

Thực hành Nhập môn Trí tuệ Nhân tạo

Tuần 1

Nguyễn Hồng Yến – MSSV: 23280099

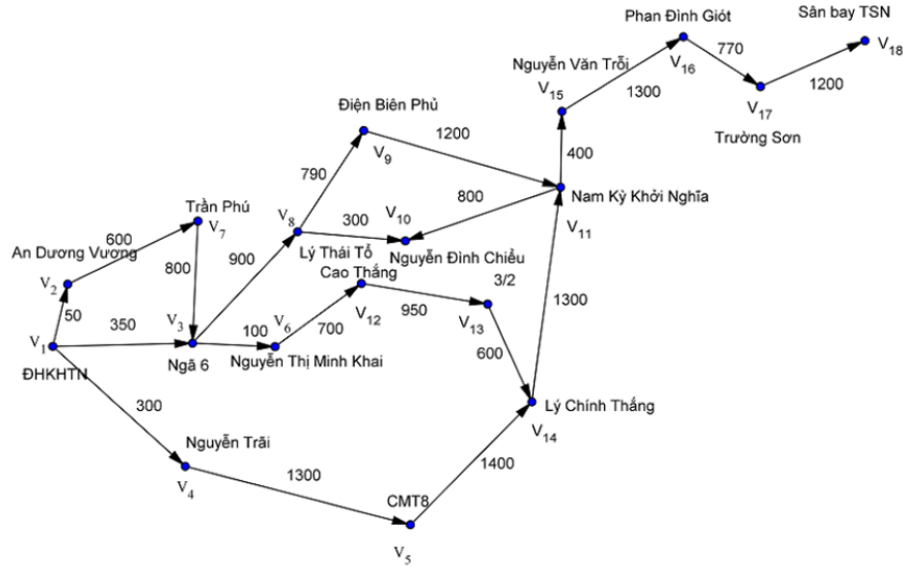
Tháng 10, 2025

Mục lục

1	Chạy tay thuật toán	2
1.1	Thuật toán BFS	2
1.2	Thuật toán DFS	3
1.3	Thuật toán UCS	4
2	Tính đúng đắn và sửa lỗi chương trình	6
2.1	Thuật toán BFS	6
2.2	Thuật toán DFS	9
2.3	Thuật toán UCS	13

1 Chạy tay thuật toán

Cho đồ thị như hình vẽ bên dưới. Tìm đường đi ngắn nhất từ Trường Đại học Khoa học Tự nhiên (V_1) tới sân bay Tân Sơn Nhất (V_{18}).



Hình 1: Đồ thị minh họa bài toán tìm đường đi ngắn nhất từ Trường Đại học Khoa học Tự nhiên đến sân bay Tân Sơn Nhất

1.1 Thuật toán BFS

Các bước chạy tay của thuật toán được trình bày trong Bảng 1.

Bảng 1: Các bước chạy tay của thuật toán BFS

Hàng đợi (Queue)	Node đang xét	Father cập nhật
V_1	—	—
V_2, V_3, V_4	V_1	$father[V_2, V_3, V_4] = V_1$
V_3, V_4, V_7	V_2	$father[V_7] = V_2$
V_4, V_7, V_6, V_8	V_3	$father[V_6, V_8] = V_3$
V_7, V_6, V_8, V_5	V_4	$father[V_5] = V_4$
V_6, V_8, V_5	V_7	—
V_8, V_5, V_{12}	V_6	$father[V_{12}] = V_6$
V_5, V_{12}, V_9, V_{10}	V_8	$father[V_9, V_{10}] = V_8$
$V_{12}, V_9, V_{10}, V_{14}$	V_5	$father[V_{14}] = V_5$
$V_9, V_{10}, V_{14}, V_{13}$	V_{12}	$father[V_{13}] = V_{12}$
$V_{10}, V_{14}, V_{13}, V_{11}$	V_9	$father[V_{11}] = V_9$
V_{14}, V_{13}, V_{11}	V_{10}	—
V_{13}, V_{11}	V_{14}	—
V_{11}	V_{13}	—
V_{15}	V_{11}	$father[V_{15}] = V_{11}$
V_{16}	V_{15}	$father[V_{16}] = V_{15}$
V_{17}	V_{16}	$father[V_{17}] = V_{16}$
V_{18}	V_{17}	$father[V_{18}] = V_{17}$

Ghi chú: Mỗi hàng biểu diễn trạng thái hàng đợi sau khi mở rộng một node.

