

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**



**BÁO CÁO
LAB 1: UNINFORMED SEARCH
AND INFORMED SEARCH
MÔN: CƠ SỞ TRÍ TUỆ NHÂN TẠO**

Sinh viên thực hiện:

Họ và tên: Nguyễn Hải Đăng
MSSV: 20120049

Giảng viên phụ trách:

Lý thuyết: GS. TS. Lê Hoài Bắc
Thực hành: Nguyễn Bảo Long

Ngày 26 tháng 3 năm 2023

Mục lục

1	Tổng quan về bài toán tìm kiếm	4
1.1	Các thành phần trạng thái của bài toán tìm kiếm	4
1.2	Informed search và Uninformed search	4
1.3	Giải pháp (mã giả) giải bài toán tìm kiếm	4
2	Thuật toán DFS	6
2.1	Ý tưởng thuật toán	6
2.2	Mã giả thuật toán	6
2.3	Tính chất của thuật toán	7
2.3.1	Tính đầy đủ	7
2.3.2	Tính tối ưu	7
2.3.3	Độ phức tạp về thời gian	7
2.3.4	Độ phức tạp về không gian	7
2.4	Kết quả và giải thích cách chạy thuật toán	7
3	Thuật toán BFS	9
3.1	Ý tưởng thuật toán	9
3.2	Mã giả thuật toán	9
3.3	Tính chất của thuật toán	10
3.3.1	Tính đầy đủ	10
3.3.2	Tính tối ưu	10
3.3.3	Độ phức tạp về thời gian	10
3.3.4	Độ phức tạp về không gian	10
3.4	Kết quả và giải thích cách chạy thuật toán	10
4	Thuật toán UCS	12
4.1	Ý tưởng thuật toán	12
4.2	Mã giả thuật toán	12
4.3	Tính chất của thuật toán	13
4.3.1	Tính đầy đủ	13
4.3.2	Tính tối ưu	13
4.3.3	Độ phức tạp về thời gian	13
4.3.4	Độ phức tạp về không gian	14
4.4	Kết quả và giải thích cách chạy thuật toán	14
5	Thuật toán A*	16
5.1	Ý tưởng thuật toán	16
5.2	Mã giả thuật toán	16
5.3	Tính chất của thuật toán	18
5.3.1	Tính đầy đủ	18
5.3.2	Tính tối ưu	18
5.3.3	Độ phức tạp về thời gian	18

5.3.4	Độ phức tạp về không gian	18
5.4	Hàm <i>heuristic</i> của thuật toán	18
5.5	Kết quả và giải thích cách chạy thuật toán	19
6	Một số thuật toán tìm kiếm khác	20
6.1	Greedy	20
6.2	Dijkstra	21
7	So sánh các thuật toán	23
7.1	UCS, Greedy và A*	23
7.2	UCS và Dijkstra	23
8	Các tài liệu liên quan đến bài Lab	25
8.1	Link video của bài Lab	25
8.2	Tài liệu tham khảo	25

Phần 1

Tổng quan về bài toán tìm kiếm

1.1 Các thành phần trạng thái của bài toán tìm kiếm

- Trạng thái ban đầu: trạng thái ban đầu của bài toán tìm kiếm.
- Trạng thái kế tiếp: các trạng thái mà thuật toán tìm kiếm có thể đi đến từ trạng thái hiện tại.
- Trạng thái đích: trạng thái mà ta cần tìm đến.
- Hành động: các hành động mà thuật toán tìm kiếm có thể thực hiện để đi từ trạng thái hiện tại đến trạng thái kế tiếp.
- Chi phí: chi phí của mỗi hành động.

1.2 Informed search và Uninformed search

- Uninformed search: khi không có thông tin gì về bài toán, thuật toán sẽ thực hiện tìm kiếm ngẫu nhiên từ trạng thái ban đầu đến khi đạt được trạng thái đích.
- Informed search: khi có thông tin bổ sung về bài toán, thuật toán sẽ sử dụng thông tin đó để giảm thiểu số lần thực hiện tìm kiếm. Ví dụ: DFS, BFS, UCS, A*, ...

1.3 Giải pháp (mã giả) giải bài toán tìm kiếm

```
function tìm_kiếm(bài_toán):  
    khởi_tạo trạng_thái_ban_đầu  
    khởi_tạo trạng_thái_kế_tiếp  
    while trạng_thái_ban_đầu != trạng_thái_đích:  
        trạng_thái_kế_tiếp = lấy_trạng_thái_kế_tiếp(bài_toán, trạng_thái_ban_đầu)  
        trạng_thái_ban_đầu = trạng_thái_kế_tiếp  
    return trạng_thái_ban_đầu
```

Trong đó:

- bài_toán là bài toán tìm kiếm cần giải quyết.
- trạng_thái_ban_đầu là trạng thái ban đầu của bài toán tìm kiếm.
- trạng_thái_đích là trạng thái đích cần tìm.

- `lấy_trạng_thái_kế_tiếp` là một hàm được định nghĩa để lấy ra trạng thái kế tiếp của bài toán tìm kiếm từ trạng thái hiện tại, có thể là hàm tìm kiếm theo chiều rộng (BFS), tìm kiếm theo chiều sâu (DFS), hoặc các thuật toán tìm kiếm khác.

Phần 2

Thuật toán DFS

2.1 Ý tưởng thuật toán

Thuật toán DFS (Depth-First Search) là một trong những thuật toán cơ bản để tìm kiếm đường đi trên đồ thị. Ý tưởng của thuật toán là sử dụng ngăn xếp (stack) để lưu trữ các đỉnh liền kề của đỉnh đang xét.

Dầu tiên, ta khởi tạo một ngăn xếp rỗng và thêm đỉnh đầu tiên vào đó. Sau đó, ta lấy đỉnh đầu tiên từ ngăn xếp và đánh dấu nó đã được thăm. Tiếp theo, ta tìm tất cả các đỉnh liền kề với đỉnh đang xét, chưa được thăm và đưa chúng vào trong ngăn xếp. Lặp lại quá trình này cho đến khi ngăn xếp trống.

Để lưu trữ trạng thái của các đỉnh, ta sử dụng hai tập hợp `open_set` và `close_set`. `open_set` là tập hợp các đỉnh chưa được thăm, còn `close_set` là tập hợp các đỉnh đã được thăm. Khi một đỉnh được thăm, ta di chuyển nó từ `open_set` sang `close_set`.

Ngoài ra, trong quá trình tìm kiếm đường đi, ta cần lưu lại đỉnh cha của mỗi đỉnh. Điều này giúp ta có thể truy vết lại đường đi từ đỉnh đích về đỉnh nguồn.

Để thực hiện thuật toán DFS, ta cần đưa các đỉnh và cạnh của đồ thị vào một biểu diễn phù hợp, ví dụ như ma trận kề hoặc danh sách kề. Sau đó, ta áp dụng thuật toán DFS trên biểu diễn này.

