TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI

TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

======***======

BÁO CÁO THỰC NGHIỆM MÔN HỌC
TRÍ TUỆ NHÂN TẠO

Đề tài

Ứng dụng A* để tìm đường đi ngắn nhất trong Google Maps mini

Giảng viên hướng dẫn: **Mai Thanh Hồng**

Nhóm : 15

Họ và tên sinh viên:	Mã SV:
Đào Xuân Thắng	2023604198
Nguyễn Công Tuấn	2023602166
Bùi Minh Hiếu	2023606921
Tống Trần Quang Lộc	2023607837

Lớp : 20251IT6094001

Hà Nội, tháng 11 năm 2025

MỤC LỤC

LỜI NÓI ĐẦU

Trong thời đại công nghệ 4.0, việc tìm kiếm và tối ưu hóa đường đi ngắn nhất trong các mạng lưới giao thông trở thành một vấn đề quan trọng và thiết thực. Với sự phát triển nhanh chóng của các thành phố và nhu cầu di chuyển của con người, việc lập kế hoạch tuyến đường hiệu quả không chỉ giúp tiết kiệm thời gian mà còn giảm thiểu chi phí vận tải và ô nhiễm môi trường (Russell & Norvig, 2021).

Thuật toán tìm kiếm A* là một trong những phương pháp tiêu biểu của trí tuệ nhân tạo, được sử dụng để tìm đường đi ngắn nhất với hiệu quả cao nhờ cơ chế heuristic thông minh. A* đã được ứng dụng rộng rãi trong robot, game, hệ thống bản đồ số và các ứng dụng điều hướng (Hart, Nilsson & Raphael, 1968). Việc hiểu và triển khai thuật toán này trong một mô hình Google Maps mini giúp sinh viên nắm vững lý thuyết đồng thời phát triển kỹ năng lập trình và tư duy giải thuật.

Tuy nhiên, hiện nay phần lớn các nghiên cứu và tài liệu liên quan đến A* đều tập trung vào lý thuyết hoặc ứng dụng trong các bản đồ lớn với dữ liệu GPS thực tế. Trong khi đó, việc mô phỏng một bản đồ mini để thực hành thuật toán còn ít được khai thác, dẫn đến khoảng trống trong việc liên kết lý thuyết và thực hành trong môi trường học tập.

Trước thực trạng này, việc phát triển một ứng dụng Google Maps mini sử dụng thuật toán A* để tìm đường đi ngắn nhất là rất cấp thiết. Nó vừa giúp sinh viên nắm chắc kiến thức lý thuyết về heuristic search, vừa tạo cơ hội thử nghiệm, đánh giá thuật toán trong môi trường trực quan, dễ hiểu và kiểm soát được dữ liệu.

Từ những lý do đó, nhóm thực hiện đề tài "*Ứng dụng A để tìm đường đi ngắn nhất trong Google Maps mini**" với mục tiêu xây dựng một mô hình bản đồ giả lập, áp dụng thuật toán A* để tìm đường đi tối ưu, đồng thời phân tích hiệu quả của thuật toán trong các điều kiện khác nhau.

CHƯƠNG 1: CƠ SỞ LÝ THUYẾT

1.1. Tìm kiếm trong không gian trạng thái

1.1.1. Khái niệm trạng thái

-Giải bài toán trong không gian trạng thái, trước hết phải xác định dạng mô tả trạng thái bài toán sao cho bài toán trở nên đơn giản hơn, phù hợp bản chất vật lý của bài toán (Có thể sử dụng các xâu ký hiệu, véctơ, mảng hai chiều, cây, danh sách,…).

- Mỗi trạng thái chính là mỗi hình trạng của bài toán, các tình trạng ban đầu và tình trạng cuối của bài toán gọi là trạng thái đầu và trạng thái cuối.

1.1.2 Toán tử chuyển trạng thái

Toán tử là các phép biến đổi từ trạng thái này sang trạng thái khác.

Có hai cách dùng để biểu diễn các toán tử:

- Biểu diễn như một hàm xác định trên tập các trạng thái và nhận giá trị cũng trong tập này.
- Biểu diễn dưới dạng các quy tắc sản xuất S →A có nghĩa là nếu có trạng thái S thì có thể đưa đến trạng thái A.

1.1.3. Không gian trạng thái của bài toán

- Không gian trạng thái là tập tất cả các trạng thái có thể có và tập các toán tử của bài toán.

- Không gian trạng thái là một bộ bốn, Ký hiệu: K= (T, S, G, F).

Trong đó:

+ T: tập tất cả các trạng thái có thể có của bài toán.

+ S: trạng thái đầu.

+ G: tập các trạng thái đích.

+ F: tập các toán tử

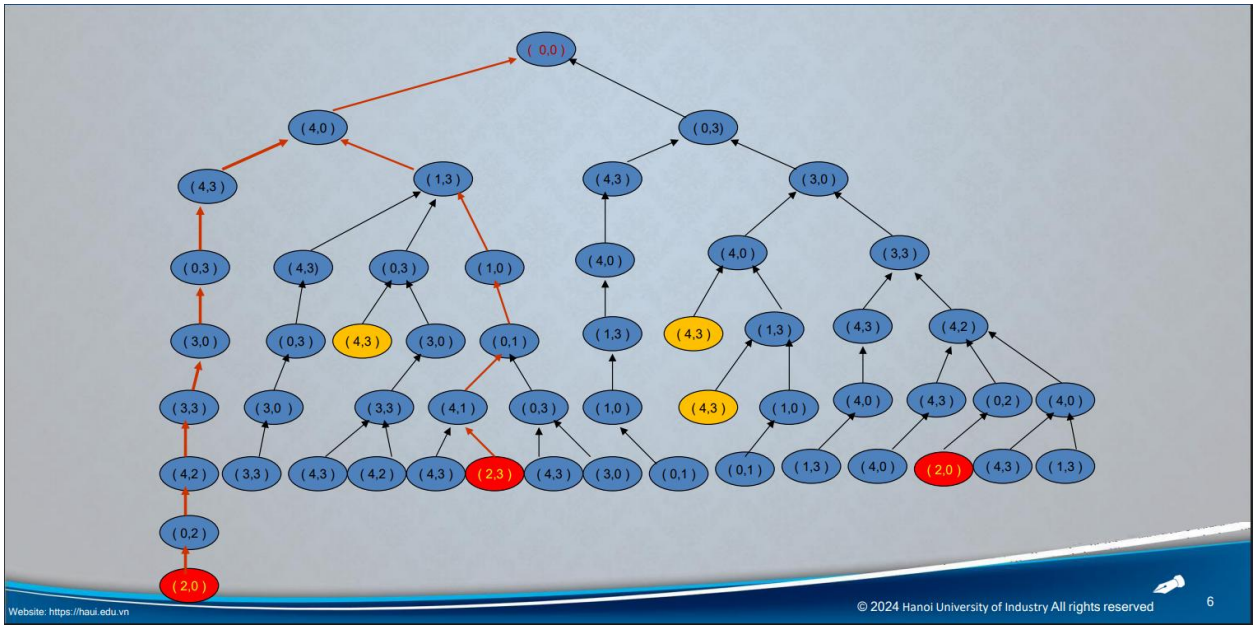
Ví dụ : Không gian trạng thái của bài toán đong nước là bộ bốn T, S, G, F xác định như sau:

T ={(x,y) / 0 <= x <= m; 0 <=y<=n}

S =(0,0)

G = {(x,y) ∈ T | x = k ∨ y = k}

F = Tập các thao tác đong đầy, đổ ra hoặc đổ sang bình khác thực hiện trên một bình.



Hình 1.1 : *Minh họa cây trạng thái của bài toán đong nước*

1.2. Heuristic trong bài toán tìm kiếm

1.2.1. Khái niệm

Tìm kiếm heuristic là một phương pháp tìm kiếm có sử dụng thông tin bổ sung để đánh giá và lựa chọn các trạng thái tiềm năng trong không gian tìm kiếm. Không giống như tìm kiếm mù, vốn mở rộng trạng thái theo một quy tắc cố định (như BFS hoặc DFS), tìm kiếm heuristic dựa vào một hàm heuristic để ước lượng chi phí hoặc lợi ích của một bước đi nhất định, giúp tăng tốc độ tìm kiếm lời giải.

- Heuristic chỉ là một phỏng đoán chứa các thông tin về bước tiếp theo sẽ được chọn dùng trong việc giải quyết một vấn đề.

- Heuristic là những tri thức được rút ra từ những kinh nghiệm, "trực giác" của con người.

- Heuristic có thể là những tri thức đúng hoặc sai.

Vì các heuristic sử dụng những thông tin hạn chế nên chúng ít khi có khả năng đoán trước chính xác cách hành xử của không gian trạng thái ở những giai đoạn xa hơn.

1.2.2. Chức năng của Heuristic

Các chương trình giải quyết những vấn đề trí tuệ nhân tạo sử dụng Heuristic cơ bản theo hai dạng:

- Vấn đề có thể thể không có giải pháp chính xác vì những điều không rõ ràng trong diễn đạt vấn đề hoặc trong các dữ liệu có sẵn.
- Vấn đề có thể có giải pháp chính xác, nhưng chi phí tính toán để tìm ra nó không cho phép.

