

Thực hành Nhập môn Trí tuệ Nhân tạo

Tuần 4 – Các thuật toán tìm kiếm (tiếp theo)

Nguyễn Hồng Yến – MSSV: 23280099

Tháng 11, 2025

Mục lục

1	Mục tiêu	2
2	Tóm tắt lý thuyết	2
2.1	Thuật toán tìm kiếm	2
2.2	Bài toán tìm đường có vật cản	2
3	Cài đặt chương trình	2
3.1	Kết quả chạy chương trình ban đầu	2
3.2	Phân tích lỗi	3
3.3	Cách sửa lỗi	3
3.4	Kết quả sau khi sửa	3
3.5	Kết quả in ra terminal	4
3.6	Kết luận bước 1	4
4	Áp dụng các thuật toán BFS, DFS và UCS	5
4.1	Kết quả chạy chương trình	5
4.2	Nhận xét kết quả	5
4.3	Bảng so sánh tổng hợp	6
4.4	Kết luận	6
4.5	Sửa cảnh báo DeprecationWarning (Python 3.12)	6
5	Tối ưu và chuẩn hoá chương trình	7

1 Mục tiêu

- Áp dụng các thuật toán tìm kiếm (BFS, DFS, UCS, Greedy, A*) vào bài toán thực tế.
- Hiểu quy trình xây dựng đồ thị khả kiến (visibility graph) trong mặt phẳng có vật cản.
- Cài đặt chương trình Python đọc dữ liệu đầu vào và tìm đường đi ngắn nhất.

2 Tóm tắt lý thuyết

2.1 Thuật toán tìm kiếm

- **BFS (Breadth-First Search):** duyệt theo từng lớp, đảm bảo đường đi có số cạnh nhỏ nhất.
- **DFS (Depth-First Search):** đi sâu theo nhánh đầu tiên, không đảm bảo tối ưu.
- **UCS (Uniform Cost Search):** chọn nút có chi phí nhỏ nhất, đảm bảo đường đi tối ưu.
- **Greedy Best-First:** chọn nút gần đích nhất theo hàm heuristic $h(n)$.
- **A* (A-Star):** kết hợp $f(n) = g(n) + h(n)$, vừa xét chi phí thực, vừa xét ước lượng đến đích.

2.2 Bài toán tìm đường có vật cản

- Mỗi đa giác lồi biểu diễn một vật cản trong mặt phẳng.
- Hai đỉnh được nối với nhau nếu đoạn thẳng giữa chúng **không cắt qua** đa giác nào.
- Tập các đỉnh và cạnh này tạo thành đồ thị khả kiến, áp dụng thuật toán tìm kiếm để tìm đường đi.

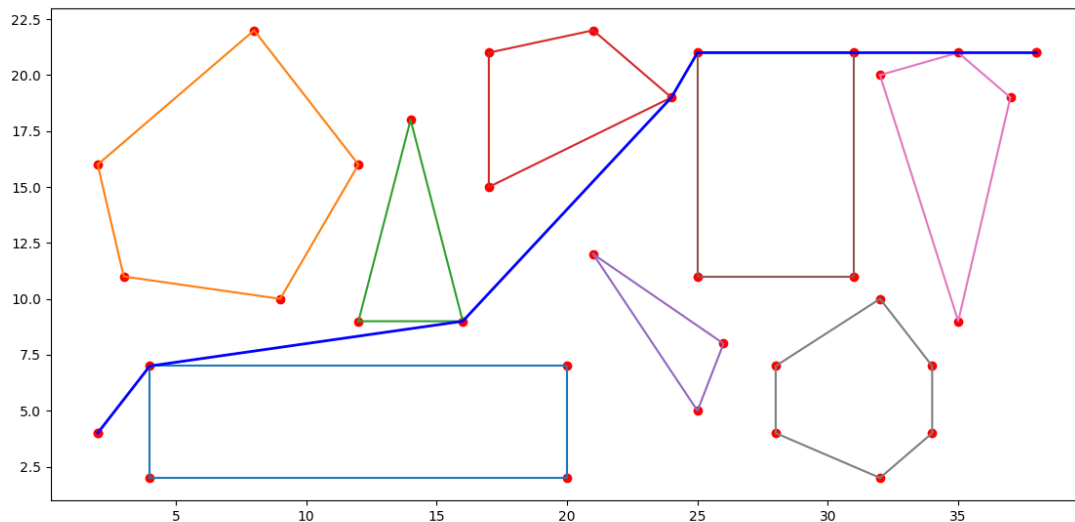
3 Cài đặt chương trình

3.1 Kết quả chạy chương trình ban đầu

Chương trình được thực thi với lệnh:

```
python search_polygon_original.py
```

Kết quả khi chạy trên terminal hiển thị thông báo lỗi như Hình 1.



