KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
227 Nguyễn Văn Cừ, Phường 4, Quận 5, TP.HCM
Điện Thoại: (08) 38.354.266 - Fax:(08) 38.350.096

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

BÀI TẬP CÁ NHÂN
MÔN CƠ SỞ TRÍ TUỆ NHÂN TẠO

ĐỀ TÀI:

NGHIÊN CỨU VỀ THUẬT TOÁN TÌM KIẾM ĐỒ THỊ

Giảng viên phụ trách : **Lê Hoài Bắc**
Nguyễn Bảo Long

Họ và tên sinh viên : **Phan Quốc Huy**
Mã số sinh viên : **21120082**
Lớp : **21_22**

Thành phố Hồ Chí Minh – tháng 10 năm 2023

Mục lục

I. GIỚI THIỆU

Tìm kiếm trên đồ thị là một trong những bài toán khó và quan trọng trong Lý thuyết đồ thị, lĩnh vực Toán học và Khoa học máy tính. Với yêu cầu tìm kiếm và duyệt các đỉnh hoặc cạnh trong đồ thị, nó giúp ta giải quyết các vấn đề phức tạp liên quan đến các mối quan hệ giữa các đối tượng và hỗ trợ tối ưu hóa trong đồ thị. Vì vậy, việc xây dựng một thuật toán đồ thị một cách hệ thống và tối ưu đã luôn thu hút sự quan tâm của nhiều nhà nghiên cứu trên nhiều lĩnh vực.

Ở bài viết này, ta sẽ tập trung nghiên cứu 4 thuật toán quan trọng được sử dụng rộng rãi hiện nay, bao gồm: *DFS, BFS, UCS và AStar*. Ngoài ra, một số thuật toán được đề cập khác bao gồm: *Greedy Search, Dijsktra.*

II. PHÂN LOẠI

Có nhiều cách phân loại các thuật toán tìm kiếm trên đồ thị, trong đó phổ biến hơn cả ta chia ra làm hai loại: *Uninformed Search (Blind Search) và Informed Search (Heuristic Search).*

1. Uninformed Search

Các thuật toán thuộc phân loại này có đặc điểm nổi bật là không chứa bất cứ dữ liệu hay thông tin bổ sung nào về cấu trúc của đồ thị hay vị trí của đối tượng đang xét đối với mục tiêu tìm kiếm. Thay vào đó, nó chỉ tiến hành tìm kiếm dựa trên thông tin cơ bản được cung cấp từ yêu cầu đặt ra.

Loại thuật toán này hoạt động bằng cách duyệt qua tất cả các nút trong đồ thị một cách tổng quát mà không xét đến độ ưu tiên cho các nút đó. Do đó, chúng cần tốn nhiều tài nguyên, thời gian và số lần thử hơn để tìm được tới mục tiêu.

Tuy vậy, *Uninformed Search* luôn đảm bảo duyệt qua tất cả các vị trí nút trong đồ thị và luôn hoàn thành nhiệm vụ tìm kiếm nếu giải giải pháp tìm kiếm cho bài toán đó có tồn tại.

Một số thuật toán tiêu biểu bao gồm: *DFS, BFS, UCS, DLS, ...*

2. Informed Search

Trái ngược với *Uninformed Search*, thuật toán *Informed Search* sử dụng thêm thông

tin đặc biệt để hỗ trợ quá trình tìm kiếm mục tiêu. Chẳng hạn như còn cách mục tiêu bao xa, chi phí đường dẫn, độ ưu tiên, cách tiếp cận đến mục tiêu, … Nhờ vậy, quá trình tìm kiếm và tài nguyên sử dụng sẽ được tối ưu hơn thuật toán *Uninformed Search.*

Thuật toán *Informed Search* sử dụng một hàm số gọi là *Heuristic*, hàm này được sử dụng để ước tính khoảng cách hoặc chi phí tìm kiếm cho các khả năng ở hiện tại và chọn ra kết quả tối ưu nhất. Đây là một hàm rất quan trọng trong thuật toán *Informed Search* vì nó quyết định đến việc thuật toán có tìm được mục tiêu hay không. Cho nên, việc xác định một hàm *Heuristic* đủ tốt và thông minh rất quan trọng, nếu có sai sót trong hàm *Heuristic*, nó có thể dẫn đến việc thuật toán bỏ sót các khả năng và giải pháp, từ đó dẫn đến không hoàn thành nhiệm vụ tìm kiếm.

Một số thuật toán tiêu biểu bao gồm: *AStar, Best-First Search, Greedy Search…*

❖ So sánh Uninformed Search và Informed Search:

Tiêu chí	Informed Search	Uninformed Search
Sử dụng dữ liệu bổ sung	Sử dụng dữ liệu bổ sung cho quá trình tìm kiếm.	Không sử dụng dữ liệu bổ sung cho quá trình tìm kiếm.
Hiệu suất	Nhanh hơn.	Chậm hơn.
Tính hoàn chỉnh	Luôn hoàn chỉnh.	Có thể hoàn chỉnh hoặc không.
Chi phí	Thấp hơn.	Cao hơn.
Thời gian chạy	Ít hơn.	Nhiều hơn.
Hỗ trợ xác định phương hướng	Được hỗ trợ xác định phương hướng tối ưu.	Không được hỗ trợ xác định phương hướng tối ưu.

Độ phức tạp	Khó hơn.	Dễ hơn.
Tính hiệu quả	Hiệu quả hơn về chi phí và hiệu suất. Chi phí phát sinh ít hơn và tốc độ tìm kiếm nhanh hơn.	Tương đối kém hiệu quả hơn vì chi phí phát sinh cao hơn và tốc độ tìm kiếm còn chậm.
Yêu cầu tính toán	Thấp hơn.	Cao hơn.
Kích thước của bài toán	Có phạm vi rộng về mặt xử lý các vấn đề tìm kiếm lớn.	Khó giải quyết được các bài toán lớn.
Ví dụ các thuật toán	Greedy Search A* Search AO* Search Hill Climbing Algorithm	Depth First Search (DFS) Breadth First Search (BFS) Branch and Bound

III. CÁC THUẬT TOÁN TIÊU BIỂU

1. Thuật toán Depth-First Search (DFS)

1.1. Giới thiệu

Thuật toán *Depth-First Search - Tìm kiếm theo chiều sâu*, là một thuật toán tìm kiếm trên đồ thị dựa trên việc duyệt qua hết 1 nhánh các đỉnh theo chiều sâu trước khi quay lại kiểm tra các nhanh khác.

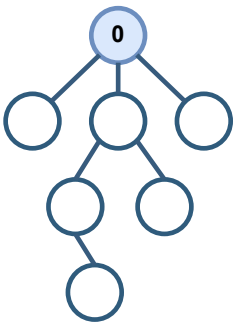
1.2. Cách thức hoạt động

- Thuật toán bắt đầu từ đỉnh gốc, tiếp tục đi đến các đỉnh kề theo một nhánh bằng các cạnh của đồ thị. Quá trình này lặp lại cho tới khi tìm thấy mục tiêu hoặc tất cả các đỉnh đều được duyệt qua.
- Nếu tại một đỉnh nào đó không còn đỉnh nào kề chưa duyệt qua nữa thì quay trở lại tiếp tục tìm con đường khác của đồ thị.
- Thuật toán sử dụng cấu trúc dữ liệu *ngăn xếp (stack)* lưu trữ các đỉnh đã duyệt qua để đảm bảo các đỉnh chỉ được duyệt một lần duy nhất.