1.2.3. Ưu điểm của Heuristic

Thuật giải Heuristic thể hiện cách giải bài toán với các đặc tính sau:

- Thường tìm được lời giải tốt (Nhưng không chắc là lời giải tốt nhất).

- Giải bài toán theo thuật giải Heuristic thường dễ dàng và nhanh chóng đưa ra kết quả hơn so với giải thuật tối ưu, vì vậy chi phí thấp hơn.

- Thuật giải Heuristic thường thể hiện khá tự nhiên, gần gũi với cách suy nghĩ và hành động con người.

1.2.4. Nhược điểm của Heuristic

Mặc dù hàm Heuristic mang lại hiệu quả cao về tốc độ tìm kiếm, nhưng nó vẫn tồn tại những nhược điểm cố hữu mà người thiết kế thuật toán cần cân nhắc:

- Không đảm bảo tính tối ưu: Nếu hàm heuristic thiết kế không chuẩn (không thỏa mãn tính *chấp nhận được - admissible*), thuật toán có thể đưa ra kết quả sai hoặc đường đi không phải là ngắn nhất.

- Dễ mắc kẹt tại cực trị địa phương (Local Optima): Thuật toán có thể bị "đánh lừa", dừng lại ở một kết quả có vẻ tốt trong phạm vi hẹp nhưng thực tế chưa phải là đích đến tốt nhất.

- Phụ thuộc vào bài toán cụ thể: Không có công thức chung cho mọi trường hợp. Người lập trình phải tự thiết kế hàm riêng cho từng bài toán dựa trên đặc thù của dữ liệu.

- Chi phí tính toán: Nếu hàm heuristic quá phức tạp, thời gian để máy tính xử lý nó có thể làm chậm chương trình, triệt tiêu lợi ích về tốc độ.

1.2.5. Thiết kế thuật giải Heuristic

Có nhiều phương pháp để thiết kế thuật giải Heuristic, trong đó người ta sử dụng 2 nguyên lý cơ bản sau:

-*Nguyên lý vét cạn thông minh* : Khi không gian tìm kiếm D lớn, ta thường tìm cách giới hạn lại không gian tìm kiếm này hoặc thực hiện một kiểu dò tìm đặc biệt dựa vào đặc thù của bài toán để nhanh chóng tìm ra mục tiêu.

- Mở rộng : áp dụng *nguyên lý thứ tự* để sắp xếp cấu trúc không gian khảo sát nhằm đạt lời giải nhanh hơn.

-*Nguyên lý tham lam* : Lấy tiêu chuẩn tối ưu nhất trên phạm vi toàn cục của bài toán để làm tiêu chuẩn chọn lựa hành động cho phạm vi cục bộ của từng bước trong quá trình tìm kiếm lời giải.

- Ví dụ điển hình : *Tìm kiếm leo đồi* - thuật toán này dựa trên hàm Heuristic để chọn trạng thái tốt nhất kế tiếp mà không quay lui.

Tư tưởng của thuật giải được thể hiện qua 2 bước :

i, Nếu trạng thái bắt đầu cũng là trạng thái đích thì thoát và báo là tìm được lời giải.

ii, Lặp lại cho đến khi trạng thái kết thúc hoặc cho đến khi Ti không tồn tại 1 trạng thái nào tốt hơn trạng thái hiện tại.

+ Đặt S bằng tập tất cả các trạng thái có thể có của Ti và tốt hơn Ti.

+ Xác định Tmax là trạng thái tốt nhất trong tập S: Ti=Tmax

Và lặp lại cho đến khi thông báo dừng.

1.2.6. So sánh thuật toán tìm kiếm mù với Heuristic

	Tìm kiếm mù(Blind search)	**Tìm kiếm Heuristic(Heuristic search)**
Nguyên lý hoạt động	Duyệt không gian trạng thái một cách hệ thống (cơ học) mà không biết đích nằm ở đâu cho đến khi tìm thấy nó**.**	Sử dụng hàm đánh giá h(n) (kinh nghiệm/ước lượng) để định hướng và ưu tiên duyệt các lối đi có triển vọng đến đích nhất.

Thông tin sử dụng	Chỉ dựa vào trạng thái hiện tại và các luật chuyển trạng thái.	Dựa vào trạng thái hiện tại + Hàm Heuristic (ước lượng khoảng cách tới đích).
Hiệu quả	Thấp.	Cao.
Tính tối ưu	Thường đảm bảo tìm ra kết quả tốt nhất (ví dụ: BFS luôn tìm đường ngắn nhất trong đồ thị không trọng số).	Không đảm bảo (trừ thuật toán A* với heuristic chấp nhận được). Đôi khi chấp nhận kết quả "đủ tốt" để đổi lấy tốc độ.
Độ phức tạp	Tốn nhiều tài nguyên bộ nhớ và thời gian tính toán khi không gian bài toán lớn.	Tiết kiệm tài nguyên hơn, phù hợp với các bài toán có không gian trạng thái lớn và phức tạp.

1.3. Các thuật toán tìm kiếm Heuristic

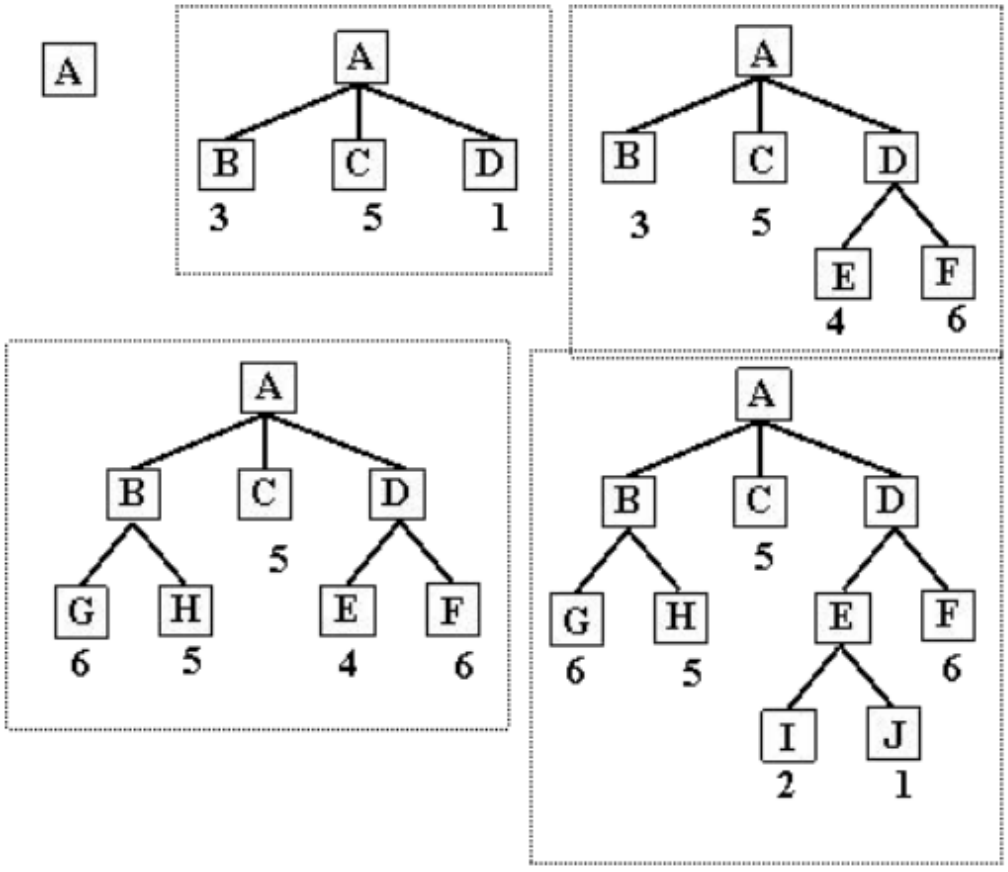
1.3.1. Tìm kiếm tối ưu (Best-First Search-BeFS)

Ưu điểm của tìm kiếm theo chiều sâu là không phải quan tâm đến sự mở rộng của tất cả các nhánh. Ưu điểm của tìm kiếm chiều rộng là không bị sa vào các đường dẫn bế tắc (các nhánh cụt). Tìm kiếm tối ưu (Best-First Search-BeFS) sẽ kết hợp hai phương pháp trên cho phép ta đi theo một con đường duy nhất tại một thời điểm, nhưng đồng thời vẫn xét được những hướng khác. Nếu con đường đang đi không triển vọng bằng những con đường đang quan sát, ta sẽ chuyển sang đi theo một trong số các con đường này.

Một cách cụ thể, tại mỗi bước của tìm kiếm BeFS, ta chọn đi theo trạng thái có khả năng cao nhất trong số các trạng thái đã được xét *cho đến thời điểm đó*. BeFS khác với tìm kiếm leo đồi là chỉ chọn trạng thái có khả năng cao nhất trong số các trạng thái kế tiếp có thể đến được từ trạng thái hiện tại. Như vậy, với tiếp cận này, ta sẽ ưu tiên đi vào những nhánh tìm kiếm có khả năng nhất (giống tìm kiếm leo đồi), nhưng ta sẽ không bị liên quan trong các nhánh này vì nếu càng đi sâu vào một hướng mà ta phát hiện ra rằng hướng này càng đi thì càng xấu, đến mức nó xấu hơn cả những hướng mà ta chưa đi, thì

ta sẽ không đi tiếp hướng hiện tại nữa mà chọn đi theo một hướng tốt nhất trong số những hướng chưa đi. Đó là tư tưởng chủ đạo của tìm kiếm tối ưu.

