**BÁO CÁO ĐỒ ÁN 1 – ROBOT TÌM ĐƯỜNG**

MÔN HỌC: CƠ SỞ TRÍ TUỆ NHÂN TẠO

NHÓM: 25

THÀNH VIÊN:

1. Bùi Hữu Nghĩa - 1712613
2. Nguyễn Anh Nam - 1712604
3. **Mức độ hoàn thành**

|  |  |  |
| --- | --- | --- |
| Mức | Yêu cầu | Hoàn thành |
| 1 – 40% | Cài đặt thành công 1 thuật toán tìm đường | 100% |
| 2 – 30% | Cài đặt ít nhất 3 thuật toán khác nhau | 100% |
| 3 – 30% | Đi qua các điểm đón | 100% |
| 4 – 10% | Vật cản di dộng | 0% |
| 5 – 10% | Không gian 3 chiều | 0% |

1. **Chi tiết**

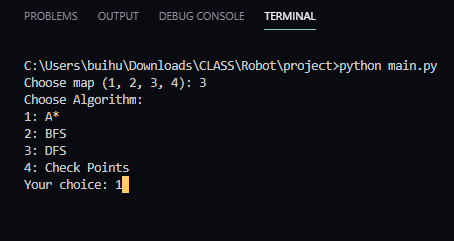
Chuẩn bị: đầu tiên ta cần cài python 3 và thư viện opencv, mở terminal tại thư mục chứa file

Dùng lệnh dưới để cài opencv

pip install opencv-python

Tiếp theo ta chạy chương trình bằng lệnh:

python main.py

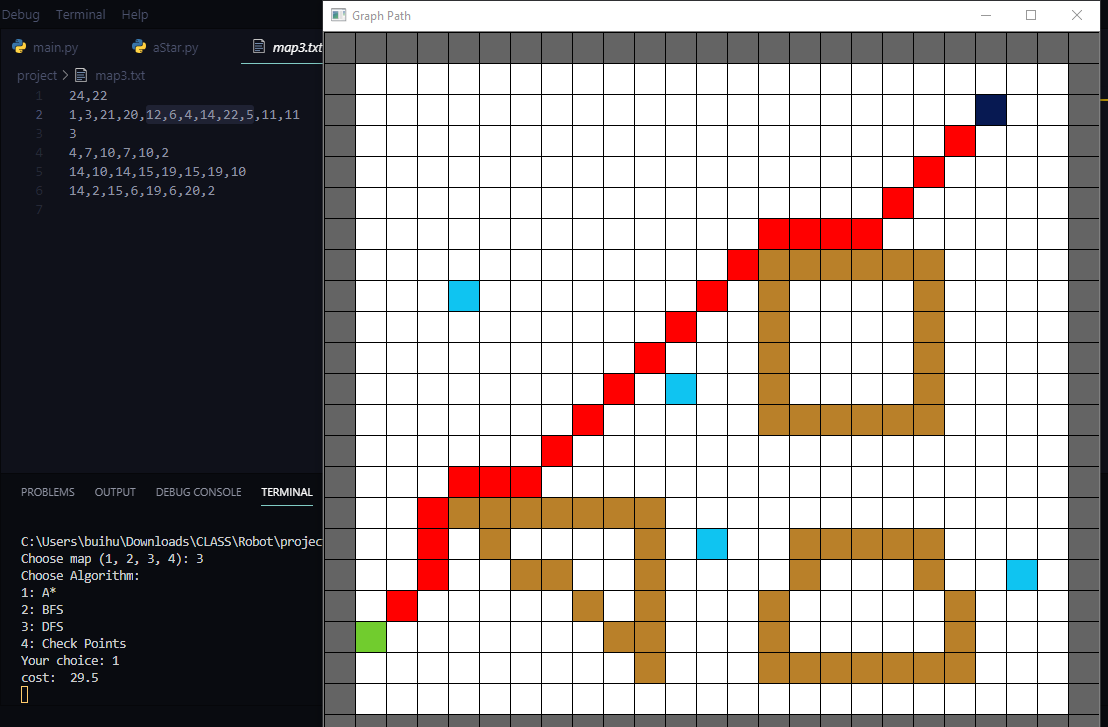


Có 4 tùy chọn map(1 - 4) và 3 thuật toán (A\*, BFS và DFS)

