TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Lê Phạm Hoàng Trung

SEARCH IN GRAPH
(TÌM KIẾM TRONG ĐỒ THỊ)

ĐỒ ÁN 1 THỰC HÀNH CƠ SỞ TRÍ TUỆ NHÂN TẠO
CHƯƠNG TRÌNH CHÍNH QUY

Tp. Hồ Chí Minh, tháng 10/2023

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Lê Phạm Hoàng Trung - 21120157

SEARCH IN GRAPH
(TÌM KIẾM TRONG ĐỒ THỊ)

ĐỒ ÁN 1 THỰC HÀNH CƠ SỞ TRÍ TUỆ NHÂN TẠO
CHƯƠNG TRÌNH CHÍNH QUY

GIÁO VIÊN HƯỚNG DẪN

GS.TS. Lê Hoài Bắc

Giảng viên Nguyễn Bảo Long

Tp. Hồ Chí Minh, tháng 10/2023

Mục lục

Danh sách hình

Danh sách bảng

Chương 1

Biểu diễn bài toán tìm kiếm

1.1 Biểu diễn bài toán tìm kiếm - Các thành phần của bài toán tìm kiếm

Khi giải quyết một bài toán bằng phương pháp tìm kiếm (bài toán tìm kiếm) trước hết phải biểu diễn được bài toán đó. Một bài toán thường gồm có vấn đề (yêu cầu) và lời giải cho bài toán. Phương pháp tìm kiếm được sử dụng để tìm ra lời giải trong không gian tìm kiếm. Biểu diễn một bài toán tìm kiếm thường bao gồm năm thành phần:

- Không gian trạng thái Q (tập hữu hạn các trạng thái): Cho biết tất cả các trạng thái có thể có của bài toán
- Trạng thái ban đầu $S \subseteq Q$ (tập khác rỗng gồm các trạng thái ban đầu): Nơi bắt đầu việc tìm kiếm, ví dụ: Du khách sẽ xuất phát từ thành phố A thì thành phố A chính là trạng thái ban đầu
- Một tập các trạng thái đích $G \subseteq Q$ (tập khác rỗng gồm các trạng thái đích/kết thúc): Các trạng thái đích này đều thuộc không gian trạng thái. Tuỳ vào bài toán mà có một hay nhiều trạng thái đích khác nhau. Trong bài toán đi du lịch thì trạng thái đích chính là thành phố cần đến của du khách. Tuy nhiên, trong những bài toán khác thì trạng thái đích chỉ được mô tả bởi một vài điều kiện nào đó chứ không phải là một tập các trạng thái xác định nữa. Như trong cờ vua, trạng thái đích chính là tình huống "chiếu bí" tức là tình huống đối phương không thể ngăn cản được việc quân vua của mình bị tấn công
- Hàm trạng thái con $succs(x)$: Phát sinh trạng thái kế tiếp từ một trạng thái cụ thể x, nhận một trạng thái x làm đầu vào và trả về

một tập các trạng thái có thể đến từ x trong một bước. Hàm trạng thái con, cùng với trạng thái ban đầu, tạo nên không gian trạng thái của bài toán. Không gian này bao gồm các trạng thái mà ta sẽ xem xét đến trong quá trình tìm kiếm

- Hàm chi phí đường đi $cost(x, x')$: Dùng để tính chi phí cho lời giải của bài toán. Hàm nhận 2 trạng thái x và x' làm đầu vào và trả về chi phí di chuyển một bước từ x đến x'. Hàm chi phí chỉ được định nghĩa khi x' là trạng thái con của x.

1.2 Mã giả chung để giải quyết các bài toán tìm kiếm

```
function Search_Tree(problem, frontier) return solution or failure
    frontier <- Add(Create_Node(Initial_State[problem]), frontier)
    loop do
        if Is_Empty(frontier) then return failure
        node <- Remove_Front(frontier)
        if Goal_Test[problem] on State[node] is true
            then return Solution(node)
        frontier <- Add_All(Expand(node, problem), frontier)

function Expand(node, problem) return a set of nodes
    successor_states <- initialize an empty set
    for each (action, result) in Child[problem](State[node]) do
        s <- a new node
        State[s] <- result
        Parent_Node[s] <- node
        Action[s] <- action
        Path_Cost[s] <- Path_Cost[node] + Step_Cost(action, State[node],
    result)
        Depth[s] <- Depth[node] + 1
        Add s to successor_states
    return successor_states
```

- Is_Empty(frontier): Trả về true nếu hàng đợi rỗng
- Remove_Front(frontier): Trả về phần tử đầu tiên trong hàng đợi
- Add_All(Node_list, frontier): Thêm một loạt các phần tử vào hàng đợi và trả về hàng đợi mới.

Tuỳ vào kiểu hàng đợi (frontier) mà thao tác thêm vào và lấy ra có sự khác biệt, có 3 kiểu hàng đợi chính:

- Queue/FIFO (First-in-First-out)
- Stack/LIFO (Last-in-Last-out)
- Priority Queue (Hàng đợi ưu tiên)

1.3 Uninformed Search - Informed Search

1.3.1 Uninformed Search (Tìm kiếm không có thông tin/Tìm kiếm mù)

Bài toán tìm kiếm mù là những bài toán không có thêm bất kỳ thông tin bổ sung nào về nút đích ngoại trừ những thông tin được cung cấp trong định nghĩa bài toán. Các bước để đạt được trạng thái mục tiêu từ trạng thái đầu chỉ khác nhau ở thứ tự và độ dài của hành động. Các thuật toán tìm kiếm mù phổ biến: tìm kiếm theo chiều rộng (BFS), tìm kiếm theo chiều sâu (DFS), tìm kiếm chi phí đồng nhất (UCS) và tìm kiếm theo chiều sâu có giới hạn.

1.3.2 Informed Search (Tìm kiếm có thông tin)

Các chiến lược tìm kiếm có thông tin dựa trên tri thức có được từ bài toán bên cạnh định nghĩa của bài toán để tìm ra lời giải hiệu quả hơn nhiều so với các chiến lược tìm kiếm mù. Cách tiếp cận tổng quát ta sẽ xem xét gọi là **tìm kiếm tối ưu**, nó dựa trên một **hàm đánh giá** để chọn ra nút tốt nhất để mở rộng. Thông thường hàm đánh giá này sẽ tính toán khoảng cách tới đích (còn bao xa nữa sẽ tới đích) như vậy sẽ chọn nút có giá trị khoảng cách nhỏ nhất). Chữ "tối ưu" không có nghĩa là lúc nào cũng lấy được nút tốt nhất mà có nghĩa là lấy được nút *được xem là* tốt nhất theo tiêu chí của hàm đánh giá. Nếu hàm đánh giá này phản ánh chính xác bài toán thì khi này nút được chọn sẽ thật sự tốt nhất, nếu không thì nó sẽ làm lệch hướng tìm kiếm. Có một họ rất nhiều thuật toán dạng này với những hàm đánh giá khác nhau. Điểm mấu chốt của những

Loại tìm kiếm	**Ưu điểm**	**Nhược điểm**
Uninformed Search	Dễ hiểu và cài đặt. Thích hợp cho bài toán đơn giản.	Không sử dụng thông tin bổ sung. Không hiệu quả cho bài toán phức tạp.
Informed Search	Sử dụng thông tin bổ sung từ hàm heuristic. Hiệu quả cho bài toán phức tạp và có khả năng tối ưu hóa đường đi.	Đòi hỏi việc xác định hàm heuristic. Cần nhiều tài nguyên tính toán.

Bảng 1.1: Bảng so sánh hai chiến lược tìm kiếm

thuật toán này là ở hàm *heuristic* (kí hiệu $h(n)$) với $h(n)$ = xấp xỉ chi phí của đường đi (chi phí thấp nhất) từ nút thứ n đến nút đích. Ví dụ: Có thể xấp xỉ bằng khoảng cách đường chim bay từ thành phố A đến thành phố B.

Các hàm *heuristic* là dạng thông dụng nhất để đưa tri thức về bài toán vào trong thuật toán tìm kiếm. Các thuật toán tìm kiếm có thông tin phổ biến: AStar search, Best-First search và Greedy search.

Chương 2

Các thuật toán tìm kiếm mù

2.1 Tìm kiếm theo chiều rộng (Breadth-first search - BFS)

2.1.1 Ý tưởng

Breadth-First Search (BFS) là một thuật toán tìm kiếm trong đồ thị thường được áp dụng để khám phá đồ thị theo chiều rộng từ một đỉnh ban đầu. Thuật toán này xây dựng một cây tìm kiếm theo chiều rộng, bắt đầu bằng việc mở rộng đỉnh gốc, sau đó là tất cả các đỉnh con của nó, và tiếp tục như vậy. Tóm lại, chúng ta thực hiện việc duyệt qua các đỉnh ở mức độ sâu d trước khi di chuyển xuống độ sâu $d + 1$. Nguyên tắc này xuất phát từ việc chúng ta muốn xem xét tất cả các con đường có thể đạt được đích trong n bước. Do chúng ta chưa biết giá trị cụ thể của n, chúng ta cần kiểm tra tất cả các đường đi có thể trong thứ tự từ $n = 1, 2, 3, ...$ cho đến khi đạt được đích. Cách làm này tương tự như việc trải rộng trên cây tìm kiếm, và do đó, phương pháp này được gọi là "tìm kiếm theo chiều rộng". Ý tưởng chung của BFS bao gồm các bước sau:

- Từ một đỉnh, ta tìm các đỉnh kề rồi duyệt qua các đỉnh này.
- Tiếp tục tìm các đỉnh kề của đỉnh vừa xét rồi duyệt tiếp đến khi đi qua hết tất cả các đỉnh có thể đi.

2.1.2 Pseudo code (Mã giả)

```
function BFS (G, s)
    //Where G is the graph and s is the source node let Q be queue.
    //Inserting s in queue until all its neighbour vertices are marked.
    Q.enqueue( s )
    V = () // Visited set or closed set
```

```
6
7     // mark s as visited.
8     V.add(s)
9
10    while (Q is not empty)
11        //Removing that vertex from queue, whose neighbour will be visited
12        v  =  Q.dequeue( )
13        if Is_Goal(v):
14            return path
15
16        //processing all the neighbours of v
17        for all neighbours w of v in Graph G
18            if w is not in V
19                //Stores w in Q to further visit its neighbour
20                Q.enqueue(w)
21                // mark w as visited.
22                V.add(w)
```

trong đó:

- Q là một hàng đợi Queue, ở trong giáo trình thì còn gọi là tập mở (open_set)
- V là một tập đóng, tức những đỉnh đã được duyệt rồi (closed_set)
- enqueue là một hàm trong Queue, giúp thêm một phần tử vào cuối hàng đợi
- dequeue là một hàm trong Queue, giúp lấy ra phần tử đầu tiên trong hàng đợi.

2.1.3 Phân tích thuật toán

Một thuật toán tìm kiếm để giải quyết bài toán có thể cho ra một lời giải hoặc thất bại (kết thúc nhưng không tìm thấy lời giải hoặc lặp vô hạn không kết thúc). Ta sẽ đánh giá thuật toán tìm kiếm dựa trên bốn yếu tố sau:

- Tính đầy đủ: Thuật toán có bảo đảm tìm thấy lời giải nếu có hay không?
- Tính tối ưu: Thuật toán có bảo đảm tìm thấy lời giải tối ưu không? (Sẽ tìm thấy đường đi có chi phí nhỏ nhất?)

