạng thái của mỗi agent: Điểm bắt đầu, điểm kết thúc, trạng thái di chuyển (lên, xuống, trái, phải, đứng yên), ô nào đi được, chi phí cho mỗi ô đi

- Điểm bắt đầu: Chính là điểm hiện tại của xe đó trên bản đồ
- Điểm kết thúc: Chính là điểm Goal của xe đó trên bản đồ, trường hợp đối với các xe phụ nếu tới Goal sẽ random lại điểm kết thúc và đi tiếp. Nếu xe chính tới được đích thì coi như là bài toán đã giải quyết thành công.
- Trạng thái di chuyển: Lên, xuống, trái, phải một ô. Đối với trường hợp nếu tạm thời không kiếm ra được đường đi mà còn xăng với thời gian thì xe tạm thời đứng yên. Trường hợp xe hết xăng hoặc hết thời gian mà không tới được đích thì xe sẽ dừng vĩnh viễn ở ô đó (đánh dấu ô đó -1)
- Các ô đi được: Tất cả các ô đi được ngoại trừ ô đánh dấu -1, nếu trường hợp có xe khác chiếm ô mà xe chính định đi qua, xe đó sẽ nhìn ô bị chiếm như -1, tức là vẫn không đi được.
- Chi phí mỗi ô đi: Đi qua một ô bình thường thì lượng xăng giảm đi một, thời gian giảm đi một. Qua trạm xăng thì lượng xăng được đổ đầy dung tích lại, còn thời gian tiếp tục giảm đi theo thời gian chờ của trạm xăng đó. Qua các trạm dừng xe, thời gian giảm đi tùy theo thời gian chờ, lượng xăng giảm đi một.

### 6.1.2 Thuật toán

- Khởi tạo trạng thái ban đầu: Đọc file input vào, đếm xem trong file input có mấy chiếc xe (quy ước xe được đặt là chữ S với con số đằng sau, trong đó S0 là xe chính, các xe Sn với n khác 0 là xe phụ). Đưa vào hàm createState để khởi tạo list tên boards gồm n xe. Phần tử đầu tiên là xe chính.
- Cho vào vòng lặp true:

- Duyệt cái list Boards từ 0 tới n-1 (tức là turn based, xe chính sẽ đi trước)
- Gọi hàm findandsetothervehicles() để cho xe nhìn các ô khác bị xe khác chiếm là -1.
- Gọi hàm Astarsearch để tìm đường đi
- Nếu xe chính không có đường đi (Boards[0]) và lượng xăng hoặc thời gian bằng 0 thì trả về thông báo không có đường đi, không thể giải quyết được bài toán.
- Nếu không phải xe chính mà không có đường đi, thì gọi hàm GenerateNewState để tạo nên trạng thái mới (trạng thái đứng yên, None)
- Nếu kiếm ra đường đi, thì pop tập hàng đợi ưu tiên 2 lần (vì phần tử đầu là vị trí xe đang đứng hiện tại, phần tử tiếp sau đó là vị trí mới xe sẽ định đi vào). Gọi hàm GenerateNewState để cập nhật lại trạng thái mới (xe đi vào ô mới). Hàm GenerateNewState cập nhật hết các thông số trong class Board, bao gồm ma trận, lượng xăng, thời gian, địa điểm ô hiện tại.
- Sau khi tạo xong trạng thái mới, khôi phục lại bản đồ ở các ô bị chiếm (-1) thành kí hiệu của các xe.

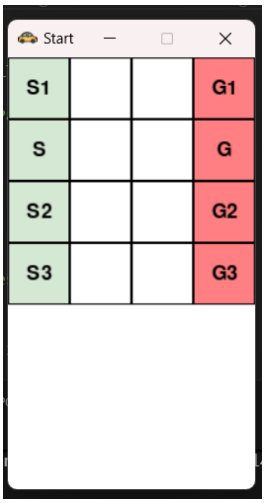
- Truyền trạng thái (ma trận) của xe này vào xe kế tiếp được duyệt, như vậy xe kế tiếp sẽ biết được môi trường của mình thay đổi ra sao để tìm đường đi cho phù hợp.
- Ta phải kiểm tra liên tục sau mỗi trạng thái là xe chính tới đích của nó chưa, nếu rồi thì thoát khỏi vòng for và cả vòng lặp True. Kết thúc bài toán thành công.
- Nếu xe chính chưa tới đích mà vẫn còn xăng và thời gian thì tiếp tục lặp lại vòng True và cứ thực hiện tuần tự các bước trình bày như ở trên.