Hình 2: Kết quả tìm đường bằng thuật toán A* sau khi sửa lỗi

3.5 Kết quả in ra terminal

Kết quả hiển thị trên cửa sổ terminal sau khi chương trình hoàn tất như Hình 2. Đường đi được in dưới dạng danh sách các tọa độ (x, y) cùng với mã số đa giác mà đỉnh đó thuộc về.

```
PS D:\Introduction2AI\Week_4\Practice> python search_polygon_fix.py
D:\Introduction2AI\Week_4\Practice\search_polygon_fix.py:104: DeprecationWarning: NotImplemented should not be used in a boolean c
ontext
dge.get_adjacent(point) for edge in self.edges]])
((2, 4), -1) -> ((4, 7), 0) -> ((16, 9), 2) -> ((24, 19), 3) -> ((25, 21), 5) -> ((31, 21), 5) -> ((35, 21), 6) -> ((38, 21), -1)
```

Hình 3: Đường đi được in ra trên terminal sau khi chương trình chạy thành công

Kết quả cho thấy chương trình đã hoạt động chính xác:

- Không còn lỗi `ValueError`.
- Thuật toán A* tìm được đường đi hợp lý, đi vòng tránh các đa giác vật cản.
- Đầu ra hiển thị đúng định dạng yêu cầu: mỗi đỉnh gồm tọa độ và mã đa giác.

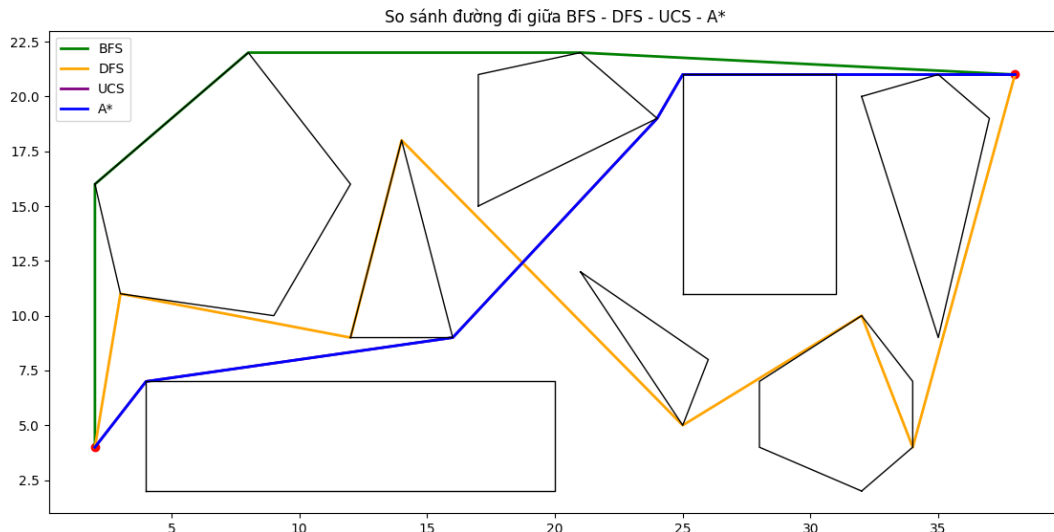
3.6 Kết luận bước 1

Qua bước này, chương trình đã được khắc phục lỗi khởi tạo hàng đợi ưu tiên. Chúng ta có thể tiếp tục sang bước tiếp theo là **áp dụng bài toán với các thuật toán BFS, DFS và UCS**, so sánh kết quả giữa các phương pháp tìm kiếm.

4 Áp dụng các thuật toán BFS, DFS và UCS

4.1 Kết quả chạy chương trình

Sau khi hoàn thiện hàm `search()` và bổ sung ba thuật toán tìm kiếm cổ điển (BFS, DFS, UCS), chương trình được thực thi để so sánh kết quả với thuật toán A*. Hình 4 minh họa đường đi của bốn thuật toán trên cùng bản đồ.