VÍ DỤ :



Hình 1.2 : *Tìm Kiếm BeFS*

Để làm rõ tư tưởng này ta xét đồ thị (hình 1). Ban đầu ta duyệt đỉnh , từ đỉnh này có thể đi đến các đỉnh B, , . chọn đỉnh để đi tiếp vì có chi phí nhỏ nhất, tiếp đến ta phát

triển đỉnh này được các đỉnh kề và F. Tuy nhiên các đỉnh này lại có chi cho lớn hơn đỉnh B do đó ta lại quay lại để duyệt đỉnh B. quá trình này cứ tiếp tục như vậy cho đến khi tìm được đỉnh đích.

Thuật giải tìm kiếm tối ưu:

1. Đặt OPEN chứa trạng thái khởi đầu T_0.
2. Cho đến khi tìm được trạng thái đích hoặc không còn nút nào trong OPEN, thực hiện :
   a. Chọn trạng thái tốt nhất (T_{max}) trong OPEN (và xóa T_{max} khỏi OPEN)
   b. Nếu T_{max} là trạng thái kết thúc thì thoát
   c. Ngược lại, tạo ra các trạng thái kế tiếp T_k có thể có từ trạng thái T_{max}. Đối với với mỗi trạng thái kế tiếp T_k thực hiện :

   Tính f(T_k); Thêm T_k vào OPEN

   BeFS khá đơn giản. Tuy vậy, trên thực tế, cũng như tìmkiếmchiều sâuvàchiềurộng, hiếm khi ta dùng BeFS một cách trực tiếp. Thông thường, người ta thườngdùngcác phiên bản của BeFS là A^T,A^{KT},A^*.

Thông tin về quá khứ và tương lai:

Thông thường, trong các phương án tìm kiếm theo kiểu BeFS, chi phí f của một trạng thái được tính dựa theo hai giá trị mà ta gọi là là g và h. Trong đó h, như đã biết,đó là một ước lượng về chi phí từ trạng thái hiện hành cho đến trạng thái đích(thông tin tương lai), còn g là chiều dài quãng đường đã đi từ trạng thái ban đầu chođến trạng thái hiện tại (thông tin quá khứ). Khi đó hàm ước lượng tổng chi phí f(n) được tính theo công thức:

$$f(n) = g(n) + h(n)$$

1.3.2. Thuật toán A^T

Thuật giải A^T là một phương pháp tìm kiếm theo kiểu BeFS với chi phí của đỉnh là giá trị hàm g (tổng chiều dài thực sự của đường đi từ đỉnh bắt đầu đến đỉnh hiện tại).

Giải thuật :

1. Đặt OPEN chứa trạng thái khởi đầu.

2. Cho đến khi tìm được trạng thái đích hoặc không còn nút nào trong OPEN, thực hiện:

   a. Chọn trạng thái (T_{max}) có giá trị g nhỏ nhất trong OPEN (và xóa T_{max} khỏi OPEN)

   b. Nếu T_{max} là trạng thái kết thúc thì thoát.

   c. Ngược lại, tạo ra các trạng thái kế tiếp T_K có thể có từ trạng thái T_{max}. Đối với mỗi trạng thái kế tiếp T_K thực hiện:

      g(T_K) = g(T_{max}) + cost(T_{max}, T_K)

      Thêm T_K vào OPEN.

***Note:** Vì chỉ sử dụng hàm g (mà không dùng hàm ước lượng h' để đánh giá độ tốt của một trạng thái nên ta cũng có thể xem A^T chỉ là một thuật toán.

1.3.3. Thuật toán A^{KT}

Thuật giải □□□ trong quá trình tìm đường đi chỉ xét đến các đỉnh và giá của chúng. Nghĩa là việc tìm đỉnh triển vọng chỉ phụ thuộc hàm g(n) (thông tin quá khứ). Tuy nhiên thuật giải này không còn phù hợp khi gặp phải những bài toán phức tạp (độ phức tạp cấp hàm mũ) do ta phải tháo một lượng nút lớn. Để khắc phục nhược điểm này, người ta sử dụng thêm các thông tin bổ sung xuất phát từ bản thân bài toán để tìm ra các đỉnh có triển vọng, tức là đường đi tối ưu sẽ tập trung xung quanh đường đi tốt nhất nếu s dụng các thông tin đặc tả về bài toán (thông tin tương lai).

Theo thuật giải này, chi phí của đỉnh được xác định:

f(n) = g(n) + h(n)

Đỉnh n được chọn nếu f(n) min.

Việc xác định hàm ước lượng h(n) được thực hiện dựa theo:

- Chọn toán tử xây dựng cung sao cho có thể loại bớt các đỉnh không liên quan và tìm ra các đỉnh có triển vọng.

- Sử dụng thêm các thông tin bổ sung nhằm xây dựng tập OPEN và cách lấy các đỉnh trong tập OPEN.

Để làm được việc này người ta phải đưa ra độ đo, tiêu chuẩn để tìm ra các đỉnh có triển vọng. Các hàm s dụng các kỹ thuật này gọi là hàm đánh giá. Sau đây là một số phương pháp xây dựng hàm đánh giá:

- Dựa vào xác suất của đỉnh trên đường đi tối ưu.

- Dựa vào khoảng cách, sự sai khác của trạng thái đang xét với trạng thái đích hoặc các thông tin liên quan đến trạng thái đích.

Giải thuật :

1. Đặt OPENchứa trạng thái khởi đầu.
2. Cho đến khi tìm được trạng thái đích hoặc không còn nút nào trong OPEN, thực hiện :
    1. Chọn trạng thái (Tmax) có giá trị f nhỏ nhất trong OPEN (và xóa Tmax khỏi OPEN)
    2. Nếu Tmaxlà trạng thái kết thúc thì thoát.
    3. Ngược lại, tạo ra các trạng thái kế tiếp Tk có thể có từ trạng thái Tmax. Đối với mỗi trạng thái kế tiếp Tk thực hiện :

        g(T_K) = g(T_{max}) + cost(T_{max}, T_K).

        Tính h'(T_K) .

        f(T_K) = g(T_K) + h'(T_K) .

        Thêm $T_K$vào OPEN.

1.4. Thuật toán A*

A* là giải thuật tìm kiếm trong đồ thị, tìm đường đi từ một đỉnh hiện tại đến đỉnh đích có sử dụng hàm để ước lượng khoảng cách hay còn gọi là hàm Heuristic.

Từ trạng thái hiện tại A* xây dựng tất cả các đường đi có thể đi dùng hàm ước lượng khoảng cách (hàm Heuristic) để đánh giá đường đi tốt nhất có thể đi. Tùy theo mỗi dạng bài khác nhau mà hàm Heuristic sẽ được đánh giá khác nhau. A* luôn tìm được đường đi ngắn nhất nếu tồn tại đường đi như thế.

1.4.1. Hàm đánh giá (Heuristic Function)

A* lưu giữ một tập các đường đi qua đồ thị, từ đỉnh bắt đầu đến đỉnh kết thúc, tập các đỉnh có thể đi tiếp được lưu trong tập OPEN.

Thứ tự ưu tiên cho một đường đi được quyết định bởi hàm Heuristic được đánh giá :

$$f(x) = g(x) + h(x)$$

Trong đó:

- g(x) là chi chi phí của đường đi từ điểm xuất phát cho đến thời điểm hiện tại.

- h(x) là hàm ước lượng chi phí từ đỉnh hiện tại đến đỉnh đích f(x) thường có giá trị càng thấp thì độ ưu tiên càng cao.

- Open: tập các trạng thái đã được sinh ra nhưng chưa được xét đến.

1.4.2. Các đặc tính của hàm Heuristic

1.4.2.1. Tính chấp nhận được(Admissibility)

Một hàm Heuristic h(n) được gọi là chấp nhận được (admissible) nếu nó không bao giờ ước lượng cao hơn chi phí thực tế để đi từ nút n đến trạng thái đích. Nói cách khác, h(n) phải luôn là một đánh giá "lạc quan" hoặc bằng với thực tế.

Điều kiện toán học của tính chấp nhận được là:

$$0 <= h(n) <= h^*(n)$$

Trong đó :

h(n): Chi phí ước tính từ nút n đến đích.

h*(n): Chi phí thực tế tối ưu (ngắn nhất) từ nút n đến đích.

Vai trò : chấp nhận được là điều kiện tiên quyết để đảm bảo độ chính xác của thuật toán.

- Nếu h(n) thỏa mãn tính chất admissible, thuật toán A* (sử dụng tìm kiếm trên cây) được chứng minh chắc chắn sẽ tìm ra đường đi tối ưu (đường đi có chi phí thấp nhất).

- Nếu h(n) vi phạm (ước lượng lớn hơn thực tế), A* có thể hội tụ nhanh hơn nhưng có nguy cơ bỏ qua đường đi tốt nhất để chọn một đường đi kém tối ưu hơn (sub-optimal).