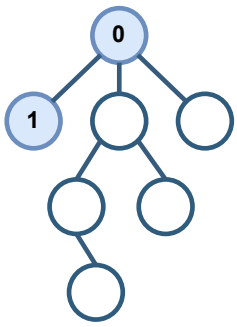
1.3. Mã giả

```
DFS(graph, start, goal):
    create stack S
    create set to track visited vertices
    push start in S
    mark start as visited

    while S is not empty
        current = S.pop()

        if current is goal:
            return "Goal found"

        for each neighbor of current:
            if neighbor is not visited:
                mark neighbor as visited
                push neighbor in S

    return "Goal not found"
```

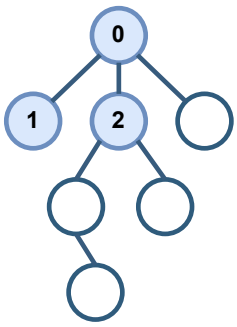
1.4. Minh họa



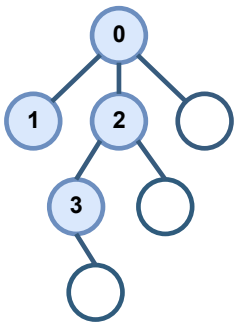
Hình 1.0



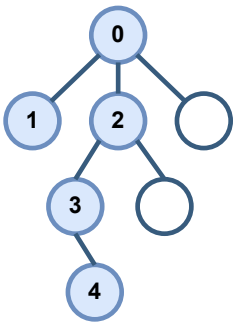
Hình 1.1



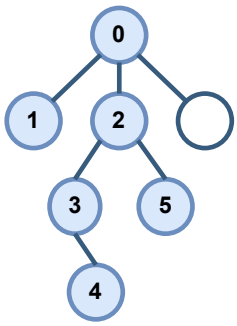
Hình 1.2



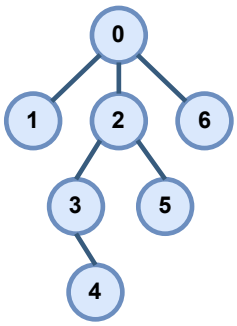
Ảnh 1.3



Hình 1.4



Hình 1.5



Ảnh 1.6

Hình 1. Minh họa cho thuật toán DFS

1.5. Phân tích

1.5.1. Độ hoàn chỉnh (Completeness)

Trong phân tích giải thuật, dữ liệu đầu vào cho các thuật toán này thường được giả định là một đồ thị hữu hạn, nghĩa là đồ thị bao gồm một tập hữu hạn các đỉnh và cạnh. Tuy nhiên, trong lĩnh vực Trí tuệ nhân tạo, dữ liệu đầu vào thường được đại diện bằng một đồ thị vô hạn.

Do đó, *DFS* được coi là một thuật toán không hoàn chỉnh khi xét trong đồ thị vô hạn, nó không đảm bảo tìm thấy mục tiêu và có thể mắc kẹt trong một phần hay một nhánh của đồ thị và không bao giờ quay trở lại.

1.5.2. Độ tối ưu (Optimality)

Thuật toán DFS không đảm bảo tính tối ưu nếu bài toán tìm kiếm có nhiều giải pháp. Lí do là vì thuật toán này thường đi sâu vào một nhánh của đồ thị mà nó gặp đầu tiên trước khi quay lại và kiểm tra các nhánh khác.

Điều này dẫn đến thuật toán DFS tuy có thể tìm ra được giải pháp sớm, nhưng không đảm bảo đó là giải pháp tối ưu nhất nếu như nó nằm ở một nhánh khác trong đồ thị mà DFS chưa kiểm tra.

1.5.3. Độ phức tạp (Complexity)

- **Độ phức tạp Thời gian (Time Complexity)** của thuật toán DFS phụ thuộc vào cấu trúc và kích thước của đồ thị. Trong trường hợp xấu nhất, độ phức tạp là $O(V + E)$ đối với đồ thị biểu diễn dưới dạng danh sách kề, và $O(V^2)$ dưới dạng ma trận kề, trong đó V là số đỉnh và E là số cạnh trong đồ thị.

- **Độ phức tạp Không gian (Space Complexity)** của DFS là $O(V)$ trong trường hợp xấu nhất, trong đó V là số đỉnh của đồ thị. Vì DFS phải lưu trữ tất cả các đỉnh trong không gian lưu trữ tạm thời là ngăn xếp (stack) trước khi quay lại và kiểm tra các đỉnh khác.

2. Thuật toán Breadth-First Search (BFS)

2.1. Giới thiệu

Khá tương đồng với *DFS*, thuật toán *Breadth-First Search - Tìm kiếm theo chiều rộng* là một thuật toán tìm kiếm trên đồ thị dựa trên việc duyệt qua hết 1 nhánh các đỉnh theo chiều sâu trước khi quay lại kiểm tra các nhanh khác.

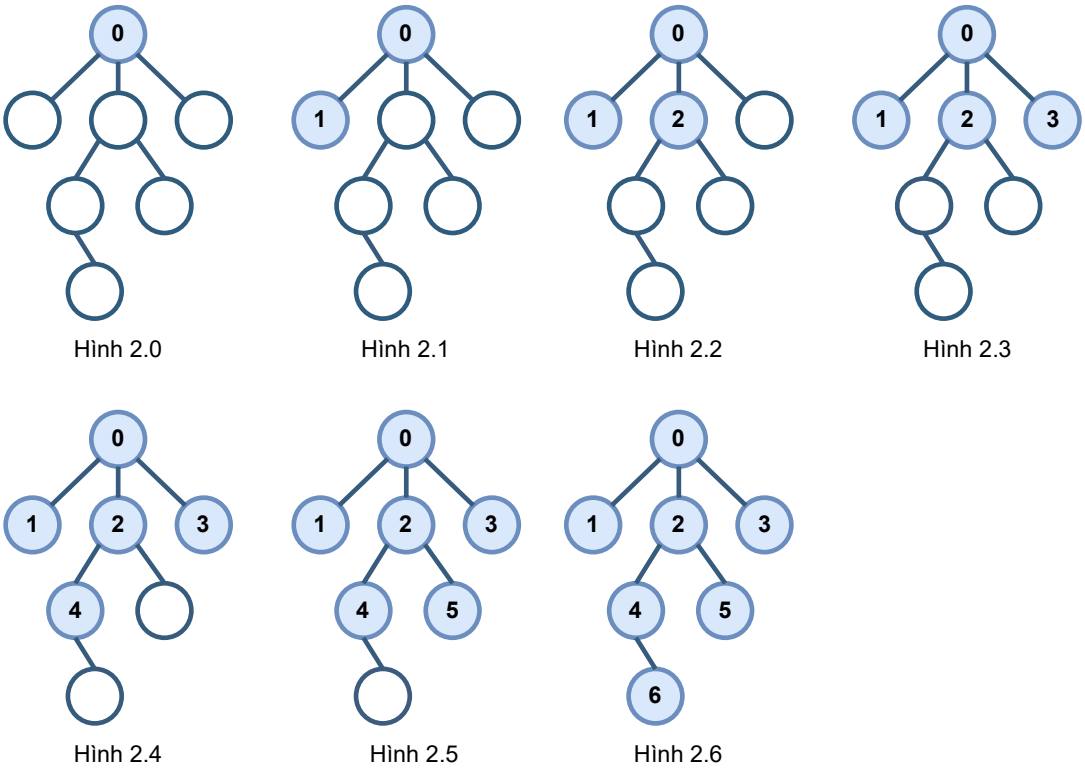
2.2. Cách thức hoạt động

- Thuật toán này viếng thăm các đỉnh theo thứ tự tăng dần *mức (level)* của đỉnh.
- Từ đỉnh đầu, thuật toán đi qua các đỉnh mức 1 sau đó đến các đỉnh mức 2… Quá trình này được tiếp tục cho đến khi tất cả các đỉnh đã được viếng thăm.
- Thuật toán sử dụng cấu trúc dữ liệu *hàng đợi (queue)* lưu trữ các đỉnh cùng mức trước khi đi tới các mức tiếp theo. Hơn nữa, hàng đợi lưu trữ các đỉnh đã viếng thăm, đảm bảo các đỉnh chỉ được duyệt một lần duy nhất.