- Độ phức tạp về thời gian: Mất bao lâu để tìm ra lời giải
- Độ phức tạp về không gian: Cần sử dụng bao nhiêu bộ nhớ trong quá trình tìm kiếm.

Trong đó độ phức tạp về thời gian và không gian thường được xem như các khía cạnh của việc đo lường độ phức tạp của bài toán. Theo lý thuyết trong khoa học máy tính, thước đo thông thường là kích thước của đồ thị không gian trạng thái, bởi vì đồ thị này được xem là một cấu trúc dữ liệu tường minh làm đầu vào cho thuật toán tìm kiếm (ví dụ như bản đồ du lịch). Trong trí tuệ nhân tạo, đồ thị này được thể hiện ngầm bởi trạng thái ban đầu và hàm trạng thái con. Độ phức tạp được thể hiện bằng ba đại được sau:

- b: Hệ số phân nhánh trung bình (số con trung bình hoặc tối đa) ($b > 1$). b đại diện cho số lượng nút con trung bình mà một nút cha có thể sinh ra trong quá trình duyệt. Nếu b lớn, nghĩa là mỗi nút cha có nhiều nút con, dẫn đến một sự gia tăng nhanh chóng trong số nút cần xem xét. Điều này thường xảy ra trên các đồ thị phức tạp hoặc cây tìm kiếm rộng.
- D: Độ dài đường đi từ nút gốc đến nút đích với số bước (chi phí) ít nhất.
- m: Độ dài đường đi dài nhất từ nút gốc đến một nút bất kì.

Đối với thuật toán tìm kiếm theo chiều rộng BFS, ta có thể thấy rằng thuật toán này đã thoả mãn **tính đầy đủ** do thuật toán luôn bảo đảm tìm thấy lời giải nếu có. Nếu nút đích gần nhất nằm ở một độ sâu d xác định trên cây tìm kiếm, thuật toán chắc chắn tìm thấy nút này sau khi đã mở rộng hết tất cả những nút ở độ sâu gần hơn (với điều kiện b hữu hạn). Nút đích gần nhất không nhất thiết là lựa chọn tối ưu, nhưng thuật toán tìm kiếm theo chiều rộng vẫn tối ưu nếu chi phí chuyển đổi là một hàm không giảm theo độ sâu của nút (chẳng hạn trong trường hợp tất cả các biến đổi đều có cùng chi phí). Suy ra thuật toán BFS cũng thoả mãn **tính tối ưu**. Vậy cả hai tiêu chuẩn đầu của việc đánh giá một thuật toán tìm kiếm BFS đều làm tốt.

Độ sâu	Số nút	Thời gian	Bộ nhớ
2	1100	11 giây	1MB
4	111100	11 giây	106MB
6	10^7	19 phút	10GB
8	10^9	31 giờ	1TB
10	10^{11}	129 ngày	101TB
12	10^{13}	35 năm	10PB
14	10^{15}	3523 năm	1EB

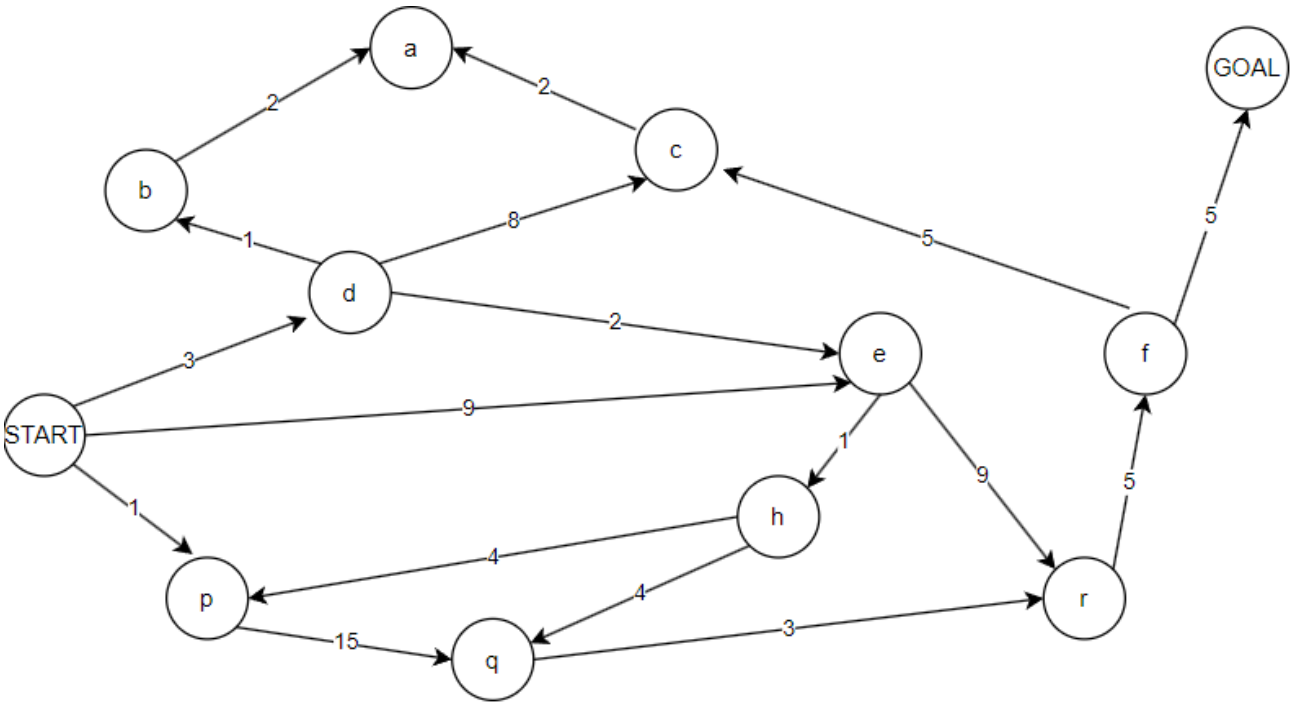
Bảng 2.1: Thể hiện thời gian chạy và bộ nhớ sử dụng của tìm kiếm theo chiều rộng. Với giả thiết là b = 10; 10000 nút/giây; 1000byte/nút

Đối với hai tiêu chuẩn tiếp theo, giả sử mọi trạng thái có thể mở rộng được tối đa b nút. Gốc của cây tìm kiếm (tức nút bắt đầu) sẽ tạo ra b nút ở mức đầu tiên, mỗi nút con mới đó sẽ tạo ra thêm b nút con nữa, tổng cộng sẽ có b^2 nút ở độ sâu thứ hai, cứ thế tiếp tục. Giả sử rằng đường đi tới đích có chiều dài là d. Trong trường hợp xấu nhất sẽ phải mở rộng các nút ở độ sâu d (trừ nút đích không mở rộng). Khi đó tổng số nút được phát sinh sẽ là:

$$b^1 + b^2 + b^3 + ... + b^d + (b^{d+1} - b^1) = O(b^{d+1})$$

Các nút được phát sinh ra đều được lưu trong bộ nhớ, do đó độ phức tạp về mặt không gian giống với độ phức tạp về mặt thời gian và bằng $O(b^{d+1})$ (tính cả nút gốc).

Độ phức tạp là một hàm mũ, như vậy thời gian cũng như không gian sử dụng sẽ tăng rất nhanh theo độ sâu. Điều này được thể hiện qua bảng 2.1 cho thấy thời gian chạy và dung lượng bộ nhớ cần cho thuật toán tìm kiếm theo chiều rộng với $b = 10$ và theo giá trị độ sâu d tăng dần. Giả sử có thể phát sinh 10000 nút trong vòng 1 giây và mỗi nút cần 1000 byte để lưu trữ.



Hình 2.1: Bài toán tìm kiếm

2.1.4 Minh hoạ mã giả

Xét đồ thị như hình 2.1, ta có thể thực hiện từng bước thuật toán BFS như sau, với Q là Queue (hàng đợi) và V là tập Visited:

- Thêm Start vào Q = {Start}
- Lấy Start ra khỏi Q, thêm vào các lân cận là d, e, p vào ta được Q = {d, e, p} và ta thêm Start vào tập visited V = {Start}
- Lấy d ra khỏi Q thêm vào V = {Start, d}, thêm các lân cận của d là b, c vào Q = {e, p, b, c}.
- Lấy e ra khỏi Q thêm vào V = {Start, d, e}, thêm các lân cận của e là r, h vào Q = {p, b, c, r, h}
- Lấy p ra khỏi Q thêm vào V = {Start, d, e, p}, thêm các lân cận của p là q vào Q = {b, c, r, h, q}
- Lấy b ra khỏi Q thêm vào V = {Start, d, e, p, b}, thêm lân cận của b là a vào Q = {c, r, h, q, a}

- Lấy c ra khỏi Q thêm vào V = {Start, d, e, p, b, c}, c không có lân cận nên Q được giữ nguyên Q = {r, h, q, a}
- Lấy r ra khỏi Q thêm vào V = {Start, d, e, p, b, c, r}, thêm lân cận f vào Q = {h, q, a, f}
- Lấy h ra khỏi Q thêm vào V = {Start, d, e, p, b, c, r, h}, h không có lân cận nên Q được giữ nguyên Q = {q, a, f}
- Lấy q ra khỏi Q thêm vào V = {Start, d, e, p, b, c, r, h, q}, q không có lân cận nên Q được giữ nguyên Q = {a, f}
- Lấy a ra khỏi Q thêm vào V = {Start, d, e, p, b, c, r, h, q, a}, a không có lân cận nên Q được giữ nguyên Q = {f}
- Lấy f ra khỏi Q thêm vào V = {Start, d, e, p, b, c, r, h, q, a, f}, f có Goal là lân cận nên thêm Goal vào Q, ta được Q = {Goal}
- Bước tiếp lấy Goal ra và trả về đường đi. Kết thúc thuật toán.

2.2 Tìm kiếm chi phí đồng nhất (Uniform-cost search - UCS)

2.2.1 Ý tưởng

Thuật toán tìm kiếm chi phí đồng nhất (Uniform Cost Search - UCS) là một sự bổ sung quan trọng so với thuật toán tìm kiếm theo chiều rộng (BFS), đặc biệt khi các bước biến đổi có chi phí khác nhau. Trong BFS, chúng ta tìm đường đi với số bước biến đổi ít nhất, trong khi UCS tập trung vào việc tìm đường đi với chi phí chuyển đổi thấp nhất. Điều này làm cho UCS thích hợp cho các bài toán mà chi phí chuyển đổi trọng số khác nhau (hình 2.1).

UCS thường được cài đặt dựa trên thuật toán tìm kiếm tổng quát và sử dụng một hàng đợi ưu tiên (priority queue) để quản lý các nút trạng thái. Việc gọi hàm Search_Tree(problem, priority_queue) sẽ kết quả trong việc tìm kiếm chi phí đồng nhất.