Ví dụ: Trong bài toán tìm đường đi trên bản đồ, khoảng cách đường thẳng (Euclidean distance) luôn là một heuristic admissible vì khoảng cách đường chim bay giữa hai điểm luôn nhỏ hơn hoặc bằng khoảng cách di chuyển thực tế trên đường bộ.

1.4.2.2. Tính nhất quán (Consistency)

Một hàm heuristic h(n) được gọi là nhất quán (consistent) hay đơn điệu (monotone) nếu giá trị ước lượng từ một nút đến đích luôn nhỏ hơn hoặc bằng chi phí đi đến nút láng giềng cộng với giá trị ước lượng từ nút láng giềng đó đến đích.

Đây là dạng tổng quát của bất đẳng thức tam giác (triangle inequality). Điều kiện toán học được biểu diễn như sau:

$$h(n) <= cost(n,p) + h(p)$$

Trong đó:

- n: Nút hiện tại.
- p: Nút láng giềng (kế tiếp) của n.
- cost(n, p): Chi phí thực tế để di chuyển từ n đến p.
- h(n), h(p): Hàm heuristic tại nút n và p.

1.4.3. Mô tả thuật toán

Giải thuật :

- Close: tập các trạng thái đã được xét đến.
- Cost(p, q): là khoảng cách giữa p, q.
- g(p): khoảng cách từ trạng thái đầu đến trạng thái hiện tại p.
- h(p): giá trị được lượng giá từ trạng thái hiện tại đến trạng thái đích.
- f(p) = g(p) + h(p)

Các bước thực hiện :

Bước 1 :

Open: = {s}

Close: = {}

Bước 2 : while(Open !={})

+ Chọn trạng thái (đỉnh) tốt nhất p trong Open (xóa p khỏi Open).

+ Nếu p là trạng thái kết thúc thì thoát.

+ Chuyển p qua Close và tạo ra các trạng thái kế tiếp q sau p.

Nếu q đã có trong Open

-Nếu g(q) > g(p) + Cost(p,q) thì

g(q) = g(p) + Cost(p,q)

f(q) = g(q) + h(q)

prev(q) = p (đỉnh cha của q là p)

Nếu q chưa có trong Open

g(q) = g(p) + cost(p, q)

f(q) = g(q) + h(q)

prev(q) = p

Thêm q vào Open

Nếu q có trong Close

Nếu g(q) > g(p) + Cost (p,q)

g(q) = g(p) + Cost(p, q)

f(q) = g(q) + h(q)

prev(q) = p

Bỏ q khỏi Close

Thêm q vào Open

Bước 3 : Không tìm được

1.4.4. Ưu, nhược điểm

Ưu điểm : Một thuật giải linh động, tổng quát, trong đó hàm chứa cả tìm kiếm chiều sâu, tìm kiếm chiều rộng và những nguyên lý Heuristic khác. Nhanh chóng tìm đến

lời giải với sự định hướng của hàm Heuristic. Chính vì thế mà người ta thường nói A* chính là thuật giải tiêu biểu cho Heuristic.

Nhược điểm : A* rất linh động nhưng vẫn gặp một khuyết điểm cơ bản- giống như chiến lược tìm kiếm chiều rộng- đó là tốn khá nhiều bộ nhớ để lưu lại những trạng thái đã đi qua.

CHƯƠNG 2: ỨNG DỤNG A* TRONG GOOGLE MAPS MINI

2.1. Giới thiệu bài toán thực tế

Trong nhiều hệ thống hiện đại như bản đồ số, robot tự hành hoặc trò chơi điện tử, việc tìm được đường đi ngắn nhất giữa hai điểm trở thành một yêu cầu quan trọng. Thay vì di chuyển ngẫu nhiên hoặc duyệt tất cả các khả năng, thuật toán tìm đường cần phải đảm bảo tối ưu về thời gian và độ chính xác. Bài toán trong đề tài được mô phỏng dưới dạng một "khu dân cư" mini gồm 35 ngôi nhà được bố trí trên lưới hai chiều. Một số ô là đường đi trống, số khác là vị trí nhà cố định. Yêu cầu đặt ra là tìm được đường đi hợp lệ và ngắn nhất từ nhà A đến nhà B trong khu vực này.

Bài toán này gần giống với bài toán thực tế như tìm đường trong khu phố, hệ thống dẫn đường xe cộ, tối ưu hoá di chuyển trong các bản đồ game và hệ robot định vị. Do bản đồ là dạng ma trận lưới và chỉ cho phép di chuyển bốn hướng, đây là một trường hợp lý tưởng để áp dụng thuật toán A*, vốn là phương pháp nổi tiếng và hiệu quả nhất trong việc tìm đường tối ưu trên lưới.

Hình 2.1: hình ảnh minh họa

2.2. Phương pháp mô hình hóa bản đồ thực tế thành bản đồ mini

Để áp dụng thuật toán A* vào bài toán tìm đường, nhóm tiến hành mô hình hóa bản đồ thực tế thành một **bản đồ mini** dưới dạng ma trận hai chiều. Mục tiêu của bước này là đơn giản hóa cấu trúc bản đồ trong khi vẫn giữ nguyên các yếu tố quan trọng phục vụ việc tìm đường.

Quá trình mô hình hóa được thực hiện theo các bước:

1. **Xác định các điểm quan trọng (nhà, vị trí, giao điểm, ô trống)**
   Mỗi vị trí có ý nghĩa trong bản đồ thật được nhóm quy đổi thành một ô trong ma trận. Các vị trí được đánh số theo thứ tự tăng dần để tiện truy cập và xử lý trong chương trình.

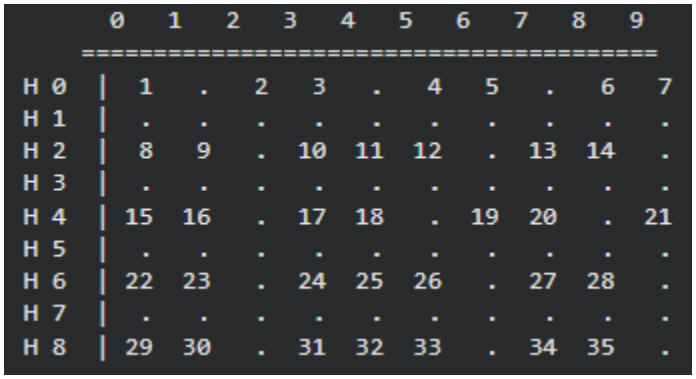
2. **Quy ước ký hiệu**

- Các ô chứa vị trí/nhà được ký hiệu bằng số (1, 2, 3, …).
- Các ô đường đi hoặc khoảng trống không chứa vật cản được ký hiệu bằng dấu chấm (“.”).
- Cấu trúc này đảm bảo thuật toán dễ dàng phân biệt giữa node có thể đi qua và node đại diện cho “mốc” cần tìm đường.

3. **Chuyển bản đồ thật sang dạng lưới đều (grid-based)**
   Dựa vào bố cục thực tế, nhóm chia bản đồ thành các hàng và cột. Mỗi ô tương ứng với một vị trí trong không gian.
   Cách làm này giúp việc áp dụng heuristic như **Manhattan** hoặc **Euclidean** trở nên trực quan.

4. **Xây dựng quan hệ láng giềng giữa các ô**
   Mỗi ô (node) được kết nối với các ô kề cạnh (trên, dưới, trái, phải).
   Nếu cần mô phỏng đường dạng chéo, nhóm có thể bổ sung thêm kết nối đường chéo, nhưng trong mô hình này nhóm chỉ dùng 4 hướng để đơn giản hóa.

5. **Kết quả mô hình hóa**
   Sau quá trình chuyển đổi, bản đồ mini có thể trông như sau:



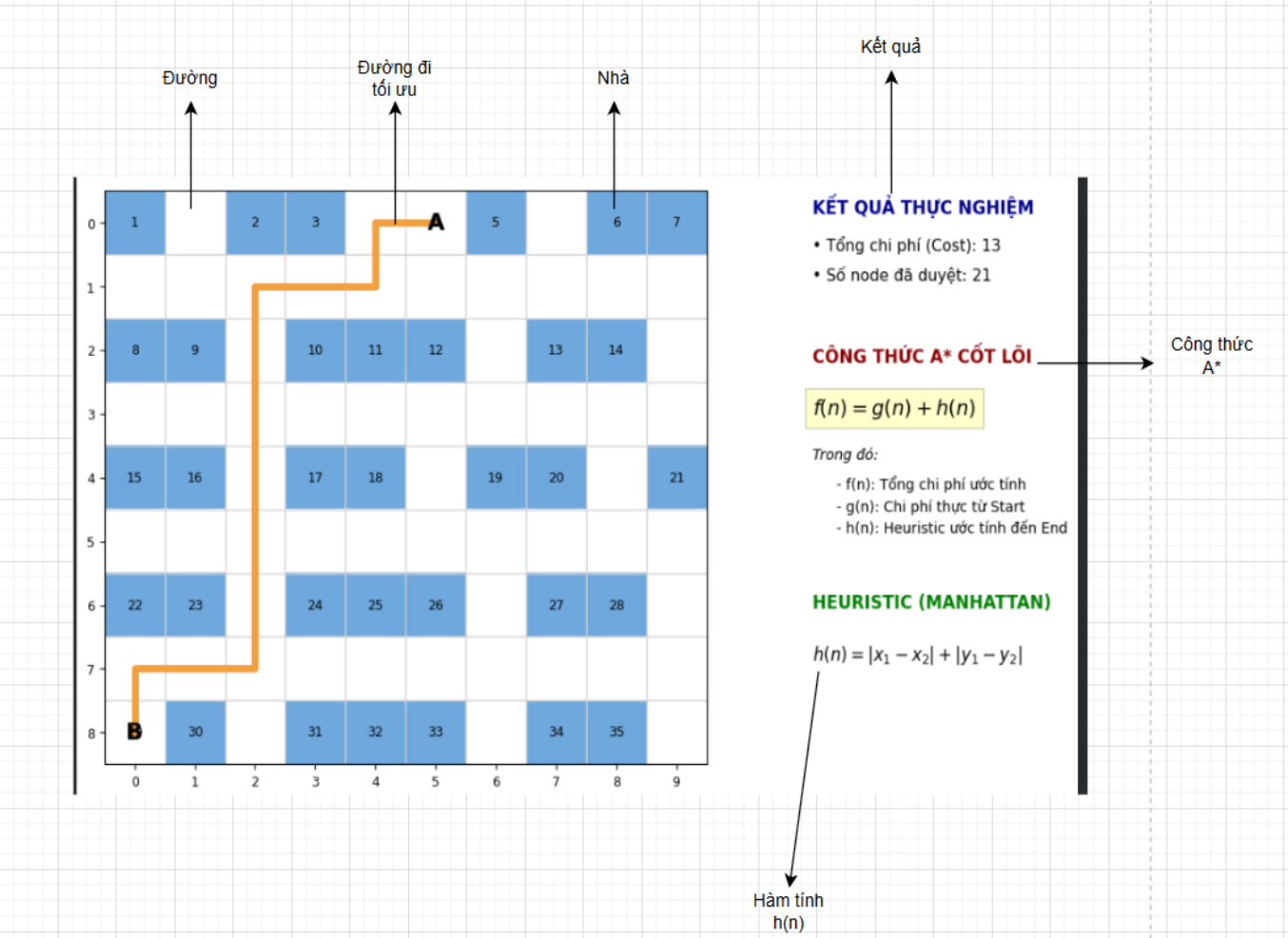
Hình 2.2: Bản đồ mini sau khi mô hình hóa

2.3. Triển khai bài toán trên môi trường lập trình

Trong phần này, nhóm xây dựng một chương trình Python nhằm mô phỏng bản đồ mini và áp dụng thuật toán A* để tìm đường đi tối ưu. Chương trình cho phép người dùng nhập hai vị trí bất kỳ làm điểm bắt đầu và điểm kết thúc. Sau khi nhận dữ liệu, hệ thống sẽ hiển thị bản đồ ban đầu, thực thi thuật toán A*, và trực quan hóa kết quả bằng thư viện **Matplotlib**.

Đường đi được tìm thấy sẽ được biểu diễn bằng một đường màu nổi bật trên bản đồ, kèm theo các thông tin như tổng chi phí di chuyển, số lượng node đã duyệt và công thức A* được minh họa trực tiếp trên giao diện. Phần triển khai không chỉ cung cấp đường đi cụ thể mà còn mô tả rõ quy trình hoạt động của thuật toán, giúp người xem dễ dàng hiểu cách A* lựa chọn hướng đi tối ưu.

Người dùng có thể thử nhiều cặp vị trí khác nhau để quan sát sự thay đổi về đường đi và số lượng node được duyệt. Qua đó, nhóm đánh giá được mức độ hiệu quả của thuật toán A* trong từng tình huống khác nhau trên bản đồ mini.



Hình 2.3: Kết quả chạy bài toán

2.4. Áp dụng thuật toán A*

2.4.1 Nguyên lý thuật toán A*

Thuật toán A* là một phương pháp tìm đường tối ưu trên đồ thị, kết hợp giữa tìm kiếm chi phí thực tế và ước lượng khoảng cách đến đích. Trên bản đồ mini, mỗi ô trong ma trận được coi là một node, và thuật toán sẽ tìm đường đi ngắn nhất từ điểm xuất phát đến điểm kết thúc dựa trên tổng chi phí $f(n)$:

$$f(n) = g(n) + h(n)$$

Trong đó:

- $g(n)$là chi phí thực tế từ điểm bắt đầu đến node n.
- $h(n)$là giá trị heuristic ước lượng chi phí còn lại từ node nđến điểm đích.

Đối với bản đồ dạng lưới và di chuyển theo 4 hướng, heuristic Manhattan được sử dụng:

$$h(n) = | x_n - x_{goal} | + | y_n - y_{goal} |$$

Giá trị này đảm bảo **không đánh giá quá cao chi phí thực** và giúp thuật toán luôn tìm được đường đi ngắn nhất.

2.4.2 Quy trình xử lý trên bản đồ mini

1. **Khởi tạo:**

- Thuật toán nhận điểm **bắt đầu (A)** và điểm **kết thúc (B)**.
- Node bắt đầu được thêm vào **danh sách OPEN** (danh sách các node cần duyệt).
- Gán chi phí thực tế $g(A) = 0$và chi phí tổng ước tính $f(A) = h(A)$.

2. **Duyệt node:**

- Chọn node trong OPEN có **giá trị f(n) thấp nhất** để mở rộng.
- Di chuyển node này sang **CLOSED list** (danh sách các node đã duyệt).
- Quá trình này đảm bảo thuật toán luôn mở rộng node **tiềm năng tối ưu nhất trước**, tránh lãng phí thời gian trên các hướng không hiệu quả.

3. **Xác định các node lân cận:**

- Với node hiện tại, thuật toán kiểm tra các **ô lân cận đi được** (trên, dưới, trái, phải).
- Các ô này phải là đường đi trống hoặc là điểm kết thúc.
- Mỗi node lân cận này được xem là ứng viên tiếp theo để di chuyển.

4. **Cập nhật chi phí và đường đi:**

- Tính chi phí tạm thời từ Start tới node lân cận $g_{\text{temp}} = g(\text{hiện tại}) + 1$.
- Nếu node chưa duyệt hoặc chi phí mới rẻ hơn, cập nhật:
    - $g(n) = g_{\text{temp}}$
    - $f(n) = g(n) + h(n)$
    - Ghi lại **node cha** để phục hồi đường đi sau khi tìm xong.

5. **Lặp lại quá trình:**

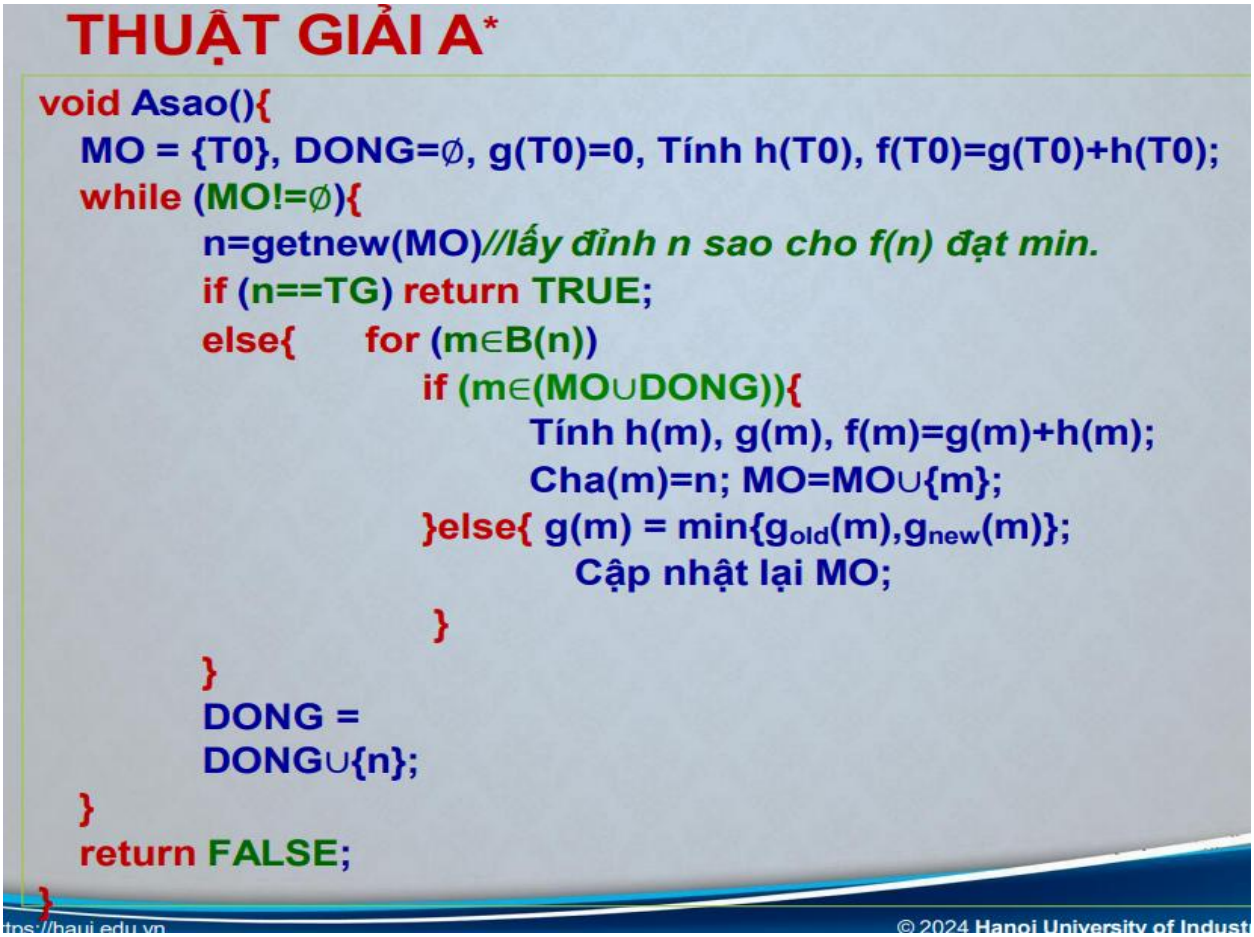
- Thuật toán tiếp tục chọn node có $f(n)$nhỏ nhất từ OPEN.
- Lặp lại bước 2–4 cho đến khi **tìm thấy node kết thúc** hoặc hết các node trong OPEN.