Lần lượt chọn 1 trong các tùy chọn trên và enter

* **Mức 1, 2: cài đặt thành công 3 thuật toán**

Chạy chương trình với map đã cho và 1 trong 3 thuật toán đã được cài đặt (A\*, BFS, DFS)

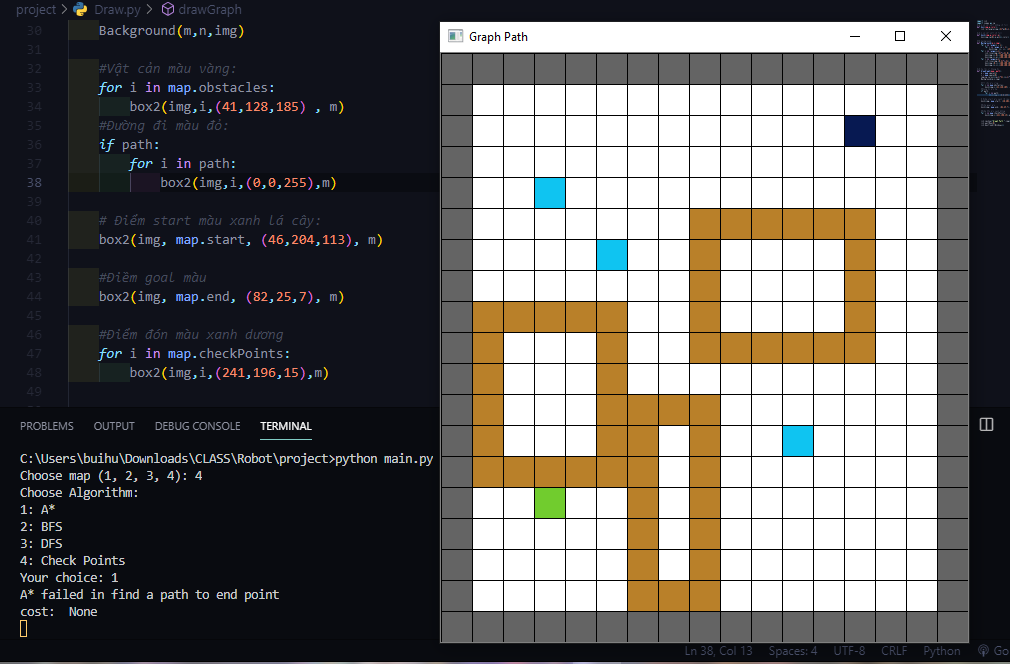


Map được chọn là map3 và thuật toán A\*, trong đó:

* Điểm màu xanh lá cây là start
* Điểm màu xanh đậm là end(goal)
* Các điểm màu xanh nhạt là điểm đón
* Các điểm màu cam là vật cản
* Các điểm màu đỏ là đường đi
* Cho phép đi chéo với chi phí 1.5