2.3. Mã giả

```
BFS(graph, start, goal):
    create queue Q
    create set to track visited vertices
    enqueue start into Q
    mark start as visited

    while Q is not empty:
        current = dequeue from Q

        if current is goal:
            return "Goal found"

        for each neighbor of current:
            if neighbor is not visited:
                enqueue neighbor into Q
                mark neighbor as visited

    return "Goal not found"
```

2.4. Minh họa



Hình 2. Minh họa cho thuật toán BFS

2.5. Phân tích

2.5.1. Độ hoàn chỉnh (Completeness)

Trái ngược với *DFS*, thuật toán *BFS* được coi là một thuật toán hoàn chỉnh trong bất kì đồ thị nào, kể cả đồ thị vô hạn. Vì thuật toán này sẽ duyệt qua và kiểm tra tất cả các đỉnh trên cùng một mức rồi mới chuyển sang mức tiếp theo. Do vậy, nó đảm bảo luôn tìm thấy mục tiêu và cuối cùng sẽ tìm thấy mục tiêu nếu tồn tại trong đồ thị.

2.5.2. Độ tối ưu (Optimality)

Thuật toán *BFS* được xem là tối ưu khi và chỉ khi nó duyệt qua đồ thị không trọng số. Trường hợp đặc biệt thuật toán *BFS* được xem là tối ưu là khi toàn bộ trọng số trong đồ thị mà nó duyệt qua đều tương đương.

2.5.3. Độ phức tạp (Complexity)

❖ **Độ phức tạp Thời gian (Time Complexity)** của thuật toán *BFS* cũng phụ

thuộc vào cấu trúc của đồ thị. Trong trường hợp xấu nhất, độ phức tạp là $O(V + E)$, trong đó V là số đỉnh và E là số cạnh trong đồ thị.

- **Độ phức tạp Không gian (Space Complexity)** của *BFS* trong trường hợp xấu nhất có thể lên đến $O(V)$ với V là số đỉnh trong đồ thị. Trong trường hợp này, *BFS* có thể phải lưu trữ tất cả các đỉnh trong không gian lưu trữ hàng đợi tạm thời (queue).

3. Thuật toán Uniform Cost Search (UCS)

3.1. Giới thiệu

Thuật toán *Uniform Cost Search - Tìm kiếm chi phí đều*, là một thuật toán duyệt hay tìm kiếm trên một cây hoặc đồ thị có trọng số. Việc tìm kiếm bắt đầu tại đỉnh gốc và tiếp tục bằng cách duyệt các đỉnh tiếp theo sao cho tính từ đỉnh gốc, tổng trọng số hay chi phí của đường đi là thấp nhất.

3.2. Cách thức hoạt động

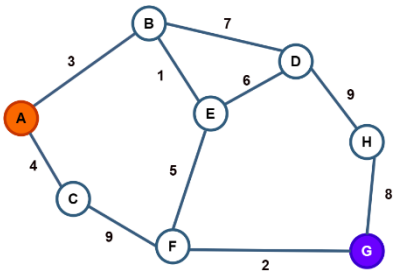
- Thuật toán này viếng thăm các đỉnh theo thứ tự ưu tiên *trọng số (chi phí) tích lũy* nhỏ nhất từ đỉnh gốc đến mỗi đỉnh, đảm bảo tìm ra đường đi có tổng chi phí thấp nhất mà không quan tâm đến số bước phải đi.
- Thuật toán sử dụng cấu trúc dữ liệu *hàng đợi ưu tiên (priority queue)* lưu trữ các đỉnh với độ ưu tiên là trọng số của cạnh nối với đỉnh kế tiếp trên đường đi.

3.3. Mã giả

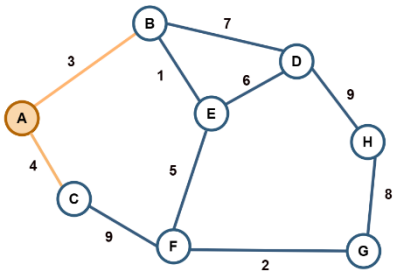
```
UCS(graph, start, goal):
    create priority queue PQ
    create dictionary cost to track the cost of reaching each vertex
    create set to track visited vertices

    enqueue start into PQ with cost 0
    set cost of start to 0

    while PQ is not empty:
        current, current_cost = dequeue from PQ

        if current is goal:
```

```
            return "Goal found"

        if current_cost > cost[current]:
            -- Skip if there's a cheaper path to this current vertex
            continue

        for each neighbor, edge_cost in neighbors(current):
            total_cost = current_cost + edge_cost

            if neighbor is not visited or total_cost < cost[neighbor]:
                enqueue neighbor into PQ with total_cost
                set cost[neighbor] to total_cost
                mark neighbor as visited

    return "Goal not found"
```

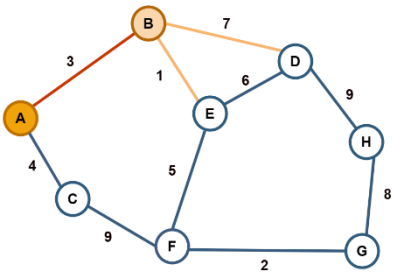
3.4. Minh họa



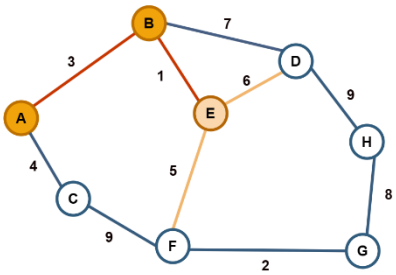
Hình 3.1



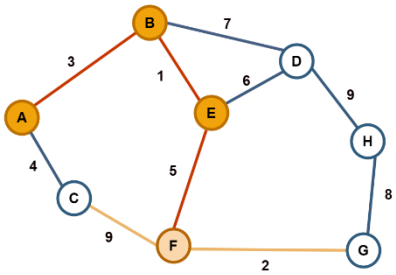
Hình 3.2



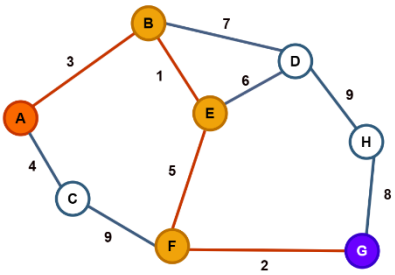
Hình 3.3



Hình 3.4



Hình 3.5



Hình 3.6

Hình 3. Minh họa cho thuật toán UCS

3.5. Phân tích

3.5.1. Độ hoàn chỉnh (Completeness)

Thuật toán *UCS* đảm bảo tính hoàn chỉnh trong đồ thị có trọng số khi và chỉ khi mọi bước di chuyển có chi phí lớn hơn một giá trị hằng số dương ϵ.

Hằng số ϵ là một giá trị dương rất nhỏ, được sử dụng để biểu diễn giới hạn

chi phí tối thiểu cho một bước đi trong các thuật toán như *UCS*. Nó đảm bảo con đường tìm ra là hoàn chỉnh khi mọi bước trên đó có chi phí không nhỏ hơn ϵ, hay còn có nghĩa là không bước nào có chi phí bằng 0 hoặc gần bằng 0.