Một điểm mạnh của UCS là khả năng tìm kiếm đường đi tối ưu trong trường hợp các bước biến đổi có chi phí khác nhau. Nó sẽ duyệt qua các nút trạng thái với chi phí thấp nhất đầu tiên, đảm bảo rằng nó luôn tìm kiếm theo chi phí tối ưu.

2.2.2 Pseudo code (Mã giả)

```
function UCS_Search(problem, priority_queue) return solution or failure
    // Add the first node to the priority_queue
    priority_queue <- Add(Create_Node(Initial_State[problem]),
   priority_queue)
    loop do
        if Is_Empty(priority_queue) then return failure
        node <- Get_Lowest_Cost(priority_queue)
        if Goal_Test[problem] on State[node] is true
            then return Solution(node)
        priority_queue <- Add_All(Expand(node, problem), priority_queue)
```

trong đó:

- priority_queue: là một hàng đợi ưu tiên, là một loại cấu trúc dữ liệu trong đó ta có thể thêm vào và lấy ra các phần tử dựa trên độ ưu tiên của mỗi phần tử. Đối với thuật toán tìm kiếm chi phí đồng nhất, độ ưu tiên của các phần tử (trạng thái) chính là chi phí đường đi tới trạng thái đó, trạng thái nào có chi phí đường đi thấp hơn thì độ ưu tiên sẽ cao hơn và trạng thái đó sẽ được lấy ra khỏi hàng đợi trước.

- Get_Lowest_Cost: hàm lấy ra phần tử có chi phí thấp nhất (độ ưu tiên cao nhất) và xoá phần tử đó khỏi hàng đợi.

Lưu ý, với mô tả thuật toán như trên, thuật toán sẽ dừng khi **đích được lấy ra khỏi hàng đợi ưu tiên**. Dòng lệnh kiểm tra là đích được thực hiện ngay sau thao tác lấy một phần tử ra khỏi hàng đợi. Nếu ta thực hiện kiểm tra đích ngay sau thao tác mở rộng một nút thì có thể sẽ bỏ qua đường đi ngắn nhất.

2.2.3 Phân tích thuật toán

Thuật toán tìm kiếm chi phí đồng nhất không quan tâm đến số bước của đường đi mà chỉ quan tâm đến chi phí tổng cộng. Do đó thuật toán có thể bị lặp vô hạn, nếu gặp một nút có chi phí chuyển đổi tới cùng trạng thái đó bằng 0 (ví dụ ứng với hành động không làm gì cả). Chúng ta có thể đảm bảo được **tính đầy đủ** với điều kiện chi phí tại mỗi bước biến đổi lớn hơn hoặc bằng một hằng số c dương đủ nhỏ. Điều kiện này cũng đảm bảo tính tối ưu. Điều này đồng nghĩa với hàm chi phí sẽ tăng theo đường đi, việc mở rộng thêm nút sẽ làm tăng chi phí của đường đi. Do đó, khi bắt gặp nút đích đầu tiên thì đó cũng chính là **lời giải tối ưu**.

Xét hai tiêu chí độ phức tạp thời gian và không gian:

- Độ phức tạp thời gian: Độ phức tạp thời gian của UCS phụ thuộc vào một số yếu tố:

  - b: Hệ số phân nhánh trung bình, tức là số nút con trung bình mà một nút cha có thể sinh ra. Thường, b được xem xét là một tham số đầu vào của thuật toán và tùy thuộc vào cụ thể của bài toán. Đối với một bài toán tìm kiếm trên đồ thị, b có thể biến đổi dựa trên cấu trúc của đồ thị và chi phí chuyển đổi giữa các trạng thái.
  - C^*: Chi phí của đường đi tối ưu, tức là chi phí thấp nhất để đi từ trạng thái ban đầu đến trạng thái đích.
  - e: Chi phí chuyển đổi ít nhất. Đây là giá trị nhỏ nhất trong tất cả các chi phí chuyển đổi giữa các trạng thái. Nếu không có ràng buộc về giá trị e, giá trị này có thể là 1 (khi tất cả các bước có chi phí tối thiểu là 1).

  Độ phức tạp thời gian của UCS được ước tính bằng công thức $O(b^{1+\lfloor \frac{C^*}{e} \rfloor})$. Điều này có nghĩa rằng độ phức tạp tăng theo cấp số mũ với $\frac{C^*}{e}$. Nếu C^* lớn hoặc e nhỏ, thì độ phức tạp thời gian của UCS có thể trở nên rất lớn. Khi tất cả các bước có chi phí chuyển đổi bằng nhau (điều này xảy ra khi $e = 1$), thì công thức trở thành

$O(b^{C^*})$, tức là độ phức tạp tăng theo cấp số mũ với C^*. Điều này có thể là một độ phức tạp rất lớn nếu C^* lớn.

- Độ phức tạp không gian: Độ phức tạp không gian của UCS phụ thuộc vào số lượng trạng thái đã mở rộng và lưu trữ trong hàng đợi ưu tiên. UCS thường lưu trữ tất cả các trạng thái mở rộng trong hàng đợi ưu tiên, sắp xếp theo chi phí. Điều này có nghĩa rằng độ phức tạp không gian của UCS phụ thuộc vào:

  – Số lượng trạng thái trạng thái cần mở rộng để tìm đường đi tối ưu.

  – Kích thước của hàng đợi ưu tiên, nơi các trạng thái được lưu trữ và quản lý.

  Độ phức tạp không gian thường được xác định bởi số lượng trạng thái đã mở rộng, và nó cũng phụ thuộc vào cấu trúc của đồ thị tìm kiếm và chi phí chuyển đổi giữa các trạng thái. UCS có thể lưu trữ một số lượng lớn trạng thái trong hàng đợi ưu tiên, đặc biệt nếu có nhiều trạng thái có chi phí gần nhau. Điều này có thể dẫn đến độ phức tạp không gian lớn, đặc biệt đối với các bài toán với đồ thị lớn hoặc trạng thái không gian lớn. Ngoài ra, độ phức tạp không gian của UCS có thể được kiểm soát bằng cách sử dụng một chiến lược gom nhóm (pruning) hoặc giới hạn số lượng trạng thái được lưu trữ trong hàng đợi ưu tiên.

Tổng quát lại, thuật toán tìm kiếm chi phí đồng nhất chủ yếu dựa trên chi phí của đường đi nên độ phức tạp khó xác định bởi giá trị của b và d (độ sâu) nên ta dùng C^* là chi phí của lời giải tối ưu và e là chi phí chuyển đổi ít nhất. Độ phức tạp về thời gian và không gian của thuật toán là $O(b^{1+[C^*/e]})$ và có thể lớn hơn b^d nhiều. Bởi vì tìm kiếm chi phí đồng nhất có thể và thông thường sẽ mở rộng hướng tìm kiếm theo những nhánh mà ban đầu có chi phí thấp trước khi đi vào con đường đúng mà ban đầu có chi phí cao (tổng chi phí vẫn thấp hơn). Khi chi phí chuyển đổi đều bằng nhau thì $b^{1+[C^*/e]}$ sẽ chỉ còn là b^d.

2.2.4 Minh hoạ mã giả

Với đồ thị trong hình 2.1, ta có các bước thực hiện thuật toán UCS như sau, đặt PQ là Priority Queue:

1. Đầu tiên, ta thêm nút Start vào PQ với chi phí để đến được Start là 0, PQ = {(Start, 0)}

2. Tiếp theo ta lấy nút có chi phí thấp nhất ra khỏi PQ, ở đây ta lấy được nút Start. Sau đó ta mở rộng nút Start tìm được các nút lân cận là p, d, e. Ta thêm các nút này vào PQ theo thứ tự chi phí đến tăng dần, khi này PQ = {(p, 1), (d, 3), (e, 9)}

3. Tiếp theo ta lấy nút p có chi phí thấp nhất (1) ra khỏi PQ, và mở rộng nút p ta tìm được các nút (q, 1 + 15 = 16), thêm nút này vào PQ theo thứ tự chi phí tăng dần ta được PQ = {(d, 3), (e, 9), (q, 16)}

4. Tiếp theo ta lấy nút d có chi phí thấp nhất (3) ra khỏi PQ và mở rộng nút d, ta tìm được các nút (b, 3 + 1 = 4), (c, 3 + 8 = 11), thêm vào ta được PQ = {(b, 4), (e, 5), (c, 11), (q, 16)}

5. Tiếp theo ta lấy nút b có chi phí thấp nhất (4) ra khỏi PQ và mở rộng nút b ta được nút (a, 4 + 2 = 6), ta thêm vào PQ được PQ = {(e, 5), (a, 6), (c, 11), (q, 16)}

6. Tiếp theo ta lấy nút e có chi phí thấp nhất (5) ra khỏi PQ và mở rộng nút e ta được nút (h, 5 + 1 = 6), thêm vào ta được PQ = {(a, 6), (h, 6), (c, 11), (r, 14), (q, 16)}

7. Tiếp theo ta lấy nút a có chi phí thấp nhất (6) ra khỏi PQ và mở rộng nút a, do không có nút lân cận nên PQ chỉ bỏ nút a và giữ nguyên PQ = {(h, 6), (c, 11), (r, 14), (q, 16)}

8. Tiếp theo ta lấy nút h có chi phí thấp nhất (6) ra khỏi PQ và mở rộng nút h, các nút lân cận của h là p và q, do p đã xét rồi nên ta không tính vào, ta xem xét nút q xem có tìm được chi phí thấp hơn nút q hiện tại trong hàng đợi hay không, ta được (q, 6 + 4 = 10) <

(q, 16) nên ta bỏ nút q trong hàng đợi hiện tại và thế bằng (q, 10), ta được PQ = {(q, 10), (c, 11), (r, 14)}

9. Tiếp theo ta lấy nút q có chi phí thấp nhất (10) ra khỏi hàng đợi, mở rộng q ta được nút r. Tuy r đã có trong PQ nhưng ta vẫn xét để xem có tìm được nút r có chi phí thấp hơn hay không, ta có r mới (r, 10 + 3 = 13) < r(14) trong hàng đợi nên ta thế nút r trong hàng đợi bằng nút (r, 13) mới, ta được PQ = {(c, 11), (r, 13)}

10. Tiếp theo ta lấy nút c có chi phí thấp nhất (11) ra khỏi hàng đợi, mở rộng nút c ta không tìm được nút lân cận nào nên giữ nguyên PQ = {(r, 13)}

11. Tiếp theo ta lấy nút r có chi phí thấp nhất (13) trong PQ, mở rộng nút r ta được nút (f, 13 + 5 = 18) và thêm vào ta PQ ta được PQ = {(f, 18)}

12. Tiếp theo ta lấy nút f có chi phí (18) ra khỏi PQ, mở rộng nút f ta được nút Goal có chi phí là (Goal, 18 + 5 = 23), thêm Goal vào ta được PQ = {(Goal, 23)}

13. Lấy Goal ra, thuật toán kết thúc, đường đi ngắn nhất là: Start -> d -> e -> h -> q -> r –> f -> Goal với chi phí là 23.