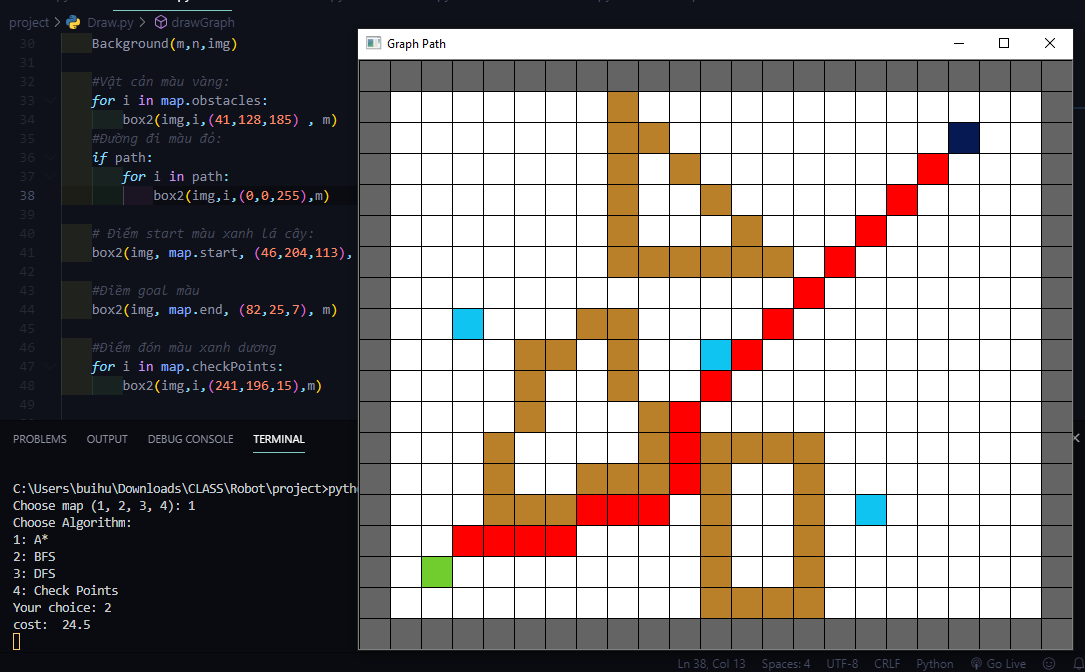
**Trường hợp không có đường đi**

Ta chạy chương trình với map4 và thuật toán bất kì đã cho

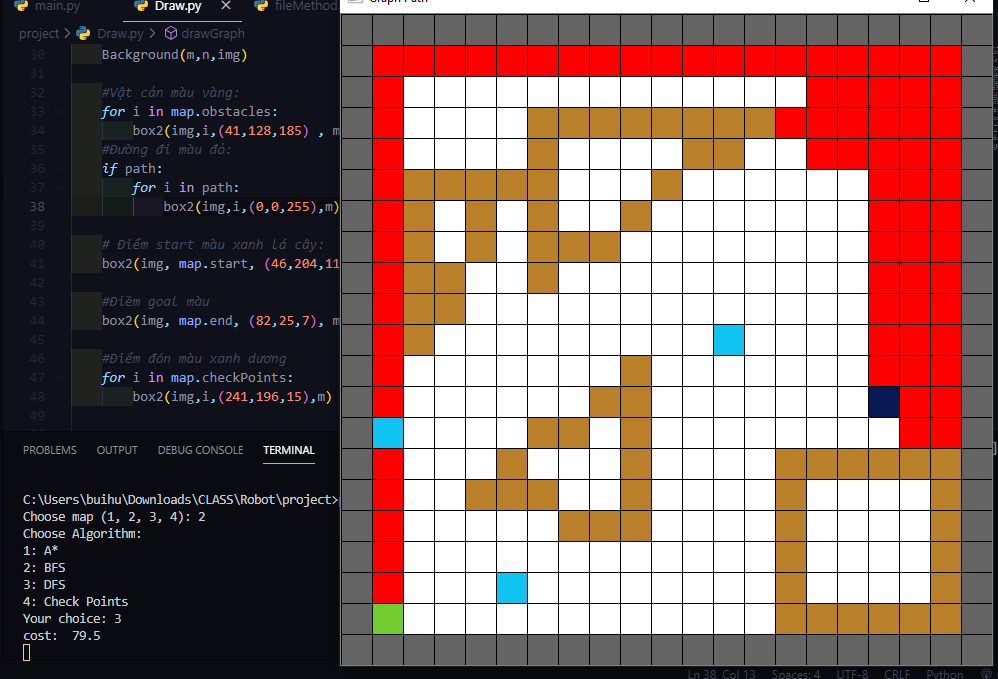


Như ta thấy các vật cản đã che hết đường đi của start, vì không tìm được đường đi nên chi phí bằng None