6. **Truy xuất đường đi tối ưu:**

- Khi đạt điểm kết thúc, thuật toán **quay ngược từ node kết thúc về node bắt đầu** theo node cha đã lưu, tạo ra đường đi ngắn nhất.
- Đồng thời, tổng chi phí $g(B)$và số node đã duyệt được ghi lại để đánh giá hiệu quả thuật toán.

7. **Hiển thị kết quả:**

- Đường đi được vẽ trên bản đồ mini, điểm Start và End được đánh dấu rõ ràng.
- Hiển thị thông tin: **tổng chi phí, số node duyệt**, và minh họa công thức A* trực quan.



Hình 2.4: Mô tả quá trình xử lí logic bài toán

2.5. Cài đặt và phân tích cơ chế hoạt động

Sau khi nghiên cứu lý thuyết, nhóm em tiến hành hiện thực thuật toán A* bằng ngôn ngữ Python. Để thuật toán hoạt động hiệu quả trên mô hình lưới (Grid Map), nhóm tập trung xử lý hai thành phần cốt lõi: hàm ước lượng (Heuristic) và vòng lặp tìm kiếm chính.

1. Hàm Heuristic (ước lượng khoảng cách)

Trong mô hình lưới cho phép di chuyển theo bốn hướng (lên, xuống, trái, phải), khoảng cách Manhattan được lựa chọn thay vì khoảng cách Euclid. Hàm Manhattan được tính theo công thức:

$$h(n) = | x_n - x_{goal} | + | y_n - y_{goal} |$$

Hàm này biểu diễn số bước tối thiểu cần di chuyển theo trục ngang và dọc để tới đích. Nhờ đó, thuật toán được định hướng đúng mà không bị ảnh hưởng bởi các đường chéo—những hướng di chuyển không tồn tại trên mô hình lưới 4 hướng.

```
# Hàm Heuristic Manhattan: |x1 - x2| + |y1 - y_2|
def heuristic(nut, dich):
    return abs(nut[0] - dich[0]) + abs(nut[1] - dich[1])
```

Hình 2.5: Hàm heuristic

Biện luận lựa chọn Heuristic (Manhattan vs. Euclidean):

Trong bài toán tìm đường trên lưới ô vuông (Grid Map), việc lựa chọn hàm heuristic ảnh hưởng trực tiếp đến hiệu suất của thuật toán A*. Nhóm đã cân nhắc hai loại khoảng cách phổ biến:

- Khoảng cách Euclidean (Đường chim bay): $h(n) = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$.

  Đây là khoảng cách ngắn nhất về mặt hình học. Tuy nhiên, trong mô hình lưới chỉ cho phép di chuyển 4 hướng (ngang/dọc), chúng ta không thể đi chéo. Do đó, Euclidean thường đưa ra ước lượng nhỏ hơn nhiều so với chi phí thực tế (underestimate), khiến thuật toán A* phải mở rộng nhiều node hơn mức cần thiết, làm chậm tốc độ tìm kiếm.

- Khoảng cách Manhattan: $h(n) = |x_1-x_2| + |y_1-y_2|$.

  Công thức này tính tổng số bước phải đi theo trục ngang và dọc, mô phỏng chính xác cách di chuyển trong ma trận. Manhattan cung cấp một ước lượng "chặt" hơn (tighter lower bound) và sát với thực tế hơn so với Euclidean. Chính vì vậy, việc sử dụng Manhattan giúp thuật toán A* định hướng tốt hơn, hội tụ về đích nhanh hơn mà vẫn đảm bảo tìm ra đường đi tối ưu (Admissible).

2. Cơ chế tìm kiếm đường đi tối ưu (Thuật toán A*)

Thuật toán A* là trung tâm của quá trình tìm đường. Thay vì duyệt lần lượt tất cả các ô như thuật toán loang BFS, A* sử dụng **Hàng đợi ưu tiên (Priority Queue)** để quản lý tập các ô đang chờ được mở rộng, gọi là **Open Set**.

Quy trình hoạt động như sau:

Tại mỗi bước, thuật toán luôn chọn ô có chi phí dự kiến nhỏ nhất theo công thức:

$$f(n) = g(n) + h(n)$$

Trong đó:

$g(n)$: chi phí thực tế từ điểm bắt đầu đến ô hiện tại

$h(n)$: chi phí ước lượng từ ô hiện tại đến đích

- Khi mở rộng một ô, thuật toán duyệt qua tất cả các ô hàng xóm. Nếu một ô là vật cản, nó sẽ bị bỏ qua ngay lập tức.
- Nếu ô hàng xóm có thể đi được, thuật toán tính chi phí mới g_{new}. Nếu g_{new}**nhỏ hơn** chi phí cũ của ô đó, thuật toán sẽ:

  + Cập nhật lại giá trị $g(n)$và $f(n)$

  + Cập nhật parent (dùng để truy vết đường đi sau khi tìm xong)

  + Đưa ô đó trở lại Open Set nếu cần thiết

- Các ô sau khi đã mở rộng hoàn tất sẽ được đưa vào **Closed Set**, giúp thuật toán không xử lý lại chúng, tránh lặp vô hạn và tối ưu thời gian.

```python
# Vòng lặp chính: Luôn chọn node có f(n) thấp nhất từ hàng đợi ưu tiên (mo)
while mo:
    _, nut_hien_tai = heapq.heappop(mo)

    # Nếu đến đích -> Dừng và trả về đường đi
    if nut_hien_tai == dich:
        return truy_vet_duong_di(cha, dich), g_diem[dich], so_node_duyet

    # Xét các ô hàng xóm
    for xom in hang_xom(nut_hien_tai, batdau, dich, luoi):
        # Tính toán chi phí mới: g_tam = chi phí cũ + 1 bước đi
        g_tam = g_diem[nut_hien_tai] + 1

        # Nếu tìm thấy đường đi tốt hơn đến ô hàng xóm này
        if xom not in g_diem or g_tam < g_diem[xom]:
            g_diem[xom] = g_tam
            # Công thức A*: f(n) = g(n) + h(n)
            f_new = g_tam + heuristic(xom, dich)
            heapq.heappush(mo, (f_new, xom))
```

Hình 2.6: Đoạn code vòng lặp chính xử lý việc chọn đỉnh tối ưu

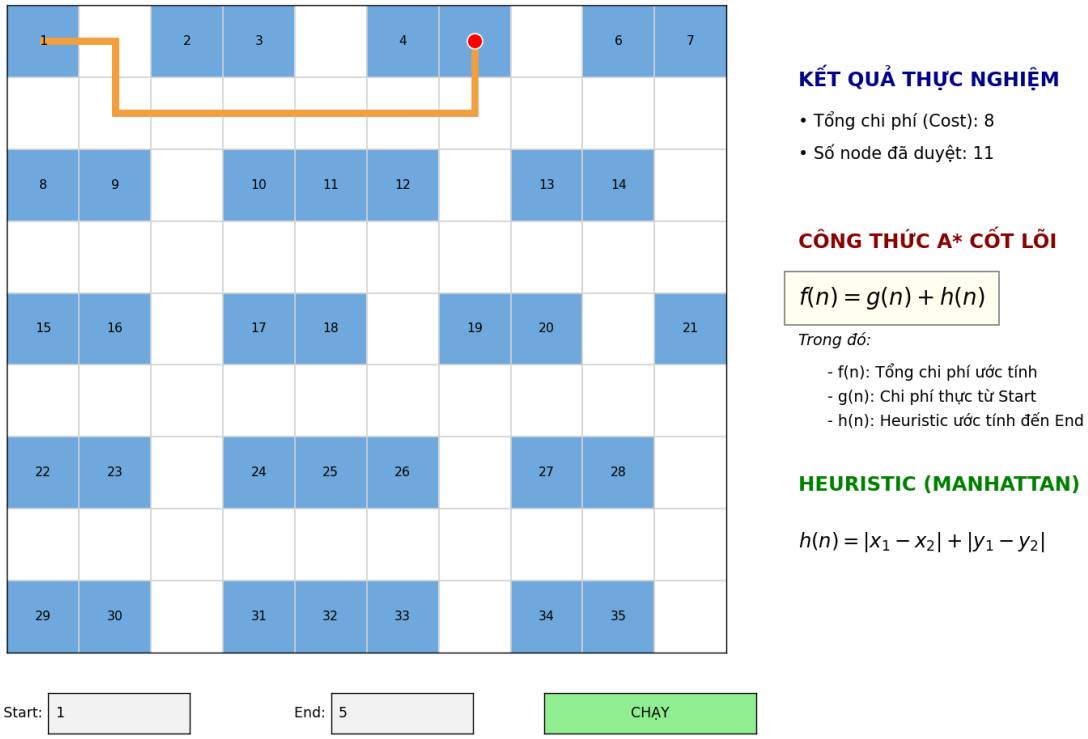
Việc sử dụng heapq (cấu trúc dữ liệu dạng Heap) giúp thao tác lấy phần tử có giá trị nhỏ nhất chỉ tốn độ phức tạp O(logN). Điều này góp phần cải thiện đáng kể hiệu suất của thuật toán, đặc biệt khi kích thước bản đồ lớn.