2.3 Tìm kiếm theo chiều sâu (Depth-first search - DFS)

2.3.1 Ý tưởng

Thuật toán tìm kiếm theo chiều sâu luôn mở rộng nút sâu mãi theo một hướng, chỉ khi nào không mở rộng được nữa thì chuyển sang hướng khác. Thuật toán có thể cài đặt dựa trên thuật toán tìm kiếm tổng quát, sử dụng hàm DFS với tham số frontier là một hàng đợi LIFO (Last in First out) hay còn được gọi là Stack (ngăn xếp). Theo nguyên tắc này thì những nút được mở rộng sau (có độ sâu lớn hơn) sẽ được lấy ra trước. Một cách cài đặt khác, thường được sử dụng cho thuật toán tìm kiếm theo chiều sâu,

đó là sử dụng một hàm đệ quy gọi lại chính nó lần lượt cho mỗi nút trên cây. Có thể tóm tắt lại ý tưởng của thuật toán DFS như sau:

- Từ một đỉnh, ta đi qua sâu vào từng nhánh. Khi đã duyệt hết nhánh của đỉnh, lùi lại để duyệt đỉnh tiếp theo.
- Thuật toán dừng khi đi qua hết tất cả các đỉnh có thể đi qua.

2.3.2 Pseudo code (Mã giả)

Sử dụng Stack

```
function Depth_First_Search(problem) returns solution or failure
    stack <- create an empty stack
    initial_node <- Create_Node(Initial_State[problem])
    push initial_node onto stack

    while stack is not empty do
        node <- pop the top node from stack

        if Goal_Test[problem] on State[node] is true
            then return Solution(node)

        for each successor in Expand(problem) do
            push successor onto stack

    return failure
```

Sử dụng đệ qui

```
function Depth_First_Search(problem) returns solution or failure
    return Recursive_DFS(Create_Node(Initial_State[problem]), problem)

function Recursive_DFS(node, problem) returns solution or failure
    if Goal_Test[problem] on State[node] is true
        then return Solution(node)
    else for each successor in Expand(problem) do
        result <- Recursive_DFS(successor, problem)
        if result is not failure then return result
    return failure
```

2.3.3 Phân tích thuật toán

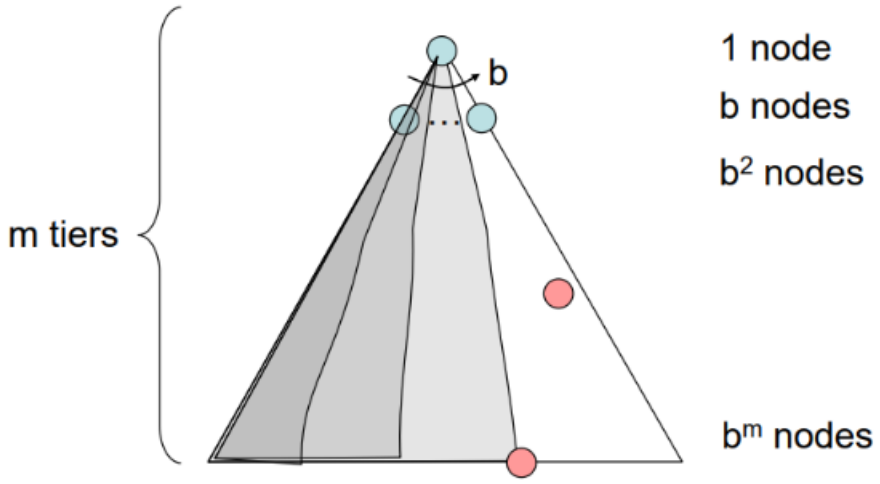
- **Tính đầy đủ**: DFS có thể đảm bảo tính đầy đủ trong một số trường hợp và điều kiện. Điều này có nghĩa là nếu có một đường đi tới đích, thì DFS sẽ tìm thấy nó. Tuy nhiên, trong trường hợp đồ thị có chu trình, DFS có thể bị kẹt trong vòng lặp vô hạn mà không bao giờ tìm thấy lời giải. Điều này có thể xảy ra nếu thuật toán không theo dõi các trạng thái đã thăm và có thể lặp đi lặp lại trên các chu trình. Để đảm bảo tính đầy đủ, cần phải kiểm tra các trạng thái đã thăm và tránh lặp lại.

- **Tính tối ưu**: DFS không đảm bảo tính tối ưu. Nghĩa là nó có thể tìm thấy lời giải, nhưng không nhất thiết phải là lời giải tối ưu. Điều này xảy ra vì DFS theo đường đi đầu tiên mà nó gặp, và có thể không kiểm tra xem có một đường đi ngắn hơn nằm ở một nhánh khác. Do đó, nó có thể dẫn đến tìm thấy một lời giải có chi phí cao hơn so với lời giải tối ưu. Để đảm bảo tính tối ưu, cần phải kiểm tra các lời giải tìm thấy và so sánh chúng để chọn ra lời giải tốt nhất.

- **Độ phức tạp thời gian**: Độ phức tạp thời gian của thuật toán DFS phụ thuộc vào cấu trúc của đồ thị hoặc cây tìm kiếm và có thể thay đổi tùy thuộc vào cách mà đồ thị được tổ chức.

  - Trong trường hợp tốt nhất, nếu nút đích là ngay bên cạnh nút gốc, DFS có thể tìm thấy lời giải ngay lập tức. Độ phức tạp thời gian trong trường hợp tốt nhất là O(1).
  - Trong trường hợp xấu nhất, nếu cây tìm kiếm là một đường dài với đích nằm ở cuối, DFS sẽ phải duyệt toàn bộ đường đi trước khi tìm thấy lời giải. Do đó, độ phức tạp thời gian trong trường hợp xấu nhất có thể là $O(b^m)$ (hình 2.2), trong đó b là hệ số phân nhánh trung bình, đại diện cho số lượng con của mỗi nút, m là độ sâu tối đa.

- **Độ phức tạp không gian**: Độ phức tạp không gian của DFS phụ thuộc vào việc lưu trữ các nút trong ngăn xếp (stack). Trong trường hợp xấu nhất, khi cây tìm kiếm là một đường dài nhất, DFS sẽ lưu



Hình 2.2: Độ phức tạp của DFS

trữ tất cả các nút trên đường đi trong ngăn xếp. Điều này có thể dẫn đến độ phức tạp không gian là $O(b*m+1)$ nút (+1 do tính luôn nút gốc), với b và m như đã nêu ở trên.

2.3.4 Minh hoạ mã giả

Với đồ thị như hình 2.1, ta có thuật toán DFS làm từng bước như sau, với S là ngăn xếp (Stack) và V là tập nút đã thăm (Visited), ở đây ta giả sử ưu tiên mở theo thứ tự alphabet:

- S = {Start}
- Lấy Start ra khỏi S, thêm vào V = {Start}, mở rộng nút Start ta được p, e, d và thêm vào S = {p, e, d}
- Lấy d ra khỏi S, thêm vào V = {Start, d}, mở rộng nút d ta được b, c và thêm vào S = {p, e, c, b}
- Lấy b ra khỏi S, thêm vào V = {Start, d, b}, mở rộng nút b ta được a, thêm vào S = {p, e, c, a}
- Lấy a ra khỏi S, thêm vào V = {Start, d, b, a}, mở rộng nút a ta thấy không có lân cận nên S = {p, e, c} được giữ nguyên

- Lấy c ra khỏi S, thêm vào V = {Start, d, b, a, c}, mở rộng nút c ta thấy không có lân cận nên S = {p, e} được giữ nguyên
- Lấy e ra khỏi S, thêm vào V = {Start, d, b, a, c, e}, mở rộng nút e ta được các nút lân cận là h, r và thêm vào S = {p, h, r}
- Lấy r ra khỏi S, thêm vào V = {Start, d, b, a, c, e, r}, mở rộng nút r có các nút lân cận là f, thêm vào S = {p, h, f}
- Lấy f ra khỏi S, thêm vào V = {Start, d, b, a, c, e, r, f}, f có các nút lân cận là Goal, thêm vào S = {p, h, Goal}
- Lấy Goal ra, kết thúc thuật toán.

Chương 3

Các thuật toán tìm kiếm có thông tin

3.1 Tìm kiếm AStar (A*)

3.1.1 Ý tưởng

Xét bài toán tìm đường - bài toán mà A* thường được dùng để giải. A* xây dựng tăng dần tất cả các tuyến đường từ điểm xuất phát cho tới khi nó tìm thấy một đường đi chạm tới đích. Tuy nhiên, cũng như tất cả các thuật toán tìm kiếm có thông tin (informed search), nó chỉ xây dựng các tuyến đường "có vẻ" dẫn về phía đích.

Để biết những tuyến đường nào có khả năng sẽ dẫn tới đích, A* sử dụng một "đánh giá heuristic" về khoảng cách từ điểm bất kỳ cho trước tới đích. Trong trường hợp tìm đường đi, đánh giá này có thể là khoảng cách đường chim bay - một đánh giá xấp xỉ thường dùng cho khoảng cách của đường giao thông.

Điểm khác biệt của A* đối với tìm kiếm theo lựa chọn tốt nhất là nó còn tính đến khoảng cách đã đi qua. Điều đó làm cho A* "đầy đủ" và "tối ưu", nghĩa là, A* sẽ luôn luôn tìm thấy đường đi ngắn nhất nếu tồn tại một đường đi như thế. A* không đảm bảo sẽ chạy nhanh hơn các thuật toán tìm kiếm đơn giản hơn. Trong một môi trường dạng mê cung, cách duy nhất để đến đích có thể là trước hết phải đi về phía xa đích và cuối cùng mới quay lại. Trong trường hợp đó, việc thử các nút theo thứ tự "gần đích hơn thì được thử trước" có thể gây tốn thời gian.

Rõ hơn vấn đề trên, thuật toán A* đánh giá nút dựa trên chi phí đi từ nút gốc đến nút đó - $g(n)$, cộng với chi phí từ nút đó đến đích - $h(n)$:

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

Do giá trị $g(n)$ cho biết chi phí đường đi từ nút gốc đến nút n và $h(n)$ là ước lượng chi phí đường đi ngắn nhất từ nút n đến đích nên ta có $f(n) =$ ước lượng chi phí của lời giải "tốt nhất" qua n.

Do đó, nếu ta định tìm lời giải tốt nhất thì cách dễ thấy nhất là lấy nút có giá trị: $g(n) + h(n)$ nhỏ nhất. Hơn nữa, nếu hàm heuristic $h(n)$ thoả một số điều kiện nhất định thì tìm kiếm A* sẽ đầy đủ và tối ưu.