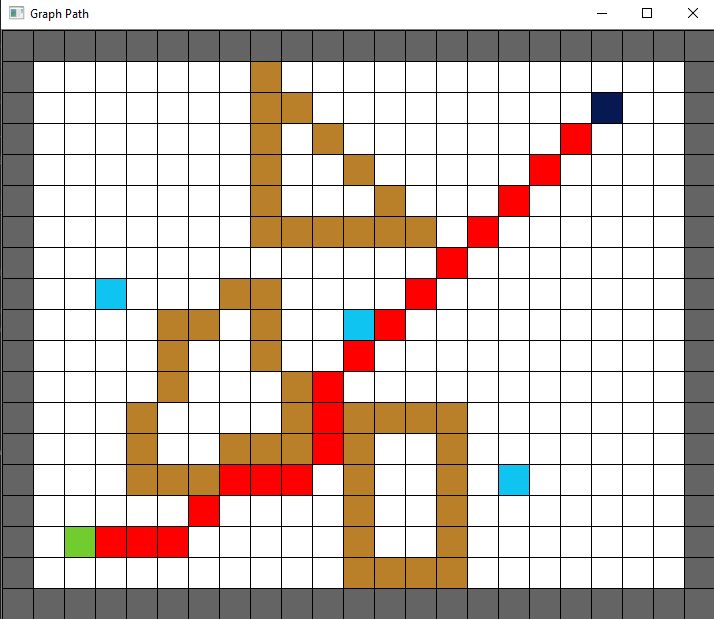
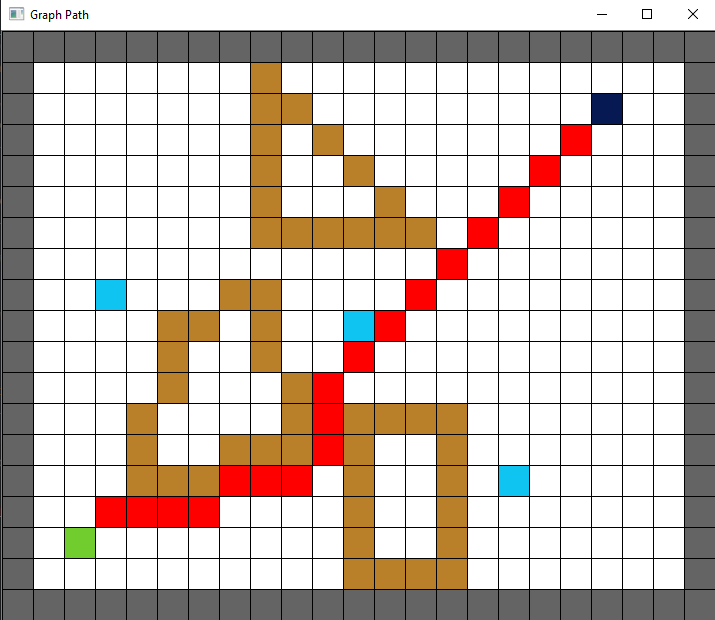
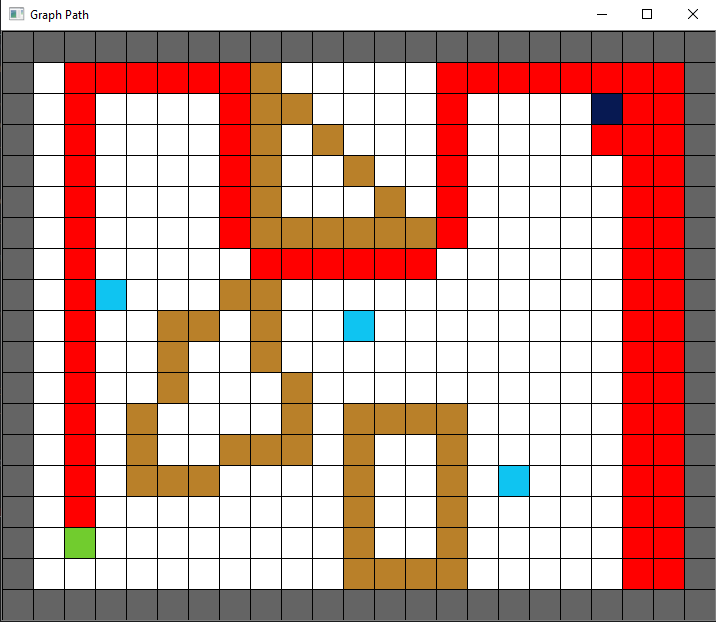
Thuật toán BFS – map1



