ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

----------&----------

CƠ SỞ TRÍ TUỆ NHÂN TẠO

LAB 01: SEARCH IN GRAPH

SVTH: 21120280 Lý Minh Khuê Lớp 21_21

Thành phố Hồ Chí Minh - 2023

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

----------&----------

CƠ SỞ TRÍ TUỆ NHÂN TẠO

| Đề tài |
SEARCH IN GRAPH

| Giáo viên hướng dẫn |

Thầy Lê Hoài Bắc
Thầy Nguyễn Bảo Long

Thành phố Hồ Chí Minh – 2023

LỜI CẢM ƠN

Lời đầu tiên, tụi em xin gửi cảm ơn chân thành đến thầy Lê Hoài Bắc và thầy Nguyễn Bảo Long, giảng viên bộ môn Cơ sở trí tuệ nhân tạo, Trường Đại học Khoa học Tự nhiên Thành phố Hồ Chí Minh đã giúp đỡ tận tình, tạo điều kiện để tụi em hoàn thành báo cáo một cách tốt nhất.

Sau đề tài nghiên cứu báo cáo đó, tụi em đã học hỏi và tích lũy được nhiều kinh nghiệm để hoàn thiện và phát triển bản thân. Tuy nhiên, do khả năng tiếp thu thực tế còn nhiều hạn hẹp, kiến thức chưa sâu rộng. Mặc dù bản thân đã cố gắng hết sức nhưng chắc chắn báo cáo khó tránh khỏi những thiếu sót, kính mong thầy xem xét và góp ý để bài báo cáo của tụi em được hoàn thiện và tốt hơn.

Xin chân thành cảm ơn!

MỤC LỤC

LỜI CẢM ƠN 3
MỤC LỤC 4
BÀI TOÁN TÌM KIẾM 6
1. Bài toán tìm kiếm: 6
1.1. Định nghĩa bài toán tìm kiếm: 6
1.2. Thành phần chính của bài toán tìm kiếm: 6
1.3. Mã giả chung để giải quyết bài toán tìm kiếm: 6
1.4. Minh họa bằng ví dụ: 7
2. Uninformed Search vs. Informed Search: 11
2.1. Uninformed Search: 11
2.2. Informed Search: 12
2.3. So sánh Uninformed Search và Informed Search 13
NGHIÊN CỨU THUẬT TOÁN 14
1. Depth-first-search (DFS): 14
1.1. Ý tưởng: 14
1.2. Mã giả 14
1.3. Đặc điểm: 15
1.4. Minh họa: 16
2. Breadth-first-search (BFS): 19
2.1. Ý tưởng: 19
2.2. Mã giả 19
2.3. Đặc điểm: 19
2.4. Minh họa: 20
3. Uniform-cost-search (UCS): 23
3.1. Ý tưởng: 23
3.2. Mã giả 23
3.3. Đặc điểm: 24
3.4. Minh họa: 24
4. AStar Search: 27
4.1. Ý tưởng: 27
4.2. Mã giả 28

4.3. Đặc điểm: 28
4.4. Hàm Heuristic: 29
4.5. Một số hàm Heuristic trong AStar: 30
4.6. Minh họa: 31
So sánh các thuật toán 35
1. UCS, Greedy, AStar 35
2. UCS, Dijsktra 36
Cài đặt thuật toán 37
1. DFS: 37
2. BFS: 38
3. UCS 39
4. AStar 40
TÀI LIỆU THAM KHẢO 41

BÀI TOÁN TÌM KIẾM

1. Bài toán tìm kiếm:

1.1. Định nghĩa bài toán tìm kiếm:

Bài toán tìm kiếm là một khái niệm cơ bản trong khoa học máy tính và trí tuệ nhân tạo. Nó có thể được **định nghĩa** như sau:

Một bài toán tìm kiếm liên quan đến việc tìm đường đi từ trạng thái ban đầu đến trạng thái đích bằng cách điều hướng qua không gian trạng thái thông qua việc sử dụng một chuỗi hành động.

1.2. Thành phần chính của bài toán tìm kiếm:

Trạng thái ban đầu (S0): Đây là điểm bắt đầu của bài toán. Nó đại diện cho trạng thái hiện tại mà quá trình tìm kiếm bắt đầu.

Trạng thái đích (Sg): Trạng thái đích là trạng thái mà thuật toán tìm kiếm cố gắng đạt được. Giải pháp được tìm thấy khi trạng thái hiện tại trùng với trạng thái đích.

Không gian trạng thái: Không gian trạng thái bao gồm tất cả các trạng thái có thể đạt được từ trạng thái ban đầu bằng cách áp dụng một chuỗi hành động.

Hành động: Đây là các bước di chuyển hoặc thao tác có thể thực hiện để chuyển từ một trạng thái này sang trạng thái khác.

Hàm chi phí: Hàm chi phí ấn định một chi phí bằng số cho mỗi hành động, cho biết mức độ tốn kém hoặc mong muốn thực hiện một hành động cụ thể. Chi phí của một đường đi từ trạng thái ban đầu đến trạng thái mục tiêu là tổng chi phí của các hành động trên đường đi đó.

1.3. Mã giả chung để giải quyết bài toán tìm kiếm:

MÃ GIẢ CHUNG ĐỂ GIẢI QUYẾT BÀI TOÁN TÌM KIẾM

```
Input: trạng thái ban đầu (S0)
Output: trạng thái kết thúc (Sg)
1:  add S0 to OPEN
2:  loop
3:      get current_state from OPEN
4:      remove current_state from OPEN
5:      if current_state is Sg
6:          return path
7:      add current_state to CLOSED
8:      foreach action in get_possible_actions(current_state)
9:          next_state = apply_action(current_state, action)
10:         if next_state not in OPEN and next_state not in CLOSED
11:             add next_state to OPEN
12: end loop
13: return "No path found"
```

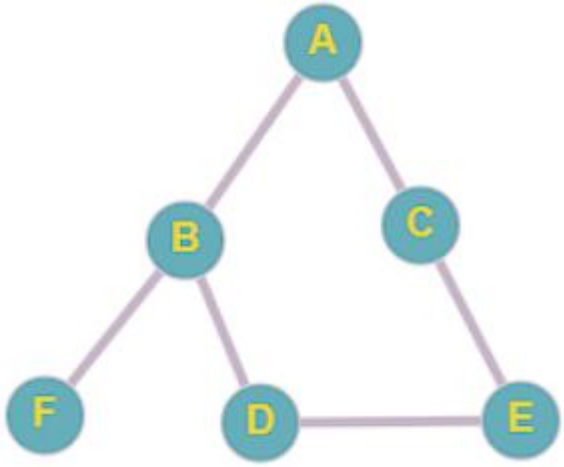
Chú thích:

- ***OPEN***: là một hàng đợi lưu trạng thái chưa được kiểm tra.
- ***CLOSED***: là tập hợp lưu trạng thái đã được kiểm tra.
- ***current_state***: trạng thái hiện tại
- ***get_possible_actions(current_state)***: trả về danh sách các hành động có thể thực hiện từ trạng thái hiện tại.
- ***apply_action(current_state, action)***: thực hiện hành động trên trạng thái hiện tại để tạo ra trạng thái mới.
- ***Path***: đường đi

1.4. Minh họa bằng ví dụ:

Để minh họa cách các thuật toán tìm kiếm hoạt động, chúng ta sẽ sử dụng một ví

dụ đơn giản với một đồ thị có 5-6 nút. Chúng ta sẽ sử dụng cùng một đồ thị để so sánh hiệu suất của các thuật toán.