2.6. Kết quả thực nghiệm (Experimental Results)

Để đánh giá độ ổn định và khả năng hoạt động của thuật toán, nhóm em tiến hành thực nghiệm trên 3 kịch bản đại diện cho các tình huống tìm đường thực tế.

Kịch bản 1: Tìm đường cơ bản

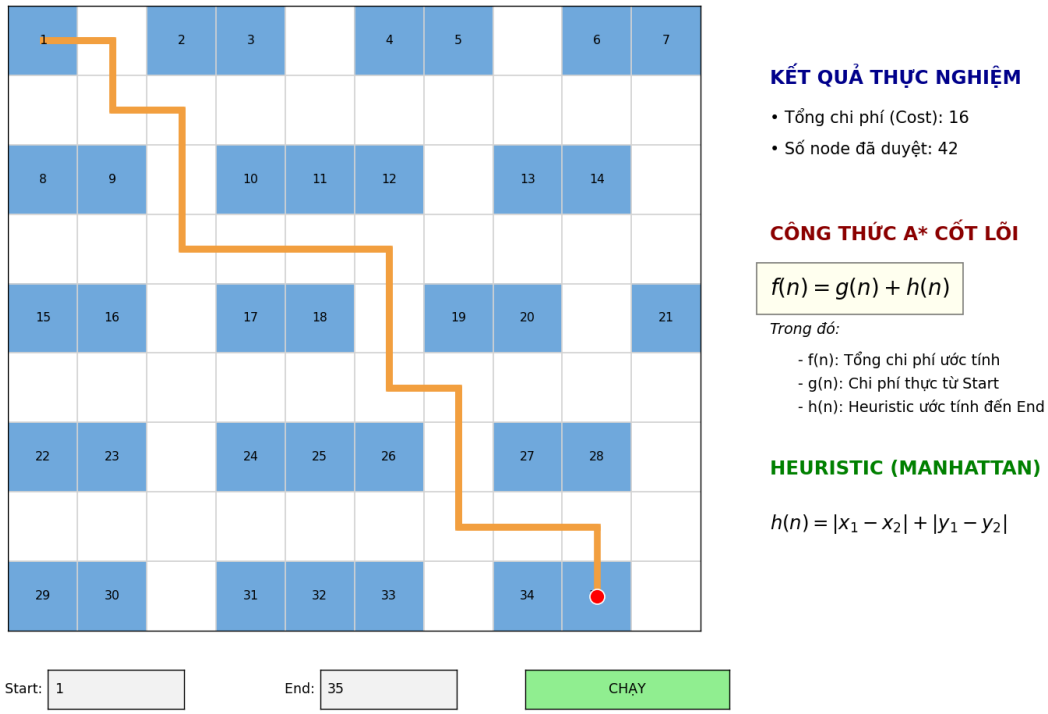
- **Thiết lập:** Tìm đường từ nhà số 1 đến nhà số 5
- **Quan sát:** Đường đi trực tiếp bị chắn bởi các nhà số 2 và 3. Thuật toán tự động lựa chọn hướng vòng xuống dưới để tránh vật cản.
- **Kết quả:** Tìm được đường hợp lệ với tổng chi phí **8 bước**.



Hình 2.7: Kết quả tìm đường né vật cản đơn giản

Kịch bản 2: Tìm đường phức tạp (Xuyên bản đồ)

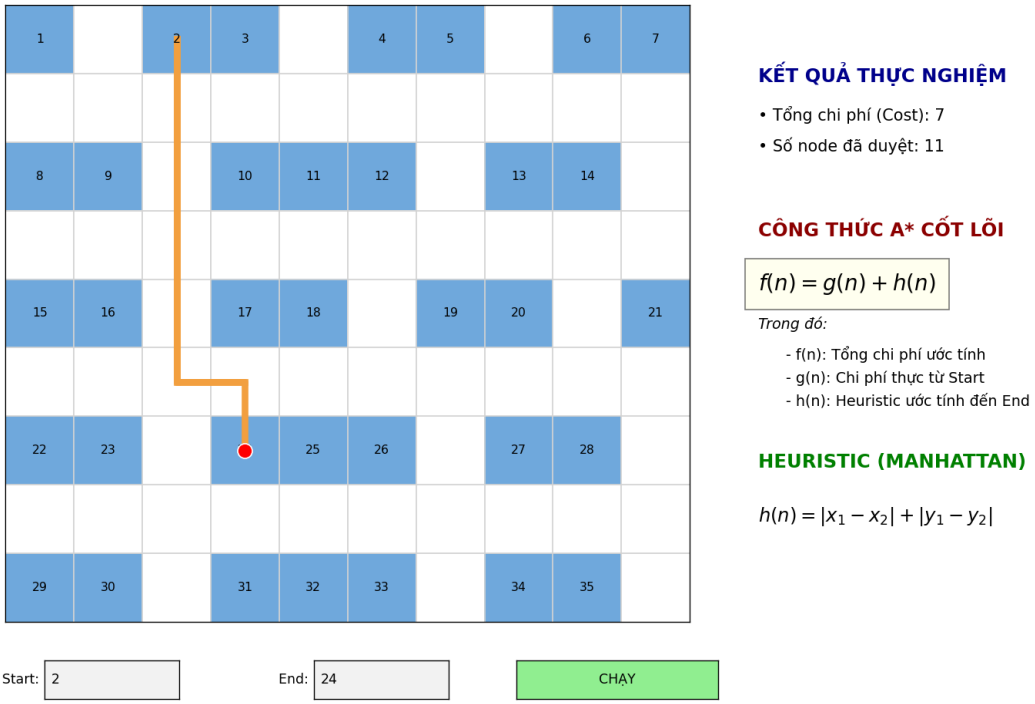
- **Thiết lập:** Đi từ góc trái trên (1) đến góc phải dưới (35).
- **Kết quả:** Đây là bài toán khó nhất. A* đã tìm được đường đi tối ưu, luồn qua các khe hẹp giữa các dãy nhà.
- **Tổng chi phí:** 16 bước



Hình 2.8: Kết quả thử nghiệm kịch bản phức tạp

Kịch bản 3: Xử lý vật cản dạng "góc chết"

- **Thiết lập:** Đi từ nhà số 2 đến nhà số 24
- **Đặc điểm:** Nhà 24 bị khuất sau một dãy nhà tạo thành "góc chết".
- **Kết quả:** Thuật toán không bị kẹt, mà lựa chọn hướng vòng hợp lý để tiếp cận đích.



Hình 2.9: Thuật toán xử lý vật cản dạng góc chết

2.7. Đánh giá và Phân tích kết quả (Evaluation & Analysis)

Dựa trên các số liệu thu thập được, chúng em có những phân tích sâu hơn về hiệu năng:

Bảng tổng hợp số liệu:

	A	B	C	D	E	F
1	Kịch bản	Start → End	Chi phí thực (g(r	Số Node đã duy	Tỷ lệ duyệt bản đồ	
2	1	1 → 5	8	11	≈12%	
3	2	1 → 35	16	42	≈46%	
4	3	2 → 24	7	11	≈12%	

Hình 2.10: Bảng phân tích kết quả

Trong **kịch bản 2** (mức độ phức tạp cao nhất), để tìm được đường đi dài 16 bước, thuật toán chỉ cần mở rộng **42 node** trên tổng số **90 node** của bản đồ. Điều này có nghĩa là A* chỉ duyệt **chưa tới 50%** không gian tìm kiếm.

Kết quả này chứng minh rằng **hàm Heuristic Manhattan đã phát huy hiệu quả rất tốt**. Nó hoạt động như một "la bàn định hướng", liên tục ưu tiên các node tiến gần mục tiêu, từ đó:

- giảm số phép tính cần thực hiện,
- rút ngắn thời gian tìm kiếm,
- tránh việc dò mù, duyệt lan rộng không cần thiết như thuật toán Dijkstra (không dùng heuristic).

Độ chính xác

Trong toàn bộ các trường hợp thử nghiệm, kết quả thu được đều cho thấy đường đi mà thuật toán tìm được **luôn là đường ngắn nhất**. Không ghi nhận trường hợp nào A* đi vòng hoặc lựa chọn đường kém tối ưu.