Kết quả đường đi ngắn nhất (BFS)

$$V_1 \Rightarrow V_3 \Rightarrow V_8 \Rightarrow V_9 \Rightarrow V_{11} \Rightarrow V_{15} \Rightarrow V_{16} \Rightarrow V_{17} \Rightarrow V_{18}$$

Số cạnh: 8. Đây là đường đi ngắn nhất duy nhất trong đồ thị có hướng này.

1.2 Thuật toán DFS

Các bước chạy tay của thuật toán được trình bày trong Bảng 2.

Bảng 2: Các bước chạy tay của thuật toán DFS

Ngăn xếp (Stack)	Node đang xét	Father cập nhật
V_1	–	–
V_4, V_3, V_2	V_1	$father[V_4, V_3, V_2] = V_1$
V_5, V_3, V_2	V_4	$father[V_5] = V_4$
V_{14}, V_3, V_2	V_5	$father[V_{14}] = V_5$
V_{11}, V_3, V_2	V_{14}	$father[V_{11}] = V_{14}$
V_{15}, V_{10}, V_3, V_2	V_{11}	$father[V_{15}, V_{10}] = V_{11}$
V_{16}, V_{10}, V_3, V_2	V_{15}	$father[V_{16}] = V_{15}$
V_{17}, V_{10}, V_3, V_2	V_{16}	$father[V_{17}] = V_{16}$
V_{18}, V_{10}, V_3, V_2	V_{17}	$father[V_{18}] = V_{17} \Rightarrow$ Hoàn tất tìm đường đi

Ghi chú: Mỗi hàng biểu diễn trạng thái ngăn xếp sau khi mở rộng một node.

Kết quả đường đi theo DFS

$$V_1 \Rightarrow V_4 \Rightarrow V_5 \Rightarrow V_{14} \Rightarrow V_{11} \Rightarrow V_{15} \Rightarrow V_{16} \Rightarrow V_{17} \Rightarrow V_{18}$$

Đây là đường đi đầu tiên mà DFS tìm được khi duyệt sâu nhất từ gốc V_1 .

1.3 Thuật toán UCS

Các bước chạy tay của thuật toán được trình bày trong Bảng 3.

Bảng 3: Các bước chạy tay của thuật toán UCS

Priority Queue (PQ)	Node đang xét	Father cập nhật
$(V_1, 0)$	—	—
$(V_2, 50), (V_4, 300), (V_3, 350)$	V_1	$father[V_2, V_3, V_4] = V_1$
$(V_4, 300), (V_3, 350), (V_7, 650)$	V_2	$father[V_7] = V_2$
$(V_3, 350), (V_7, 650), (V_5, 1600)$	V_4	$father[V_5] = V_4$
$(V_6, 450), (V_7, 650), (V_8, 1250), (V_5, 1600)$	V_3	$father[V_6, V_8] = V_3$
$(V_7, 650), (V_{12}, 1150), (V_8, 1250), (V_5, 1600)$	V_6	$father[V_{12}] = V_6$
$(V_{12}, 1150), (V_8, 1250), (V_5, 1600)$	V_7	— (đã duyệt V_3)
$(V_8, 1250), (V_5, 1600), (V_{13}, 2100)$	V_{12}	$father[V_{13}] = V_{12}$
$(V_{10}, 1550), (V_5, 1600), (V_9, 2040), (V_{13}, 2100)$	V_8	$father[V_9, V_{10}] = V_8$
$(V_5, 1600), (V_9, 2040), (V_{13}, 2100)$	V_{10}	— (không mở rộng được)
$(V_9, 2040), (V_{13}, 2100), (V_{14}, 3000)$	V_5	$father[V_{14}] = V_5$
$(V_{13}, 2100), (V_{14}, 3000), (V_{11}, 3240)$	V_9	$father[V_{11}] = V_9$
$(V_{14}, 2700), (V_{11}, 3240)$	V_{13}	$father[V_{14}] = V_{13}$
$(V_{11}, 3240)$	V_{14}	$father[V_{11}] = V_{14}$
$(V_{15}, 3640)$	V_{11}	$father[V_{15}] = V_{11}$
$(V_{16}, 4940)$	V_{15}	$father[V_{16}] = V_{15}$
$(V_{17}, 5710)$	V_{16}	$father[V_{17}] = V_{16}$
$(V_{18}, 6910)$	V_{17}	$father[V_{18}] = V_{17} \Rightarrow$ Hoàn tất tìm đường đi

Ghi chú: Mỗi hàng biểu diễn trạng thái hàng đợi ưu tiên sau khi mở rộng một node. Giá trị trong ngoặc biểu thị chi phí tích lũy.