3.1.2 Pseudo code (Mã giả)

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from
// node n.

function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather
    // than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the
    // cheapest path from the start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to
    n // currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

```

```
    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our
    // current best guess as to
    // how cheap a path could be from start to finish if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(Log(N)) time if openSet is a min-
        // heap or a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current
    to // neighbor
            // tentative_gScore is the distance from start to the neighbor
            // through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one.
                // Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

3.1.3 Phân tích thuật toán

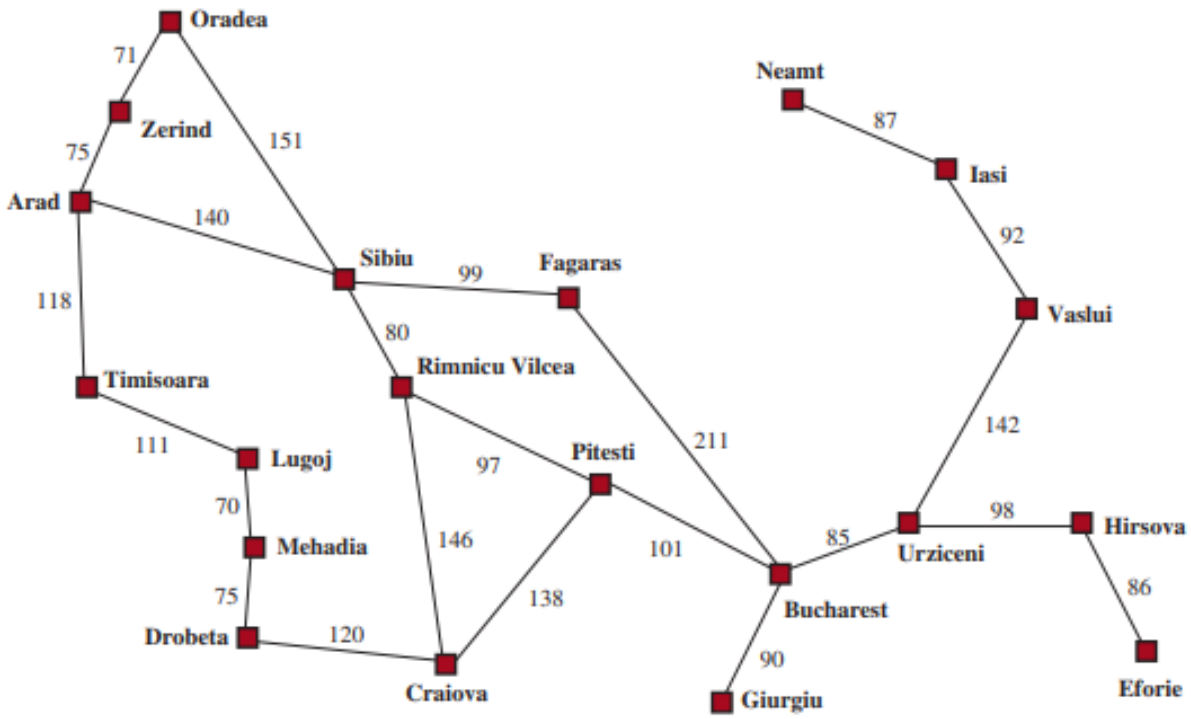
- **Độ phức tạp thời gian**:
  - Độ phức tạp thời gian của thuật toán A* phụ thuộc vào hàm heuristic (được ký hiệu là h). Trong trường hợp xấu nhất, khi không gian tìm kiếm không giới hạn, số lượng nút được mở rộng theo cấp số nhân dựa trên độ sâu của lời giải (đường đi ngắn nhất) d và hệ số phân nhánh trung bình b: $O(b^d)$.
  - Điều này giả định rằng trạng thái mục tiêu tồn tại và có thể truy cập từ trạng thái ban đầu. Nếu không có đường đi, không

gian trạng thái sẽ vô hạn và thuật toán A* sẽ lặp vô hạn.

  - Độ phức tạp thời gian của A* là một đa thức bởi vì không gian tìm kiếm tạo thành một cây với một trạng thái mục tiêu duy nhất.
  - Hàm heuristic h phải thỏa mãn điều kiện sau: $|h(x) - h^*(x)| = O(log(h^*(x))$. Trong đó, h^* là chi phí chính xác để đi từ trạng thái x đến trạng thái mục tiêu. Điều này có nghĩa rằng sai số của heuristic h không tăng nhanh hơn logarit của "heuristic hoàn hảo" h^* trả về khoảng cách thực tế từ trạng thái x đến trạng thái mục tiêu.

- Hàm heuristic:
  - Hàm heuristic đóng vai trò quan trọng trong hiệu suất thực tế của thuật toán A*.
  - Hàm heuristic tốt cho phép A* loại bỏ một số lượng lớn các nút không cần thiết trong quá trình tìm kiếm.
  - Chất lượng của heuristic có thể được đánh giá bằng hệ số phân nhánh trung bình hiệu quả b^*, mà có thể được xác định thông qua thực nghiệm trên một trường hợp vấn đề cụ thể. Hệ số này thỏa mãn phương trình sau: $N + 1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d$. Trong đó, N là số lượng nút được tạo bởi việc mở rộng và d là độ sâu của lời giải.

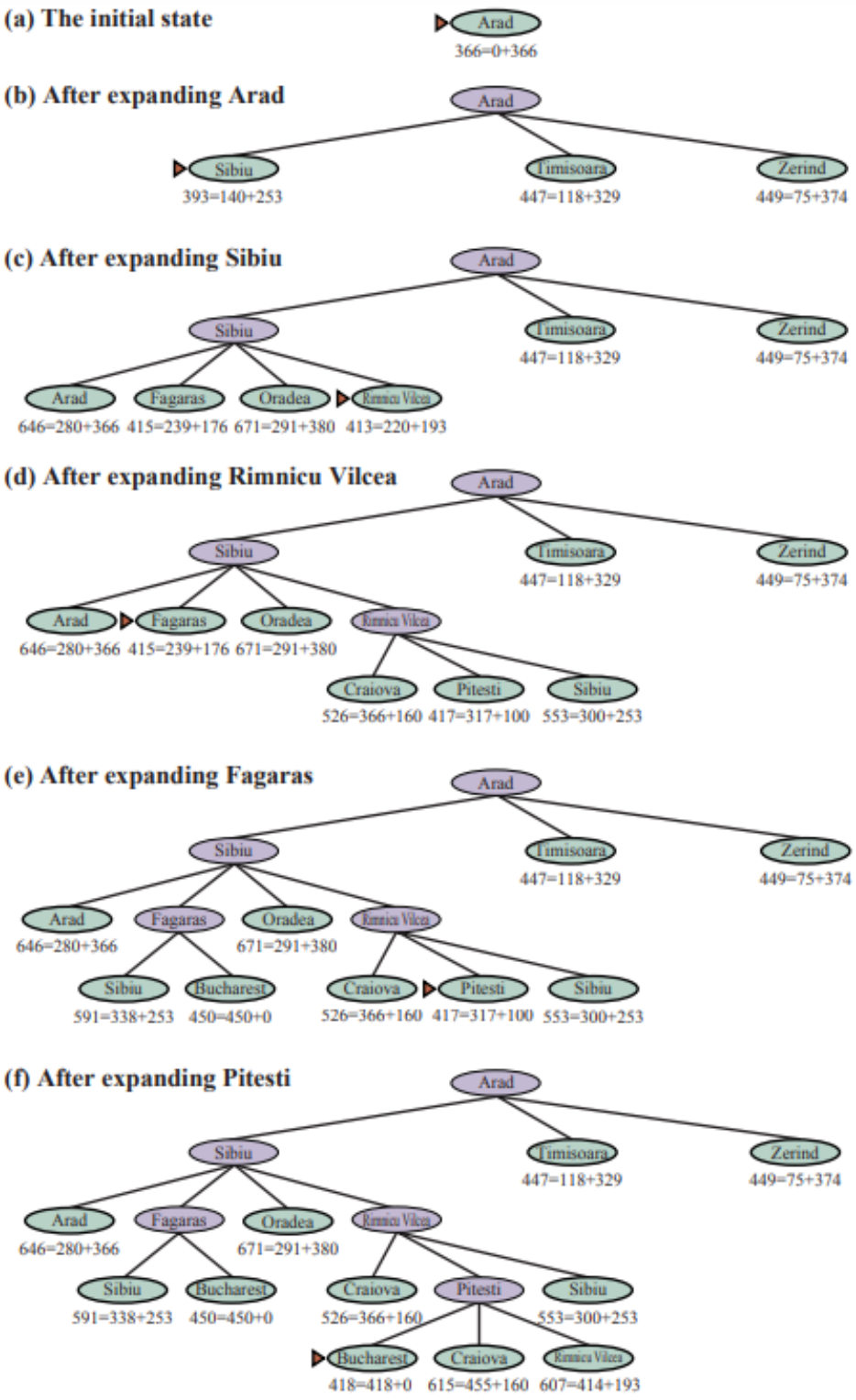
- **Độ phức tạp không gian**:
  - Độ phức tạp không gian của A* tương tự như của tất cả các thuật toán tìm kiếm đồ thị khác, bởi vì nó duy trì tất cả các nút được tạo trong bộ nhớ.
  - Điều này là một trong nhược điểm lớn nhất của thuật toán A*, và đã thúc đẩy sự phát triển của các thuật toán tìm kiếm heuristic giới hạn bộ nhớ như Iterative Deepening A* và SMA*.



Hình 3.1: Bản đồ đường giữa các thành phố của Romania, với khoảng cách đường đi tính bằng dặm.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Hình 3.2: Bảng heuristic đánh giá khoảng cách đường thẳng từ từng nút đến nút đích



Hình 3.3: Các bước trong thuật toán tìm kiếm A* để đi đến đích Bucharest. Các nút được đánh nhãn với $f = g+h$, trong đó h là giá trị đường đi thẳng từ từng nút đến Bucharest

3.1.4 Minh hoạ mã giả

Xét đồ thị minh hoạ cho thuật toán tìm kiếm A* như hình 3.1, có bảng heuristic cho từng nút (thành phố) như bảng 3.2, ta có các bước thực hiện như trong hình 3.3, dưới đây là giải thích cách từng bước làm được thực hiện:

- Ở bước 1, ta mở nút Arad, chi phí để đến Arad là 0, tức $g(Arad) = 0$ và chi phí heuristic của Arad là 366, tức $h(Arad) = 366$, từ đó ta tính được $f(Arad) = g(Arad) + h(Arad) = 0 + 366 = 366$
- Ở bước 2, từ nút Arad sau khi mở ta tìm được 3 lân cận là Sibiu, Timisoara và Zerind, ta xét hàm f của từng nút như bên dưới:
  - Xét Sibiu, ta có $g(Sibiu) = g(Arad) + d(Arad, Sibiu) = 0 + 140 = 140$ và $h(Sibiu) = 253$ nên $f(Sibiu) = g(Sibiu) + h(Sibiu) = 140 + 253 = 393$.
  - Xét Timisoara, ta có $g(Timisoara) = g(Arad) + d(Arad, Timisoara) = 0 + 118 = 118$ và $h(Timisoara) = 329$ nên $f(Timisoara) = g(Timisoara) + h(Timisoara) = 118 + 329 = 447$
  - Xét Zerind, ta có $g(Zerind) = g(Arad) + d(Arad, Zerind) = 0 + 75 = 75$ và $h(Zerind) = 374$ nên $f(Zerind) = g(Zerind) + h(Zerind) = 75 + 374 = 449$
- Ở bước 3, ta lấy nút ở bước 2 có chi phí f nhỏ nhất để mở là nút Sibiu $f(Sibiu) = 393$ bé hơn $f(Timisoara)$ và $f(Zerind)$, sau khi mở Sibiu ta tìm được các nút lân cận là Arad, Fagaras, Oradea và Rimnicu Vilcea, ta xét hàm f của từng nút như bên dưới:
  - Xét Arad, ta có $g(Arad) = g(Sibiu) + d(Sibiu, Arad) = 140 + 140 = 280$ và $h(Arad) = 366$ nên $f(Arad) = g(Arad) + h(Arad) = 280 + 366 = 646$
  - Xét Fagaras, ta có $g(Fagaras) = g(Sibiu) + d(Sibiu, Fagaras) = 140 + 99 = 239$ và $h(Fagaras) = 176$ nên $f(Fagaras) = g(Fagaras) + h(Fagaras) = 239 + 176 = 415$