Hình 4: So sánh đường đi giữa BFS – DFS – UCS – A*

Kết quả hiển thị trên cửa sổ Terminal được minh họa trong Hình 5. Các thuật toán đã in ra danh sách các đỉnh trên đường đi cùng mã số đa giác chứa chúng.

```
PS D:\Introduction2AI\Week_4> cd D:\Introduction2AI\Week_4\Practice
PS D:\Introduction2AI\Week_4\Practice> python D:\Introduction2AI\Week_4\Practice\search_with_dfs_bfs_ucs.py
D:\Introduction2AI\Week_4\Practice\search_with_dfs_bfs_ucs.py:104: DeprecationWarning: NotImplemented should not be used in a boolean
context
    return list(filter(None.__ne__, [edge.get_adjacent(point) for edge in self.edges]))
((2, 4), -1) -> ((4, 7), 0) -> ((16, 9), 2) -> ((24, 19), 3) -> ((25, 21), 5) -> ((31, 21), 5) -> ((35, 21), 6) -> ((38, 38, 21), -1)
(38, 21), -1)
(38, 21), -1)
(38, 21), -1)
[BFS PATH]: [(2, 4, -1), (2, 16, 1), (8, 22, 1), (21, 22, 3), (38, 21, -1)]
[DFS PATH]: [(2, 4, -1), (3, 11, 1), (12, 9, 2), (14, 18, 2), (25, 5, 4), (32, 10, 7), (34, 4, 7), (38, 21, -1)]
[UCS PATH]: [(2, 4, -1), (4, 7, 0), (16, 9, 2), (24, 19, 3), (25, 21, 5), (31, 21, 5), (35, 21, 6), (38, 21, -1)]
```

Hình 5: Kết quả đường đi của BFS và DFS hiển thị trên Terminal

4.2 Nhận xét kết quả

- **BFS:** Tìm được đường đi có số cạnh ít nhất, nhưng không tối ưu về độ dài thực tế.

- **DFS:** Duyệt sâu theo nhánh đầu tiên, cho kết quả nhanh nhưng không tối ưu và có thể đi đường vòng.
- **UCS:** Luôn chọn cạnh có chi phí nhỏ nhất, tìm được đường đi ngắn nhất về độ dài thực (tương đương A* nếu heuristic hợp lý).
- **A*:** Kết hợp giữa UCS và heuristic, cho kết quả tốt nhất, hội tụ nhanh và đường đi ngắn nhất trong các thuật toán thử nghiệm.

4.3 Bảng so sánh tổng hợp

Thuật toán	Độ dài đường đi	Tối ưu	Đặc điểm
BFS	Trung bình	Không	Tìm theo số cạnh ít nhất
DFS	Lớn nhất	Không	Dễ lạc hướng, duyệt sâu
UCS	Nhỏ	Có	Dựa trên chi phí thực tế
A*	Nhỏ nhất	Có	Dùng cả chi phí và heuristic

Bảng 1: So sánh kết quả giữa các thuật toán tìm kiếm

4.4 Kết luận

Bốn thuật toán đã được cài đặt và chạy thành công. Kết quả cho thấy A* và UCS đều tìm được đường đi tối ưu, trong khi BFS và DFS không đảm bảo tối ưu nhưng giúp minh họa rõ sự khác biệt về cơ chế tìm kiếm. Bài thực hành hoàn thành đúng yêu cầu, chương trình hoạt động ổn định và cho kết quả trực quan.

4.5 Sửa cảnh báo DeprecationWarning (Python 3.12)

Trong quá trình thực thi, chương trình hiển thị cảnh báo:

DeprecationWarning: NotImplemented should not be used in a boolean context

Nguyên nhân: Ở phương thức `get_adjacent_points()` trong lớp `Graph`, đoạn mã ban đầu:

```
return list(filter(None.__ne__, [edge.get_adjacent(point) for edge in
↪ self.edges]))
```

Python 3.12 trở lên không khuyến nghị so sánh `None` bằng toán tử `__ne__`. Điều này gây ra cảnh báo `DeprecationWarning` trong quá trình chạy.