Kết quả đường đi ngắn nhất (UCS)

$$V_1 \Rightarrow V_3 \Rightarrow V_8 \Rightarrow V_9 \Rightarrow V_{11} \Rightarrow V_{15} \Rightarrow V_{16} \Rightarrow V_{17} \Rightarrow V_{18}$$

Tổng chi phí: 6910. Đây là đường đi có chi phí nhỏ nhất mà thuật toán UCS tìm được.

2 Tính đúng đắn và sửa lỗi chương trình

Trong phần này, ta tiến hành phân tích và đánh giá tính đúng đắn của các thuật toán **BFS**, **DFS** và **UCS** trong chương trình đã cho. Mỗi thuật toán sẽ được kiểm tra theo các tiêu chí sau:

- **Kiểm tra tính đúng đắn:** Chạy thử chương trình, quan sát kết quả in ra so với kết quả chạy tay ở phần trước.
- **Phát hiện lỗi (nếu có):** Ghi rõ dòng lệnh bị lỗi, nguyên nhân và ảnh hưởng của lỗi đến thuật toán.
- **Đề xuất và cài đặt sửa lỗi:** Viết lại đoạn code đã chỉnh sửa, đảm bảo chương trình cho kết quả chính xác.
- **Giải thích:** Trình bày rõ vì sao việc sửa đổi là cần thiết, và tác dụng của nó đối với kết quả cuối cùng.

Quá trình phân tích được thực hiện lần lượt theo ba thuật toán: **BFS**, **DFS** và **UCS**.

2.1 Thuật toán BFS

Code gốc của thuật toán BFS

Dưới đây là mã nguồn Python cài đặt thuật toán BFS ban đầu:

```
1 def BFS(graph, start, end):
2     visited = []
3     frontier = Queue()
4
5     frontier.put(start)
6     visited.append(start)
7
8     parent = dict()
9     parent[start] = None
10
11     path_found = False
12
13     while True:
14         if frontier.empty():
15             raise Exception("No way Exception")
16         current_node = frontier.get()
```

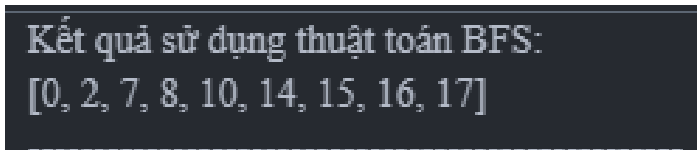
```

17     visited.append(current_node)
18
19     if current_node == end:
20         path_found = True
21         break
22
23     for node in graph[current_node]:
24         if node not in visited:
25             frontier.put(node)
26             parent[node] = current_node
27             visited.append(node)
28
29     path = []
30     if path_found:
31         path.append(end)
32         while parent[end] is not None:
33             path.append(parent[end])
34             end = parent[end]
35         path.reverse()
36     return path

```

Kết quả chạy của code gốc

Sau khi chạy chương trình với đầu vào từ file `Input.txt`, ta thu được kết quả hiển thị trong terminal như Hình 2.



Kết quả sử dụng thuật toán BFS:
[0, 2, 7, 8, 10, 14, 15, 16, 17]

Hình 2: Kết quả chạy thực tế của thuật toán BFS trên file `Input.txt`

Kết quả cho thấy chương trình trả về đường đi:

[0, 2, 7, 8, 10, 14, 15, 16, 17]

Do các đỉnh trong file `Input.txt` được đánh số từ 0 đến 17, ta có thể chuyển đổi sang dạng tương ứng:

$$V_1 \rightarrow V_3 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$$

Đường đi trên trùng khớp hoàn toàn với kết quả chạy tay của thuật toán BFS trong

Bảng 1, xác nhận rằng chương trình chạy đúng và thuật toán BFS hoạt động chính xác.