Hình 1. Đồ thị có 6 nút(vẽ bằng https://graphonline.ru/en/)

- Trạng thái ban đầu (S_0) là nút A.
- Trạng thái mục tiêu (S_g) là nút E.
- Bước 1: Khởi tạo
  - Thêm trạng thái ban đầu (A) vào ***OPEN***: ***OPEN*** = [A]
  - Không có trạng thái nào được kiểm tra: ***CLOSED*** = []
- Bước 2: Kiểm tra ***OPEN*** trống hay không
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = A
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = []
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm A vào ***CLOSED***: ***CLOSED*** = [A]
  - Lấy các hành động có thể từ ***current_state***: ***get_possible_actions(A)***:= [B, C]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Thực hiện hành động B từ A: ***apply_action(A, B)*** = B

- Thực hiện hành động C từ A: ***apply_action(A, C)*** = C
- Thêm B và C vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [B, C]

- Bước 3: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = B
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [C]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B]
  - Lấy các hành động có thể từ ***current_state***: ***get_possible_actions(B):***= [F, D]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Thực hiện hành động F từ B: ***apply_action(B, F)*** = F
    - Thực hiện hành động D từ B: ***apply_action(B, D)*** = D
    - Thêm F và D vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [C, F, D]
- Bước 4: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = C
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [F, D]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B, C]
  - Lấy các hành động có thể từ ***current_state***: ***get_possible_actions(C):***= [E]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Thực hiện hành động E từ C: ***apply_action(C, E)*** = E
    - Thêm F và D vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [F, D, E]

- Bước 5: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = F
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [D, E]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B, C, F]
  - Lấy các hành động có thể từ ***current_state***: ***get_possible_actions(F)***:= []
  - Không có hành động nào nên qua bước tiếp theo
- Bước 6: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = D
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [E]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B, C, F, D]
  - Lấy các hành động có thể từ ***current_state***: ***get_possible_actions(D)***:= []
  - Không có hành động nào nên qua bước tiếp theo
- Bước 7: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = E
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = []
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Đúng
  - Trả về đường đi từ E về A.

2. Uninformed Search vs. Informed Search:

2.1. Uninformed Search:

Uninformed Search (Tìm kiếm không đủ thông tin), còn được gọi là tìm kiếm mù, dựa hoàn toàn vào cấu trúc của bài toán và không sử dụng bất kỳ kiến thức hay thông tin về bài toán ngoài trạng thái ban đầu và những hành động có thể thực hiện. Thay vào đó, các thuật toán này xác định không gian tìm kiếm một cách có hệ thống nhưng không xem xét chi phí để đạt được mục tiêu hoặc khả năng tìm ra giải pháp.

Trong phần này, chúng ta sẽ minh họa hai thuật toán tìm kiếm không đủ thông tin phổ biến: Tìm kiếm theo chiều sâu (DFS) , Tìm kiếm theo chiều rộng (BFS) và Tìm kiếm chi phí đều (UCS).

Dưới đây là một số đặc điểm chính của thuật toán tìm kiếm mù:

- Tìm kiếm có hệ thống: Thuật toán tìm kiếm không xác định khám phá không gian tìm kiếm có hệ thống, bằng cách mở rộng tất cả các nút con của một nút (ví dụ: BFS) hoặc bằng cách khám phá càng sâu càng tốt trong một đường dẫn duy nhất trước khi quay lại (ví dụ: DFS).
- Không có thuật giải: Các thuật toán tìm kiếm thiếu thông tin không sử dụng thông tin bổ sung, chẳng hạn như thuật giải hoặc ước tính chi phí, để hướng dẫn quá trình tìm kiếm.
- Triển khai đơn giản: Các thuật toán tìm kiếm không chính xác thường dễ triển khai và dễ hiểu, khiến chúng trở thành điểm khởi đầu tốt cho các thuật toán phức tạp hơn.
- Không hiệu quả trong các bài toán phức tạp: Các thuật toán tìm kiếm không chính xác có thể không hiệu quả trong các bài toán phức tạp với không gian tìm kiếm lớn, dẫn đến số lượng trạng thái được khám phá tăng theo cấp số nhân.

2.2. Informed Search:

Informed Search (Tìm kiếm có thông tin) là một loại thuật toán tìm kiếm sử dụng thông tin bổ sung để hướng dẫn quá trình giải quyết vấn đề một cách hiệu quả hơn so với các thuật toán tìm kiếm không có thông tin. Thông tin này có thể là dạng hàm Heuristics, ước tính chi phí hoặc dữ liệu liên quan khác để ưu tiên mở rộng và duyệt các trạng thái.

Các ví dụ về các thuật toán tìm kiếm có thông tin bao gồm tìm kiếm A*, tìm kiếm Best-First và tìm kiếm Greedy.

Dưới đây là phiên bản dịch của các đặc điểm chính của các thuật toán tìm kiếm có thông tin trong lĩnh vực Trí tuệ nhân tạo:

- Sử dụng Heuristics (Thông tin chướng ngại học): Các thuật toán tìm kiếm có thông tin sử dụng thông tin chướng ngại học hoặc thông tin bổ sung để hướng dẫn quá trình tìm kiếm và ưu tiên việc mở rộng các nút.
- Hiệu quả hơn: Các thuật toán tìm kiếm có thông tin được thiết kế để hiệu quả hơn so với các thuật toán tìm kiếm không có thông tin như tìm kiếm theo chiều rộng hoặc tìm kiếm theo chiều sâu bằng cách tránh sự khám phá của các con đường không khả thi và tập trung vào những con đường có triển vọng hơn.
- Định hướng đến mục tiêu: Các thuật toán tìm kiếm có thông tin được thiết kế để định hướng đến mục tiêu, có nghĩa là chúng được tạo ra để tìm giải pháp cho một vấn đề cụ thể.
- Dựa trên chi phí: Các thuật toán tìm kiếm có thông tin thường sử dụng ước tính dựa trên chi phí để đánh giá các nút, chẳng hạn như chi phí ước tính để đạt được mục tiêu hoặc chi phí của một con đường cụ thể.
- Ưu tiên: Các thuật toán tìm kiếm có thông tin ưu tiên việc mở rộng các trạng thái dựa trên thông tin bổ sung có sẵn, thường dẫn đến giải quyết vấn đề hiệu quả hơn.

- Tính tối ưu: Các thuật toán tìm kiếm có thông tin có thể đảm bảo giải pháp tối ưu nếu thông tin Heuristics được sử dụng là có thể chấp nhận được nếu nó không bao giờ đánh giá quá cao chi phí đạt được mục tiêu, tức là chi phí mà nó ước tính để đạt được mục tiêu không cao hơn chi phí thấp nhất có thể từ trạng thái hiện tại trong đường đi.

2.3. So sánh Uninformed Search và Informed Search

Tiêu chí	Informed Search	Uninformed Search
Chiến lược	Tìm kiếm có thông tin, sử dụng hàm Heuristics	Tìm kiếm theo một hệ thống
Hiệu suất	Tìm ra giải pháp nhanh hơn khi hàm Heuristics hiệu quả	Tìm ra giải pháp chậm hơn
Độ hoàn thành	Có thể không hoàn thành hoặc đi xa khỏi lời giải nếu hàm Heuristics không phù hợp. Có thể lặp vô tận	Luôn hoàn thành
Chi phí	Chi phí thấp	Chi phí cao
Kích thước bài toán tìm kiếm	Có thể xử lý các bài toán tìm kiếm lớn	Tìm ra giải pháp cho một bài toán tìm kiếm lớn là khó khi phải thời gian tìm kiếm lâu.

NGHIÊN CỨU THUẬT TOÁN

1. Depth-first-search (DFS):

1.1. Ý tưởng:

DFS xác định không gian tìm kiếm bằng cách duyệt càng sâu càng tốt dọc theo một nhánh trước khi quay lại.