2.2 Mã giả thuật toán

```
function DFS(graph, start, goal):
    open_set = [start]
    closed_set = []
    father = {}

    while open_set:
        current = open_set.pop()

        if current == goal:
            path = []
            while current in father:
                path.append(current)
                current = father[current]
            path.append(start)
            return path

        for neighbor in graph[current]:
            if neighbor not in closed_set:
                open_set.append(neighbor)
                father[neighbor] = current
```

```

    path.reverse()
    return path

closed_set.append(current)

for neighbor in graph.get_neighbors[current]:
    if neighbor not in closed_set and neighbor not in open_set:
        open_set.append(neighbor)
        father[neighbor] = current

return None

```

Thuật toán sử dụng các tập hợp `open_set`, `close_set` và `father` để lưu trữ trạng thái của các đỉnh. Trong quá trình thực hiện, hàm lặp lại các bước sau cho đến khi `open_set` trống: lấy một đỉnh từ `open_set` để xét, đánh dấu đỉnh đang xét đã được thăm, xét các đỉnh kề với đỉnh đang xét và thêm chúng vào `open_set` nếu chưa được thăm. Nếu tìm thấy đường đi từ đỉnh bắt đầu đến đỉnh đích, hàm trả về một danh sách các đỉnh trên đường đi đó. Nếu không tìm thấy đường đi, hàm trả về `None`.

2.3 Tính chất của thuật toán

2.3.1 Tính đầy đủ

Thuật toán DFS sẽ tìm được đường đi từ đỉnh bắt đầu đến đỉnh đích nếu đường đi đó tồn tại trên đồ thị. Tuy nhiên, nếu đồ thị có chu trình vô hạn, thuật toán sẽ không kết thúc. Do đó, thuật toán DFS không đảm bảo tính đầy đủ tuyệt đối.

2.3.2 Tính tối ưu

Thuật toán DFS không đảm bảo tìm được đường đi ngắn nhất từ đỉnh bắt đầu tới đỉnh đích, trừ khi đồ thị không có trọng số hoặc có trọng số không âm.

2.3.3 Độ phức tạp về thời gian

Độ phức tạp thời gian của thuật toán DFS là $O(V + E)$, trong đó V là số lượng đỉnh của đồ thị và E là số lượng cạnh của đồ thị. Trong trường hợp xấu nhất, thuật toán DFS có thể duyệt qua tất cả các đỉnh và cạnh trên đồ thị.

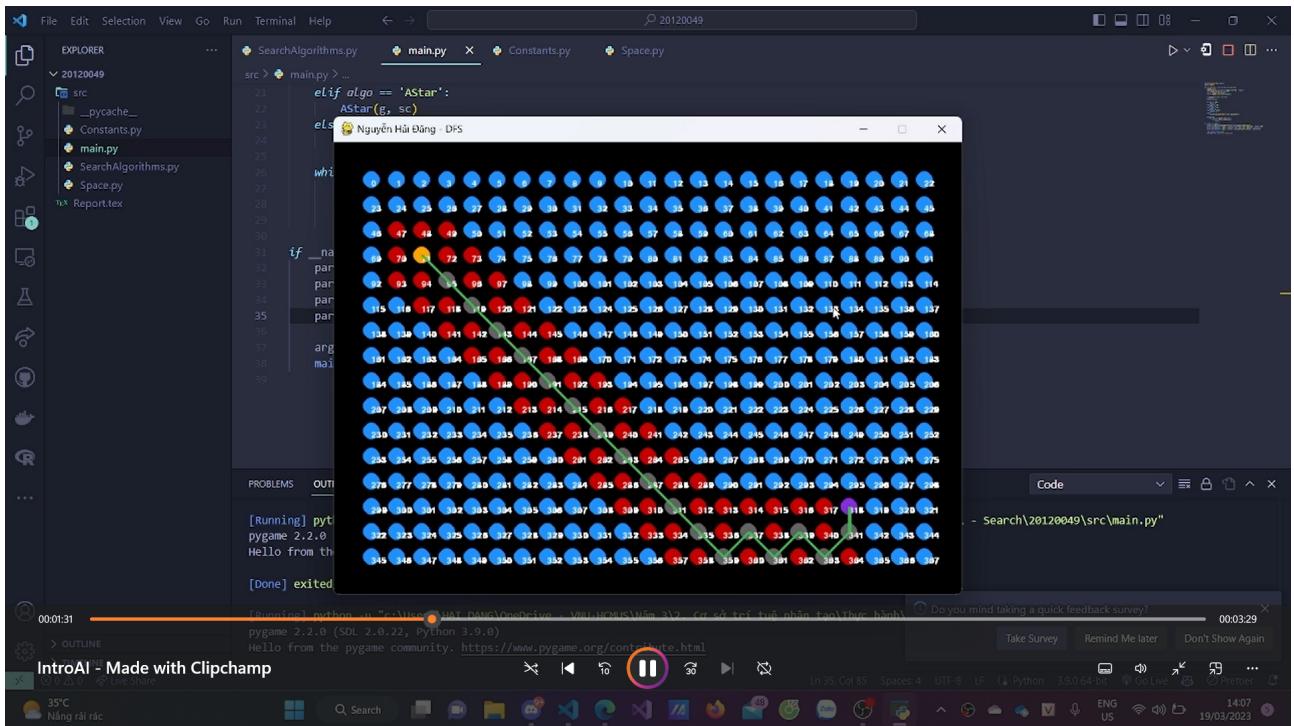
2.3.4 Độ phức tạp về không gian

Độ phức tạp không gian của thuật toán DFS phụ thuộc vào cách lưu trữ đồ thị. Nếu sử dụng ma trận kề để lưu trữ đồ thị, độ phức tạp không gian của thuật toán DFS là $O(V^2)$. Nếu sử dụng danh sách kề để lưu trữ đồ thị, độ phức tạp không gian của thuật toán DFS là $O(V + E)$, trong đó V là số lượng đỉnh của đồ thị và E là số lượng cạnh của đồ thị.

2.4 Kết quả và giải thích cách chạy thuật toán

Hàm nhận vào hai tham số là đối tượng đồ thị và đối tượng `pygame.Surface`.

Các biến `open_set`, `closed_set` và `father` được sử dụng để lưu trữ các đỉnh được duyệt qua và đường đi tìm được.



Hình 2.1: Kết quả khi chạy thuật toán DFS.

Các bước chương trình chạy bao gồm:

1. Đỉnh bắt đầu là màu cam và đỉnh kết thúc là màu tím.
2. Đánh dấu nút hiện tại là màu vàng trên màn hình.
3. Nếu nút hiện tại là điểm kết thúc (điểm đích), thì tìm đường đi từ **start** đến **goal**, đánh dấu các nút trên đường đi màu xám, nối các nút trên đường đi bằng đường thẳng màu xanh lá cây trên màn hình và trả về đường đi đó.
4. Nếu nút hiện tại không phải là điểm kết thúc, đưa nó vào danh sách **closed_set**.
5. Duyệt qua tất cả các nút kề của nút hiện tại, đánh dấu chúng là màu đỏ và thêm vào **open_set**.
6. Vẽ đồ thị lên màn hình, đánh dấu nút hiện tại là màu xanh dương và tạm dừng một khoảng thời gian ngắn để hiển thị quá trình tìm kiếm.