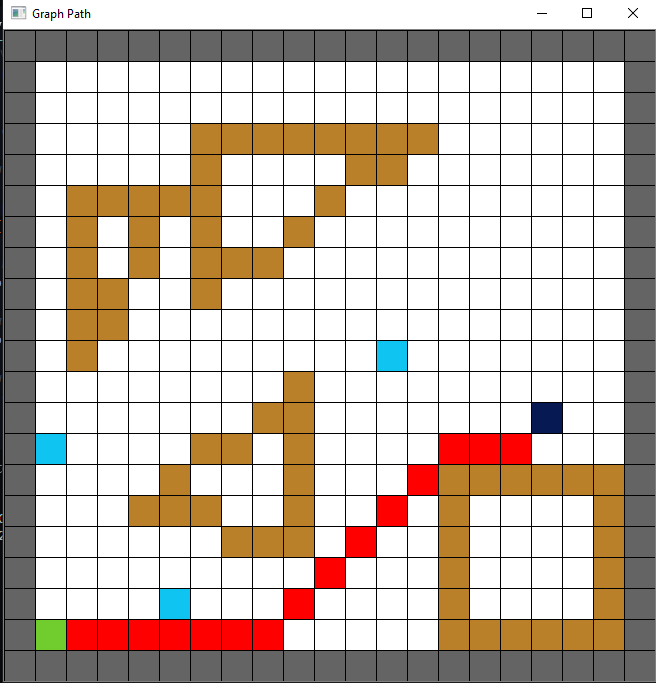
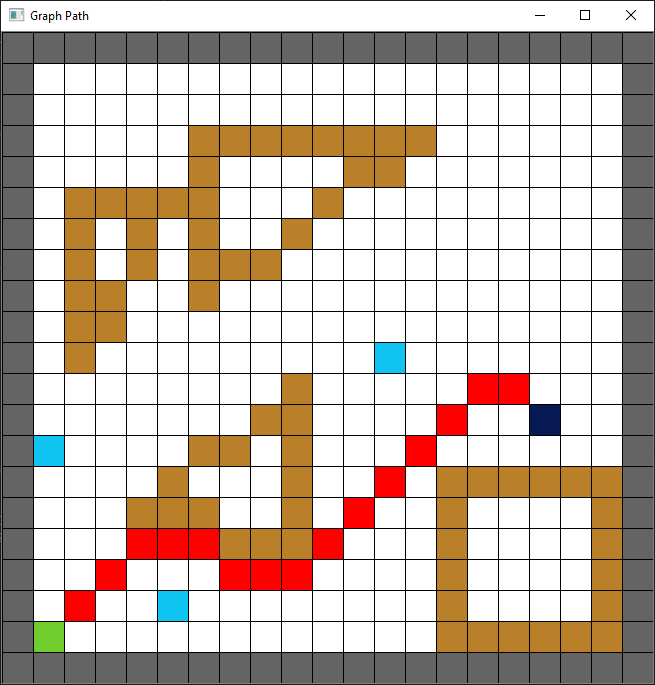
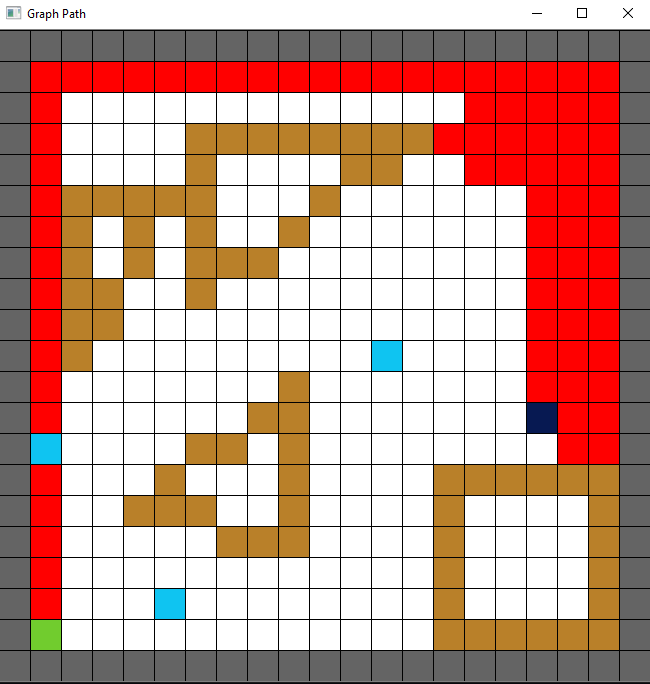
Thuật toán DFS – map2