Nó hoạt động như sau:

- Bắt đầu ở trạng thái ban đầu.
- Duyệt càng sâu càng tốt dọc theo một nhánh cho đến khi không còn trạng thái nào để duyệt hoặc tìm thấy trạng thái mục tiêu.
- Quay lại và duyệt các nhánh khác.
- Tiếp tục cho đến khi tìm thấy trạng thái mục tiêu hoặc không còn trạng thái nào để duyệt

1.2. Mã giả

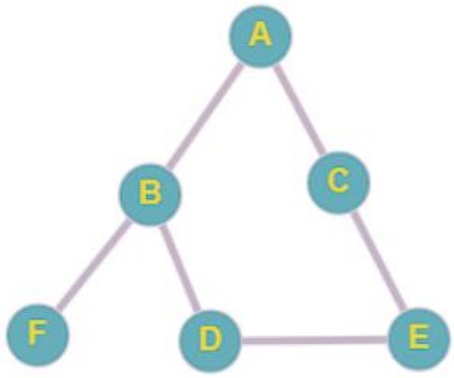
DEPTH-FIRST-SEARCH (DFS)

```
    Input: trạng thái ban đầu (S0)
    Output: trạng thái kết thúc (Sg)
1:  add S0 to OPEN
2:  loop
3:      get the first element from OPEN and assign to current_state
4:      remove current_state from OPEN
5:      if current_state is Sg
6:          return path
7:      add current_state to CLOSED
8:      foreach neighbour in get_neighbours(current_state)
9:          if neighbour not in OPEN and neighbour not in CLOSED
10:             add neighbour to OPEN as the first element
11              path[neighbour] = current_state
12: end loop
13: return "No path found"
```

1.3. Đặc điểm:

- **Độ hoàn thành:** Nó phụ thuộc vào không gian tìm kiếm. Nếu không gian tìm kiếm là hữu hạn thì **DFS** sẽ hoàn tất. Tuy nhiên, nếu có vô số lựa chọn thay thế, nó có thể không tìm ra giải pháp.
  Nhưng nếu ta sử dụng ***CLOSED*** set để lưu các vị trí đã qua thì DFS sẽ luôn tìm ra giải pháp nếu có.
- **Tính tối ưu:** DFS không đảm bảo đường đi ngắn nhất tới trạng thái mục tiêu. Vì nó tìm ra giải pháp "trái cùng ", bất kể độ sâu hay chi phí.
- **Độ phức tạp thời gian:**
  - DFS sẽ duyệt qua mỗi nút một lần nên độ phức tạp sẽ ít nhất là $\boldsymbol{O(V^h)}$ với h là chiều cao của cây.
  - Ở cấu trúc dữ liệu đồ thị thì nếu đồ thị biểu diễn dưới ma trận kề thì độ phức tạp là $\boldsymbol{O(V^2)}$còn sử dụng danh sách kề thì độ phức tạp là $\boldsymbol{O(V+E)}$
  - Với **V** là số đỉnh và **E** là số cạnh.
- **Độ phức tạp không gian:** DFS sẽ chỉ lưu trữ nhiều bộ nhớ trên stack từ gốc đến lá dài nhất trong cây. Nói cách khác, việc sử dụng không gian là $\boldsymbol{O(h)}$ với $\boldsymbol{h}$ là chiều cao của cây.

1.4. Minh họa:



Hình 2. Đồ thị gồm 6 nút (DFS)

- Trạng thái ban đầu (S_0) là nút A.
- Trạng thái mục tiêu (S_g) là nút E.
- Bước 1: Khởi tạo
  - Thêm trạng thái ban đầu (A) vào ***OPEN***: ***OPEN*** = [A]
  - Không có trạng thái nào được kiểm tra: ***CLOSED*** = []
- Bước 2: Kiểm tra ***OPEN*** trống hay không
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = A
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = []
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm A vào ***CLOSED***: ***CLOSED*** = [A]
  - Lấy các hành động có thể từ ***current_state***: ***get_neighbours(A)***:= [B, C]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Thêm B và C vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [B, C]
    - path[B] = A

- path[C] = A

- Bước 3: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = B
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [C]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B]
  - Lấy các hành động có thể từ ***current_state***: ***get_neighbours (B):***= [F, D]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Thêm F và D vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [F, D, C]
    - path[F] = B
    - path[D] = B
- Bước 4: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = F
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [D, C]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B, F]
  - Lấy các hành động có thể từ ***current_state***: ***get_neighbours (F):***= []
  - Không có hành động nào nên qua bước tiếp theo
- Bước 5: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = D
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [C]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B, F, D]

- Lấy các hành động có thể từ ***current_state***: ***get_neighbours(D)***:= [E]
- Thêm các trạng thái tiếp theo vào ***OPEN***
  - Thêm E vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [E]
  - path[D] = E

- Bước 6: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = E
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = []
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Đúng
  - Trả về đường đi từ E về A: A → B → D → E . (path[E] = D, path[D] = B, path[B] = A

2. Breadth-first-search (BFS):

2.1. Ý tưởng:

BFS xác định không gian tìm kiếm bằng cách duyệt theo chiều rộng, duyệt tất cả các trạng thái trong cùng một độ sâu trước khi chuyển sang trạng thái ở mức khác.

Nó hoạt động như sau:

- Bắt đầu ở trạng thái ban đầu.
- Duyệt tất cả các trạng thái cùng cấp trước khi duyệt xuống.
- Quay lại và duyệt các nhánh khác.
- Tiếp tục cho đến khi tìm thấy trạng thái mục tiêu hoặc không còn trạng thái nào để duyệt

2.2. Mã giả

BREADTH-FIRST-SEARCH (DFS)

```
Input: trạng thái ban đầu (S0)
Output: trạng thái kết thúc (Sg)
1:  add S0 to OPEN
2:  loop
3:      get the first element from OPEN and assign to current_state
4:      remove current_state from OPEN
5:      if current_state is Sg
6:          return path
7:      add current_state to CLOSED
8:      foreach neighbour in get_neighbours(current_state)
9:          if neighbour not in OPEN and neighbour not in CLOSED
10:             add neighbour to OPEN as the last element
11:             path[neighbour] = current_state
12: end loop
13: return "No path found"
```

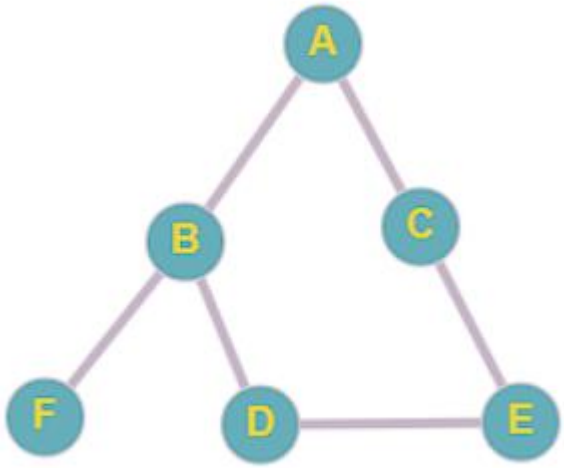
2.3. Đặc điểm:

- **Độ hoàn thành:** Nó phụ thuộc vào không gian tìm kiếm. Nếu không gian tìm kiếm là hữu hạn thì **BFS** sẽ hoàn tất. Tuy nhiên, nếu có vô số lựa chọn thay thế, nó có thể không tìm ra giải pháp.

Nhưng nếu ta sử dụng ***CLOSED*** set để lưu các vị trí đã qua thì **BFS** sẽ luôn tìm ra giải pháp nếu có.

- **Tính tối ưu: BFS** đảm bảo đường đi ngắn nhất(khi chi phí của tất cả đường đều là 1)
- **Độ phức tạp thời gian:** $\boldsymbol{O(V^s)}$
- **Độ phức tạp không gian:** $\boldsymbol{O(V^s)}$
- **V**ới $\boldsymbol{V^s}$ là số nút và $\boldsymbol{h}$ là chiều cao của cây.