## 6.2 Các test case

- Các thầy đọc từ trên xuống dưới, từ trái qua phải để tiện theo dõi các trạng thái. Mỗi lượt minh họa hình ảnh là tất cả chiếc xe đều di chuyển. Đi qua mỗi test case thì ảnh minh họa được đếm lại từ đầu là 1.

### 6.2.1 Test case 1

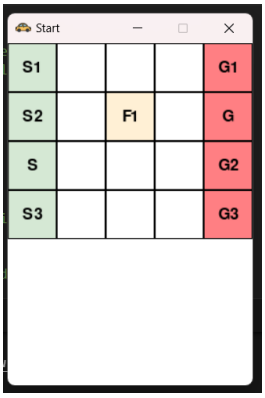
Khởi đầu là test case cơ bản nhất, ma trận 4x5 gồm có 4 chiếc xe. Lượng xăng là 3, thời gian là 3.






### 6.2.2 Test case 2

- Test case tiếp theo cũng là 4x5, có thêm trạm xăng, lượng xăng là 3, thời gian là 10. Tình huống này đặc biệt ở chỗ sẽ có 3 xe cùng nhau tranh đoạt trạm F1 bơm nhiên liệu.







- Tại hình số 3 qua hình số 4, vì xe 0 và xe 1 chỉ còn nhiên liệu là 1 nên phải đi tới ô F1 mà xe 2 đang đứng để đổ xăng, nhưng không tới được vì xe 2 đang đứng đó nên chọn đứng im, nên ta thấy xe 2 ở hình 4 được di chuyển tiếp.
- Hạn chế của thuật toán ở đây: Ở hình 3 qua hình 4, vì xe nhìn các xe khác với góc nhìn xem ô đó là -1, nên khi xe 2 vô Goal của xe 0 thì xe 0 sẽ không tìm thấy được goal nên phải lựa chọn đứng yên, tạo điều kiện cho xe 1 xuống đổ xăng trước.



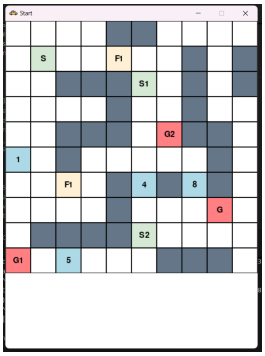

Hình 43: Đây là list các dictionary ghi nhận lại điểm start và goal được random tương ứng với index của các xe, ta thấy xe 1 và 2 đều random 1 lần

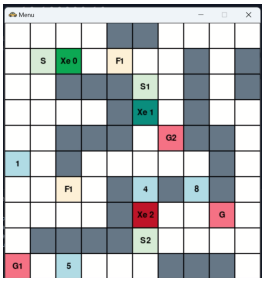


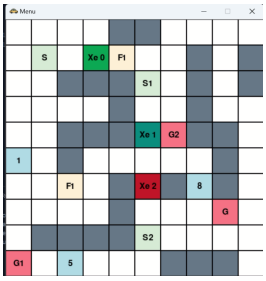
- Xe 2 sau khi được random điểm bắt đầu mới thì cũng hết xăng, nên dừng tại vị trí 2,4
- Xe 1 sau khi được random điểm bắt đầu mới thì cũng đã hết xăng, vậy nên dừng tại vị trí 0,4
- Xe 0 đã di được tới đích, tuy nhiên với nhược điểm của thuật toán đã nói trên, thời gian đi của xe 0 không thực sự tối ưu.
- Xe 3, tất nhiên, vì không đủ nhiên liệu để tới điểm G3 nên nó không di chuyển xuyên suốt cả trạng thái.

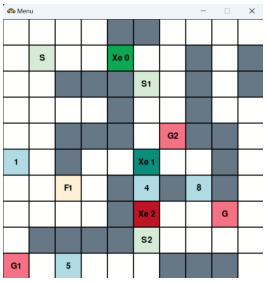
### 6.2.3 Test case 3

- Test case tiếp theo là ma trận 10x10 với time là 20, fuel là 20 gồm 3 chiếc xe.