3.5.2. Độ tối ưu (Optimality)

Thuật toán *UCS* hoàn toàn tối ưu trong việc tìm kiếm đường đi ngắn nhất. Nó luôn duyệt qua các đỉnh ưu tiên theo chi phí thấp nhất nên đảm bảo không bỏ sót đường đi tối ưu nào nếu nó tồn tại.

3.5.3. Độ phức tạp (Complexity)

- **Độ phức tạp Thời gian (Time Complexity)** của thuật toán *UCS* trong trường hợp xấu nhất là $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, trong đó b là hệ số phân nhánh của cây, C^* là tổng chi phí của đường đi tối ưu và ϵ được giả sử là chi phí tối thiểu cho mỗi bước đi.

- **Độ phức tạp Không gian (Space Complexity)** của *UCS* cũng tương tự như trên, với $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$ trong trường hợp xấu nhất.

4. Thuật toán AStar (A*)

4.1. Giới thiệu

*A** là một thuật toán tìm kiếm trên đồ thị thuộc phân loại *Informed Search*. Thuật toán này tìm một đường đi từ đỉnh khởi đầu đến đỉnh mục tiêu sao cho quãng đường là ngắn nhất.

*A** sử dụng một hàm *heuristic* để đánh giá đường đi tốt nhất có thể đi. Điều này làm cho *A** trở nên "thông minh" hơn nhiều so với các thuật toán *Uninformed Search* khác. Vì vậy mà thuật toán này được sử dụng rộng rãi trong lĩnh vực trí tuệ nhân tạo, giải thuật tìm kiếm và ứng dụng tìm đường đi trong bản đồ.

4.2. Cách thức hoạt động

Từ đỉnh xuất phát, *A** xây dựng tất cả các đường đi dùng hàm ước lược khoảng cách *heuristic* để đánh giá đường đi tốt nhất có thể đi. Tùy theo mỗi dạng bài khác nhau mà hàm *heuristic* sẽ được đánh giá khác nhau. *A** luôn tìm được đường đi ngắn nhất

nếu đường đi đó tồn tại trong đồ thị.

Thuật toán sử dụng cấu trúc dữ liệu *hàng đợi ưu tiên (priority queue)* lưu trữ các đỉnh với độ ưu tiên dựa trên hàm đánh giá $f(x) = g(x) + h(x)$. Trong đó, $f(x)$ được tính bằng tổng chi phí của đường đi tính từ đỉnh xuất phát đến đỉnh x hiện tại ($g(x)$) và giá trị của hàm *heuristic* ước lượng chi phí từ đỉnh x hiện tại đến đỉnh mục tiêu.

4.3. Một số hàm Heuristic

- **Khoảng cách Manhattan (Manhattan Distance)** là khoảng cách giữa 2 điểm trong không gian Euclid với hệ tọa độ Descartes. Khoảng cách này được tính bằng tổng của độ lệch theo chiều ngang và chiều dọc giữa 2 điểm đó. Phổ biến trong tìm kiếm trên bản đồ ô lưới, bàn cờ… với giới hạn 4 hướng di chuyển.

- **Khoảng cách Euclidean (Euclidean Distance)** cũng tương tự như *Manhattan,* được tính bằng căn bậc hai tổng bình phương độ lệch theo chiều ngang và chiều dọc giữa 2 điểm đó. Phù hợp trong tìm kiếm trong không gian 2D hoặc 3D với khả năng di chuyển theo bất cứ hướng nào.

- **Khoảng cách đường chéo (Diagonal Distance)** là khoảng cách giữa 2 điểm sử dụng độ chênh lệch lớn nhất giữa chiều ngang và chiều dọc 2 điểm đó.

  – Nó được sử dụng trong đồ thị di chuyển 8 hướng với chi phí đường chéo với đường thẳng là khác nhau. Trong không gian 2D, khoảng cách trên được tính bằng công thức sau với D và D2 lần lượt là chi phí cho đường thẳng và đường chéo:

```
def heuristic(node) =
dx = abs(node.x - goal.x)
dy = abs(node.y - goal.y)
return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

  – Khi $D = 1$ và $D2 = 1$, khoảng cách trên được gọi là *Khoảng cách Chebyshe.*

  – Khi $D = 1$ và $D2 = \sqrt{2}$, khoảng cách trên được gọi là *Khoảng cách Octile.*

4.4. Tính thu nạp được (Admissible)

Hàm *heuristic* có tính chất thu nạp được nghĩa là nó không bao giờ đánh giá cao hơn chi phí nhỏ nhất thực sự của việc tìm được tới mục tiêu. Một thuật toán A* với hàm *heuristic* có tính chất thu nạp được đảm bảo rằng nó sẽ tìm thấy một đường đi tối ưu nếu như nó tồn tại.

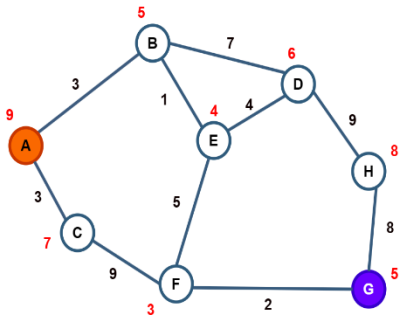
Phát biểu một cách hình thức, với mọi nút x, y trong đó y là nút tiếp theo của x:

$$h(x) \leq g(y) - g(x) + h(y)$$

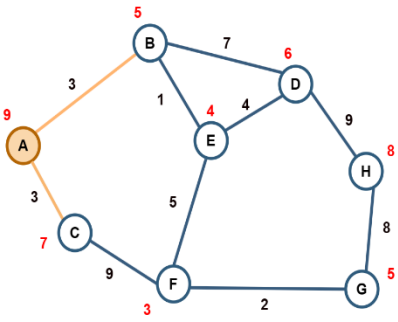
4.5. Mã giả

```
AStar(graph, start, goal, heuristic):
    create priority queue PQ
    create dictionary to track the cost of reaching each vertex
    create set to track visited vertices

    enqueue start into PQ with cost 0 + heuristic(start, goal)
    set cost of start to 0

    while PQ is not empty:
        current, current_cost = dequeue from PQ

        if current is goal:
            return "Goal found"

        if current_cost > cost[current]:
            -- Skip if there's a cheaper path to this current vertex
            continue

        for each neighbor, edge_cost in neighbors(current):
            total_cost = cost[current] + edge_cost

            if neighbor is not visited or total_cost < cost[neighbor]:
                enqueue neighbor into PQ with total_cost
                + heuristic(neighbor, goal)