2.4. Minh họa:



Hình 3. Đồ thị gồm 6 nút (BFS)

- Trạng thái ban đầu ($\boldsymbol{S_0}$) là nút A.
- Trạng thái mục tiêu ($\mathbf{S_g}$) là nút E.
- Bước 1: Khởi tạo
  - Thêm trạng thái ban đầu (A) vào ***OPEN***: ***OPEN*** = [A]
  - Không có trạng thái nào được kiểm tra: ***CLOSED*** = []
- Bước 2: Kiểm tra ***OPEN*** trống hay không
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = A
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = []
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E)

không: Không phải

- Thêm A vào ***CLOSED***: ***CLOSED*** = [A]
- Lấy các hành động có thể từ ***current_state***: ***get_neighbours(A)***:= [B, C]
- Thêm các trạng thái tiếp theo vào ***OPEN***
  - Thêm B và C vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [B, C]
  - path[B] = A
  - path[C] = A

- Bước 3: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = B
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [C]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B]
  - Lấy các hành động có thể từ ***current_state***: ***get_neighbours (B)***:= [F, D]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Thêm F và D vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [C, F, D]
    - path[F] = B
    - path[D] = B
- Bước 4: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = C
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [F, D]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B, C]

- Lấy các hành động có thể từ ***current_state***: ***get_neighbours (C)***:= [E]
- Thêm các trạng thái tiếp theo vào ***OPEN***
  - Thêm E vào ***OPEN*** (nếu chúng không thuộc ***CLOSED*** và ***OPEN***): ***OPEN*** = [F, D, E]
  - path[E] = C

- Bước 5: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = F
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [D, E]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B, C, F]
  - Lấy các hành động có thể từ ***current_state***: ***get_neighbours(F)***:= []
  - Không có hành động nào nên qua bước tiếp theo
- Bước 6: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = D
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = [E]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm ***current_state*** vào ***CLOSED***: ***CLOSED*** = [A, B, C, F, D]
  - Lấy các hành động có thể từ ***current_state***: ***get_neighbours(D)***:= []
  - Không có hành động nào nên qua bước tiếp theo
- Bước 7: Lặp lại bước 2
  - Lấy trạng thái hiện tại từ ***OPEN***: ***current_state*** = E
  - Xóa trạng thái hiện tại trong ***OPEN***: ***OPEN*** = []
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Đúng
  - Trả về đường đi từ E về A: A → C → E. (path[E] = C, path[C] = A)

3. Uniform-cost-search (UCS):

3.1. Ý tưởng:

UCS duyệt các nút dựa trên chi phí của chúng từ trạng thái ban đầu.

Nó hoạt động như sau:

- Bắt đầu ở trạng thái ban đầu.
- Duyệt tất cả các trạng thái có chi phí thấp nhất trước.
- Tiếp tục duyệt các trạng thái dựa trên chi phí tới khi tìm thấy trạng thái mục tiêu hoặc không còn trạng thái nào để duyệt.

3.2. Mã giả

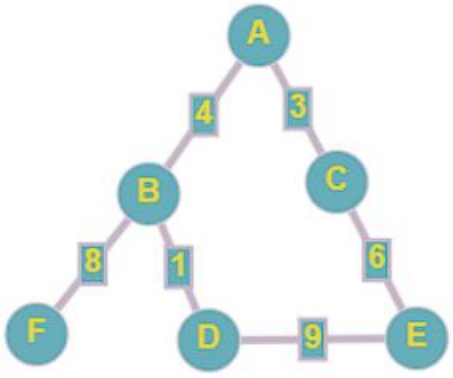
UNIFORM-COST-SEARCH (UCS)

```
    Input: trạng thái ban đầu (S0)
    Output: trạng thái kết thúc (Sg)
1:  add (0, S0) to OPEN
2:  loop
        sort OPEN in ascending order based on cost
3:      get the first element from OPEN and assign to cost, current_state
4:      remove current_state from OPEN
5:      if current_state is Sg
6:          return path
7:      add current_state to CLOSED
8:      foreach neighbour in get_neighbours(current_state)
9:          if neighbour not in CLOSED
10             tentative_cost = cost + distance_between(neighbor, current_state)
11             if neighbour not in OPEN
12:                Add (tentative_cost, neighbour) to OPEN as the last element
13:                path[neighbour] = current_state
14:            if tentative_cost < [cost of neighbour in OPEN]
15:                change the cost of neighbour in OPEN to tentative_cost
16:                path[neighbour] = current_state
17: end loop
18: return "No path found"
```

3.3. Đặc điểm:

- **Độ hoàn thành:** Nếu đồ thị tìm kiếm là hữu hạn thì phiên bản tìm kiếm đồ thị của **UCS** đã hoàn tất. Nếu đồ thị là vô hạn nhưng không có nút nào có số lượng nút kề vô hạn và tất cả các cạnh đều có chi phí dương thì **UCS** cũng sẽ hoàn thành.
- **Tính tối ưu: UCS** luôn luôn tối ưu (khi mọi nút với chi phí thấp hơn đều đã duyệt qua)
- **Độ phức tạp thời gian:** $\boldsymbol{O(l^{(1+\frac{C}{\varepsilon})})}$
- **Độ phức tạp không gian:** $\boldsymbol{O(l^{(1+\frac{C}{\varepsilon})})}$
- **V**ới $\boldsymbol{l}$ là số level ,$\boldsymbol{C}$ là chi phí tới đích.
- Với mỗi bước sẽ gần đích hơn một đoạn $\boldsymbol{\varepsilon}$.
- $\boldsymbol{+1}$ là do bắt đầu tại khoảng cách 0 và kết thúc tại $\boldsymbol{\frac{C}{\varepsilon}}$.

3.4. Minh họa:



Hình 4. Đồ thị gồm 6 nút (UCS)

- Trạng thái ban đầu ($\boldsymbol{S_0}$) là nút A.
- Trạng thái mục tiêu ($\mathbf{S_g}$) là nút E.

- Bước 1: Khởi tạo
  - Thêm trạng thái ban đầu (A) vào ***OPEN***: ***OPEN*** = [(0,A)]
  - Không có trạng thái nào được kiểm tra: ***CLOSED*** = []
- Bước 2: Kiểm tra ***OPEN*** trống hay không
  - **Sort** ***OPEN*** theo cost.
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** **= 0,** ***current_state*** = A
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = []
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm A vào ***CLOSED***: ***CLOSED*** = [A]
  - Lấy các trạng thái có thể từ ***current_state***: ***get_neighbours(A)***:= [B, C]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Tính ***tentative_cost*** = ***cost*** + distance_between(B, A) = 0 + 4 = 4
    - Thêm B vào ***OPEN*** (vì B không thuộc ***OPEN***): ***OPEN*** = [(***tentative_cost***, B)]= [(4, B)]
    - path[B] = A
    - Tính ***tentative_cost*** = ***cost*** + distance_between(C, A) = 0 + 3 = 3
    - Thêm C vào ***OPEN*** (vì C không thuộc ***OPEN***): ***OPEN*** = [(3, B), (***tentative_cost***, C)]= [(4, B), (3, C)]
    - path[C] = A
- Bước 3: Lặp lại bước 2
  - **Sort** ***OPEN*** theo cost. ***OPEN*** = [(3, C), (4, B)]
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** = 3, ***current_state*** = C
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = [(4, B)]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm C vào ***CLOSED***: ***CLOSED*** = [A, C]

- Lấy các trạng thái có thể từ ***current_state***: ***get_neighbours(C)***:= [E]
- Thêm các trạng thái tiếp theo vào ***OPEN***
  - Tính ***tentative_cost*** = ***cost*** + distance_between(E, C) = 3 + 6 = 9
  - Thêm E vào ***OPEN*** (vì E không thuộc ***OPEN***): ***OPEN*** = [(4, B), (***tentative_cost***, E)] = [(4, B), (9, E)]
  - path[E] = C