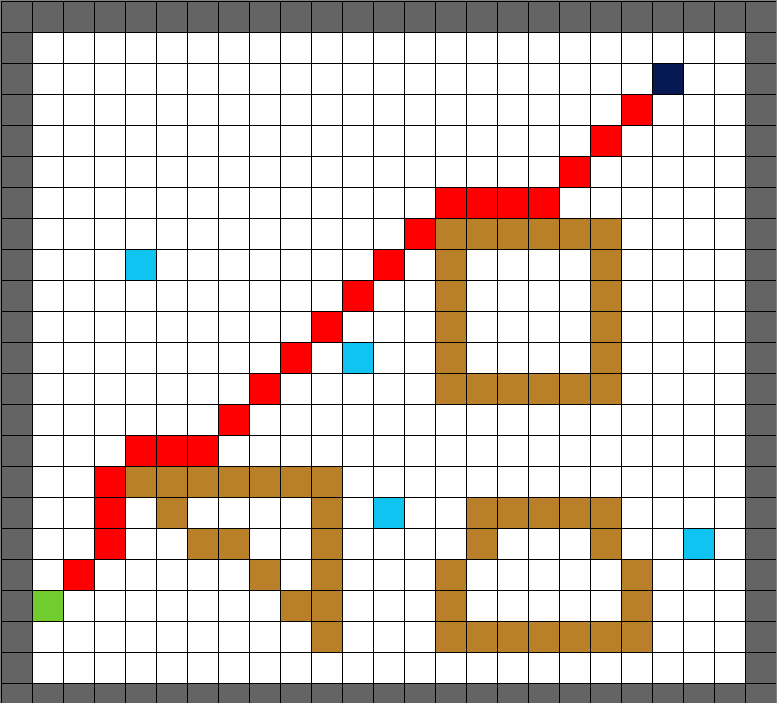
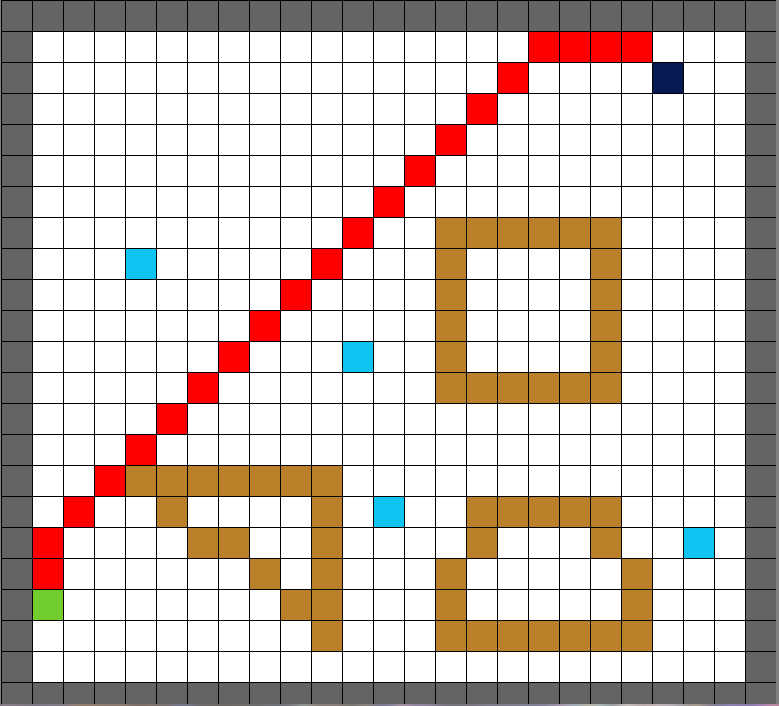
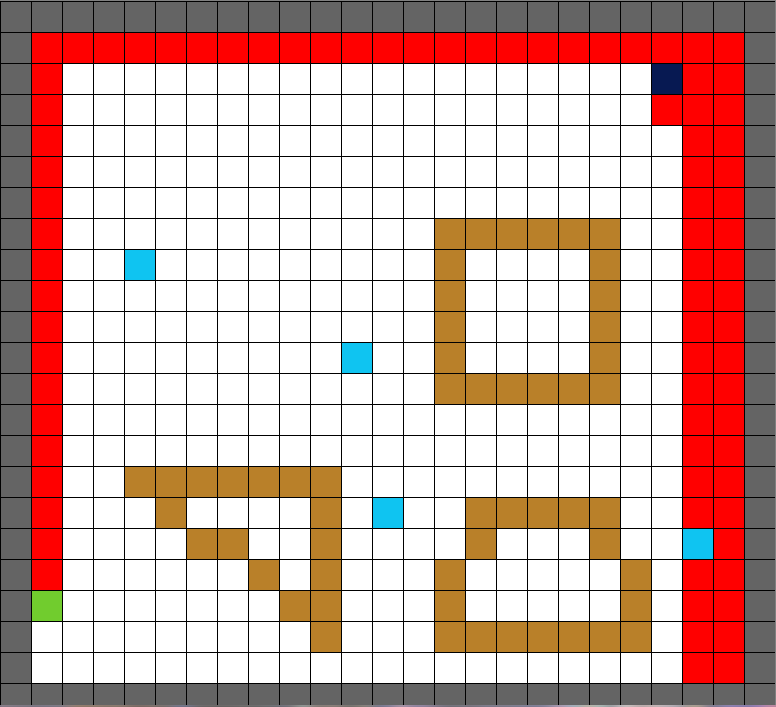
**Kết qủa khi chạy từng thuật toán (A\*, BFS, DFS) với từng map**