Phần 3

Thuật toán BFS

3.1 Ý tưởng thuật toán

Thuật toán BFS (Breadth First Search) là một thuật toán tìm kiếm trên đồ thị rất phổ biến trong lĩnh vực khoa học máy tính. Ý tưởng của thuật toán BFS là tìm kiếm một đồ thị hay một cấu trúc dữ liệu theo chiều rộng, nghĩa là duyệt tất cả các đỉnh kề với đỉnh hiện tại trước khi di chuyển đến các đỉnh kề tiếp theo.

Ý tưởng của thuật toán BFS là duyệt qua các đỉnh của đồ thị theo chiều rộng. Để làm được điều này, ta sử dụng hai tập hợp `open_set` và `close_set` để lưu trữ danh sách các đỉnh đã được duyệt qua và chưa được duyệt qua. `open_set` chứa các đỉnh chưa được thăm, còn `close_set` chứa các đỉnh đã được thăm.

Ngoài ra, ta sử dụng một mảng `father` để lưu lại đỉnh cha của mỗi đỉnh trong đồ thị, giúp tìm kiếm và tái tạo đường đi từ đỉnh đầu tiên đến đỉnh cuối cùng.

Việc sử dụng `open_set`, `close_set` và `father` giúp cho việc tìm kiếm trở nên dễ dàng và hiệu quả hơn. Tuy nhiên, nếu đồ thị quá lớn, việc lưu trữ các tập hợp này sẽ tốn nhiều bộ nhớ và làm chậm thuật toán.

3.2 Mã giả thuật toán

```
function BFS(graph, start, goal):
    open_set = [start]
    close_set = []
    father = {}

    while open_set:
        current = open_set.pop(0) #Đây là điểm khác biệt với DFS
        close_set.append(current)

        if current == goal:
            path = [current]
            while current != start:
                current = father[current]
                path.append(current)
            path.reverse()
            return path
```

```

for neighbor in graph.get_neighbors[current]:
    if neighbor not in close_set and neighbor not in open_set:
        open_set.append(neighbor)
        father[neighbor] = current

return None

```

Thuật toán BFS sử dụng tập hợp `open_set` để lưu trữ các nút chưa được xét và tập hợp `close_set` để lưu trữ các nút đã được xét. Thuật toán sẽ lấy nút đầu tiên của `open_set` để xét, sau đó thêm nút đó vào `close_set`. Tiếp theo, thuật toán duyệt qua các nút kề của nút hiện tại và thêm các nút kề chưa được xét vào `open_set`.

Nếu tìm được đích, thuật toán sẽ sử dụng từ điển `father` để truy vết đường đi từ đích đến đầu. Ngược lại, nếu không tìm được đường đi, thuật toán trả về giá trị `None`.

3.3 Tính chất của thuật toán

3.3.1 Tính đầy đủ

Thuật toán BFS sẽ tìm ra đường đi từ đỉnh xuất phát tới đỉnh đích (nếu có tồn tại đường đi) nếu đồ thị là liên thông. Tuy nhiên, nếu đồ thị không liên thông thì BFS chỉ tìm ra đường đi từ đỉnh xuất phát tới các đỉnh thuộc cùng thành phần liên thông với đỉnh xuất phát.

3.3.2 Tính tối ưu

Nếu các cạnh trong đồ thị có cùng trọng số hoặc không có trọng số, BFS sẽ tìm ra đường đi ngắn nhất từ đỉnh xuất phát tới tất cả các đỉnh trong đồ thị, đồng thời đường đi đó cũng là đường đi ngắn nhất từ đỉnh xuất phát tới đỉnh đó. Tuy nhiên, nếu các cạnh có trọng số khác nhau, thì BFS không đảm bảo tìm được đường đi ngắn nhất.

3.3.3 Độ phức tạp về thời gian

Độ phức tạp thời gian của BFS là $O(V + E)$, trong đó V là số đỉnh và E là số cạnh trong đồ thị. Đây là độ phức tạp thời gian tốt nhất có thể đạt được cho các thuật toán duyệt đồ thị.

Trong trường hợp tệ nhất của BFS, khi đồ thị là một đường thẳng dài gồm V đỉnh liên tiếp nhau, thì BFS sẽ phải duyệt qua tất cả các đỉnh, tức là thực hiện V lần duyệt.

3.3.4 Độ phức tạp về không gian

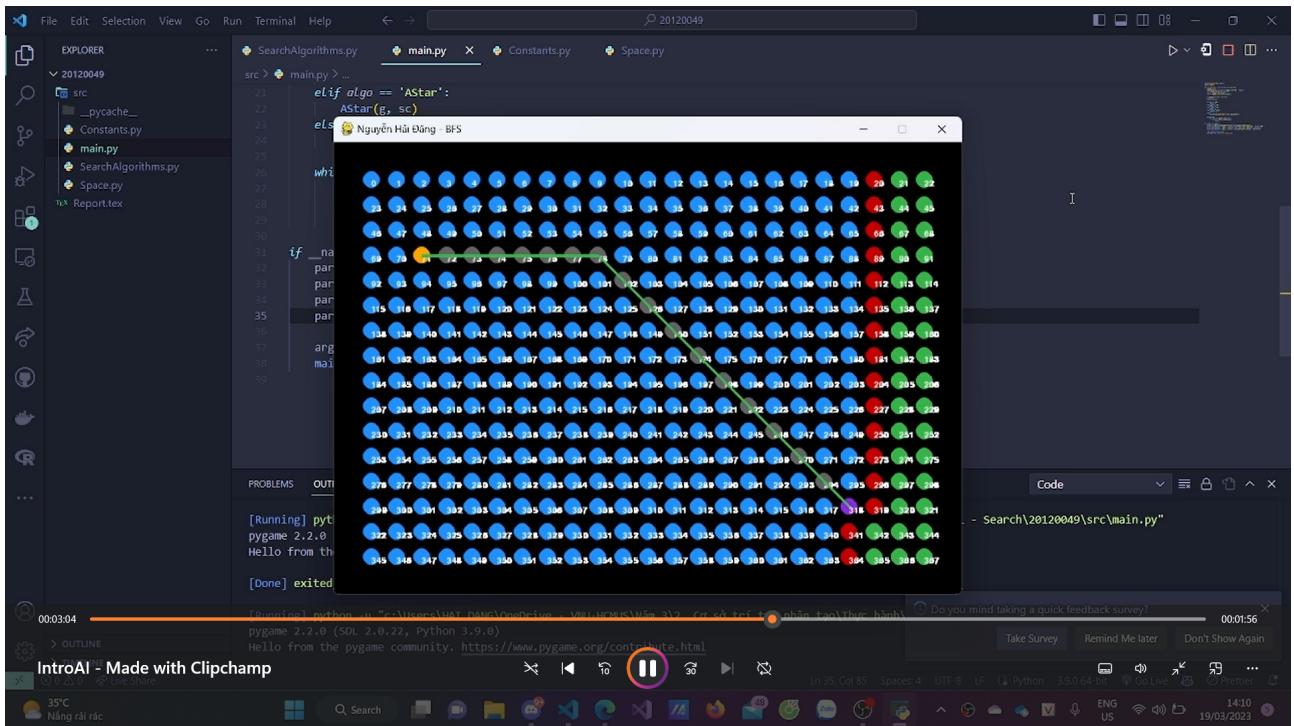
Độ phức tạp không gian của BFS cũng là $O(V + E)$, do trong quá trình duyệt BFS sẽ lưu trữ thông tin của tất cả các đỉnh và các cạnh trong đồ thị. Điều này có nghĩa là BFS sẽ sử dụng một lượng bộ nhớ tương đối lớn, đặc biệt là đối với các đồ thị lớn.

