TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Báo cáo Đồ án 1: Search in Graph

Môn học: Cơ sở trí tuệ nhân tạo - CSC14003

Sinh viên thực hiện:
Trương Thành Nhân (21120105)

Giảng viên hướng dẫn:
Thầy Nguyễn Bảo Long

TP. Hồ Chí Minh

Ngày 30 tháng 10 năm 2023

Mục lục

1 Thông tin chung

1.1 Thông tin sinh viên

- Họ tên: Trương Thành Nhân
- MSSV: 21120105
- Email: 21120105@student.hcmus.edu.vn

1.2 Thông tin đồ án

- **Tên đồ án:** Search in Graph (*DFS, BFS, UCS, AStar.*)
- **Mục tiêu đồ án:** Thông qua đồ án, sinh viên có thể hiểu được về "framework" của những thuật toán tìm kiếm trong đồ thị. Các thuật toán được đề cập đến trong đồ án bao gồm: DFS, BFS, UCS, AStar.

1.3 Đánh giá mức độ hoàn thành

STT	Công việc	Đánh giá
1	Study and present search algorithms in graph	4
2	Compare the algorithms with each other	2
3	Implement the algorithms	2.5
1	Research, present, compare and implement other search algorithms	1
	Tổng	9.5

Bảng 1: Đánh giá mức độ hoàn thành

2 Bài toán tìm kiếm và một số giải thuật

2.1 Giới thiệu về bài toán tìm kiếm

2.1.1 Vấn đề tìm kiếm (search problem)

Vấn đề tìm kiếm (search problem) là những bài toán liên quan đến việc tìm kiếm các giải pháp, thông tin về vấn đề được đặt ra trong một không gian hoặc tập hợp dữ liệu xác định. Khái niệm này xuất hiện trong nhiều lĩnh vực khác nhau như khoa học máy tính (CS), trí tuệ nhân tạo (AI), tối ưu hóa, truy xuất thông tin,...

Trong Toán học về lý thuyết độ phức tạp giải thuật, vấn đề tìm kiếm là những bài toán được biểu diễn bằng quan hệ nhị phân. Xét về mặt trực giác, vấn đề bao gồm việc tìm kiếm cấu trúc y bên trong đối tượng x. Một thuật toán được cho là có thể giải quyết vấn đề nếu tồn tại ít nhất một cấu trúc tương ứng và một cấu trúc phù hợp được tạo ra. Nếu không, thuật toán sẽ kết thúc và trả về kết quả thích hợp (không tìm thấy hoặc tìm thấy) [1].

Trong Trí tuệ nhân tạo (AI), **tìm kiếm (search)** là một kỹ thuật giải quyết vấn đề cơ bản nhất nhằm khám phá các giai đoạn liên tiếp nhau trong quá trình giải quyết vấn đề. Tìm kiếm cung cấp một "framework" để mô hình hóa các vấn đề. Hầu hết các vấn đề đều có thể được mô hình hóa dưới dạng vấn đề tìm kiếm. [2]

Một số ví dụ về vấn đề tìm kiếm:

- **Tìm kiếm trong chuỗi:** Tìm kiếm một chuỗi con cụ thể trong một chuỗi lớn.
  Ví dụ: Máy tìm kiếm dữ liệu (Search Engines).
- **Tìm kiếm đường đi ngắn nhất:** Tìm đường đi ngắn nhất từ một điểm đến điểm khác trên đồ thị.
  Ví dụ: Tìm đường đi ngắn nhất giữa hai thành phố trên bản đồ cho trước.
- **Tìm kiếm tối ưu:** Tìm giải pháp tối ưu trong những giải pháp có thể

xảy thể cho một vấn đề cụ thể.
Ví dụ: Tìm giao dịch chứng khoán tốt nhất trong một chuỗi giao dịch.

- **Tìm kiếm đường đi trong trò chơi:** Tìm đường đi tối ưu để đạt được mục tiêu, kết quả tốt nhất trong trò chơi.
  Ví dụ: Tìm nước đi tốt nhất tiếp theo trong bàn cờ. Tìm bước đi tiếp theo trong trò chơi 8-puzzle. Giải đường đi cho trò chơi mê cung.

- **Tìm kiếm đường đi, trạm dừng của xe buýt:** Tìm các đường đi phù hợp hoặc các điểm dừng của xe buýt phù hợp với lộ trình cho trước.
  Ví dụ: Tìm đường đi từ nhà đến trường.