                set cost[neighbor] to total_cost
                mark neighbor as visited

    return "Goal not found"
```

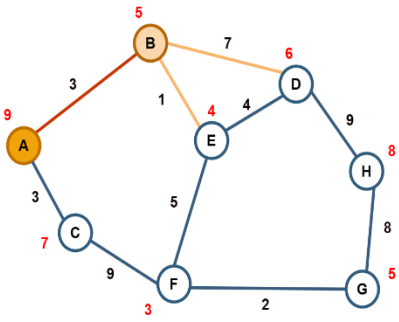
4.6. Minh họa



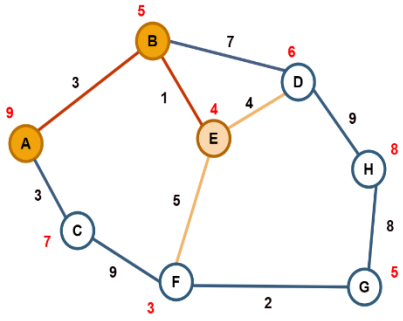
Hình 4.1



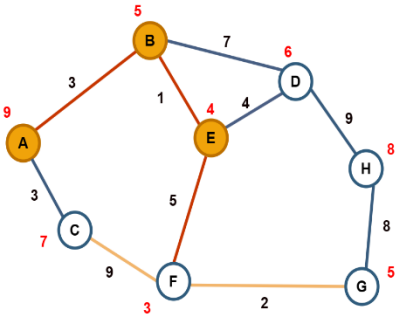
Hình 4.2



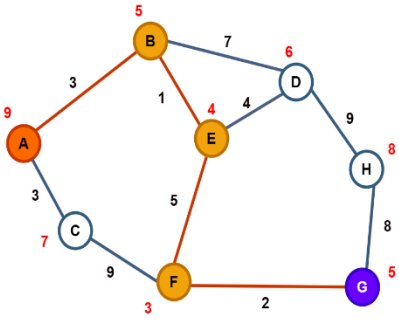
Hình 4.3



Hình 4.4



Hình 4.5



Hình 4.6

*Hình 4. Minh họa cho thuật toán A**

4.7. Phân tích

4.7.1. Độ hoàn chỉnh (Completeness)

Tương tự *UCS*, thuật toán *A** cũng đảm bảo tính hoàn chỉnh, nó sẽ luôn luôn tìm thấy lời giải nếu tồn tại. Với điều kiện là chi phí cho mỗi bước đi lớn hơn hoặc bằng một hằng số dương ϵ và phải có hữu hạn các đỉnh kề, cũng như chi phí cho các đường đi không là giá trị âm.

4.7.2. Độ tối ưu (Optimality)

Thuật toán *A** hoàn toàn tối ưu trong việc tìm kiếm đường đi ngắn nhất. Tính tối ưu của thuật toán phụ thuộc nhiều vào hàm *heuristic*, nếu hàm *heuristic* mang tính chất thu nạp được (admissible), nghĩa là nó không bao giờ vượt quá chi phí nhỏ nhất thực sự của việc đi tới đỉnh mục tiêu, thì *A** sẽ tìm ra được đường đi tối ưu nhất có thể.

4.7.3. Độ phức tạp (Complexity)

- **Độ phức tạp Thời gian (Time Complexity)** của *A** phụ thuộc nhiều vào hàm *heurisitc*. Xét trên lĩnh vực *Trí tuệ nhân tạo*, không gian tìm kiếm cho thuật toán là một cây vô hạn với hệ số phân nhánh b, vì vậy độ phức tạp thời gian cho trường hợp xấu nhất là $O(b^d)$ với d chính là độ sâu của đường đi tối ưu.

- **Độ phức tạp Không gian (Space Complexity)** của A* cũng tương tự như trên, với $O(b^d)$ trong trường hợp xấu nhất.

IV. SO SÁNH CÁC THUẬT TOÁN

1. UCS, Greedy Search & A*

UCS, Greedy và A* đều là các thuật toán tìm đường đi ngắn nhất trong đồ thị có trọng số. Cả ba đều sử dụng một tập hợp các đỉnh mở (open set) để theo dõi các đỉnh đã được thăm và tập hợp các đỉnh đóng (closed set) để theo dõi các đỉnh chưa được thăm. Tuy nhiên, cả ba đều có một số điểm khác biệt sau:

Tiêu chí	UCS	Greedy Search	A*
Bản chất	Tìm đường đi với tổng chi phí tối thiểu tới đỉnh mục tiêu cụ thể.	Chọn đỉnh gần mục tiêu nhất tại thời điểm đang xét mà không quan tâm đến độ tối ưu của đường đi.	Sự kết hợp giữa UCS và Greedy Search.
Ưu tiên	Dựa trên chi phí tích lũy đến đỉnh gần nhất.	Dựa trên ước tính khoảng cách đến mục tiêu ngắn nhất.	Dựa trên cả chi phí tích lũy và ước tính khoảng cách nhỏ nhất đến mục tiêu.
Tính tối ưu	Đảm bảo tìm đường đi ngắn nhất nhưng không đảm bảo số bước ít nhất.	Tìm được đường đi nhanh nhất nhưng không đảm bảo ngắn nhất.	Đảm bảo tìm đường đi tối ưu vì kết hợp cả 2 tính chất của UCS và Greedy Search.
Sử dụng hàm	Sử dụng chi phí thực tế đến từ đỉnh hiện tại.	Sử dụng hàm heuristic.	Sử dụng cả chi phí thực tế và hàm heuristic.

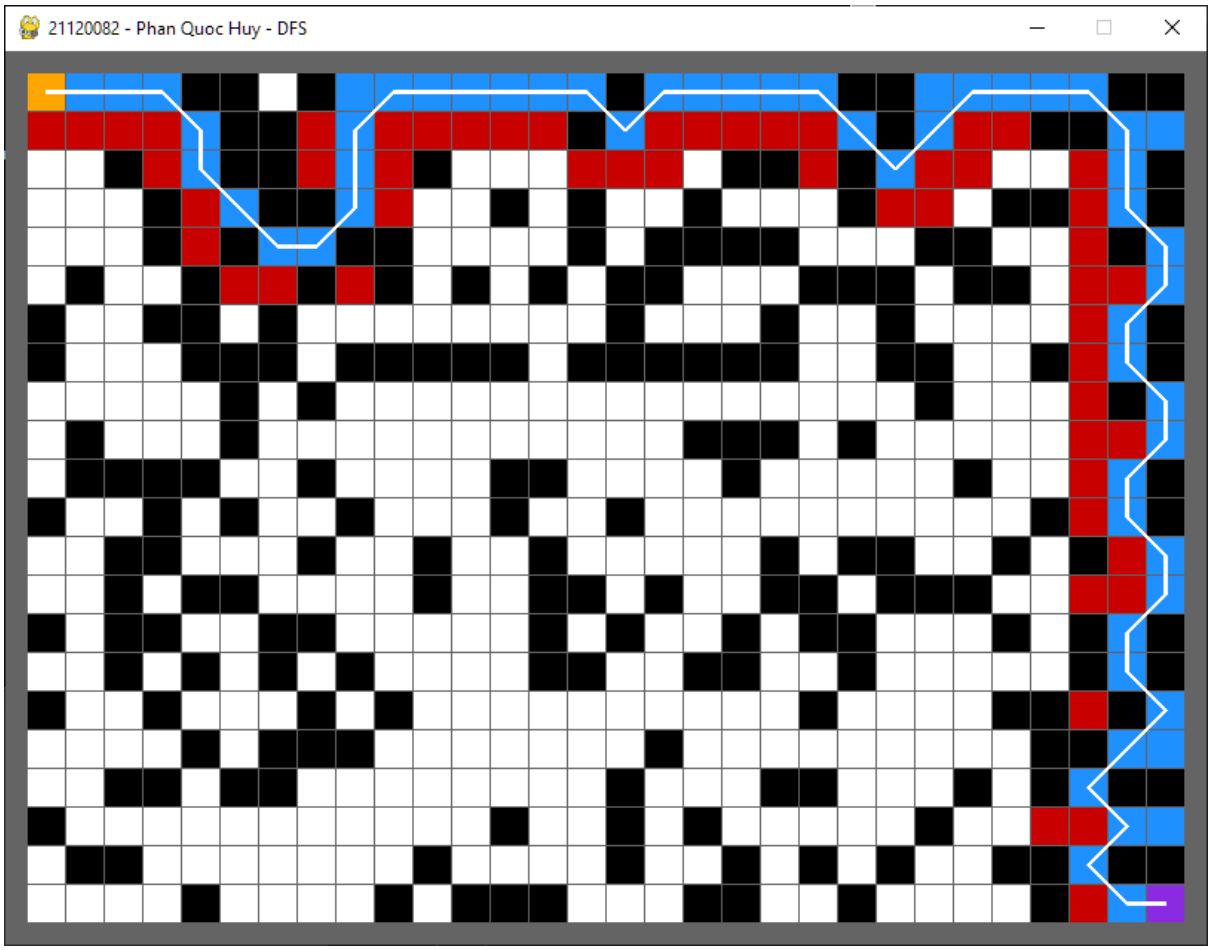
2. UCS & Dijkstra