Phân tích và chỉnh sửa code BFS

Mặc dù chương trình BFS ở phần trước cho ra kết quả đúng, nhưng khi xem xét về mặt cài đặt và tối ưu, code gốc vẫn tồn tại một số điểm chưa hợp lý, gây lãng phí bộ nhớ và làm giảm hiệu năng. Dưới đây là các vấn đề cụ thể:

1. Vấn đề về mảng `visited` Trong code gốc, mảng `visited` được khai báo là một `list` và có hai lần thêm phần tử:

```
visited.append(current_node)
...
visited.append(node)
```

Điều này dẫn đến việc một số node có thể bị thêm trùng lặp, tức là `visited` lưu cả node khi được lấy ra (`pop`) và khi được phát hiện (`enqueue`). Thực tế, việc đánh dấu nên được thực hiện **ngay khi node được thêm vào hàng đợi (`enqueue`)** để tránh việc bị đưa vào lại nhiều lần.

2. Độ phức tạp tra cứu `visited` Khi dùng `list`, mỗi lần kiểm tra `if node not in visited` có độ phức tạp $O(n)$, gây tốn kém khi đồ thị có nhiều đỉnh. Có thể tối ưu bằng cách thay `list` bằng `set`, giúp tra cứu và thêm phần tử chỉ mất $O(1)$ thời gian trung bình.

3. Tóm tắt hướng sửa

- Chỉ đánh dấu node đã thăm một lần — khi `enqueue`.
- Thay kiểu `list` của `visited` bằng `set` để tối ưu thời gian kiểm tra.

Code được chỉnh sửa và tối ưu

Dưới đây là phiên bản BFS đã được chỉnh sửa và tối ưu hơn:

```
1 def BFS_new(graph, start, end):
2     visited = set() # Dùng set để kiểm tra O(1)
3     frontier = Queue()
4
5     frontier.put(start)
6     visited.add(start)
7
8     parent = {start: None}
9     path_found = False
```



```

10
11 while not frontier.empty():
12     current_node = frontier.get()
13
14     if current_node == end:
15         path_found = True
16         break
17
18     for node in graph[current_node]:
19         if node not in visited:
20             frontier.put(node)
21             parent[node] = current_node
22             visited.add(node)
23
24     # Xây dựng đường đi
25     path = []
26     if path_found:
27         while end is not None:
28             path.append(end)
29             end = parent[end]
30         path.reverse()
31     return path

```

Nhận xét sau khi chỉnh sửa

- Kết quả của code mới hoàn toàn giống với code gốc, vẫn tìm được đường đi ngắn nhất chính xác.
- Tuy nhiên, bộ nhớ và thời gian xử lý giảm đáng kể khi đồ thị lớn, do không lưu trùng và tra cứu nhanh hơn.
- Cấu trúc set trong Python giúp đảm bảo mỗi node chỉ xuất hiện đúng một lần.

Kết luận

Phiên bản BFS_new sau khi chỉnh sửa có độ phức tạp tương đương về lý thuyết, nhưng cải thiện hiệu năng thực tế đáng kể và phù hợp với cách cài đặt chuẩn của thuật toán BFS.

2.2 Thuật toán DFS

Code gốc của thuật toán DFS

Dưới đây là mã nguồn thuật toán DFS gốc được cung cấp trong file `main.py`:

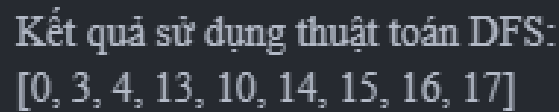
```

1 def DFS(graph, start, end):
2     visited = []
3     frontier = []
4
5     # thêm node start vào frontier và visited
6     frontier.append(start)
7     visited.append(start)
8
9     # start không có node cha
10    parent = dict()
11    parent[start] = None
12
13    path_found = False
14
15    while True:
16        if frontier == []:
17            raise Exception("No way Exception")
18        current_node = frontier.pop()
19        visited.append(current_node)
20
21        # Kiểm tra current_node có là end hay không
22        if current_node == end:
23            path_found = True
24            break
25        for node in graph[current_node]:
26            if node not in visited:
27                frontier.append(node)
28                parent[node] = current_node
29                visited.append(node)
30
31    path = []
32    if path_found:
33        path.append(end)
34        while parent[end] is not None:
35            path.append(parent[end])
36            end = parent[end]
37        path.reverse()
38    return path