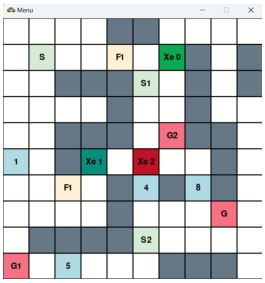
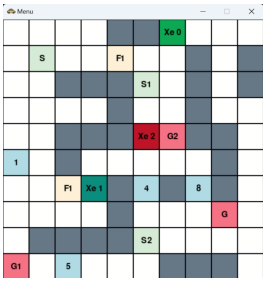
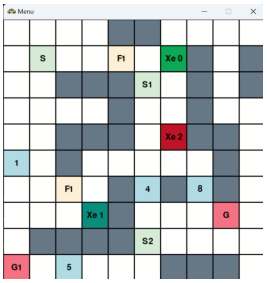




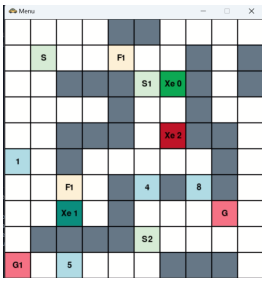
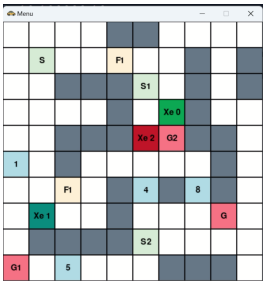
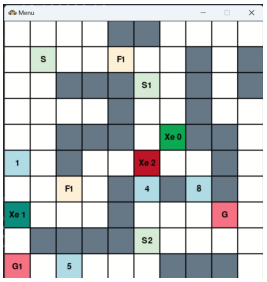
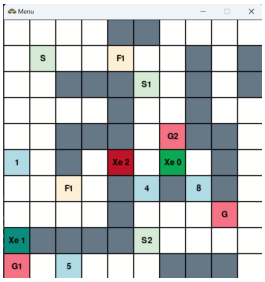
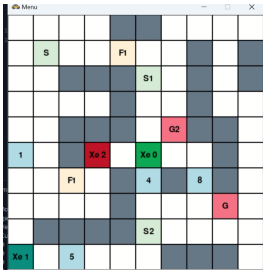
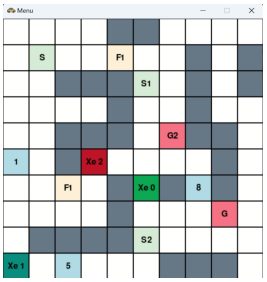




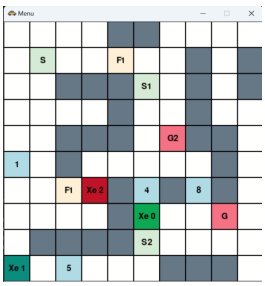
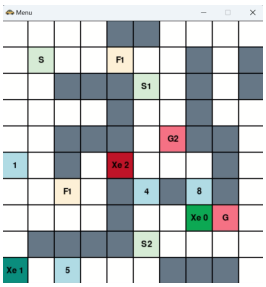
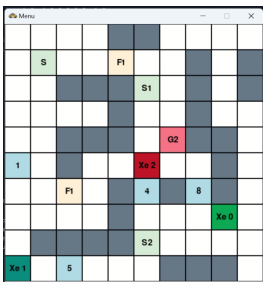
- Vì những trạng thái tiếp theo chưa quan trọng nên em sẽ tua tới trạng thái xe 0 ở vị trí 1,6.

- Như vậy có thể thấy xe chính (số 0) không đi tối ưu được thời gian, từ hình 5 qua hình 6, nếu xe chính đi theo hướng bên phải thì sẽ có chi phí là 11. Còn xe chính đi xuống thì khi muốn tới đích chắc chắn sẽ qua ô toll booth là 4 hoặc là 8, vì vậy thời gian đi sẽ nhiều hơn 11.