- Xét Oradea, ta có $g(Oradea) = g(Sibiu) + d(Sibiu, Oradea) = 140+151 = 291$ và $h(Oradea) = 380$ nên $f(Oradea) = g(Oradea)+h(Oradea) = 291 + 380 = 671$
- Xét Rimnicu Vilcea, ta có $g(RimnicuVilcea) = g(Sibiu) + d(Sibiu, RimnicuVilcea) = 140+80 = 220$ và $h(RimnicuVilcea) = 193$ nên $f(RimnicuVilcea) = g(RimnicuVilcea)+h(RimnicuVilcea) = 220 + 193 = 413$

- Ở bước 4, ta lấy nút ở bước 3 có chi phí f nhỏ nhất để mở là nút Rimnicu Vilcea, sau khi mở ta tìm được các nút lân cận là Craiova, Pitesti, Sibiu, ta xét hàm f của từng nút như bên dưới:
  - Xét Craiova, ta có $g(Craiova) = g(RimnicuVilcea)+d(RimnicuVilcea, C$ $220 + 146 = 366$ và $h(Craiova) = 160$ nên $f(Craiova) = g(Craiova) + h(Craiova) = 366 + 160 = 526$
  - Xét Pitesti, ta có $g(Pitesti) = g(RimnicuVilcea)+d(RimnicuVilcea, Pit$ $220+97 = 317$ và $h(Pitesti) = 100$ nên $f(Pitesti) = g(Pitesti)+h(Pitesti) = 317 + 100 = 417$
  - Sibiu, ta có $g(Sibiu) = g(RimnicuVilcea)+d(RimnicuVilcea, Sibiu) = 220 + 80 = 300$ và $h(Sibiu) = 253$ nên $f(Sibiu) = g(Sibiu) + h(Sibiu) = 300 + 253 = 553$

- Ở bước 5, ta lấy nút ở bước 4 có chi phí f nhỏ nhất để mở là nút Fagaras, sau khi mở ta tìm được các nút lân cận là Sibiu và Bucharest, ta xét hàm f của từng nút như bên dưới:
  - Xét Sibiu, ta có $g(Sibiu) = g(Fagaras) + d(Fagaras, Sibiu) = 239 + 99 = 338$ và $h(Sibiu) = 253$ nên $f(Sibiu) = g(Sibiu) + h(Sibiu) = 338 + 253 = 591$
  - Bucharest, ta có $g(Bucharest) = g(Fagaras)+d(Fagaras, Bucharest) = 239 + 211 = 450$ và $h(Bucharest) = 0$ nên $f(Bucharest) = g(Bucharest) + h(Bucharest) = 450 + 0 = 450$

- Ở bước 6, ta lấy nút ở bước 5 có chi phí f nhỏ nhất để mở là nút Pitesti, sau khi mở ta tìm được các nút lân cận là Bucharest, Craiova,

Rimnicu Vilcea, ta xét hàm f của từng nút như bên dưới:

- Xét Bucharest, ta có $g(Bucharest) = g(Pitesti)+d(Pitesti, Bucharest) = 317 + 101 = 418$ và $h(Bucharest) = 0$ nên $f(Bucharest) = g(Bucharest) + h(Bucharest) = 418 + 0 = 418$
- Xét Craiova, ta có $g(Craiova) = g(Pitesti)+d(Pitesti, Craiova) = 317 + 138 = 455$ và $h(Craiova) = 160$ nên $f(Craiova) = g(Craiova) + h(Craiova) = 455 + 160 = 615$
- Xét Rimnicu Vilcea, ta có $g(RimnicuVilcea) = g(Pitesti) + d(Pitesti, RimnicuVilcea) = 317+97 = 414$ và $h(RimnicuVilcea) = 193$ nên $f(RimnicuVilcea) = g(RimnicuVilcea)+h(RimnicuVilcea) = 414 + 193 = 607$

- Ở bước cuối cùng, ta chọn ra nút ở bước 6 có chi phí f nhỏ nhất để mở là nút Bucharest, và nút Bicharest cũng chính là thành phố đích mà chúng ta cần tới nên khi lấy được nó ra khỏi hàng đợi thì cũng chính là tìm được đường đi bằng thuật toán A*, do đó chúng ta kết thúc thuật toán thành công.

Chương 4

So sánh các thuật toán tìm kiếm

4.1 So sánh 4 thuật toán DFS, BFS, UCS, AStar

Thuật toán	**DFS**	**BFS**	**UCS**	**AStar**
Tính đầy đủ	Có	Có	Có	Có
Tính tối ưu	Không	Có	Có	Có
Độ phức tạp	$O(b^m)$	$O(b^{d+1})$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b^d)$

Bảng 4.1: Bảng so sánh 4 thuật toán DFS, BFS, UCS, AStar

4.2 So sánh 3 thuật toán UCS, Greedy và Astar

Bảng 4.2: Bảng so sánh 3 thuật toán UCS, Greedy và AStar

Tính chất so sánh	**UCS**	**Greedy**	**AStar**
Định nghĩa	UCS là một thuật toán tìm kiếm Uninformed, dựa vào việc duyệt các nút dựa trên chi phí tích lũy	Greedy Search là một thuật toán tìm kiếm Informed, dựa vào ước đo heuristic để đưa ra quyết định	A* là một thuật toán tìm kiếm Informed kết hợp cả chi phí đồng đều và heuristic

Điểm mạnh	Tìm kiếm đường dẫn ngắn nhất với chi phí thấp nhất	Tìm kiếm nhanh bằng cách chọn nút mà heuristic cho là gần mục tiêu nhất	Kết hợp thông tin về chi phí tích luỹ đã đi qua (hàm g) và một ước đo heuristic (hàm h) để dự đoán tổng chi phí $f = g + h$, giúp tìm kiếm đường dẫn tối ưu nhanh hơn
Điểm yếu	Không có kiến thức về mục tiêu, không biết nên đi về đâu, có thể dẫn đến mất thời gian khi có nhiều lựa chọn	Không đảm bảo tìm được đường dẫn tối ưu, có thể dẫn đến đi vào các "ngõ cụt" khi heuristic không hoàn hảo	Tuỳ thuộc vào chất lượng của heuristic, nếu heuristic không tốt, A* có thể bị chệch và tốn thời gian

Độ phức tạp	Độ phức tạp trong trường hợp tồi nhất có thể là $O(b^{1+d})$ với b là số nút con trung bình mỗi nút cha, d là độ sâu của nút mục tiêu	Độ phức tạp trong trường hợp tồi nhất là $O(b^m)$, trong đó m là số nút tối đa trong hàng đợi ưu tiên	Độ phức tạp trong trường hợp tồi nhất cũng là $O(b^d)$ nhưng nó có khả năng giảm thiểu tối đa số nút tối nghiệm được duyệt
Ứng dụng trong thực tế	Thường được sử dụng khi cần tìm kiếm đường dẫn ngắn nhất trong các trường hợp không có thông tin trước về vị trí mục tiêu, ví dụ trong việc tìm đường đi giữa hai điểm trên bản đồ	Sử dụng khi có heuristic tốt và muốn tìm kiếm gần mục tiêu nhanh chóng. Ví dụ, trong hệ thống hướng dẫn lái xe để tìm đường đi nhanh nhất tới mục tiêu, mặc dù không đảm bảo là tối ưu.	Phù hợp cho hầu hết các bài toán tìm kiếm trong thực tế. Nó kết hợp chi phí và thông tin heuristic, giúp tìm kiếm nhanh và tối ưu. Ví dụ, trong lập lịch công việc, trò chơi video, và quy hoạch địa lý.

4.3 So sánh 2 thuật toán UCS và Dijkstra

Bảng 4.3: So sánh UCS và Dijkstra

Tính chất so sánh	**UCS**	**Dijkstra**
Định nghĩa	UCS là một thuật toán tìm kiếm Uninformed, dựa vào việc duyệt các nút dựa trên chi phí tích lũy, nhưng có một lưu ý là **UCS chỉ tìm đường đi ngắn nhất giữa 2 nút**	Dijkstra là một thuật toán tìm kiếm thông tin hoàn toàn (thông tin mọi chi phí ngắn nhất) dựa vào việc duyệt các nút dựa trên chi phí tích lũy, **Dijkstra tìm kiếm đường đi ngắn nhất từ một nút đến tất cả các nút trong đồ thị**
Điểm mạnh	Tìm kiếm đường dẫn ngắn nhất với chi phí thấp nhất	Tìm kiếm đường dẫn ngắn nhất với chi phí thấp nhất
Điểm yếu	Không có kiến thức về mục tiêu, không biết nên đi về đâu, có thể dẫn đến mất thời gian khi có nhiều lựa chọn	Không xử lý được đường dẫn âm (negative-weight edges), có thể dẫn đến kết quả sai lệch
Độ phức tạp	Độ phức tạp trong trường hợp tồi nhất có thể là $O(b^{1+d})$ với b là số nút con trung bình mỗi nút cha, d là độ sâu của nút mục tiêu	Độ phức tạp trong trường hợp tồi nhất cũng là $O(b^{1+d})$ với b là số nút con trung bình mỗi nút cha, d là độ sâu của nút mục tiêu
Ứng dụng trong thực tế	Thường được sử dụng khi cần tìm kiếm đường dẫn ngắn nhất trong các trường hợp không có thông tin trước về vị trí mục tiêu, ví dụ trong việc tìm đường đi giữa hai điểm trên bản đồ	Thường được sử dụng trong các bài toán tìm đường đi trong đồ thị có trọng số không âm, ví dụ trong hệ thống định tuyến mạng (routing)