- Bước 4: Lặp lại bước 2
  - **Sort** ***OPEN*** theo cost. ***OPEN*** = [(4, B), (9, E)]
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** = 4, ***current_state*** = B
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = [(9, E)]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm C vào ***CLOSED***: ***CLOSED*** = [A, C, B]
  - Lấy các trạng thái có thể từ ***current_state***: ***get_neighbours(B)***:= [F, D]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Tính ***tentative_cost*** = ***cost*** + distance_between(F, B) = 4 + 8 = 12
    - Thêm E vào ***OPEN*** (vì F không thuộc ***OPEN***): ***OPEN*** = [(9, E), (***tentative_cost***, F)] = [(9, E), (12, F)]
    - path[F] = B
    - Tính ***tentative_cost*** = ***cost*** + distance_between(D, B) = 4 + 1 = 5
    - Thêm D vào ***OPEN*** (vì D không thuộc ***OPEN***): ***OPEN*** = [(9, E), (12, F), (***tentative_cost***, D)] = [(9, E), (12, F), (5, D)]
    - path[D] = B

- Bước 5: Lặp lại bước 2
  - **Sort** ***OPEN*** theo cost. ***OPEN*** = [(5, D), (9, E), (12, F)]
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** = 5, ***current_state*** = D
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = [(9, E), (12, F)]

- Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
- Thêm C vào ***CLOSED***: ***CLOSED*** = [A, C, B, D]
- Lấy các trạng thái có thể từ ***current_state***: ***get_neighbours(D)***:= [E]
- Thêm các trạng thái tiếp theo vào ***OPEN***
  - Tính ***tentative_cost*** = ***cost*** + distance_between(D, E) = 5 + 9 = 14
  - Kiểm tra E thuộc ***OPEN***: ***Đúng***
  - Kiểm tra ***tentative_cost*** < ***[cost of E in OPEN](9): Sai***

- Bước 6: Lặp lại bước 2
  - **Sort** ***OPEN*** theo cost. ***OPEN*** = [(9, E), (12, F)]
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** = 9, ***current_state*** = E
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = [(12, F)]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Đúng
  - Trả về đường đi từ E về A: A → C → E. (path[E] = C, path[C] = A)

4. AStar Search:

4.1. Ý tưởng:

AStar duyệt các nút dựa trên chi phí của chúng từ trạng thái ban bầu cộng với giá trị dự đoán Heuristics của chúng tới trạng thái đích.

Nó hoạt động như sau:

- Bắt đầu ở trạng thái ban đầu.
- Duyệt tất cả các trạng thái có chi phí cộng với giá trị dự đoán Heuristics thấp nhất trước.
- Tiếp tục duyệt các trạng thái dựa trên chi phí tới khi tìm thấy trạng thái mục tiêu hoặc không còn trạng thái nào để duyệt.

4.2. Mã giả

ASTAR SEARCH (A* SEARCH)

```
Input: trạng thái ban đầu (S0)
Output: trạng thái kết thúc (Sg)
1:  add (0, S0) to OPEN
2:  g_cost = list n elements with [100,000] value
3:  loop
4:      sort OPEN in ascending order based on cost
5:      get the first element from OPEN and assign to node_cost, current_state
6:      remove current_state from OPEN
7:      if current_state is Sg
8:          return path
9:      add current_state to CLOSED
10:     foreach neighbour in get_neighbours(current_state)
11:         if neighbour not in CLOSED
                tentative_cost = g_cost[current_state] +
12:             distance_between(neighbor, current_state)
                + heuristic(neighbour, Sg)
13:             if neighbour not in OPEN
14:                 Add (tentative_cost, neighbour) to OPEN as the last element
15:                 path[neighbour] = current_state
16                  g_cost[neighbour] = g_cost[current_state] +
                    distance_between(neighbor, current_state)
17:             if tentative_cost < [cost of neighbour in OPEN]
18:                 change the cost of neighbour in OPEN to tentative_cost
19:                 path[neighbour] = current_state
20:                 g_cost[neighbour] = g_cost[current_state] +
                    distance_between(neighbor, current_state)
        end for
21: end loop
22: return "No path found"
```

4.3. Đặc điểm:

- **Độ hoàn thành:** AStar sẽ hoàn tất nếu có đường đi tới đích và thỏa điều kiện

  Điều kiện:

- Đồ thị tìm kiếm là hữu hạn
- Tất cả các cạnh chi phí đều dương
- Hàm heuristic đáng tin

- **Tính tối ưu:** Tìm kiếm A* là tối ưu nếu phương pháp phỏng đoán được sử dụng vừa có thể chấp nhận được vừa ổn định. Một heuristic có thể chấp nhận được không bao giờ đánh giá quá cao chi phí thực sự từ một trạng thái đến mục tiêu và một heuristic ổn định thỏa mãn bất đẳng thức tam giác.
- **Độ phức tạp về thời gian:** Trong trường hợp xấu nhất, độ phức tạp về thời gian của A* là $O(b\hat{}d)$, trong đó b là hệ số phân nhánh (số lượng kế thừa tối đa cho bất kỳ trạng thái nào) và d là độ sâu của trạng thái mục tiêu nông nhất. Tuy nhiên, trong thực tế, A* có thể nhanh hơn đáng kể khi có một phương pháp tìm kiếm tốt hướng dẫn tìm kiếm.
- **Độ phức tạp về không gian:** Độ phức tạp về không gian của A* được xác định bởi số lượng nút tối đa phải được lưu trữ trong bộ nhớ trong quá trình tìm kiếm. Trong trường hợp xấu nhất, nó cũng là $O(b\hat{}d)$. Độ phức tạp của không gian bị ảnh hưởng bởi kích thước của không gian trạng thái và hệ số phân nhánh.
- **Độ phức tạp về thời gian và không gian** của tìm kiếm A* phụ thuộc vào nhiều yếu tố khác nhau, bao gồm chất lượng của phương pháp heuristic và cấu trúc của không gian tìm kiếm. Trong trường hợp xấu nhất, độ phức tạp về thời gian và không gian của tìm kiếm A* là theo cấp số nhân. Tuy nhiên, khi sử dụng phương pháp heuristic, A* thường có thể tìm ra giải pháp hiệu quả hơn nhiều.

4.4. Hàm Heuristic:

- **Hàm heuristic $h(n)$** cho A* ước tính chi phí tối thiểu từ bất kỳ đỉnh n nào tới đích. Điều quan trọng là chọn một hàm heuristic tốt.

  Tuy nhiên, hàm Heuristic mang tính chất của thuật giải nên việc chọn một

hàm Heuristic sẽ quyết định thuật toán A* search có tối ưu về mặt chi phí, có tìm được điểm đích hay không.

- **Admissible heuristic**(hàm heuristic đáng tin cậy) là hàm Heuristic mà sẽ không đánh giá cao hơn chi phí thực tế để đạt được trạng thái đích từ bất kì trạng thái nào.

$$h(n) \leq h^*(n)$$

Với:

$h(n)$ là chi phí dự tính từ trạng thái n tới đích.

$h^*(n)$ là chi phí thực(tối ưu) từ trạng thái n tới đích.

4.5. Một số hàm Heuristic trong AStar:

- **Euclidean Distance:**

Trong trường hợp ta có thể duyệt theo 8 hướng. Thì việc sử dụng Euclidean Distance là tối ưu.

```
def euclidean_distance(node: Node, goal: Node):
    dx = node.rect.centerx - goal.rect.centerx
    dy = node.rect.centery - goal.rect.centery
    return sqrt(dx * dx + dy * dy)
```

Hình 5. Euclidean_distance trích từ algo.py trong src

- **Manhattan Distance with obstacles:**

Tuy nhiên, khoảng cách Euclidean không phải lúc nào cũng được chấp nhận, đặc biệt nếu có chướng ngại vật. Để làm cho khoảng cách Euclide được chấp nhận khi có chướng ngại vật, bạn có thể kết hợp nó với một phương pháp phỏng đoán bổ sung để tính các chướng ngại vật.

```
# Calculates the Manhattan distance with obstacles
# obstacle_cost is the cost of moving through an obstacle
# obstacle_cost depends on your specific problem and how you want to prioritize obstacle avoidance
def manhattan_distance_with_obstacles(node: Node, goal: Node, obstacle_cost):
    dx = abs(node.rect.centerx - goal.rect.centerx)
    dy = abs(node.rect.centery - goal.rect.centery)
    return (dx + dy) + (obstacle_cost - 1) * min(dx, dy)
```