*Map1*

*  *

*Map2*

*  *

*Map3*

Nhận xét

Ta thấy trong các trường hợp trên, thuật toán A\* là hiệu quả nhất nhờ có hàm heuristic, sau đó là thuật toán BFS và cuối cùng là DFS

Chi tiết các thuật toán

1. **A\***

Tạo ra một tập đóng và một tâp mở (hàng đợi) để lưu các điểm, thứ tự ưu tiên gán cho một đường đi {\displaystyle x}được quyết định bởi hàm f(x) = g(x) + h(x){\displaystyle f(x)=g(x)+h(x)}. Trong đó g(x) là chi phí của đường đi cho đến thời điểm hiện tại, nghĩa là tổng trọng số của các cạnh đã đi qua. h(x) {\displaystyle h(x)}hlà hàm đánh giá heuristic về chi phí nhỏ nhất để đến đích từ x

Hàm f(x) có giá trị thấp thì độ ưu tiên càng cao

* Đầu tiên ta đưa start vào tập mở rồi lặp khi nào tập mở khác rỗng
* Lấy điểm trong tập mở có chi phí(f) thấp nhất
* Kiểm tra xem có phải là đích, đúng thì kết thúc, không thì đưa vào tập đóng
* Tìm 8 đỉnh lân cận, bằng hàm getNeighbours, nếu điểm lân cận trùng với vật cản thì bỏ qua, nếu điểm lân cận thuộc tập đóng thì bỏ qua, nếu chưa có trong tập mở thì thêm vào, nếu chi phí lơn thì bỏ qua
* Tính giá trị f
* Lặp cho đến khi gặp end(goal)

Trong thuật toán còn sử dụng các hàm như cost(self, curentPos, newPos) để tính chi phí đi từ current đến new, hàm heuristicFunc(self, pos, end) để tính heuristic từ điểm hiện tại tới đích

1. **BFS**

* Chèn đỉnh gốc vào hàng đợi (đang hướng tới)
* Lấy ra đỉnh đầu tiên trong hàng đợi và quan sát nó. Nếu đỉnh này chính là đỉnh đích, dừng quá trình tìm kiếm và trả về kết quả. Nếu không phải thì chèn tất cả các đỉnh kề với đỉnh vừa thăm nhưng chưa được quan sát trước đó vào hàng đợi.
* Nếu hàng đợi là rỗng, thì tất cả các đỉnh có thể đến được đều đã được quan sát – dừng việc tìm kiếm và trả về "không thấy".
* Nếu hàng đợi không rỗng thì quay về bước 2.
* Có một mảng parents để lưu thông tin từ đó tìm ra đc đường đi và tính chi phí

Trong thuật toán này vẫn sử dụng các hàm tìm điểm lân cận và tính chi phi đi như trong A\*