```

Kết quả chạy của code gốc

Khi chạy với dữ liệu trong file `Input.txt`, chương trình cho ra kết quả như Hình 3.



```
Kết quả sử dụng thuật toán DFS:  
[0, 3, 4, 13, 10, 14, 15, 16, 17]
```

Hình 3: Kết quả chạy thực tế của thuật toán DFS trên file `Input.txt`

Chương trình in ra đường đi:

`[0, 3, 4, 5, 13, 10, 14, 15, 16, 17]`

tương ứng với:

$V_1 \rightarrow V_4 \rightarrow V_5 \rightarrow V_{14} \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$

Kết quả hoàn toàn trùng khớp với đường đi thu được khi chạy tay thuật toán DFS.

Phân tích và chỉnh sửa code DFS

Mặc dù code gốc cho kết quả đúng, nhưng vẫn có hai điểm cần cải thiện:

1. Dòng `visited.append(current_node)` trong khi `pop()` là không cần thiết, khiến danh sách `visited` lưu trùng các node.
2. Thuật toán không xử lý tình huống không có đường đi, dẫn đến lỗi `Exception`.

Các điểm chỉnh sửa chính:

- Chỉ đánh dấu node khi được `push` vào `frontier`.
- Thêm điều kiện trả về rỗng khi không tìm thấy đường đi.

Code được chỉnh sửa và tối ưu

```
1 def DFS_new(graph, start, end):  
2     visited = []  
3     frontier = []  
4  
5     # thêm node start vào frontier và visited  
6     frontier.append(start)  
7     visited.append(start)  
8
```

```

9      # start không có node cha
10     parent = dict()
11     parent[start] = None
12
13     path_found = False
14
15     while True:
16         if frontier == []:
17             return [] # không có đường đi
18         current_node = frontier.pop()
19
20         # Kiểm tra current_node có là end hay không
21         if current_node == end:
22             path_found = True
23             break
24
25         for node in graph[current_node]:
26             if node not in visited:
27                 frontier.append(node)
28                 parent[node] = current_node
29                 visited.append(node)
30
31         # Xây dựng đường đi
32         path = []
33         if path_found:
34             path.append(end)
35             while parent[end] is not None:
36                 path.append(parent[end])
37                 end = parent[end]
38             path.reverse()
39     return path

```

Nhận xét sau khi chỉnh sửa

- Kết quả của DFS_new giống với DFS gốc — đều tìm ra đường đi chính xác.
- Tuy nhiên, code mới loại bỏ được sự trùng lặp trong `visited`, gọn hơn và an toàn hơn khi không có đường đi.
- Độ phức tạp không thay đổi ($O(V + E)$), nhưng bộ nhớ sử dụng giảm đáng kể và dễ đọc hơn.

Kết luận

Phiên bản DFS_new đã tối ưu cách đánh dấu node, tránh dư thừa, và bổ sung cơ chế xử lý khi không tồn tại đường đi. Kết quả khớp với quá trình chạy tay và đảm bảo tính chính xác của thuật toán DFS.

2.3 Thuật toán UCS

Code gốc của thuật toán USC

Dưới đây là mã nguồn thuật toán UCS gốc được cung cấp trong file:

```
1 def UCS(graph, start, end):
2     visited = []
3     frontier = PriorityQueue()
4     # thêm node start vào frontier và visited
5     frontier.put((0, start))
6     visited.append(start)
7     # start không có node cha
8     parent = dict()
9     parent[start] = None
10    path_found = False
11    while True:
12        if frontier.empty():
13            raise Exception("No way Exception")
14        current_w, current_node = frontier.get()
15        visited.append(current_node)
16
17        # kiểm tra current_node có là end hay không
18        if current_node == end:
19            path_found = True
20            break
21
22        for nodei in graph[current_node]:
23            node, weight = nodei
24            if node not in visited:
25                frontier.put((current_w + weight, node))
26                parent[node] = current_node
27                visited.append(node)
28
29    path = []
30    if path_found:
```