3.4 Kết quả và giải thích cách chạy thuật toán

Hàm nhận vào hai tham số là đối tượng đồ thị và đối tượng `pygame.Surface`.

Các biến `open_set`, `closed_set` và `father` được sử dụng để lưu trữ các đỉnh được duyệt qua và đường đi tìm được.

Các bước chương trình chạy bao gồm:



Hình 3.1: Kết quả khi chạy thuật toán BFS.

1. Dỉnh bắt đầu là màu cam và đỉnh kết thúc là màu tím.
2. Dánh dấu nút hiện tại là màu vàng trên màn hình.
3. Nếu nút hiện tại là điểm kết thúc (điểm đích), thì tìm đường đi từ **start** đến **goal**, dánh dấu các nút trên đường đi màu xám, nối các nút trên đường đi bằng đường thẳng màu xanh lá cây trên màn hình và trả về đường đi đó.
4. Nếu nút hiện tại không phải là điểm kết thúc, đưa nó vào danh sách **closed_set**.
5. Duyệt qua tất cả các nút kề của nút hiện tại, dánh dấu chúng là màu đỏ và thêm vào **open_set**.
6. Vẽ đồ thị lên màn hình, dánh dấu nút hiện tại là màu xanh dương và tạm dừng một khoảng thời gian ngắn để hiển thị quá trình tìm kiếm.

Phân 4

Thuật toán UCS

4.1 Ý tưởng thuật toán

Thuật toán UCS (Uniform Cost Search) tìm kiếm đường đi có chi phí nhỏ nhất từ đỉnh xuất phát đến đỉnh đích trong đồ thị không định hướng.

Ta sử dụng các tập hợp `open_set`, `close_set` và `father` để lưu trạng thái của các đỉnh. Thuật toán bắt đầu bằng việc thêm đỉnh xuất phát vào `open_set` và gán chi phí của đỉnh này bằng 0.

Tiếp theo, lặp lại cho đến khi tìm được đỉnh đích hoặc `open_set` rỗng: lấy đỉnh có chi phí nhỏ nhất trong `open_set`, chuyển nó sang `close_set` và cập nhật lại chi phí của các đỉnh kề nếu có đường đi mới ngắn hơn.

Cuối cùng, ta sử dụng `father` để truy vết lại đường đi từ đỉnh đích đến đỉnh xuất phát.

4.2 Mã giả thuật toán

```
function UCS(graph, start, goal):
    # Khởi tạo open_set, close_set và father
    open_set = {start}
    close_set = set()
    father = {start: None}
    g = {start: 0}

    while open_set:
        # Lấy đỉnh có chi phí nhỏ nhất trong open_set
        current = min(open_set, key=g.get)

        # Nếu đỉnh hiện tại là đỉnh đích
        # Trả về đường đi từ đỉnh xuất phát đến đỉnh đích
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = father[current]
            return path[::-1]
```

```

# Chuyển đỉnh hiện tại từ open_set sang close_set
open_set.remove(current)
close_set.add(current)

# Xét các đỉnh kề với đỉnh hiện tại
for neighbor in graph[current]:
    if neighbor in close_set:
        continue

    # Cập nhật chi phí nếu đường đi mới ngắn hơn đường đi trước đó
    new_cost = g[current] + graph[current][neighbor]
    if neighbor not in open_set or new_cost < g[neighbor]:
        g[neighbor] = new_cost
        father[neighbor] = current
        open_set.add(neighbor)

# Nếu không tìm thấy đường đi từ đỉnh xuất phát đến đỉnh đích, trả về None
return None

```

Thuật toán UCS bắt đầu bằng việc thêm đỉnh xuất phát vào `open_set` và gán chi phí của đỉnh này bằng 0. Sau đó, thuật toán lặp lại các bước sau cho đến khi tìm được đỉnh đích hoặc `open_set` rỗng:

- Lấy đỉnh có chi phí nhỏ nhất trong `open_set` và chuyển nó sang `close_set`.
- Với mỗi đỉnh kề với đỉnh vừa lấy ra, nếu đỉnh này chưa được xét hoặc có chi phí nhỏ hơn chi phí của đỉnh đã xét trong `open_set`, ta cập nhật lại chi phí và thêm đỉnh vào `open_set` với đỉnh cha là đỉnh vừa lấy ra.

Khi tìm thấy đỉnh đích, ta sử dụng `father` để truy vết lại đường đi từ đỉnh đích đến đỉnh xuất phát.

4.3 Tính chất của thuật toán

4.3.1 Tính đầy đủ

Thuật toán UCS được xem là **Completeness**, vì nó sẽ tìm được đường đi giữa 2 đỉnh nếu tồn tại một đường đi giữa chúng. Mặc dù độ phức tạp thời gian của thuật toán có thể là rất cao, nhưng UCS sẽ tiếp tục tìm kiếm đường đi cho đến khi tìm thấy hoặc xác định rằng không có đường đi giữa 2 đỉnh.

4.3.2 Tính tối ưu

Khi tìm kiếm đường đi ngắn nhất, UCS luôn tìm được đường đi có chi phí thấp nhất. Thuật toán này sử dụng một hàng đợi ưu tiên để lưu trữ các đường đi tiềm năng, trong đó các đường đi có chi phí thấp hơn có ưu tiên hơn để được duyệt. Do đó, khi tìm thấy đường đi đến đích, đó là đường đi ngắn nhất.

4.3.3 Độ phức tạp về thời gian

Độ phức tạp thời gian của thuật toán UCS phụ thuộc vào số lượng đỉnh và cạnh trong đồ thị. Trong trường hợp tệ nhất, thuật toán UCS có thể phải duyệt qua tất cả các đỉnh và cạnh trong

dồ thị, do đó độ phức tạp thời gian của UCS là $O(b^{d+1})$, trong đó b là số lượng đỉnh trung bình của mỗi đỉnh và d là độ sâu của đường đi ngắn nhất.