Hình 6. Manhattan_distance_with_obstacles trích từ algo.py trong src

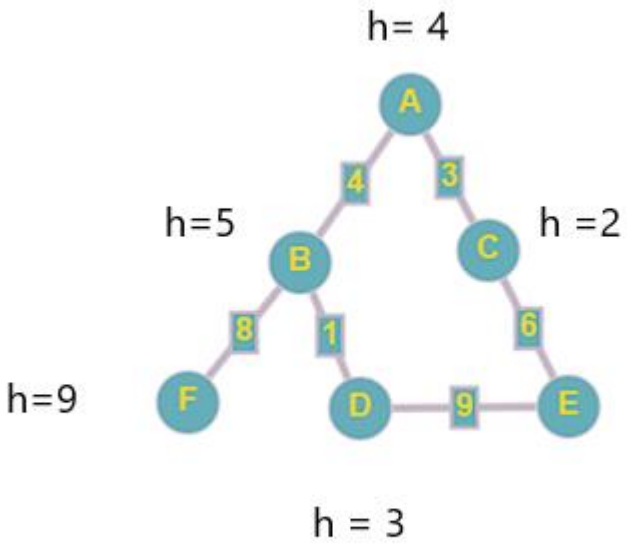
- **Hàm Heuristic tùy chỉnh:**

  Hàm Heuristic tùy chỉnh kết hợp cả Euclidean distance và Manhattan distance with obstacles.

```python
# Combines the Euclidean and Manhattan distances with obstacles using a weights parameter (0 <= weights <= 1)
# High Euclidean prioritize accuracy in estimating distances to the goal
# High Manhattan prioritize admissibility to ensure that the heuristic never overestimates the true cost
def combined_heuristic(node: Node, goal: Node, obstacle_cost = 1, euclidean_weight = 1):
    manhattan_weight = 1 - euclidean_weight
    euclidean = euclidean_weight * euclidean_distance(node, goal)
    manhattan = manhattan_weight * manhattan_distance_with_obstacles(node, goal, obstacle_cost)
    return euclidean + manhattan
```

Hình 7. Hàm combined_heuristic trích từ algo.py trong src

4.6. Minh họa:



Hình 8. Đồ thị gồm 6 nút (AStar)

- Trạng thái ban đầu (S_0) là nút A.
- Trạng thái mục tiêu (S_g) là nút E.

- Bước 1: Khởi tạo
  - Thêm trạng thái ban đầu (A) vào ***OPEN***: ***OPEN*** = [(0,A)]
  - Không có trạng thái nào được kiểm tra: ***CLOSED*** = []
- Bước 2: Kiểm tra ***OPEN*** trống hay không
  - **Sort** ***OPEN*** theo cost.
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** **= 0,** ***current_state*** = A
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = []
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm A vào ***CLOSED***: ***CLOSED*** = [A]
  - Lấy các trạng thái có thể từ ***current_state***: ***get_neighbours(A)***:= [B, C]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Tính ***tentative_cost*** = ***g_cost[A]*** + distance_between(B, A) + h(B) = 0 + 4 + 5 = 9
    - Thêm B vào ***OPEN*** (vì B không thuộc ***OPEN***): ***OPEN*** = [(***tentative_cost***, B)]= [(9, B)]
    - path[B] = A
    - ***g_cost[B] = 0 +*** distance_between(B, A) = 4
    - Tính ***tentative_cost*** = ***g_cost[A]*** + distance_between(C, A) + h(C) = 0 + 3 + 2 = 5
    - Thêm C vào ***OPEN*** (vì C không thuộc ***OPEN***): ***OPEN*** = [(9, B), (***tentative_cost***, C)]= [(9, B), (5, C)]
    - path[C] = A
    - ***g_cost[C]*** = 0 + distance_between(C, A) = 3
- Bước 3: Lặp lại bước 2
  - **Sort** ***OPEN*** theo cost. ***OPEN*** = [(5, C), (9, B)]
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** = 5, ***current_state*** = C

- Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = [(9, B)]
- Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
- Thêm C vào ***CLOSED***: ***CLOSED*** = [A, C]
- Lấy các trạng thái có thể từ ***current_state***: ***get_neighbours(C)***:= [E]
- Thêm các trạng thái tiếp theo vào ***OPEN***
  - Tính ***tentative_cost*** = ***g_cost[C]*** + distance_between(E, C) + h(E) = 3 + 6 + 0 = 9
  - Thêm E vào ***OPEN*** (vì E không thuộc ***OPEN***): ***OPEN*** = [(9, B), (***tentative_cost***, E)] = [(9, B), (9, E)]
  - path[E] = C
  - ***g_cost[E]*** = ***g_cost[C]*** + distance_between(E, C) = 3 + 6 = 9

- Bước 4: Lặp lại bước 2
  - **Sort** ***OPEN*** theo cost. ***OPEN*** = [(9, B), (9, E)]
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** = 9, ***current_state*** = B
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = [(9, E)]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm C vào ***CLOSED***: ***CLOSED*** = [A, C, B]
  - Lấy các trạng thái có thể từ ***current_state***: ***get_neighbours(B)***:= [F, D]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Tính ***tentative_cost*** = ***g_cost[B]*** + distance_between(F, B) + h(F) = 4 + 8 + 9 = 21
    - Thêm E vào ***OPEN*** (vì F không thuộc ***OPEN***): ***OPEN*** = [(9, E), (***tentative_cost***, F)] = [(9, E), (21, F)]
    - path[F] = B
    - ***g_cost[F]*** = ***g_cost[B]*** + distance_between(F, B) = 8 + 4 = 12
    - Tính ***tentative_cost*** = ***g_cost[B]*** + distance_between(D, B) +

h(D) = 4 + 1 + 3 = 8

- Thêm D vào ***OPEN*** (vì D không thuộc ***OPEN***): ***OPEN*** = [(9, E), (12, F), (***tentative_cost***, D)] = [(9, E), (21, F), (8, D)]
- path[D] = B
- ***g_cost[D]*** = ***g_cost[B]*** + distance_between(D, B) = 1 + 4 = 5

- Bước 5: Lặp lại bước 2
  - **Sort** ***OPEN*** theo cost. ***OPEN*** = [(8, D), (9, E), (21, F)]
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** = 8, ***current_state*** = D
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = [(9, E), (21, F)]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Không phải
  - Thêm C vào ***CLOSED***: ***CLOSED*** = [A, C, B, D]
  - Lấy các trạng thái có thể từ ***current_state***: ***get_neighbours(D)***:= [E]
  - Thêm các trạng thái tiếp theo vào ***OPEN***
    - Tính ***tentative_cost*** = ***cost*** + distance_between(D, E) + h(D) = 5 + 9 + 3= 17
    - Kiểm tra E thuộc ***OPEN***: ***Đúng***
    - Kiểm tra ***tentative_cost*** < ***[node_cost of E in OPEN](9): Sai***
- Bước 6: Lặp lại bước 2
  - **Sort** ***OPEN*** theo cost. ***OPEN*** = [(9, E), (21, F)]
  - Lấy trạng thái hiện tại từ ***OPEN***: ***cost*** = 9, ***current_state*** = E
  - Xóa ***current_state*** trong ***OPEN***: ***OPEN*** = [(21, F)]
  - Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu (E) không: Đúng
  - Trả về đường đi từ E về A: A → C → E. (path[E] = C, path[C] = A)