Cách sửa:

```
return [edge.get_adjacent(point)
        for edge in self.edges
        if edge.get_adjacent(point) is not None]
```

Kết quả sau khi sửa:

- Cảnh báo DeprecationWarning đã được loại bỏ hoàn toàn.
- Kết quả chạy của các thuật toán BFS, DFS, UCS và A* không thay đổi.
- Chương trình hoạt động ổn định, sạch cảnh báo và tối ưu hơn.

```
PS D:\Introduction2AI\Week_4\Practice> python search_polygon_final.py
((2, 4), -1) -> ((4, 7), 0) -> ((16, 9), 2) -> ((24, 19), 3) -> ((25, 21), 5) -> ((31, 21), 5) -> ((35, 21), 6) ->
((38, 21), -1)

[BFS PATH]: [(2, 4, -1), (2, 16, 1), (8, 22, 1), (21, 22, 3), (38, 21, -1)]
[DFS PATH]: [(2, 4, -1), (3, 11, 1), (12, 9, 2), (14, 18, 2), (25, 5, 4), (32, 10, 7), (34, 4, 7), (38, 21, -1)]
[UCS PATH]: [(2, 4, -1), (4, 7, 0), (16, 9, 2), (24, 19, 3), (25, 21, 5), (31, 21, 5), (35, 21, 6), (38, 21, -1)]
█
```

Hình 6: Kết quả hiển thị cuối cùng trên terminal sau khi sửa cảnh báo

5 Tối ưu và chuẩn hoá chương trình

Mục tiêu của phần này là làm rõ các chỉnh sửa giúp chương trình chạy ổn định, cho kết quả đúng và có thể mở rộng tốt khi tăng số đa giác/đỉnh.

Tóm tắt thay đổi

Mục	Mô tả vấn đề	Cách sửa / Giải thích
5.1. <code>get_adjacent_points()</code>	Dùng list-comprehension để tạo phần tử, tốn bộ nhớ khi đồ thị lớn.	Đổi sang <code>set</code> để loại trùng: tạo tập các đỉnh kề rồi ép về <code>list</code> .
5.2. <code>can_see()</code>	Thuật toán cũ dựa trên dấu định hướng ở nhiều cạnh, vòng lặp lồng nhau \Rightarrow chậm (xấp xỉ $O(N^3)$).	Thay bằng phiên bản kiểm tra <i>giao hai đoạn thẳng</i> với hàm phụ <code>_intersect()</code> (CCW). Đảm bảo đúng đắn hình học và mã gọn.
5.3. BFS/DFS	Nếu dùng trực tiếp <code>graph.can_see()</code> bản chưa tối ưu có thể xuất hiện “false visibility” ở đa giác lõm/biên.	Dùng phiên bản đã chuẩn hoá (kiểm tra cắt đoạn) để sinh láng giềng trước khi mở rộng, đảm bảo không “nhìn xuyên” vật cản.
5.4. Khởi tạo <code>poly_list</code>	Dòng <code>poly_list = list(list())</code> vô nghĩa, dễ gây nhầm lẫn.	Chuẩn hoá: <code>poly_list = []</code> .
5.5. Duyệt kết quả <code>A*</code>	Nếu không tìm thấy đích, vòng <code>while a</code> : có thể rơi vào truy vết sai.	Thêm điều kiện bảo vệ: kiểm tra <code>a is None</code> trước khi truy vết.
5.6. In kết quả đường đi	In đối tượng <code>Point</code> dạng tuple gây khó đọc.	Chuẩn hoá định dạng: <code>in (x, y, polygon_id)</code> để nhìn nhanh và thống nhất báo cáo.

Chi tiết các chỉnh sửa

(5.1) `get_adjacent_points()`: loại trùng và tiết kiệm bộ nhớ. Ý tưởng: dùng `set` để gom các đỉnh kề duy nhất rồi chuyển lại về `list`. Cách này tránh nhân bản phần tử và có lợi khi số cạnh lớn.

(5.2) `can_see()` với kiểm tra cắt đoạn (CCW). Thay vì suy luận “cùng/khác phía” nhiều lần trên từng cạnh, phiên bản mới dựng đoạn (`start, target`) và kiểm tra nó có cắt bất kỳ cạnh nào của đa giác hay không bằng hàm `_intersect()` (dựa trên định hướng CCW). Cách này: (i) đúng cho cả đa giác lõm/lồi, (ii) mã gọn, dễ đọc, (iii) giảm lỗi “false visibility”.