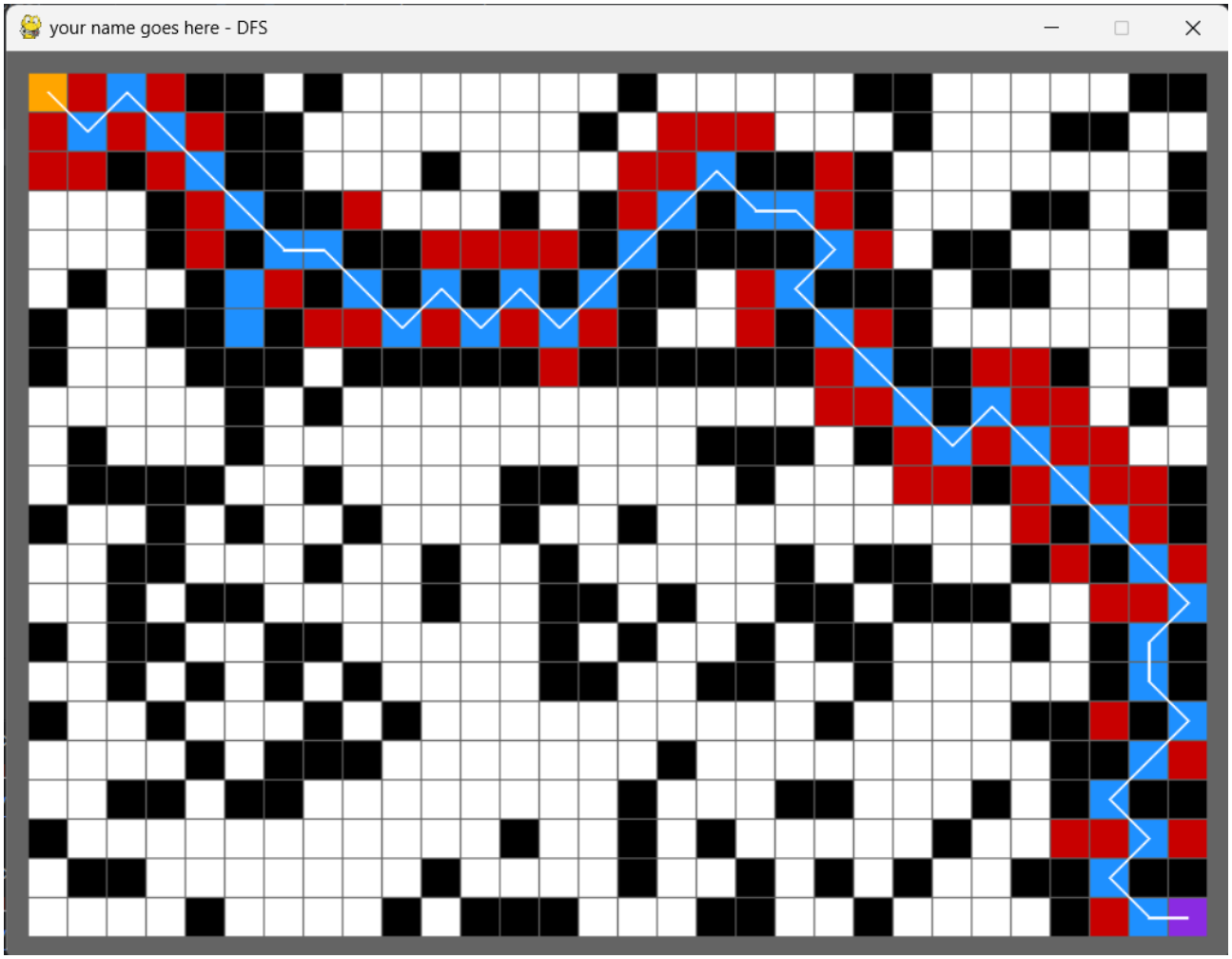
Chương 5

Cài đặt (Implementation)

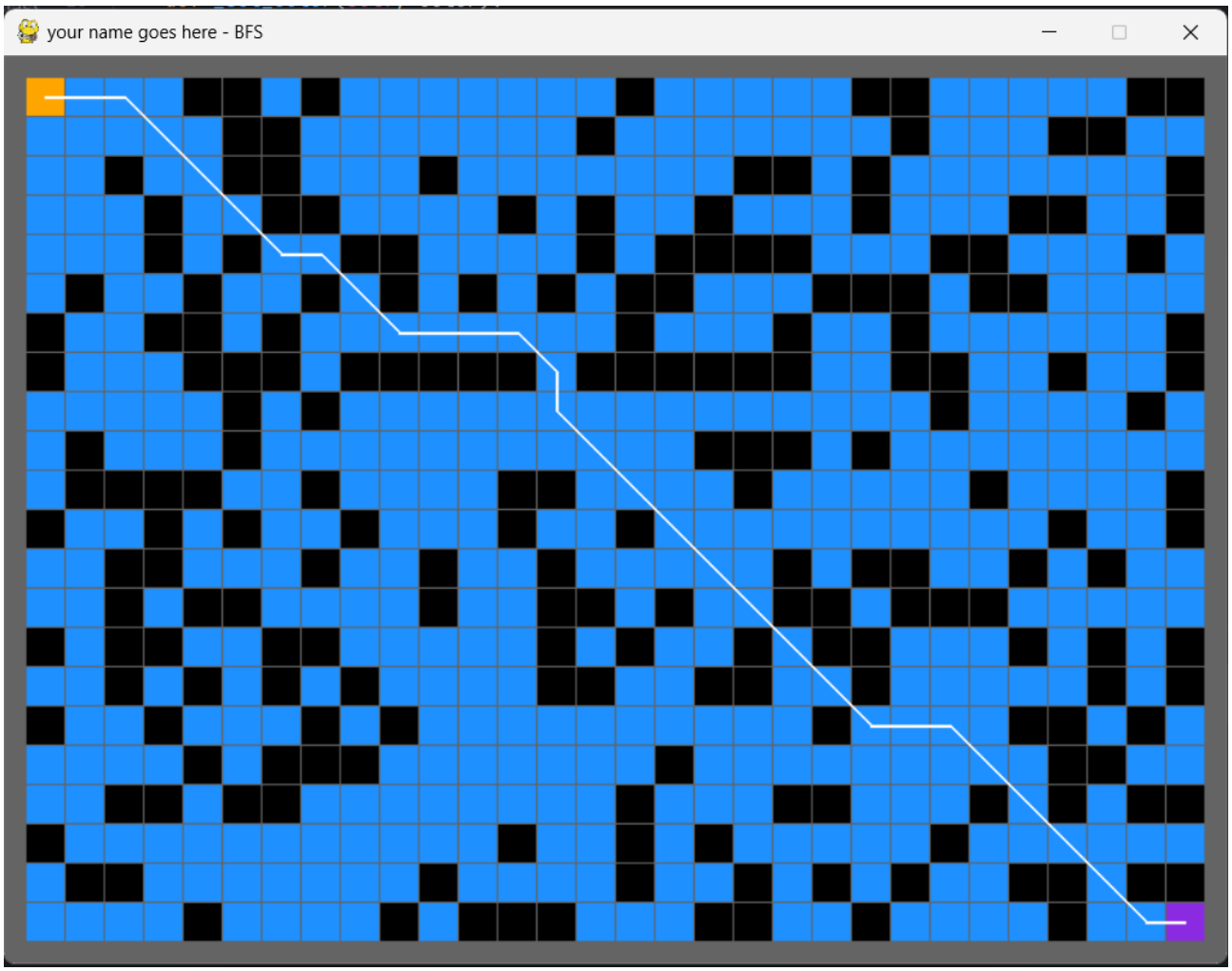
5.1 DFS

Sau khi code thì thuật toán DFS duyệt mê cung sẽ cho kết quả như hình 5.1. Sau đây là giải thích sơ lược về code DFS:

- Khởi tạo:
  - open_set: Stack ban đầu chứa ID của nút bắt đầu (nút hiện tại cần được xem xét), trong code thì thay vì dùng Stack thì ta dùng list của Python, nhưng cách thêm và lấy phần tử tương tự Stack
  - closed_set: Danh sách ban đầu rỗng, dùng để lưu trữ các nút đã được duyệt
  - father: Mảng ban đầu chứa -1, dùng để theo dõi nút cha của mỗi nút trong quá trình tìm kiếm
- Lặp: Lặp cho đến khi open_set trống (không còn nút nào cần xem xét). Lấy nút hiện tại từ open_set (nút trên cùng ngăn xếp). Kiểm tra nút đó và tô màu vàng nếu không phải nút Start và Goal.
- Kiểm tra xem nút hiện tại đã được duyệt (nằm trong closed_set) chưa, nếu chưa thì nút này sẽ được thêm vào closed_set và tô màu xanh.
- Nếu nút hiện tại là Goal, thuật toán sẽ dừng. Xây dựng đường đi từ nút Goal đến nút Start bằng mảng father. Vẽ đường đi trên màn hình.



Hình 5.1: DFS Implement



Hình 5.2: BFS Implement

- Nếu nút hiện tại không phải Goal thì thuật toán vẫn chạy tiếp, duyệt qua từng phần tử con (nút lân cận) của nút hiện tại, nếu nút con nào chưa nằm trong open_set thì thêm vào và tô màu đỏ.

5.2 BFS

Sau khi code thì thuật toán BFS sẽ như hình 5.2, sau đây là giải thích sơ lược về code BFS:

- Khởi tạo:
  - open_set: Dãy ban đầu chứa ID của nút, hoạt động như một Queue nhưng dùng list trong Python để tạo, thêm sẵn vào Queue nút bắt đầu

  - closet_set: Danh sách ban đầu rỗng, dùng để lưu trữ các nút đã được duyệt
  - father: Mảng ban đầu chứa -1, dùng để theo dõi nút cha của mỗi nút trong quá trình tìm kiếm

- Lặp: Lặp cho đến khi open_set trống (không còn nút nào cần xem xét). Lấy nút hiện tại từ open_set và thêm vào closed_set, nếu nút hiện tại là Goal thì thuật toán sẽ dừng và trả về đường dẫn từ Start đến Goal dựa vào mảng Father

- Nếu không phải Goal thì thuật toán tiếp tục chạy bằng cách xét các nút con (lân cận) của nút hiện tại. Duyệt qua từng nút con và kiểm tra xem nút con đã được duyệt và nằm trong tập open hay chưa, nếu chưa thì thêm vào open_set và tô màu đỏ