So sánh các thuật toán

1. UCS, Greedy, AStar

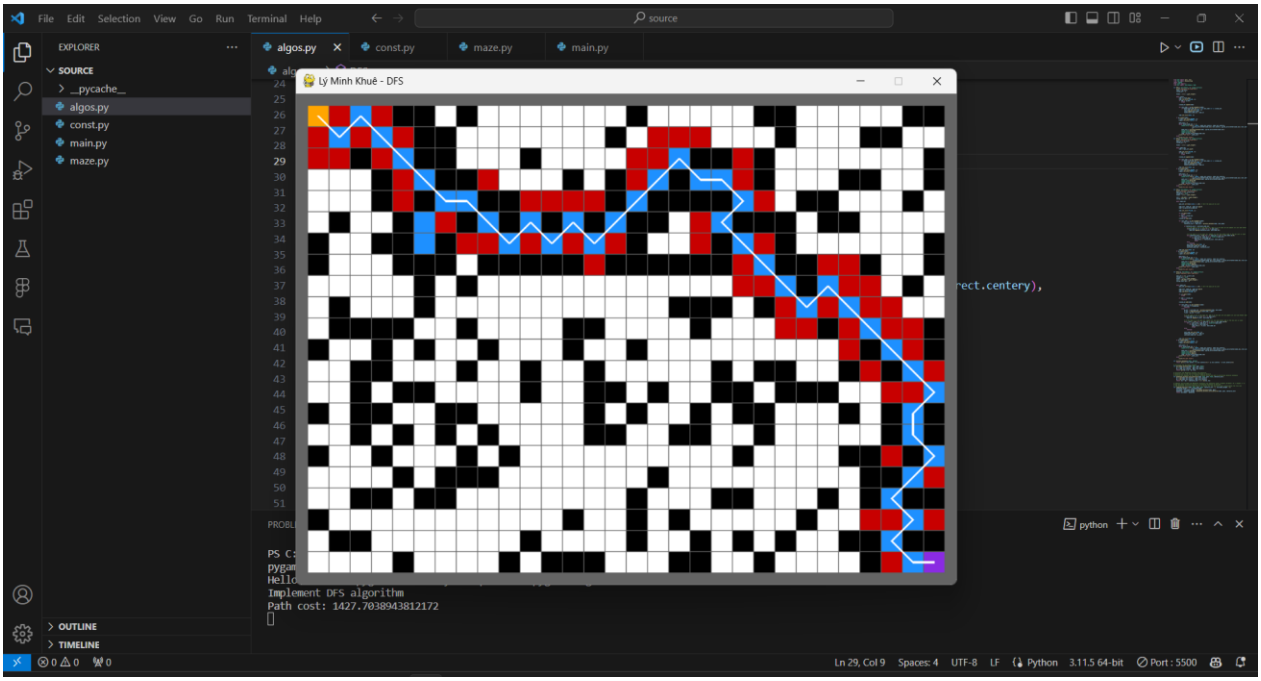
Tiêu chí	UCS	Greedy	AStar
Loại thuật toán	Uninformed Search	Informed Search	Informed Search
Chiến lược	Tìm kiếm theo một hệ thống	Tìm kiếm có thông tin, sử dụng hàm Heuristics	Tìm kiếm có thông tin, sử dụng hàm Heuristics
Hiệu suất	Chậm hơn	Không đảm bảo về đường đi ngắn nhất nên tốc độ phụ thuộc vào hàm Heuristic	Nhanh hơn UCS và Greedy
Độ hoàn thành	Đảm bảo hoàn thành	Không đảm bảo tính hoàn thành	Đảm bảo hoàn thành
Tối ưu	Đảm bảo tối ưu	Không đảm bảo tối ưu	Đảm bảo tối ưu
Phụ thuộc Heuristic	Không phụ thuộc vào Heuristic	Rất phụ thuộc vào Heuristic	Phụ thuộc vào Heuristic
Độ phức tạp thời gian	Theo cấp số mũ trong trường hợp xấu nhất	Phụ thuộc vào hàm Heuristic	Theo cấp số mũ trong trường hợp xấu nhất
Độ phức tạp không gian	Theo cấp số mũ trong trường hợp xấu nhất	Phụ thuộc vào hàm Heuristic	Theo cấp số mũ trong trường hợp xấu nhất

2. UCS, Dijsktra

Tiêu chí	UCS	AStar
Loại thuật toán	Uninformed Search	Thuật toán tìm đường đi ngắn nhất
Chiến lược	Tìm không theo hướng nào, xem xét chi phí từ trạng thái gốc	Tìm kiếm chi phí ngắn nhất từ trạng thái gốc
Hiệu suất	Nhanh hơn	Chậm hơn vì phải duyệt toàn bộ
Độ hoàn thành	Đảm bảo tìm thấy nếu tồn tại	Đảm bảo tìm thấy đường đi ngắn nhất nếu tồn tại
Tối ưu	Đảm bảo tìm thấy giải pháp tối ưu(đường đi có chi phí thấp nhất) nếu tồn tại	Đảm bảo tìm thấy đường đi ngắn nhất(chi phí thấp nhất)
Mục tiêu	Tìm đường đi ngắn nhất từ trạng thái đầu đến đích	Tìm đường đi ngắn nhất từ trạng thái đầu đến mọi trạng thái khác
Độ phức tạp thời gian	Theo cấp số mũ trong trường hợp xấu nhất và phụ thuộc vào chi phí của giải pháp tối ưu	Theo cấp số mũ trong trường hợp xấu nhất
Độ phức tạp không gian	Theo cấp số mũ trong trường hợp xấu nhất và phụ thuộc vào chi phí của giải pháp tối ưu	Theo cấp số mũ trong trường hợp xấu nhất

Cài đặt thuật toán

1. DFS:

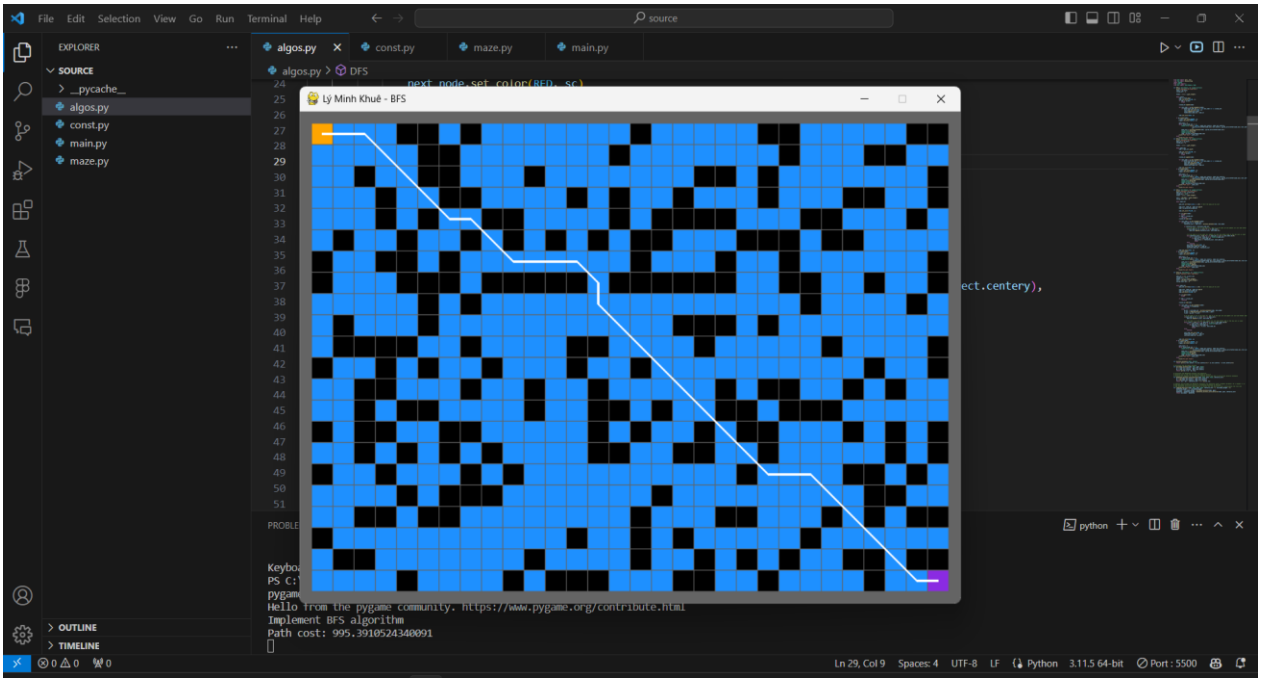


Hình 9. Cài đặt DFS

DFS cho kết quả khá nhanh khi điểm đích gần với đường đi. Di chuyển theo đường chéo là chủ yếu.