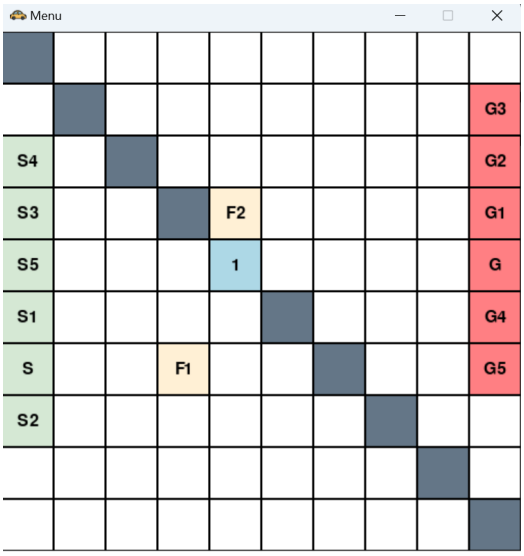
- Như vậy xe 1 đã tới đích, điểm đích được random tiếp theo là ở G2 (4,6). Xe còn fuel là 8 tuy nhiên time chỉ còn 7, vậy không thể tới đích tiếp theo được nữa, nên xe 1 sẽ đứng ngay tại vị trí 9,0

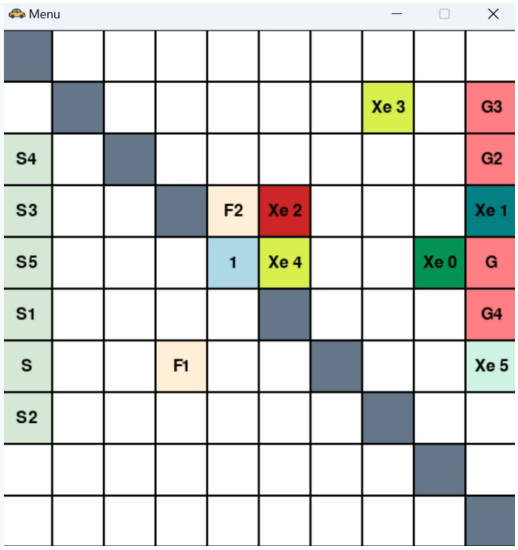




- Vậy xe 0 cuối cùng đã tới đích, tuy nhiên đường đi không thật sự tối ưu.

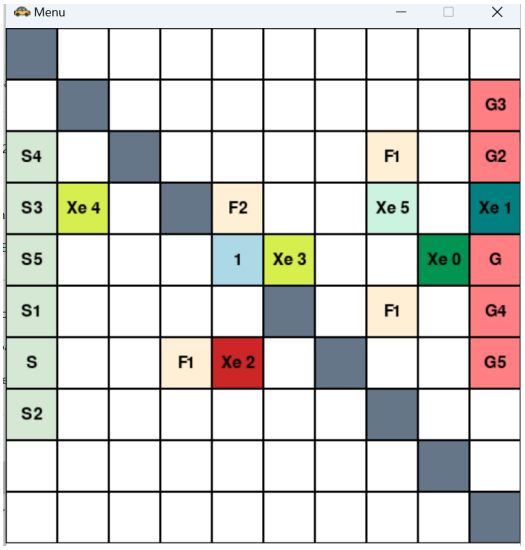
### 6.2.4 Test case 4

- Lần này là ma trận 10x10 ma trận đường chéo chính chỉ có 1 ô đi duy nhất, có 5 xe tất cả. Như vậy chắc chắn sẽ xảy ra đụng độ giữa các xe. Ở test case này có quá nhiều trạng thái nên em chỉ chụp trạng thái kế cuối.




(a) Time = 15, cost = 15. Xe đã tới đích.