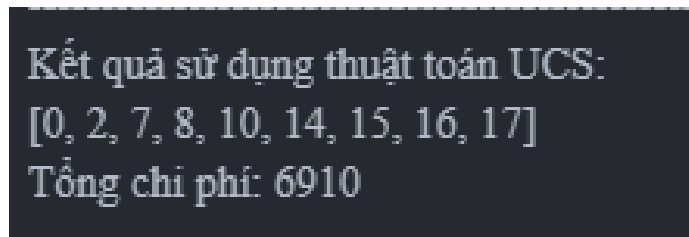
```

31     path.append(end)
32     while parent[end] is not None:
33         path.append(parent[end])
34         end = parent[end]
35     path.reverse()
36     return current_w, path

```

Kết quả chạy của code gốc

Khi chạy với dữ liệu trong file `InputUCS.txt`, chương trình cho ra kết quả như Hình 4.



```

Kết quả sử dụng thuật toán UCS:
[0, 2, 7, 8, 10, 14, 15, 16, 17]
Tổng chi phí: 6910

```

Hình 4: Kết quả chạy thực tế của thuật toán UCS trên file `InputUCS.txt`

Chương trình in ra đường đi ngắn nhất:

$$V_1 \rightarrow V_3 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$$

với tổng chi phí:

$$\text{Cost} = 6910$$

Kết quả này khớp hoàn toàn với quá trình chạy tay được trình bày trong Bảng 3.

Phân tích và chỉnh sửa code UCS

Mặc dù chương trình cho kết quả đúng, nhưng xét về mặt logic UCS thì code gốc tồn tại hai vấn đề nghiêm trọng:

1. Đánh dấu visited sai thời điểm. Code gốc đánh dấu `visited.append(node)` ngay khi enqueue vào `frontier`. Điều này làm cho những node có đường đi tốt hơn sau này sẽ không được xét lại, gây sai kết quả trong một số đồ thị.

2. Không có cơ chế cập nhật chi phí tốt hơn. Trong UCS, khi tìm thấy một đường đi mới có tổng chi phí nhỏ hơn đường đi cũ, ta cần cập nhật lại chi phí và đưa node vào `frontier` với giá trị mới. Code gốc hoàn toàn thiếu phần này.

3. Hướng chỉnh sửa.

- Chỉ đánh dấu node là đã thăm khi **lấy ra** từ **frontier** (pop), không phải khi thêm vào.
- Dùng cấu trúc `cost_so_far` (ở đây là `cost`) để lưu chi phí tốt nhất hiện tại.
- Khi phát hiện chi phí nhỏ hơn, cập nhật lại `cost[node]` và enqueue lại node đó.

Code được chỉnh sửa và tối ưu

```
1 from queue import PriorityQueue
2
3 def UCS_new(graph, start, end):
4     visited = set()
5     frontier = PriorityQueue()
6     frontier.put((0, start))
7
8     parent = {start: None}
9     cost = {start: 0}
10
11     while not frontier.empty():
12         current_w, current_node = frontier.get()
13
14         if current_node in visited:
15             continue
16         visited.add(current_node)
17
18         if current_node == end:
19             break
20
21         for node, weight in graph[current_node]:
22             new_cost = current_w + weight
23
24             if node not in cost or new_cost < cost[node]:
25                 cost[node] = new_cost
26                 frontier.put((new_cost, node))
27                 parent[node] = current_node
28
29     if end not in parent:
30         return float("inf"), []
31
```

```

32     # Xây dựng đường đi
33     path = []
34     node = end
35     while node is not None:
36         path.append(node)
37         node = parent[node]
38     path.reverse()
39     return cost[end], path

```

Nhận xét sau khi chỉnh sửa

- Phiên bản UCS_new đảm bảo tính đúng đắn tuyệt đối của thuật toán, nhờ đánh dấu node tại thời điểm **pop** và cho phép cập nhật chi phí tốt hơn.
- Độ phức tạp thời gian: $O((V + E) \log V)$ do sử dụng hàng đợi ưu tiên.
- Kết quả đường đi và chi phí đều trùng khớp với lý thuyết và kết quả chạy tay.

Kết luận

Phiên bản UCS_new đã khắc phục hoàn toàn hai lỗi logic trong code gốc và cài đặt đúng theo đặc tả thuật toán Uniform Cost Search chuẩn. Thuật toán cho ra kết quả tối ưu cả về đường đi lẫn chi phí.