Điều này khẳng định rằng với heuristic Manhattan (một heuristic **admissible** và **consistent** trong bản đồ dạng lưới 4 hướng), thuật toán A* luôn đảm bảo tính **tối ưu hoàn toàn**.

2.8. Hạn chế và Hướng phát triển (Limitations & Future Work)

Mặc dù chương trình demo đã hoạt động ổn định và cho kết quả tốt, nhóm vẫn nhận thấy một số hạn chế khi xét đến khả năng ứng dụng thực tế.

1. Hạn chế

Mô hình lưới (Grid Map) làm đường đi gấp khúc

Việc sử dụng lưới ô vuông 4 hướng khiến đường đi sinh ra có nhiều khúc cua vuông góc, chưa thật sự "mượt" như di chuyển trong thế giới thực.

Trọng số tĩnh (Cost = 1)

Hiện tại mọi ô đều có chi phí bằng nhau.
Trong thực tế, đường đi có thể có:

- đoạn đông đúc → cost cao
- đoạn nguy hiểm → hạn chế đi vào
- đoạn rộng thoáng → cost thấp

Điều này chưa được mô phỏng trong phiên bản hiện tại.

Hệ thống chỉ hoạt động trên ứng dụng Desktop

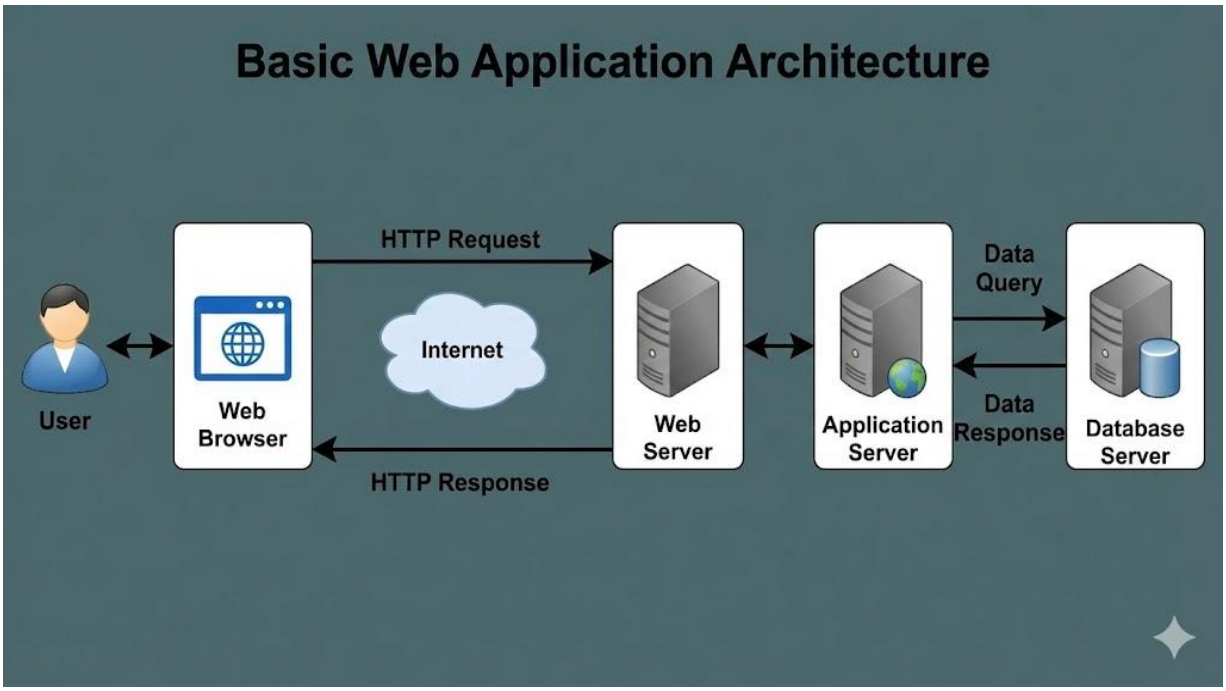
Việc triển khai local khiến khả năng chia sẻ, mở rộng dữ liệu hoặc dùng trên nhiều thiết bị còn hạn chế.

2. Đề xuất phát triển

Xây dựng hệ thống Web Map

Nhóm đề xuất nâng cấp từ ứng dụng chạy trên máy cá nhân sang **Web Application**, tương tự cơ chế hoạt động của Google Maps:

- hiển thị bản đồ và đường đi trực tiếp trên trình duyệt
- nhiều người dùng có thể sử dụng cùng lúc
- dễ quản lý, cập nhật và triển khai dữ liệu
- có thể tích hợp API (OpenStreetMap, Leaflet, Mapbox)



Hình 2.11: mô hình hệ thống web map

Bổ sung trọng số động

Nâng cấp mô hình để mỗi đường đi có chi phí riêng:

- đường đông → cost cao
- đường rẽ trái/phải → cost phụ
- địa hình khó đi → cost tăng

Điều này giúp mô phỏng thực tế chính xác và hay hơn.

- Làm mượt đường (Path Smoothing)

Áp dụng thuật toán:

+ Rubber Banding

+ Catmull–Rom Spline

+ Theta*

để giảm số góc cua, tạo đường đi liên tục và tự nhiên hơn.

- **Mở rộng hướng di chuyển (8 directions)**

Cho phép đi chéo để đường đi thực tế hơn trong bản đồ mở.

CHƯƠNG 3: HƯỚNG PHÁT TRIỂN TRONG TƯƠNG LAI VÀ KẾT LUẬN

3.1 Hướng phát triển trong tương lai

Trong thời gian tới, hệ thống Google Maps mini có thể tiếp tục được mở rộng và hoàn thiện theo nhiều hướng. Trước hết, có thể mở rộng quy mô bản đồ và nguồn dữ liệu, thay thế dữ liệu thủ công bằng dữ liệu GPS thực tế hoặc dữ liệu bản đồ mở (OpenStreetMap) nhằm tăng tính thực tiễn và độ chính xác của mô phỏng. Bên cạnh đó, hệ thống có thể xem xét tích hợp các biến thể nâng cao của thuật toán A* như Jump Point Search (JPS), Theta* hoặc kết hợp với các thuật toán tìm đường khác để so sánh hiệu suất và khả năng tối ưu trong những điều kiện khác nhau.

Ngoài ra, một hướng phát triển quan trọng là xem xét các yếu tố động trong quá trình tìm đường, chẳng hạn như thay đổi trọng số cạnh theo mật độ giao thông, thời gian di chuyển hoặc chướng ngại vật phát sinh, từ đó tiến tới mô phỏng các bài toán tìm đường trong môi trường thực tế. Hệ thống cũng có thể được cải thiện giao diện người dùng, bổ sung các chức năng tương tác, phóng to – thu nhỏ bản đồ, hiển thị từng bước tìm kiếm của thuật toán nhằm tăng tính trực quan và khả năng giảng dạy.

Cuối cùng, đề tài có thể được phát triển theo hướng ứng dụng thực tế, phục vụ cho học tập, nghiên cứu hoặc làm nền tảng cho các hệ thống hỗ trợ định tuyến thông minh trong tương lai.

3.2 Kết luận

Trong quá trình thực hiện đề tài "*Ứng dụng A để tìm đường đi ngắn nhất trong Google Maps mini**", nhóm đã xây dựng hệ thống mô phỏng bản đồ với các đỉnh là địa điểm và các cạnh là tuyến đường có trọng số, áp dụng thuật toán A* với hàm heuristic Euclid và Manhattan để tìm đường đi tối ưu. Kết quả thực nghiệm cho thấy A* hoạt động ổn định, tìm được đường đi ngắn nhất và hiệu suất chịu ảnh hưởng trực tiếp bởi hàm heuristic.

Mô phỏng Google Maps mini giúp nhóm vừa nắm vững lý thuyết tìm kiếm heuristic, vừa nâng cao kỹ năng lập trình Python và trực quan hóa dữ liệu. Mặc dù hệ thống còn hạn chế về quy mô và dữ liệu thủ công, nghiên cứu mở ra hướng phát triển tiếp theo như sử dụng

dữ liệu GPS thực, tích hợp biến thể thuật toán JPS hoặc Theta*, và cải thiện giao diện trực quan.

Nhìn chung, đề tài đạt mục tiêu: minh họa cách thuật toán A* tìm đường đi ngắn nhất, đánh giá hiệu quả heuristic, đồng thời làm nền tảng cho các nghiên cứu và ứng dụng nâng cao trong tương lai.

Tài liệu tham khảo

Buzz. (2025, 11 1). *Thuật toán tìm kiếm A*.* Retrieved from https://mytour.vn/vi/blog/bai-viet/thuat-toan-tim-kiem-a.html

tên, k. (2015). *tìm kiếm cực tiểu hàm đánh giá.* Retrieved 10 5, 2025, from https://kcntt.duytan.edu.vn/Home/ArticleDetail/vn/128/2413/tim-kiem-cuc-tieu-su-dung-ham-danh-gia-%C3%A2%E2%82%AC%E2%80%9C-thuat-toan-a

TungNV. (2016, 14 11). *Hiểu các thuật toán và triển khai đường dẫn A * với Python.* Retrieved from https://viblo.asia/p/a-pathfinding-nwmkyEjlkoW