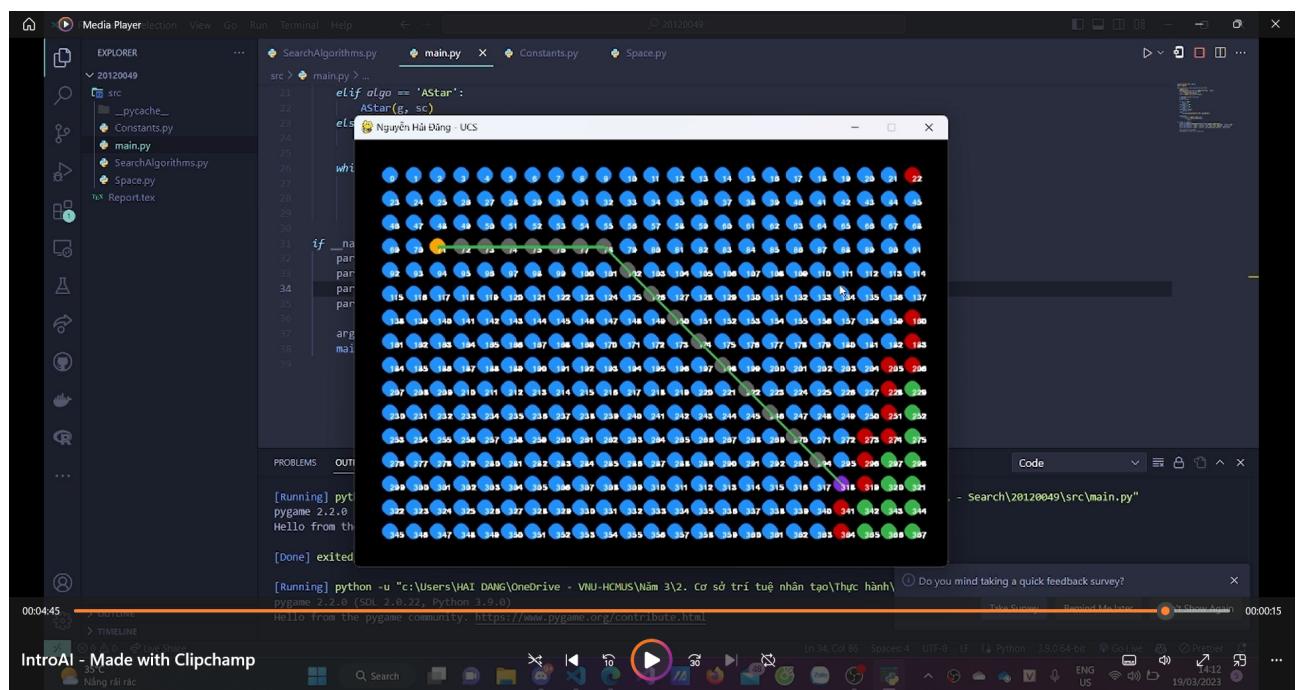
Tuy nhiên, nếu đồ thị được sắp xếp theo thứ tự tăng dần của trọng số, thì độ phức tạp thời gian của thuật toán UCS sẽ giảm xuống còn $O(d * \log(V))$, trong đó V là số lượng đỉnh trong đồ thị. Điều này là do khi sử dụng hàng đợi ưu tiên, các đường đi có chi phí thấp hơn sẽ được duyệt trước, giúp UCS tìm kiếm nhanh hơn.

4.3.4 Độ phức tạp về không gian

Dộ phức tạp không gian của thuật toán UCS phụ thuộc vào số lượng đỉnh trong đồ thị. Khi duyệt đến một đỉnh mới, thuật toán UCS sẽ lưu giữ thông tin về đỉnh đó và các đường đi tới đỉnh đó. Do đó, độ phức tạp không gian của UCS là $O(b^d)$, trong đó b là số lượng đỉnh trung bình của mỗi đỉnh và d là độ sâu của đường đi ngắn nhất.

Tuy nhiên, nếu tất cả các đỉnh đều được lưu trữ trong bộ nhớ trước khi thực hiện thuật toán UCS, thì độ phức tạp không gian của UCS sẽ là $O(V)$, trong đó V là số lượng đỉnh trong đồ thị. Điều này là do UCS không cần lưu trữ các đường đi từ đỉnh bắt đầu đến các đỉnh khác nếu chúng không phải là đường đi ngắn nhất.

4.4 Kết quả và giải thích cách chạy thuật toán



Hình 4.1: Kết quả khi chạy thuật toán UCS.

Hàm nhận vào hai tham số là đối tượng đồ thị và đối tượng `pygame.Surface`.

Đầu tiên, các giá trị mặc định ban đầu được khai báo cho biến `open_set`, `closed_set`, `father` và `cost`. Các biến này được sử dụng để lưu trữ các giá trị trong quá trình thực hiện thuật toán.

Trong quá trình thực hiện thuật toán, các đỉnh trên lưới sẽ được thay đổi màu sắc theo các trạng thái khác nhau:

1. Các đỉnh đã duyệt xong sẽ có màu xanh dương.

2. Các đỉnh lân cận của đỉnh hiện tại sẽ có màu đỏ.
3. Đỉnh đang duyệt sẽ có màu vàng.
4. Các đỉnh trên đường đi sẽ có màu xám.

Khi tìm thấy đường đi ngắn nhất, các đỉnh trên đường đi sẽ có màu xám và được nối bằng đường thẳng màu xanh lá cây.

Thuật toán được thực hiện trong vòng lặp `while`, trong đó các bước tiếp theo được thực hiện:

- Tìm ô có giá trị `cost` nhỏ nhất trong `open_set`.
- Duyệt các ô lân cận của ô hiện tại và cập nhật giá trị `cost` và `father` cho các ô lân cận.
- Nếu ô đích đã được tìm thấy, trả về đường đi ngắn nhất.
- Nếu không tìm thấy đường đi thì `raise NotImplementedError`.

Phần 5

Thuật toán A*

5.1 Ý tưởng thuật toán

Thuật toán A* là một thuật toán tìm kiếm đường đi trong đồ thị, với mục tiêu tìm đường đi từ một điểm bắt đầu đến một điểm kết thúc. Thuật toán A* sử dụng một hàm *heuristic* để ước lượng chi phí từ điểm hiện tại đến điểm đích, từ đó quyết định đường đi tiếp theo.

Thuật toán A* được xây dựng trên ý tưởng của thuật toán tìm kiếm đường đi Dijkstra. Tuy nhiên, A* sử dụng một hàm *heuristic* để ước lượng chi phí từ điểm hiện tại đến điểm đích, giúp thuật toán tìm kiếm đường đi tối ưu hơn.

Thuật toán A* sử dụng một cấu trúc dữ liệu là *priority queue* để lưu trữ các đỉnh đã duyệt qua. *Priority queue* sẽ giúp cho thuật toán tìm kiếm các đỉnh có chi phí gần điểm đích trước, giúp tối ưu thời gian tìm kiếm.

Với hàm *heuristic* phù hợp, thuật toán A* cho kết quả tìm kiếm đường đi tối ưu và nhanh chóng trong đồ thị.