1. **DFS**

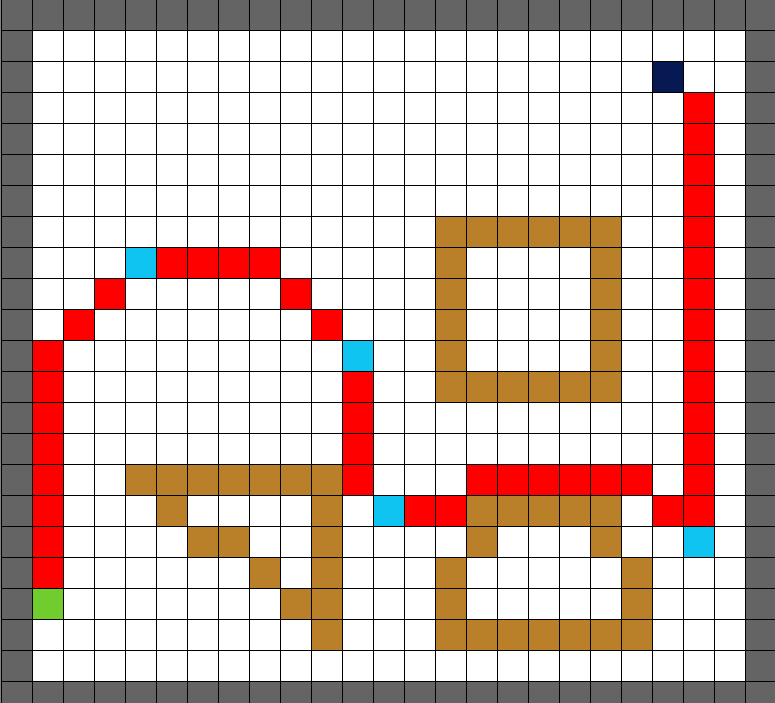
DFS sẽ lần lượt ghé thăm các đỉnh của đồ thị một cách đệ quy. Các bước trong một lần ghé thăm đỉnh current được mô tả như sau:

* Kiểm tra current có phải là địch, đúng thì kết thúc, không thì đánh dấu đã qua current
* Duyệt các đỉnh neighbour kề với current, nếu chưa qua neighbor thì gọi đệ quy để đi qua neighbor
* BFS vẫn sử dụng parent để lưu thông tin đường đi từ đó tính ra chi phí, vẫn sử dụng các hàm tìm neighbours, cost như BFS

Ngoài ra còn có hàm pointToSharp, drawLine, lineHigh, lineLow để tạo ra vật cản đầy đủ từ các đỉnh của vật cản

* **Mức 3: đi qua các điểm đón**

Ta chạy chương trình với map bất kì và tùy chọn thuật toán là 4 (Check Points), thực chất áp dụng thuật toán A\* và xử lí thêm các checkPoints



Chi tiết thuật toán:

* Đầu tiên ta sử dụng thuật toán A\* để tìm điểm gần đích nhất bằng cách tính chi phí giữa điểm Goal đến cái điểm đón (Checkpoints), nếu chi phí nào nhỏ nhất thì ta đưa điểm đó vào tập kết quả, nếu như chi phí bằng nhau thì ta lựa chọn một điểm bất kì để đưa vô tập kết quả, lần lượt ta tiếp tục tìm điểm gần nhất với điểm đón đã chọn trước đó để đưa vô tập kết quả - đây chính là cách hàm FindPriorityPoint() hoạt động, và kết quả trả về một tập hợp các điểm lần lượt ưu tiên đi trước.
* Sau khi tìm được thứ tự các điểm đón đi qua thì ta lại sử dụng thuật toán A\* để tìm đường đi ngắn nhất giữa điểm Start đến điểm đón đầu tiên, điểm đón đầu tiên đến điểm đón thứ 2,… lặp lại quá trình đến điểm đón cuối cùng, kết quả ta thu được đường đi ngắn nhất đi từ điểm Start qua các điểm đón đến điểm Goal.

1. **Tài liệu tham khảo**

Thuật toán A\*

[https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i\_thu%E1%BA%ADt\_t%C3%ACm\_ki%E1%BA%BFm\_A\*](https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_t%C3%ACm_ki%E1%BA%BFm_A*)

<https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2>

<https://github.com/tiendung1510/LearningPython>

Thuật toán BFS

<https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_r%E1%BB%99ng>

Thuật toán DFS

<https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u>

OpenCV

<https://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html?fbclid=IwAR1vRNFMzCYti0D1v6BeFjCsvKcF40m3QytbJbZAPnJXDR_I_KPq6oYLbcs>

Thuật toán tìm tọa độ các điểm trên cạnh từ các đỉnh cho trước

<https://github.com/tursilion/libti99/blob/master/vdp_bmdrawline.c>