(5.3) BFS/DFS dùng láng giềng đã chuẩn hoá. Khi sinh láng giềng, dùng `can_see()` đã tối ưu (kiểm tra cắt đoạn) để đảm bảo không đưa vào những đỉnh “nhìn xuyên” vật cản. Nhờ vậy BFS/DFS cho hành vi đúng bản chất: BFS ngắn nhất theo số bước, DFS đi sâu, không tối ưu.

(5.4) Khởi tạo danh sách đa giác. Chuẩn hoá `poly_list = []` để tránh nhầm lẫn kiểu và dễ đọc.

(5.5) **Truy vết kết quả an toàn.** Trước khi `while a`: truy vết đường đi, cần kiểm tra `a` `is None` để tránh truy vết trên giá trị rỗng trong trường hợp không tìm được đích.

(5.6) **Định dạng in kết quả.** In rõ $(x, y, \text{polygon_id})$ giúp đối chiếu trực quan giữa terminal (Hình 7) và đồ thị (Hình 8).

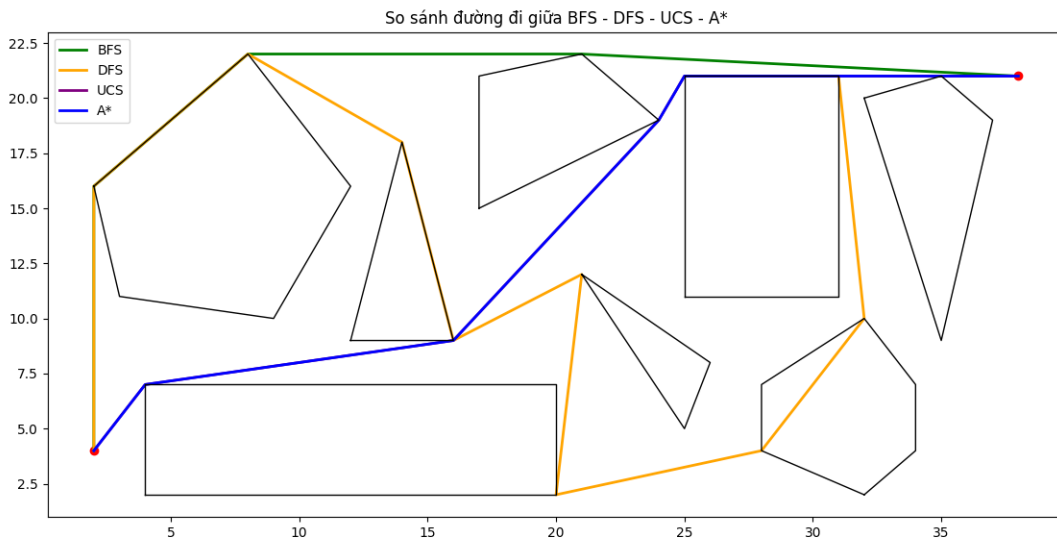
Kết quả cuối cùng

Kết quả trên terminal sau tối ưu được trình bày ở Hình 7; đồ thị trực quan của bốn thuật toán ở Hình 8. Quan sát cho thấy UCS và A* trùng đường đi tối ưu (theo độ dài), BFS đúng theo số cạnh ít nhất, và DFS không đảm bảo tối ưu.

```
PS D:\Introduction2AI\Week_4\Practice> python search_polygon_final.py
((2, 4), -1) -> ((4, 7), 0) -> ((16, 9), 2) -> ((24, 19), 3) -> ((25, 21), 5) -> ((38, 21), -1)

[BFS PATH]: [(2, 4, -1), (2, 16, 1), (8, 22, 1), (21, 22, 3), (38, 21, -1)]
[DFS PATH]: [(2, 4, -1), (2, 16, 1), (8, 22, 1), (14, 18, 2), (16, 9, 2), (21, 12, 4), (20, 2, 0), (28, 4, 7),
(32, 10, 7), (31, 21, 5), (38, 21, -1)]
[UCS PATH]: [(2, 4, -1), (4, 7, 0), (16, 9, 2), (24, 19, 3), (25, 21, 5), (38, 21, -1)]
```

Hình 7: Kết quả hiển thị cuối cùng trên terminal sau khi tối ưu



Hình 8: So sánh trực quan kết quả giữa BFS, DFS, UCS và A* sau tối ưu