Kết quả cho thuật toán tìm kiếm là đường đi 1427.7 pixels

2. BFS:



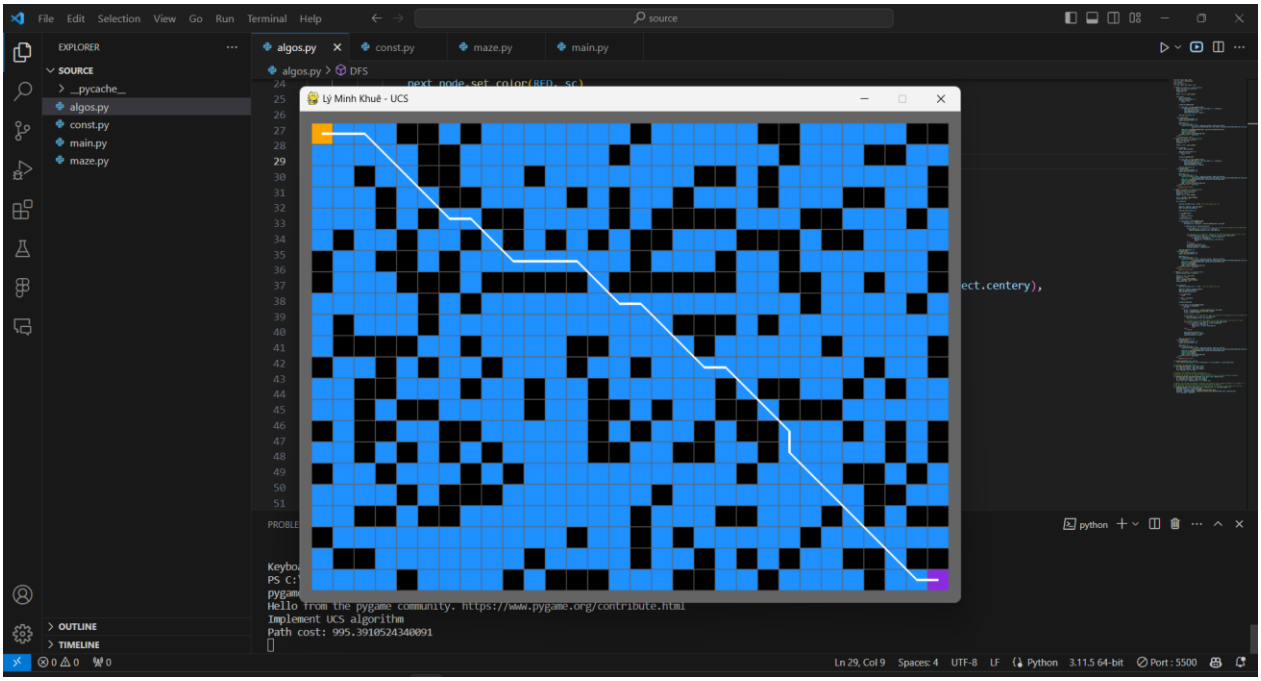
Hình 10. Cài đặt BFS

Mở rộng xung quanh điểm bắt đầu sau đó lan rộng ra cho tới điểm kết thúc.

Phải duyệt tất cả các node khi điểm kết thúc ở xa.

Kết quả cho thuật toán tìm kiếm: đường đi 995.39

3. UCS

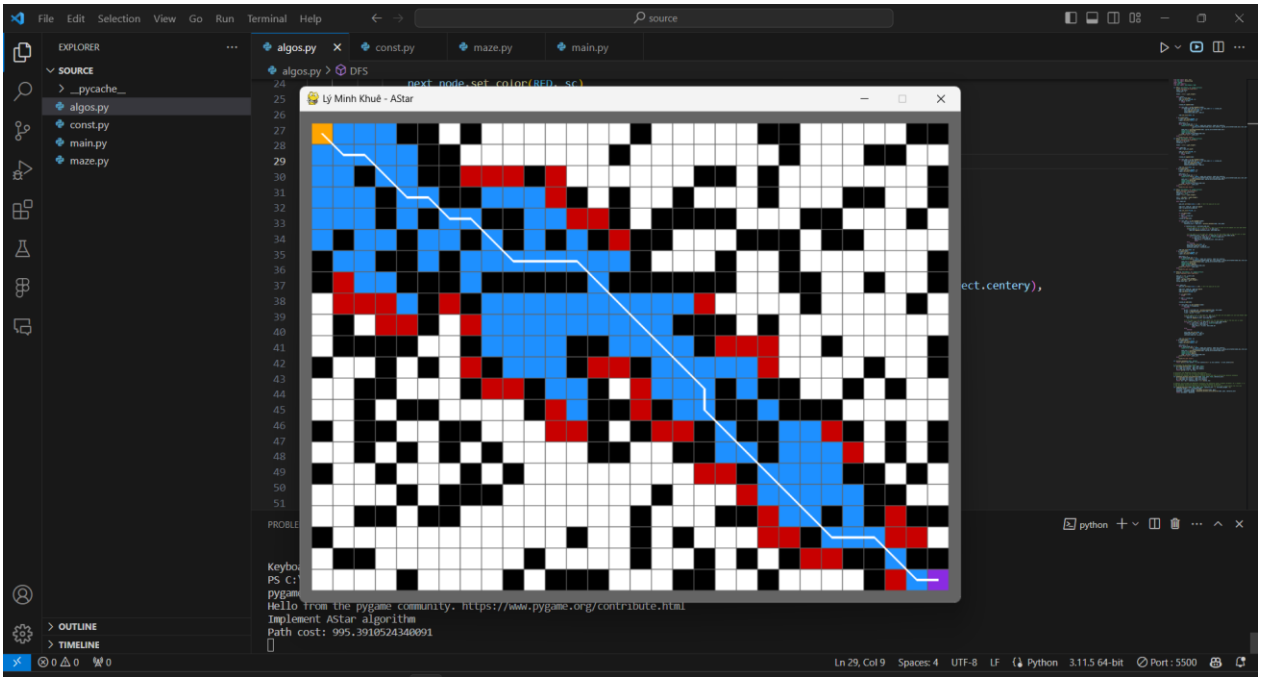


Hình 11. Cài đặt UCS

Cách thức mở rộng giống như BFS, tuy nhiên hay hướng tới điểm kết thúc nhiều hơn, đến khi gặp điểm đích thì dừng lại.

Kết quả cho thuật toán tìm kiếm: đường đi 995.39.

4. AStar



Hình 12. Cài đặt AStar

Duyệt các điểm gần với điểm đích, ít duyệt cái ô không liên quan.

Kết quả cho thuật toán tìm kiếm: đường đi 995.39

Cách tính là từ tâm ô này đến tâm ô kia trên 1 đường thẳng là 26 pixel(25 + 1(khoảng cách giữa 2 ô), còn nếu đi theo đường chéo thì $26\sqrt{2}$ pixel

Ô màu vàng là điểm bắt đầu.

Ô màu xanh là điểm đã duyệt.

Ô màu đỏ là điểm được mở rộng và chưa được duyệt.

Ô màu tím là điểm kết thúc.

TÀI LIỆU THAM KHẢO

Heuristics (stanford.edu)
[Algorithm] Các thuật toán tìm kiếm trong AI – FLINTERS Developer's Blog
Amit’s A* Pages (stanford.edu)
CTDL&GT - Graph Algorithms - Breadth First Search - Viblo
Breadth First Search or BFS for a Graph - GeeksforGeeks
Difference between Informed and Uninformed Search in AI - Testbook.com
Bài giảng của thầy Lê Hoài Bắc
Lab 1: Search in Graph của thầy Nguyễn Bảo Long