UCS và Dijkstra đều là 2 thuật toán tìm đường đi ngắn nhất trong đồ thị có trọng số. Cả hai đều bắt đầu từ đỉnh xuất phát và mở rộng đến các đỉnh có chi phí thấp nhất tính từ đỉnh ban đầu. Tuy nhiên, cả hai đều có một số điểm khác biệt sau:

Tiêu chí	UCS	Dijkstra
Bản chất	Tìm đường đi với tổng chi phí tối thiểu tới đỉnh mục tiêu cụ thể.	Tìm đường đi ngắn nhất từ đỉnh gốc đến tất cả các đỉnh trong đồ thị.
Yêu cầu bộ nhớ	Ít hơn.	Nhiều hơn.
Cách sử dụng bộ nhớ	Chỉ lưu các đỉnh neighbor của đỉnh đang xét trong suốt quá trình tìm kiếm.	Lưu tất cả các đỉnh ngay từ ban đầu.
Điều kiện dừng	Khi đỉnh mục tiêu được tìm thấy hoặc hàng đợi ưu tiên chứa các đỉnh neighbor rỗng.	Khi tất cả các đường ngắn nhất đến mọi đỉnh đều được tìm thấy.
Thời gian chạy	Ít hơn.	Nhiều hơn.
Tính ứng dụng	Thường được sử dụng trên cấu trúc cây.	Thường được sử dụng trên đồ thị tổng quát.
Độ phức tạp thời gian	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O((E+V)\log V)$
Độ phức tạp không gian	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(E+V)$

V. CÀI ĐẶT

1. DFS



Cách thức hoạt động của thuật toán DFS là duyệt đồ thị theo chiều sâu, bắt đầu từ đỉnh xuất phát và đi sâu nhất có thể theo các cạnh của đồ thị. Khi không thể đi sâu hơn nữa, thuật toán sẽ quay trở lại đỉnh trước đó và tiếp tục quá trình duyệt cho tới khi tìm được đỉnh mục tiêu.

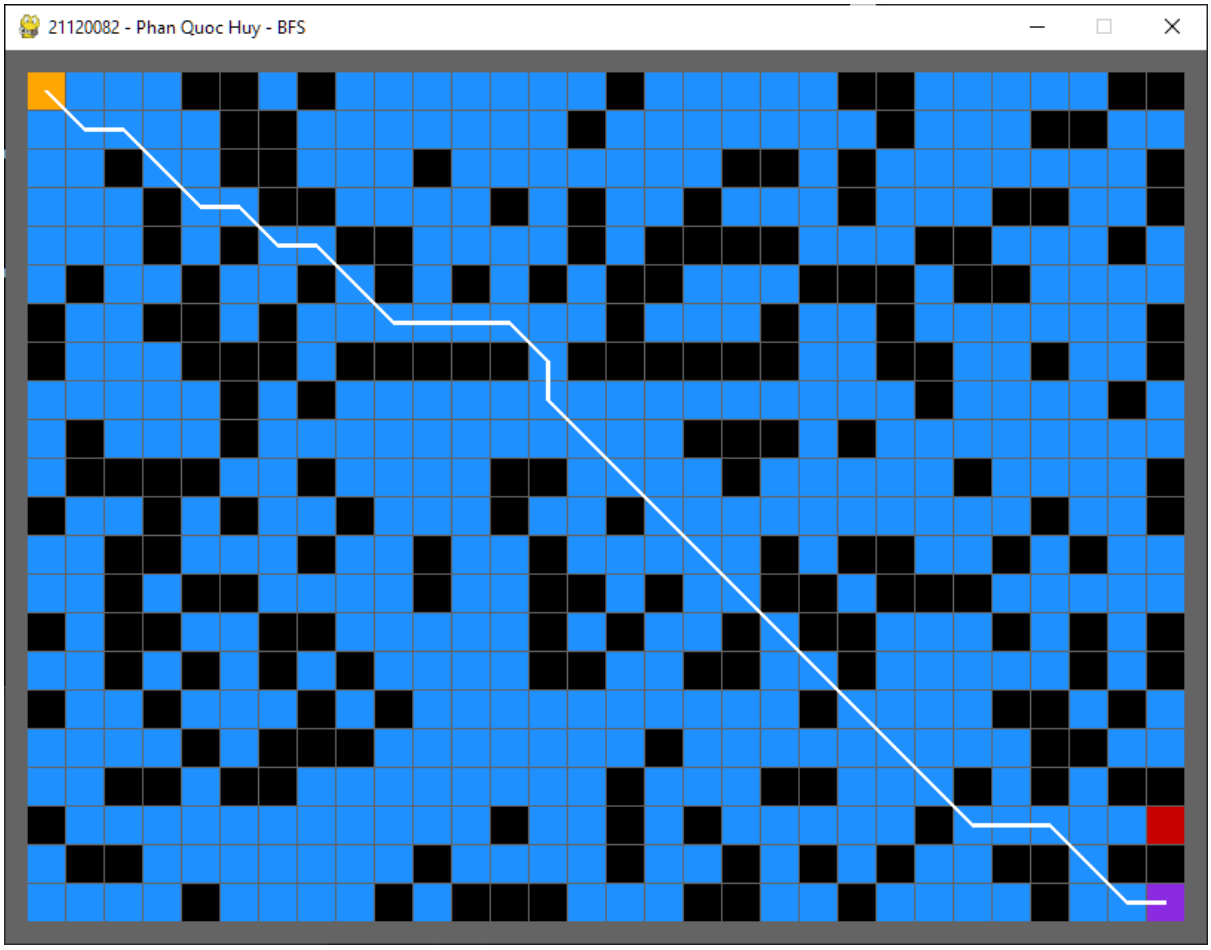
Quá trình tìm kiếm được mô tả như sau:

1. Khởi tạo hai tập dữ liệu:
    - open_set: tập dữ liệu chứa các đỉnh có thể được thăm.
    - closed_set: tập dữ liệu chứa các đỉnh đã được thăm.
2. Thêm đỉnh xuất phát vào open_set.
3. Lặp lại các bước sau cho đến khi tìm thấy mục tiêu hoặc tập open_set trống:
    - Lấy đỉnh hiện tại ra khỏi open_set và thêm vào closed_set.

- Nếu đỉnh hiện tại là đỉnh mục tiêu, thì kết thúc thuật toán.
- Duyệt qua tất cả các đỉnh lân cận `neighbor` của đỉnh hiện tại.
- Nếu một đỉnh lân cận chưa có trong open_set và closed_set, thì thêm nó vào open_set.

Kết quả của thuật toán này cho ta một đường đi từ đỉnh xuất phát đến đỉnh mục tiêu, đường đi này. DFS tuy có thể tìm ra được giải pháp sớm, nhưng không đảm bảo đó là giải pháp tối ưu nhất nếu như nó nằm ở một nhánh khác trong đồ thị mà DFS chưa kiểm tra.

2. BFS



Quá trình tìm kiếm của thuật toán BFS tương tự như DFS, nhưng thay vì sử dụng ngăn xếp cho open_set để lưu trữ các đỉnh có thể được thăm, BFS sử dụng cấu trúc dữ liệu

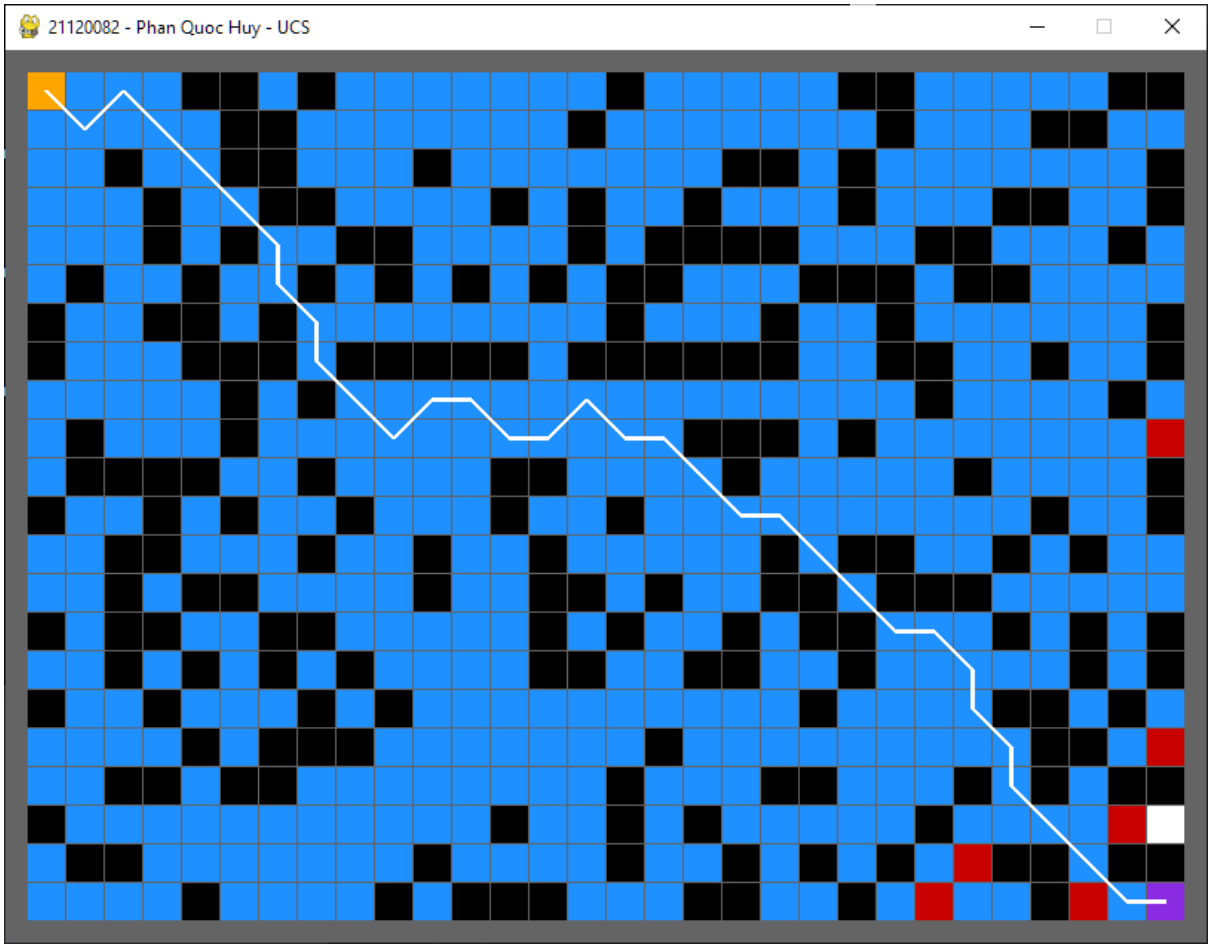
hàng đợi. Điều này dẫn đến BFS sẽ tìm kiếm các đỉnh theo chiều rộng, nghĩa là tất cả các đỉnh ở cùng mức sẽ được thăm trước khi BFS chuyển sang mức tiếp theo.

Quá trình tìm kiếm được mô tả như sau:

1. Khởi tạo hai tập dữ liệu:
    1. open_set: hàng đợi chứa các đỉnh có thể được thăm.
    2. closed_set: tập dữ liệu chứa các đỉnh đã được thăm.
2. Thêm đỉnh xuất phát vào open_set.
3. Lặp lại các bước sau cho đến khi tìm thấy mục tiêu hoặc tập open_set trống:
    1. Lấy đỉnh hiện tại ra khỏi open_set và thêm vào closed_set.
    2. Nếu đỉnh hiện tại là đỉnh mục tiêu, thì kết thúc thuật toán.
    3. Duyệt qua tất cả các đỉnh lân cận `neighbor` của đỉnh hiện tại.
    4. Nếu một đỉnh lân cận chưa có trong open_set và closed_set, thì thêm nó vào open_set.

Kết quả của thuật toán này cho ta một đường đi từ đỉnh xuất phát đến đỉnh mục tiêu, đường đi này. Trái ngược với *DFS*, thuật toán *BFS* được coi là một thuật toán hoàn chỉnh trong bất kì đồ thị nào, kể cả đồ thị vô hạn.

3. UCS



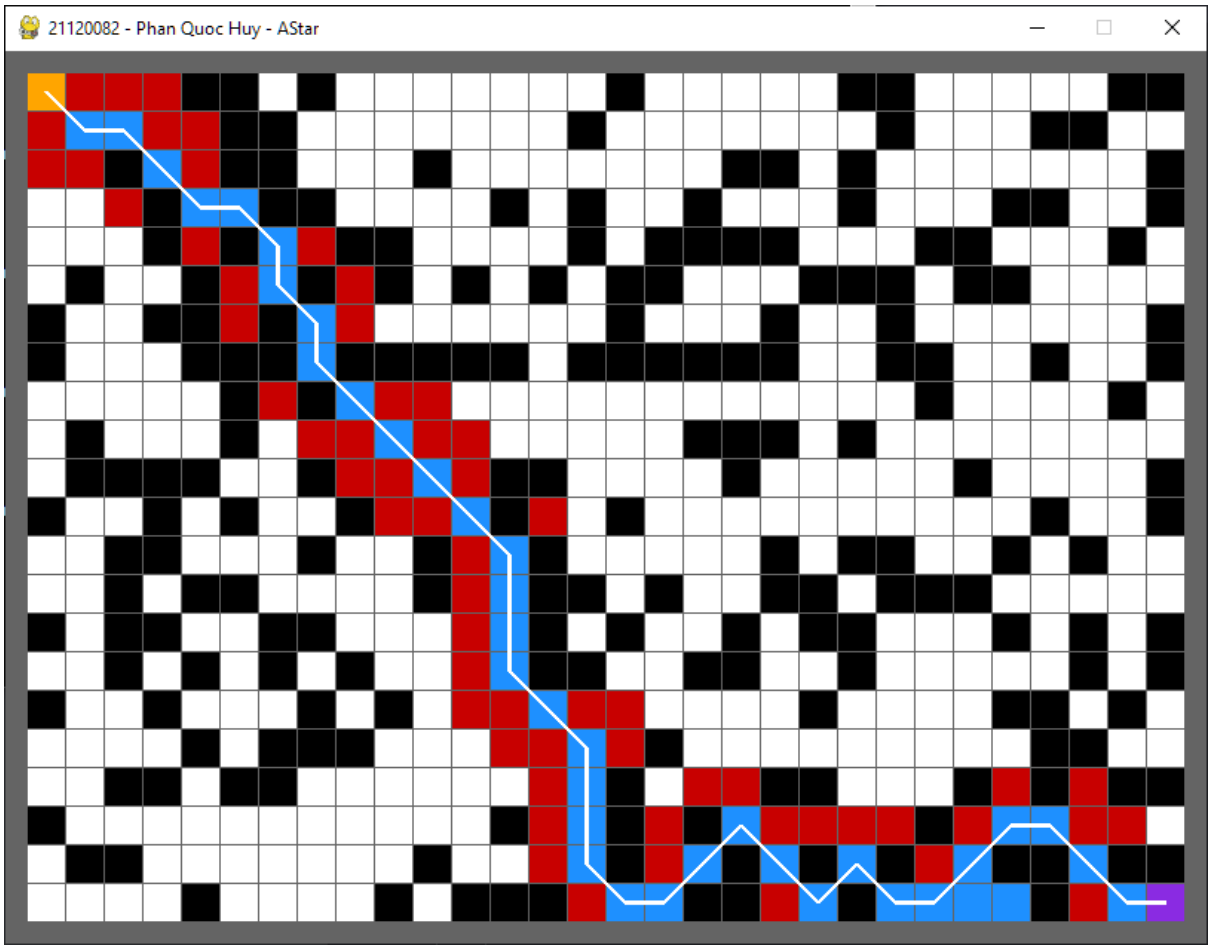
Quá trình tìm kiếm của thuật toán UCS tương tự như thuật toán BFS, nhưng thay vì sử dụng hàng đợi cho open_set để lưu trữ các đỉnh có thể được thăm, UCS sử dụng hàng đợi ưu tiên. Điều này dẫn đến UCS sẽ tìm kiếm các đỉnh theo thứ tự chi phí tăng dần, nghĩa là nút có chi phí thấp nhất sẽ được thăm trước.

Quá trình tìm kiếm được mô tả như sau:

1. Khởi tạo hai tập dữ liệu:
    1. open_set: hàng đợi ưu tiên chứa các đỉnh có thể được thăm, sắp xếp theo thứ tự chi phí tăng dần.
    2. closed_set: tập dữ liệu chứa các đỉnh đã được thăm.
2. Thêm đỉnh xuất phát vào open_set với chi phí bằng 0.
3. Lặp lại các bước sau cho đến khi tìm thấy mục tiêu hoặc open_set trống:
    1. Lấy đỉnh hiện tại ra khỏi open_set và thêm vào closed_set.

2. Nếu đỉnh hiện tại là đỉnh mục tiêu, thì kết thúc thuật toán.
3. Duyệt qua tất cả các đỉnh lân cận `neighbor` của đỉnh hiện tại.
4. Nếu một đỉnh lân cận chưa có trong open_set và closed_set, thì thêm nó vào open_set với chi phí bằng tổng chi phí của đỉnh hiện tại và chi phí đường giữa hai đỉnh.
5. Nếu một đỉnh lân cận đã có trong open_set, thì kiểm tra xem chi phí mới có nhỏ hơn chi phí cũ không. Nếu có, thì cập nhật chi phí của đỉnh lân cận và cha của nó trong open_set.

4. A*



Quá trình tìm kiếm của thuật toán A* tương tự như thuật toán UCS, nhưng thay vì chỉ sử dụng chi phí của đỉnh hiện tại để ước tính tổng chi phí đường đi, A* còn sử dụng hàm heuristic để ước tính chi phí còn lại của đường đi từ đỉnh hiện tại đến đỉnh mục tiêu.

Quá trình tìm kiếm được mô tả như sau:

1. Khởi tạo hai tập dữ liệu:
    - open_set: hàng đợi ưu tiên chứa các đỉnh có thể được thăm, sắp xếp theo thứ tự chi phí tăng dần.
    - closed_set: tập dữ liệu chứa các đỉnh đã được thăm.
2. Thêm đỉnh xuất phát vào open_set với chi phí bằng 0.
3. Lặp lại các bước sau cho đến khi tìm thấy mục tiêu hoặc open_set trống:
    - Lấy đỉnh hiện tại ra khỏi open_set và thêm vào closed_set.
    - Nếu đỉnh hiện tại là đỉnh mục tiêu, thì kết thúc thuật toán.
    - Duyệt qua tất cả các đỉnh lân cận `neighbor` của đỉnh hiện tại.
    - Nếu một đỉnh lân cận chưa có trong open_set và closed_set, thì thêm nó vào open_set với chi phí bằng tổng chi phí của đỉnh hiện tại và chi phí đường giữa hai đỉnh, cộng với giá trị của hàm heuristic.