5.2 Mã giả thuật toán

```
function A*(start, goal, heuristic):
    # Khởi tạo open set và close set
    open_set = {start}
    close_set = {}

    # Khởi tạo father, bắt đầu với rỗng
    father = {}

    # Khởi tạo g_score và f_score, bắt đầu với vô cùng
    g_score = {start: 0}
    f_score = {start: heuristic(start, goal)}

    while len(open_set) > 0:
        # Lấy ra đỉnh có f_score nhỏ nhất trong open set
        current = min(open_set, key=lambda x: f_score[x])

        # Kiểm tra xem đã tìm được đường đi tới đích chưa
        if current == goal:
```

```

        return construct_path(father, goal)

    # Di chuyển đỉnh hiện tại từ open set sang close set
    open_set.remove(current)
    close_set.add(current)

    # Duyệt qua tất cả các đỉnh kề của đỉnh hiện tại
    for neighbor in get_neighbors(current):
        # Nếu đã duyệt qua đỉnh này rồi, bỏ qua
        if neighbor in close_set:
            continue

        # Tính toán g_score của đỉnh hiện tại tới đỉnh kề
        tentative_g_score = g_score[current] + distance(current, neighbor)

        # Nếu đỉnh kề chưa được duyệt hoặc chi phí tới đỉnh kề tốt hơn
        if neighbor not in open_set or tentative_g_score < g_score[neighbor]:
            # Cập nhật father và g_score của đỉnh kề
            father[neighbor] = current
            g_score[neighbor] = tentative_g_score
            f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)

            # Nếu đỉnh kề chưa có trong open set, thêm vào
            if neighbor not in open_set:
                open_set.add(neighbor)

    # Không tìm thấy đường đi
    return None

function construct_path(father, goal):
    # Truy vết ngược từ goal để lấy ra đường đi
    path = [goal]
    while goal in father:
        goal = father[goal]
        path.append(goal)
    path.reverse()
    return path

```

Trong mã giả của thuật toán A*, `open_set` là tập hợp các đỉnh chưa được duyệt, `close_set` là tập hợp các đỉnh đã được duyệt qua. `father` là một mapping từ mỗi đỉnh đến đỉnh cha của nó trong đường đi tìm được.

Khi lặp lại vòng `while`, thuật toán A* sẽ lấy ra đỉnh có `f_score` nhỏ nhất trong `open_set`, kiểm tra xem đã tìm được đường đi tới đích chưa. Nếu tìm thấy đường đi tới đích, hàm `construct_path` sẽ truy vết ngược từ `goal` để lấy ra đường đi tìm được.

Nếu chưa tìm được đường đi tới đích, thuật toán sẽ di chuyển đỉnh hiện tại từ `open_set` sang `close_set`, duyệt qua tất cả các đỉnh kề của đỉnh hiện tại. Nếu đỉnh kề chưa được duyệt hoặc chi phí tới đỉnh kề tốt hơn, `father` và `g_score` của đỉnh kề sẽ được cập nhật. Nếu đỉnh kề chưa có trong `open_set`, nó sẽ được thêm vào.

5.3 Tính chất của thuật toán

5.3.1 Tính đầy đủ

Nếu đường đi từ đỉnh bắt đầu đến đỉnh kết thúc tồn tại, thuật toán A* sẽ tìm ra đường đi đó. Tuy nhiên, nếu không có đường đi nào từ đỉnh bắt đầu đến đỉnh kết thúc, thuật toán sẽ không tìm ra được đường đi.

5.3.2 Tính tối ưu

Nếu hàm *heuristic* được sử dụng trong thuật toán A* là một hàm *heuristic* tối ưu, thuật toán A* sẽ tìm được đường đi ngắn nhất giữa hai điểm bất kỳ trên đồ thị hoặc lưới ô vuông.

Tuy nhiên, nếu hàm *heuristic* không tối ưu, thuật toán sẽ không đảm bảo tìm được đường đi ngắn nhất.

5.3.3 Độ phức tạp về thời gian

Thời gian thực hiện của thuật toán A* phụ thuộc vào hàm *heuristic* được sử dụng. Tuy nhiên, thời gian thực hiện của A* được đánh giá là nhanh hơn so với thuật toán Dijkstra. Độ phức tạp thời gian của thuật toán A* là $O(b^d)$, trong đó b là hằng số số lượng các trạng thái kế tiếp mà mỗi trạng thái có thể có, và d là độ sâu của đường đi tốt nhất.

5.3.4 Độ phức tạp về không gian

Độ phức tạp không gian của thuật toán A* là $O(b^d)$, trong đó b là hằng số số lượng các trạng thái kế tiếp mà mỗi trạng thái có thể có, và d là độ sâu của đường đi tốt nhất. Tuy nhiên, trong thực tế, A* thường sử dụng một số lượng bộ nhớ nhỏ hơn so với Dijkstra bởi vì nó chỉ lưu trữ các đỉnh trong `open_set` có thể tối ưu nhất.

5.4 Hàm *heuristic* của thuật toán

Trong bài lab này, em sử dụng hàm tính khoảng cách *manhattan* để làm hàm *heuristic* cho thuật toán A*.

Hàm này tính khoảng cách giữa hai điểm theo hướng ngang và dọc trên lưới ô vuông, bằng cách lấy tổng giá trị tuyệt đối của hiệu giữa tọa độ ngang và dọc của hai điểm.

Hàm Manhattan có thể được biểu diễn là: $h(n) = |x - x_{goal}| + |y - y_{goal}|$

Trong đó n là điểm hiện tại, (x, y) là tọa độ của điểm hiện tại và (x_{goal}, y_{goal}) là tọa độ của điểm đích.

Ngoài ra còn có một số hàm *heuristic* khác như:

- Hàm Euclidean: là một hàm *heuristic* phổ biến được sử dụng trong tìm kiếm A*. Hàm này tính khoảng cách giữa hai điểm trên mặt phẳng theo công thức:

$$h(n) = \sqrt{(x_{goal} - x)^2 + (y_{goal} - y)^2}$$

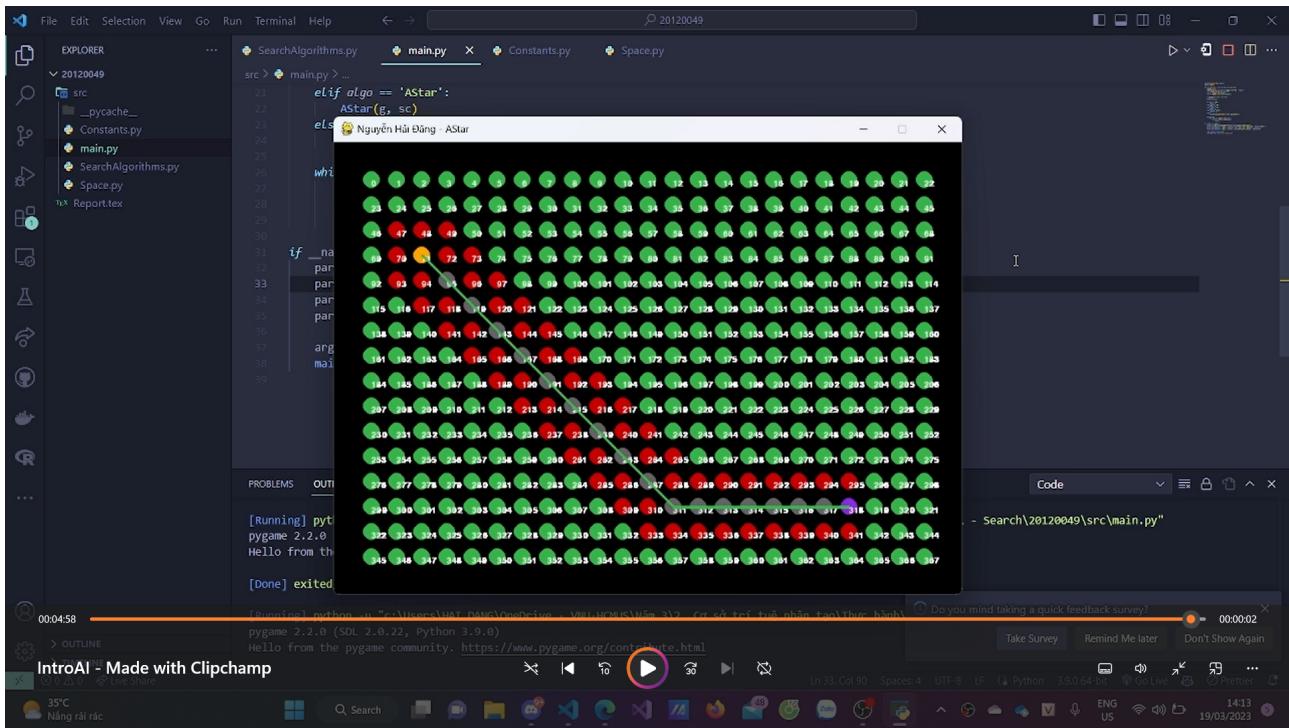
Trong đó (x_{goal}, y_{goal}) là tọa độ của ô đích, (x, y) là tọa độ của ô hiện tại đang xét.