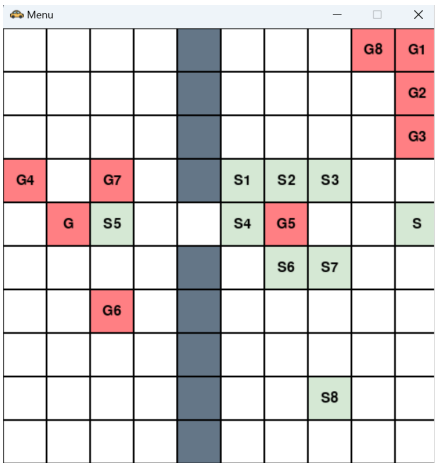


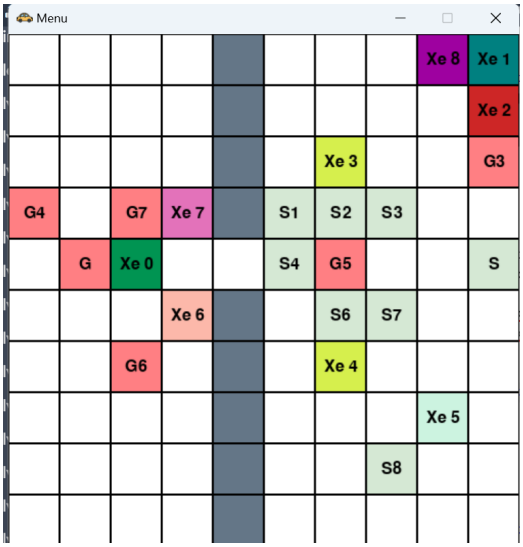

(b) Time = 15, cost = 10. Xe đã tới đích

Hình 44

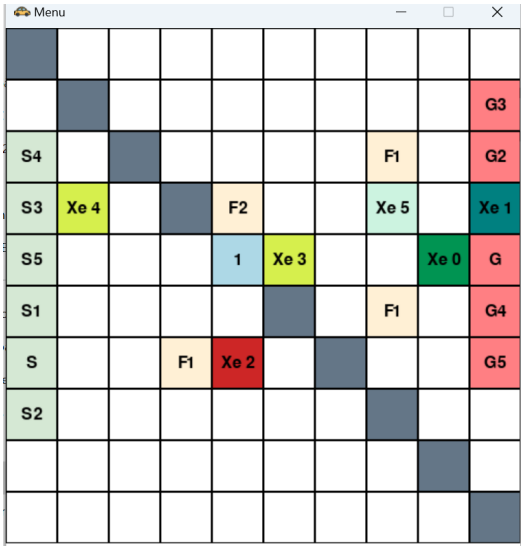
### 6.2.5 Test case 5

- Đây là test case đông xe nhất với 9 chiếc xe, ma trận 10x10, là ma trận có đường dọc là vật cản và chỉ chừa một lối đi. Xe S ở trạng thái ban đầu là bị chắn đường bởi các xe khác, các xe khác cũng bị chặn. Tùy vào các trường hợp bơm nhiều xăng và cho thời gian nhiều thì xe S sẽ tới đích. Nhưng phần lớn thì xe S không thể tới đích được. Vì có quá nhiều trạng thái nên em chỉ chụp lại trạng thái kế cuối.




(a) Time = 20, fuel = 20. Xe đã tới đích.




(b) Time = 15, fuel = 10. Xe không thể đi tới được đích

Hình 45

## 6.3 Nhận xét tổng thể

- Nhìn chung, thuật toán lv4 chạy khá ổn trong một số trường hợp cơ bản như bản đồ ít trạm xăng và ít toll booth.
- Tuy nhiên, vì code cứng ở các phương thức ở class board, đặc biệt là phương thức movevehicle, nó đi qua trạm toll booth và chuyển nó thành 0. Nên dù thuật toán lv3 chạy tốt thì các trạng thái ở lv4 xe không thể nhìn trạm toll booth (nếu trạm đó bị xe khác đi qua). Vậy thuật toán lv4 chưa thực sự tối ưu.

# 7 Visualize và đọc ghi file

## 7.1 Pygame

- Pygame là một module Python đa nền tảng dành cho việc phát triển trò chơi điện tử. Nó tích hợp cả đồ họa máy tính và thư viện âm thanh được tạo ra đặc biệt để tương thích với ngôn ngữ lập trình Python.
- Pygame giúp các nhà phát triển dễ dàng tạo ra các trò chơi đơn giản hoặc phức tạp bằng ngôn ngữ lập trình Python một cách hiệu quả và linh hoạt. Với sự hỗ trợ đa nền tảng và tính linh hoạt cao, Pygame đã trở thành công cụ phổ biến trong cộng đồng lập trình viên để tạo ra các ứng dụng giải trí đa dạng, hấp dẫn.

### 7.1.1 Cách cài đặt

- Sau khi cài đặt python, bạn có thể cài đặt thư viện pygame bằng cách sau trong cửa sổ dòng lệnh hoặc terminal:

pip install pygame

### 7.1.2 Các module chính được sử dụng

- pygame.display: Kiểm soát cửa sổ hiển thị và màn hình.
- pygame.image: Xử lý việc tải và lưu hình ảnh.
- pygame.mouse: Quản lý đầu vào từ chuột và hiển thị con trỏ.