    - Nếu một đỉnh lân cận đã có trong open_set, thì kiểm tra xem chi phí mới có nhỏ hơn chi phí cũ không. Nếu có, thì cập nhật chi phí của đỉnh lân cận và cha của nó trong open_set.

TÀI LIỆU THAM KHẢO:

1. Geeksforgeeks (2/2023), *Difference between Informed and Uninformed Search in AI*, https://www.geeksforgeeks.org/difference-between-informed-and-uninformed-search-in-ai/

2. Milos Simic (5/2023), *The Informed vs. Uninformed Search Algorithms*, https://www.baeldung.com/cs/informed-vs-uninformed-search

3. Katsu - Howkteam.vn, *Đồ thị và cây*, https://howkteam.vn/course/cau-truc-du-lieu-va-giai-thuat/do-thi-va-cay-4319

4. Vũ Quế Lâm (2/2022), *Các giải thuật tìm kiếm trên đồ thị*, https://viblo.asia/p/cac-giai-thuat-tim-kiem-tren-do-thi-1Je5EBRGKnL

5. Bùi Quang Hà (11/2020), *[Algorithm] Các thuật toán tìm kiếm trong AI*, https://labs.flinters.vn/algorithm/algorithm-cac-thuat-toan-tim-kiem-trong-ai/

6. Ashwin Ramachandran (9/2023), *Depth-first Search (DFS) Algorithm*, https://www.interviewkickstart.com/learn/depth-first-search-algorithm

7. Coppin, B. (2004). Artificial intelligence illuminated. Jones & Bartlett Learning. pp. 79–80, https://books.google.com.vn/books?id=LcOLqodW28EC&printsec=frontcover&hl=vi&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

8. Nikhil Sharma - University of California, Berkeley (2019), *CS 188, Introduction to Artificial Intelligence, Note 1*, https://inst.eecs.berkeley.edu/~cs188/sp19/assets/notes/n1.pdf

9. AAA, *CPSC 322, Practice Exercise Solutions to Uninformed Search*, https://www.cs.ubc.ca/~mack/CS322/exercises/1_ex_search_uninformed_sol.pdf

10. Andrew (4/2023), *Uniform-Cost Search (Dijkstra for large Graphs)*, https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

11. JavaTpoint, *Uninformed Search Algorithms*, https://www.javatpoint.com/ai-uninformed-search-algorithms

12. Stuart Russell and Peter Norvig (12/2009), *Artificial Intelligence: A Modern Approach, 3rd US ed*, https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf

13. Shichiki Le (8/2020), *Thuật Giải A**, https://www.iostream.vn/article/thuat-giai-a-DVnHj

14. Rachit Belwariar (3/2023), *A* Search Algorithm*, https://www.geeksforge

eks.org/a-search-algorithm/

15. Vietnam OER, Tài nguyên giáo dục Mở Việt Nam, *Giải thuật tìm kiếm A**, https://voer.edu.vn/m/giai-thuat-tim-kiem-a/d169b9dd

16. Ravikiran A S (10/2023), *A* Algorithm Concepts and Implementation*, https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm

17. Omar Khaled Abdelaziz Abdelnabi, *Shortest Path Algorithms*, https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/

18. Tantoluwa Heritage Alabi (5/2023), *What is a Greedy Algorithm? Examples of Greedy Algorithms*, https://www.freecodecamp.org/news/greedy-algorithms/#:~:text=In%20computer%20science%2C%20a%20greedy,solution%20that%20would%20be%20formed

---HẾT---