- Hàm Chebyshev: Hàm Chebyshev là một hàm *heuristic* khác được sử dụng trong tìm kiếm A*. Hàm này tính khoảng cách giữa hai điểm trên mặt phẳng theo công thức:

$$h(n) = \max(|x_{goal} - x|, |y_{goal} - y|)$$

Trong đó (x_{goal}, y_{goal}) là tọa độ của ô đích, (x, y) là tọa độ của ô hiện tại đang xét.

5.5 Kết quả và giải thích cách chạy thuật toán



Hình 5.1: Kết quả khi chạy thuật toán A*.

Hàm nhận vào hai tham số là đối tượng đồ thị và đối tượng `pygame.Surface`.

Dầu tiên, các giá trị mặc định ban đầu được khai báo cho biến `open_set`, `closed_set`, `father`, `g_score` và `f_score`. Các biến này được sử dụng để lưu trữ các giá trị trong quá trình thực hiện thuật toán.

Trong quá trình thực hiện thuật toán, các đỉnh trên lưới sẽ được thay đổi màu sắc theo các trạng thái khác nhau:

1. Thực hiện vòng lặp cho đến khi `open_set` rỗng.
2. Tìm nút có ước tính khoảng cách nhỏ nhất từ `open_set` và lưu vào `current_node`.
3. Nếu `current_node` là nút kết thúc, trả về đường đi từ điểm bắt đầu đến điểm kết thúc.
4. Nếu không, thêm `current_node` vào `closed_set` và tô màu nút đó màu vàng.
5. Duyệt qua các nút lá của `current_node`, tính toán khoảng cách tạm thời `tentative_g_score`, nếu nút lá không nằm trong `open_set`, thêm nó vào `open_set`, ngược lại, nếu $tentative_g_score \geq g_score$ của nút lá, bỏ qua nút lá đó, ngược lại, cập nhật `father`, `g_score`, `f_score` của nút lá đó và tô màu nút đó màu đỏ.
6. Vẽ đồ thị, tô màu nút đang xét màu xanh dương và tạm dừng trong một thời gian ngắn để hiển thị quá trình tìm đường.
7. Lặp lại các bước 1-6 cho đến khi tìm được đường đi hoặc `open_set` rỗng.

Phần 6

Một số thuật toán tìm kiếm khác

6.1 Greedy

Thuật toán Greedy (tạm dịch là "tham lam") trong đồ thị là một thuật toán tìm kiếm cục bộ để tìm kiếm một giải pháp tối ưu cho một vấn đề cụ thể. Ý tưởng của thuật toán Greedy là luôn chọn lựa tốt nhất trong những lựa chọn có sẵn, tại thời điểm hiện tại, mà không cân nhắc tới tương lai.

Thuật toán Greedy thường được sử dụng để giải quyết các bài toán tối ưu hóa, bao gồm cả các bài toán trên đồ thị. Cụ thể, trong bài toán này, thuật toán Greedy thường được sử dụng để tìm kiếm đường đi ngắn nhất hoặc cây khung nhỏ nhất trên đồ thị.

Cách thức hoạt động của thuật toán Greedy là từ bất kỳ đỉnh nào trên đồ thị, chọn đỉnh kề cạnh với nó có trọng số nhỏ nhất. Sau đó, từ đỉnh mới chọn, tiếp tục chọn đỉnh kề cạnh có trọng số nhỏ nhất cho đến khi tìm được đường đi ngắn nhất hoặc cây khung nhỏ nhất trên đồ thị. Tuy nhiên, thuật toán Greedy có thể không đưa ra kết quả tối ưu vì việc chọn lựa tốt nhất ở mỗi bước không nhất thiết dẫn tới kết quả tối ưu chung.

Dộ phức tạp về thời gian của thuật toán Greedy trong đồ thị là $O(|V| + |E| \log V)$ hoặc $O(V^2)$, tùy thuộc vào cách triển khai của thuật toán.

Về độ phức tạp về không gian, trong trường hợp tổng quát, độ phức tạp không gian là $O(|V| + |E|)$.

Dưới đây là mã giả của thuật toán:

```
function Greedy_shortest_path(G, start, end):
    visited = set() # tập các đỉnh đã thăm
    heap = [(0, start)] # heap lưu trữ các đỉnh và trọng số
    while heap:
        # lấy ra đỉnh có trọng số nhỏ nhất
        (weight, current_vertex) = heappop(heap)
        if current_vertex not in visited:
            visited.add(current_vertex)
            if current_vertex == end:
                return weight
            for neighbor, neighbor_weight in G[current_vertex].items():
                # thêm đỉnh kề với trọng số mới vào heap
                heappush(heap, (weight + neighbor_weight, neighbor))
    return float('inf') # không tìm thấy đường đi
```

6.2 Dijkstra

Thuật toán Dijkstra là một thuật toán tìm đường đi ngắn nhất trong đồ thị có trọng số dương. Thuật toán này đặc biệt hiệu quả khi tìm đường đi ngắn nhất từ một đỉnh đến tất cả các đỉnh còn lại trong đồ thị.

Thuật toán Dijkstra bắt đầu từ một đỉnh bất kỳ trong đồ thị và tính toán khoảng cách ngắn nhất từ đỉnh đó đến tất cả các đỉnh còn lại. Thuật toán này dựa trên một bảng lưu trữ các khoảng cách ngắn nhất đã tính được cho mỗi đỉnh trong đồ thị.

Các bước thực hiện thuật toán Dijkstra như sau:

1. Khởi tạo bảng khoảng cách: Đặt khoảng cách của đỉnh xuất phát bằng 0 và khoảng cách của các đỉnh còn lại bằng vô cùng.
2. Chọn đỉnh có khoảng cách nhỏ nhất từ đỉnh xuất phát và đánh dấu nó đã được xét.
3. Cập nhật khoảng cách: Với mỗi đỉnh kề với đỉnh vừa chọn, tính khoảng cách mới từ đỉnh xuất phát đến đỉnh đó, nếu khoảng cách mới nhỏ hơn khoảng cách hiện tại thì cập nhật khoảng cách mới.
4. Lặp lại bước 2 và 3 cho đến khi tất cả các đỉnh trong đồ thị đều đã được xét hoặc khoảng cách tới đỉnh còn lại không thay đổi.