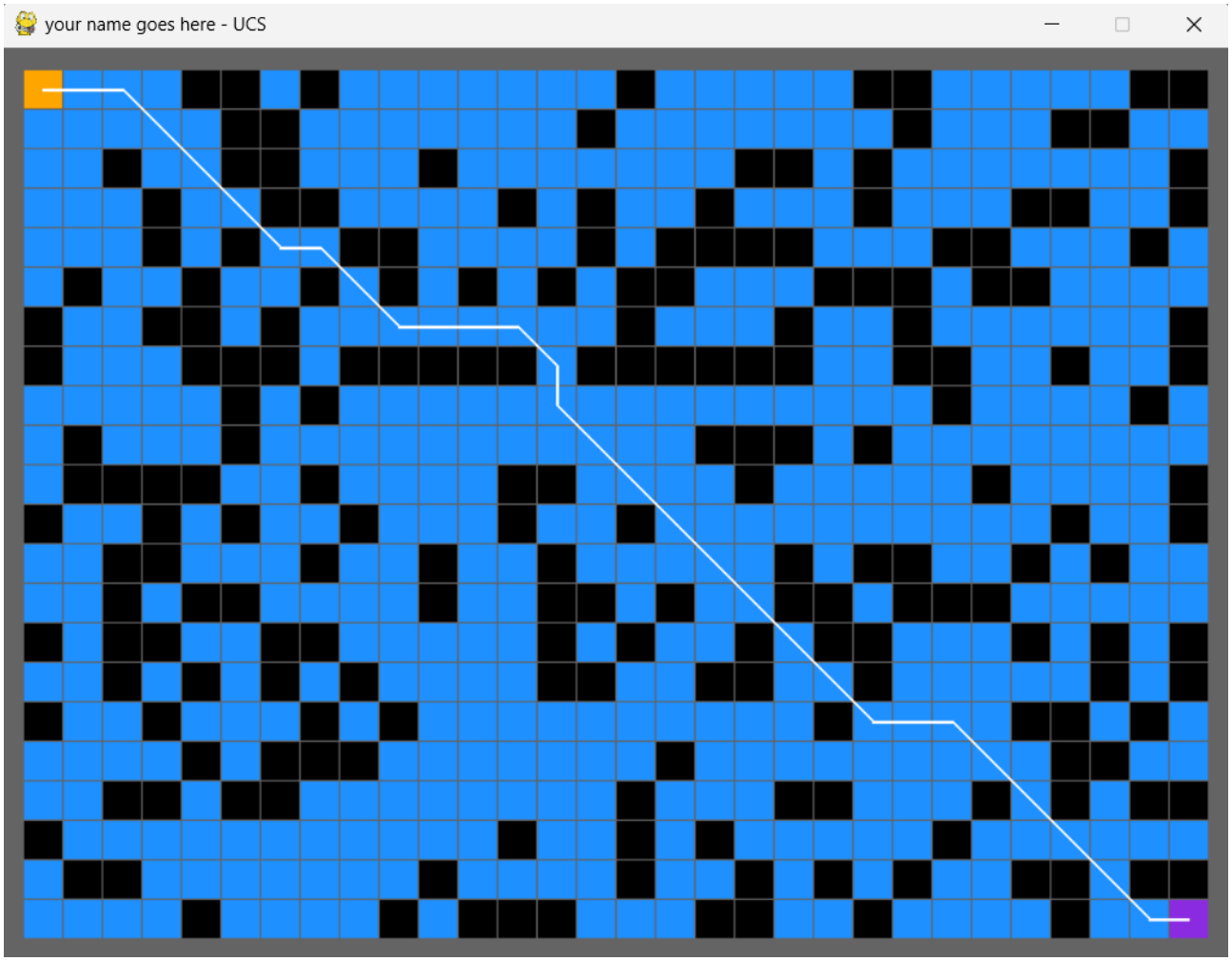
5.3 UCS

Sau khi code thì thuật toán UCS sẽ như hình 5.3, sau đây là giải thích sơ lược về code UCS

- Khởi tạo:
  - open_set: Hàng đợi ban đầu với giá trị ưu tiên (cost) bằng 0 cho nút bắt đầu
  - closed_set: Danh sách chứa các nút đã được duyệt
  - father: Mảng chứa ID của nút cha ứng với mỗi nút trong quá trình tìm kiếm
  - node_costs: Mảng chứa chi phí tới mỗi nút, ban đầu cho giá trị 100000 hoặc vô cùng

- Lặp đến khi open_set bằng rỗng

- Tìm nút có cost thấp nhất làm current_nodes. Lấy nút đó ra khỏi open_set và thêm vào closed_set



Hình 5.3: UCS Implement

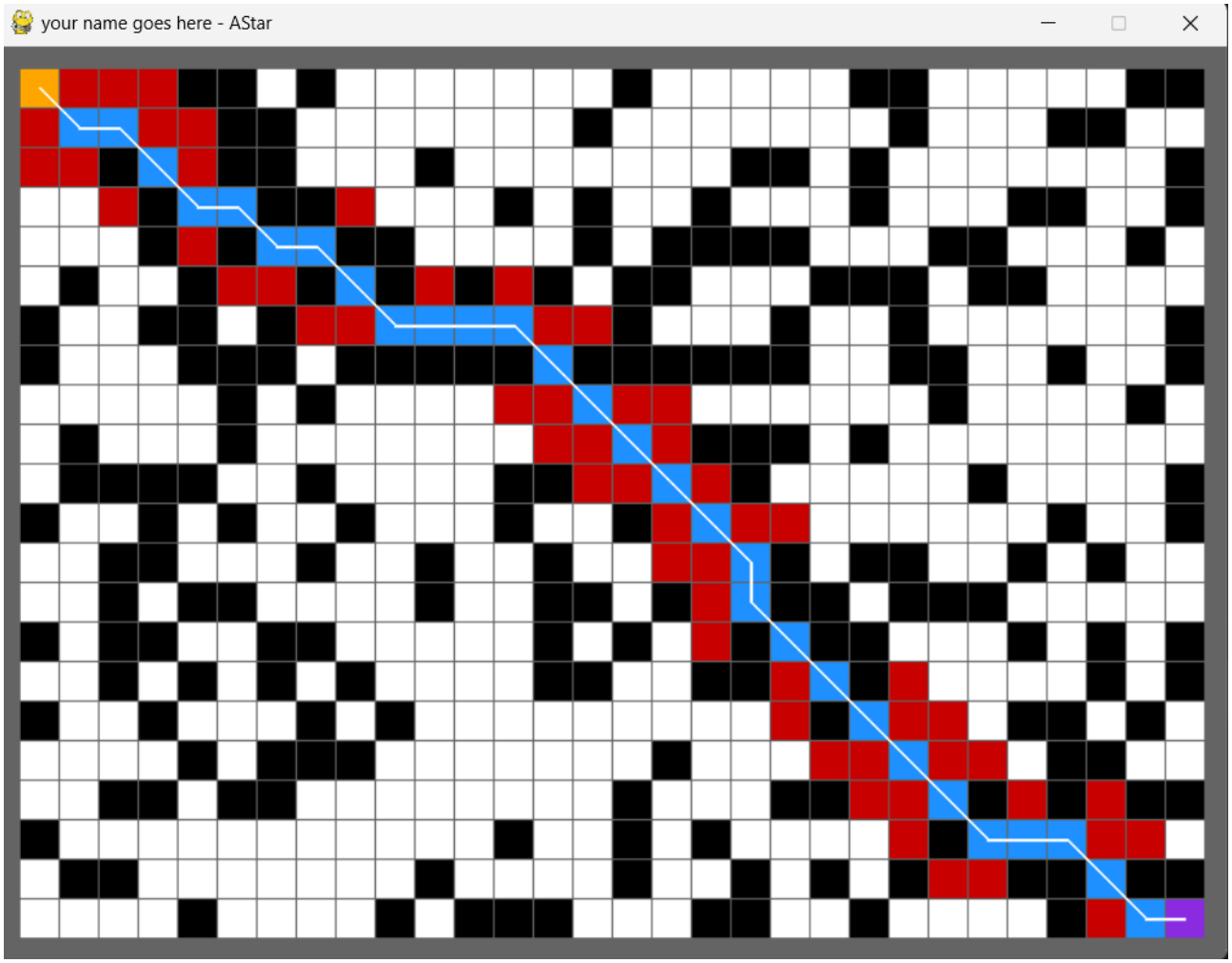
- Nếu nút hiện tại là nút kết thúc (goal node) thì kết thúc thuật toán, sau đó xây dựng đường đi từ nút kết thúc đến nút bắt đầu bằng cách theo dõi nút cha trong mảng father.
- Nếu nút hiện tại không phải Goal thì thuật toán tiếp tục bằng cách duyệt qua từng nút con (lân cận) của nút hiện tại:
  - Kiểm tra xem nút con có được duyệt hay chưa (không nằm trong closed_set)
  - Tính toán chi phí từ nút hiện tại tới nút con, trong code thì mặc định là 1, hoặc nếu có chi phí trong đồ thị thì xài chi phí đó, hoặc random (bỏ comment trong code để random)
  - Kiểm tra xem chi phí đến nút con có thấp hơn chi phí hiện tại của nút con hay không
  - Nếu chi phí đến nút con thấp hơn, cập nhật chi phí của nút con và nút cha, sau đó thêm nút con vào open_set để xem xét tiếp.

:

5.4 AStar

Sau khi code hoàn tất thì thuật toán AStar sẽ như hình 5.4, sau đây là giải thích sơ lược về code AStar:

- Khởi tạo:
  - open_set: Duyệt hàng đợi ban đầu với giá trị ưu tiên (f_cost) bằng 0 cho nút bắt đầu. Xem như list trong python là một hàng đợi ưu tiên, về cách lấy nút trong hàng đợi ưu tiên thì lấy theo thứ tự gặp nút nào đầu tiên lấy nút đó trước nếu có nhiều nút có cùng chi phí. Còn không thì vẫn lấy nút có chi phí thấp nhất trước (độ ưu tiên cao hơn)
  - closed_set: Danh sách chứa các nút đã được duyệt (ban đầu rỗng).



Hình 5.4: AStar Implement

  - father: Mảng chứa ID của nút cha của mỗi nút trong quá trình tìm kiếm.
  - g_costs: Mảng chứa chi phí thực sự để đi từ nút bắt đầu tới mỗi nút, ban đầu có giá trị lớn.
  - h_costs: Mảng chứa ước tính chi phí h(n) từ mỗi nút tới nút kết thúc, ban đầu có giá trị lớn.

- Lặp cho đến khi open_set trống (không còn nút nào cần xem xét)
- Lựa chọn nút có f_cost thấp nhất (trong open_set) để xem xét làm current_node tiếp theo
- Xem xét nút hiện tại và thêm vào closed_set. Nếu nút hiện tại là nút kết thúc (Goal) thì kết thúc thuật toán và trả về đường đi dựa vào mảng cha father
- Nếu nút hiện tại không phải Goal thì thuật toán vẫn tiếp tục bằng cách duyệt tiếp các nút con (lân cận) của nút hiện tại:
  - Kiểm tra xem nút con đã được duyệt (nằm trong closed_set) chưa. Nếu đã được duyệt, tiến hành với nút con tiếp theo.
  - Tính toán chi phí thực sự từ nút bắt đầu đến nút con (g_cost).
  - Tính toán ước tính chi phí h(n) từ nút con tới nút kết thúc (h_cost).
  - Tính tổng chi phí f_cost = g_cost + h_cost.
  - So sánh f_cost của nút con với f_cost trong open_set (nếu nút con đã có trong open_set).
  - Nếu nút con chưa có trong open_set hoặc f_cost mới thấp hơn f_cost cũ, cập nhật thông tin nút con và thêm nó vào open_set để xem xét tiếp.

Tài liệu tham khảo

[7, 6, 4, 8, 5]

References

[4] Felner, Ariel. "Position Paper: Dijkstra's Algorithm versus Uniform Cost Searchor a Case Against Dijkstra's Algorithm". In: *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS* (2011).

[5] *Giải thuật tìm kiếm A**. URL: `https://voer.edu.vn/m/giai-thuat-tim-kiem-a/d169b9dd` (visited on 10/21/2023).

[6] Lê Hoài Bắc, Tô Hoài Việt. *Cơ sở Trí tuệ Nhân tạo*. Nhà xuất bản Khoa học và Kỹ thuật, 2014.

[7] Stuart J. Russell, Peter Norvig. *Artificial Intelligence A Modern Approach Fourth Edition Global Edition*. Pearson Education, 2021.

[8] Wikipedia. *A* search algorithm*. URL: `https://en.wikipedia.org/wiki/A*_search_algorithm` (visited on 10/21/2023).