### 7.1.3 Những function chính:

- **Tạo màn hình:**
  - `init_screen(rows,cols)`: tạo màn hình dựa trên số dòng và cột có được từ file
- **Thực hiện vẽ map:**
  - `draw_map(rows,cols)`: Tô trắng màn hình và vẽ bảng theo số cột và hàng cho trước
  - `draw_cell(matrix,rows,cols)`: Tô màu một ô cụ thể với giá trị của ô khi nó khác 0
  - `draw_board(matrix,rows,cols)` : vẽ bản đồ hoàn chỉnh cho toàn bộ map với các giá trị đặc biệt : bị chặn, trạm xăng,....
  - `draw_path()` : di chuyển xe theo đường dẫn, tô màu các ô được thăm tới bước hiện tại
  - `draw_result()` : vẽ lại con đường mà chiếc xe đã đi sau khi đã đến đích
  - `draw_multiple_path(board, list_of_recorded_moves, list_of_recorded_start_goal)` : Hàm vẽ nhiều xe một lúc theo lượt
- **Hàm in màn hình menu lựa chọn:**
  - `menu()`: In màn hình menu lựa chọn level
  - `lvl1()`: Màn hình chọn map cho level 1
  - `lvl2()` : Màn hình chọn map cho level 2
  - `lvl3()` : Màn hình chọn map cho level 3
  - `lvl4()` : Màn hình chọn map cho level 4
- **Hàm xử lý Visual cho từng level:**
  - `start(board,path)`: Hàm chạy visualize chính cho level 1,2,3
  - `start_lvl4_clone(board)`: Hàm chạy visualize chính cho level 4
  - `mod_lvl1(filename, output_suffix, algo)`, `mod_lvl2(filename, output_suffix, algo)`, `mod_lvl3(filename, output_suffix, algo)`: Lấy thông tin từ file, xử lý bằng hàm `start(board, path)` và in file `mod_lvl4(filename, output_suffix)` : Lấy thông tin từ file, xử lý bằng hàm `start_lvl4_clone(board)` và in file

#### 7.1.4 Những function hỗ trợ

- **Hàm tô màu cho từng trường hợp:**
  - `hightlight_BlockedCell(row,col)` :Tô màu cho ô bị chặn
  - `hightlight_SpeacialCell(row,col)` :Tô màu cho ô có định dạng đặc biệt (trạm xăng, start,goal,...) và in ra giá trị của ô
  - `hightlight_cell(X,Y,color)` :Tô màu cho ô cụ thể với màu mong muốn (không in ra giá trị)
  - `hightlight_path(board,path)` :Tô màu ô nơi mà xe đang đứng
- **Hàm hỗ trợ (helper)** :
  - `get_font(size)` : chọn size chữ viết
  - `write_String(X,Y,string,cell)` :Viết chữ tại vị trí cho trước
  - `count_Vehicles(board)` :Đếm vào trả ra số xe có trong bảng
  - `all_path_compeleted(step_indices,list_of_recorded_moves)` : kiểm tra việc tất cả các đường đã hoàn thành hay chưa

### 7.2 Hàm đọc, ghi file

- `read_file(filepath)` : hàm đọc input từ file
- `write_file(filepath, path, algo)` : hàm ghi tiếp vào file

## 8 Tham khảo


- Khoảng cách Manhattan: https://vi.wikipedia.org/wiki/Khoòểảng_cÕách_Manhattan
- BFS: https://www.geeksforgeeks.org/breadth-first-traversal-bfs-on-a-2d-array/
- DFS: https://www.geeksforgeeks.org/depth-first-traversal-dfs-on-a-2d-array/
- UCS: https://stackoverflow.com/questions/48884990/using-uniform-cost-search-on-a-matrix-in-python
- A*: https://stackoverflow.com/questions/61795916/a-star-a-search-algorithm-on-labyrinth-matrix-in-pyth
- IDS https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-search
- Giáo trình Cơ sở trí tuệ nhân tạo - Lê Hoài Bắc, Tô Hoài Việt
- Slide Introduction to AI - giảng viên Nguyễn Tiến Huy