Kết quả của thuật toán Dijkstra là bảng khoảng cách ngắn nhất từ đỉnh xuất phát đến tất cả các đỉnh trong đồ thị. Nếu trong quá trình thực hiện thuật toán, khoảng cách tới một đỉnh không thay đổi thì ta có thể dừng thuật toán và bỏ qua các đỉnh còn lại, bởi vì các đỉnh này sẽ không có đường đi ngắn nhất mới nào hơn đường đi ngắn nhất hiện tại.

Về độ phức tạp về thời gian của thuật toán có thể là $O(|V|^2)$, $O((|V| + |E|)log|V|)$ hay $O(|V|log|V| + |E|)$ tùy vào cách lưu trữ các đỉnh.

Về độ phức tạp về không gian của thuật toán là $O(|V|)$.

Dưới đây là mã giả của thuật toán:

```
function Dijkstra(graph, start, goal):
    # Khởi tạo bảng khoảng cách và đường đi ban đầu

    # khoảng cách từ start đến mỗi đỉnh ban đầu bằng vô cực
    dist = {node: float('inf') for node in graph}

    # đỉnh trước của mỗi đỉnh ban đầu là None
    prev = {node: None for node in graph}
    dist[start] = 0 # khoảng cách từ start đến start bằng 0

    # Khởi tạo hàng đợi ưu tiên và đưa start vào hàng đợi
    # tuple (khoảng cách, đỉnh) được sử dụng để sắp xếp hàng đợi
    pq = [(0, start)]
    visited = set()

    while pq:
        # Lấy đỉnh có khoảng cách ngắn nhất từ hàng đợi
        curr_dist, curr_node = heappop(pq)
        visited.add(curr_node)

        # Nếu đỉnh hiện tại là đỉnh đích, trả về đường đi ngắn nhất
        if curr_node == goal:
            path = []
            curr = curr_node
            while curr != start:
                path.append(curr)
                curr = prev[curr]
            path.append(start)
            path.reverse()
            return path, curr_dist

        # Cập nhật khoảng cách và đường đi cho các đỉnh kề
        for neighbor, weight in graph[curr].items():
            new_dist = curr_dist + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                prev[neighbor] = curr
                heappush(pq, (new_dist, neighbor))

    return None, None
```

```

if curr_node == goal:
    path = []
    while curr_node:
        path.append(curr_node)
        curr_node = prev[curr_node]
    return path[::-1], dist[goal]

# Duyệt qua các đỉnh kề với đỉnh hiện tại
for neighbor, weight in graph[curr_node].items():
    if neighbor not in visited:
        new_dist = curr_dist + weight
        if new_dist < dist[neighbor]:
            # Cập nhật khoảng cách và đỉnh trước của đỉnh kề
            # nếu khoảng cách mới tốt hơn
            dist[neighbor] = new_dist
            prev[neighbor] = curr_node
            heappush(pq, (new_dist, neighbor))

# Nếu không tìm thấy đường đi, trả về None và khoảng cách bằng vô cực
return None, float('inf')

```

Phân 7

So sánh các thuật toán

7.1 UCS, Greedy và A*

Thuật toán	Độ phức tạp thời gian	Độ phức tạp không gian	Tính tối ưu	Tính đầy đủ
UCS	$O(b^{d+1})$	$O(b^d)$	Tối ưu	Đầy đủ
Greedy	$O(b^m)$	$O(b^m)$	Không tối ưu	Không đầy đủ
A*	$O(b^d)$	$O(b^d)$	Tối ưu	Đầy đủ

Greedy Search: Đây là thuật toán tìm kiếm đường đi nhanh nhất dựa trên hàm *heuristic* ước lượng chi phí còn lại từ nút hiện tại đến đích. Nó chọn nút lân cận mà có hàm *heuristic* nhỏ nhất và mở rộng nó. Tuy nhiên, không có đảm bảo rằng nó tìm được đường đi tối ưu, và có thể bị mắc kẹt trong một vòng lặp vô hạn khi không thể tìm ra đường đi đến đích.

A* : Đây là một thuật toán tìm kiếm đường đi tối ưu kết hợp giữa UCS và Greedy Search. Nó sử dụng hàm *heuristic* để ước lượng chi phí còn lại từ nút hiện tại đến đích và kết hợp với chi phí đã đi để tính toán chi phí tổng. Nó chọn nút lân cận mà có chi phí tổng nhỏ nhất và mở rộng nó. Thuật toán này đảm bảo tìm được đường đi tối ưu và có độ phức tạp thời gian và không gian ổn định.

7.2 UCS và Dijkstra

Thuật toán	Độ phức tạp thời gian	Độ phức tạp không gian	Tính tối ưu	Tính đầy đủ
UCS	$O(b^{d+1})$	$O(b^d)$	Tối ưu	Đầy đủ
Dijkstra	Có thể là $O(V ^2)$, $O((V + E)log V)$ hay $O(V log V + E)$ tùy vào cách lưu trữ các đỉnh	$O(V)$	Tối ưu	Đầy đủ

Dijkstra là thuật toán này tìm kiếm đường đi tối ưu dựa trên đường đi ngắn nhất từ một điểm đến tất cả các điểm còn lại. Nó duyệt qua các nút lân cận của nút hiện tại, tìm ra đường đi ngắn nhất và đánh dấu nó. Sau đó, nó chọn nút chưa đánh dấu có khoảng cách ngắn nhất từ nút ban đầu và tiếp tục mở rộng. Độ phức tạp thời gian và không gian của thuật toán Dijkstra phụ thuộc vào số lượng đỉnh và cạnh trong đồ thị.

So sánh UCS và Dijkstra trong đồ thị:

- Cả UCS và Dijkstra đều tìm kiếm đường đi tối ưu.

- UCS và Dijkstra đều có độ phức tạp thời gian và không gian khác nhau, tuy nhiên Dijkstra có độ phức tạp thời gian tốt hơn khi số cạnh nhiều hơn số đỉnh trong đồ thị.
- UCS và Dijkstra đều đầy đủ, nghĩa là nếu có đường đi từ điểm bắt đầu đến điểm kết thúc thì chúng sẽ tìm được đường đi đó.
- UCS và Dijkstra khác nhau trong cách tính toán chi phí. UCS tính toán chi phí đường đi từ nút ban đầu đến nút hiện tại, trong khi Dijkstra tính toán khoảng cách từ nút ban đầu đến nút hiện tại.

Phần 8

Các tài liệu liên quan đến bài Lab

8.1 Link video của bài Lab

- Link: <https://youtu.be/sbEo05zKEEk>.
- Title: [Search in graph] - 20120049 - HCMUS.
- Posting date: 19/03/2023.

8.2 Tài liệu tham khảo

- Artificial Intelligence: a Modern Approach, EBook, Global Edition.
- CS 188 | Introduction to Artificial Intelligence, Fall 2018: Week 1 - Uninformed Search & A* Search and Heuristics.
- Python Program for Depth First Search or DFS for a Graph - Geeksforgeeks.
- Python Program for Breadth First Search or BFS for a Graph - Geeksforgeeks.
- Uniform-Cost Search (Dijkstra for large Graphs) - Geeksforgeeks.
- A* Search Algorithm - StackAbuse.