2.1.2 Các thành phần trong một vấn đề tìm kiếm

Một vấn đề tìm kiếm có thể được định nghĩa bởi các thành phần sau:

- **Không gian trạng thái (state space):** Là một tập hợp các **trạng thái (states)** có thể của môi trường. Khi thực hiện tìm kiếm, ta chỉ nên mã hóa, quan tâm quá chi tiết đến những thông tin cần thiết của trạng thái mà không cần quan tâm quá nhiều đến những thông tin chi tiết khác nhằm hạn chế làm rối không gian tìm kiếm và khiến cho việc tìm kiếm trở nên khó khăn hơn.

  Chúng ta có thể thấy sự khác biệt giữa **Trạng thái thế giới (world state)** và **Trạng thái tìm kiếm (search state)**.

  - Trạng thái thế giới (world state) bao gồm mọi chi tiết cuối cùng của môi trường.
  - Trạng thái tìm kiếm (search/model state) chỉ giữ lại các chi tiết cần thiết trong quá trình tìm kiếm giải pháp cho một vấn đề cụ thể (**trừu tượng hóa**).
  - Ví dụ: Để tìm đường đi giữa các thành phố. Để giải quyết vấn đề, chúng ta có thể lựa chọn một mô hình với các thành phố được biểu diễn thành các node và đường đi là các cạnh nối giữa hai node với nhau. Đây là *search/model state*. Trên thực tế (*world state*) còn có

thêm nhiều chi tiết khác không liên quan, cần thiết cho việc tìm kiếm (giao thông trên mỗi con đường, những con đường "tắc" giữa các thành phố,..). Sẽ rất khó để có thể chọn được một mô hình phù hợp bao gồm tất cả những chi tiết không liên quan cho vấn đề được đề ra. [2]

Việc tìm kiếm với một mô hình có trạng thái thế giới (model state) và lựa chọn một mô hình phù hợp là điều cần thiết, quan trọng đồng thời chứng minh sự hiệu quả của giải pháp được sử dụng.

- **Trạng thái ban đầu (initial state/start state)** là nơi bắt đầu cho việc tìm kiếm, nơi tác nhân (**agent**) bắt đầu. Ví dụ trong bài toán tìm được đi từ thành phố A đến thành phố B, trạng thái ban đầu là thành phố A; trong bài toán 8 Quân hậu, nó có thể là bất kỳ vector ở hàng nào trong 8 hàng trong trường hợp sử dụng thuật toán quay lui (backtracking).

- **Kiểm tra mục tiêu (goal test)**, là một hàm xác định xem trạng thái tìm được có phải là trạng thái mục tiêu hay không. Thường thì việc kiểm tra sẽ mang tính trực quan nhưng đôi khi nó cũng được chỉ định bởi các thuộc tính trừu tượng (abstract property).

  Ví dụ trong bài toán tìm đường đi, kiểm tra mục tiêu là việc so sánh trạng thái hiện tại (**current state**) với đích đến; trong bài toán 8 quân hậu, có thể hiểu là việc kiểm tra xem có tồn tại hai quân hậu có thể "ăn" nhau trong lượt tiếp theo hay không. Mức độ hiệu quả của việc áp dụng kiểm tra mục tiêu phụ thuộc vào mô hình được lựa chọn cho vấn đề đặt ra. [2]

  Ngoài ra, có thể là một tập hợp hoặc nhiều trạng thái mục tiêu (**goal states**). Những trạng thái mục tiêu đó là một tập hợp nhỏ các trạng thái mục tiêu có thể được thay thế hoặc những trạng thái được xác định bởi một thuộc tính được áp dụng cho nhiều trạng thái. Trong một số trường hợp, "mục tiêu" (**goal**) được sử dụng cho bất kỳ mục tiêu nào trong số những trạng thái mục tiêu được tìm thấy.

- **Hành động có thể (possible actions)** của tác nhân. Gỉa sử ta có một trạng thái s, hàm **ACTIONS(s)** sẽ trả về một tập hữu hạn các hành động

có thể được thực thi bởi s, mọi phần tử trong tập hợp đều có thể áp dụng được cho s. Hay **hàm kế tiếp (successor function)**

- **Mô hình chuyển tiếp (transition model)** mô tả cụ thể việc thực hiện của từng hành động, hay còn gọi là mô hình chuyển tiếp được thể hiện bởi hàm **FUNCTION(s,a)** $= a'$, trả về trạng thái mô hình chuyển tiếp *a'* có được từ việc thực hiện hành động a ở trạng thái s. Thuật ngữ **kế thừa (successor)** để chỉ bất kỳ trạng thái nào có thể đạt được từ trạng thái cho trước.

  Qua đó, có thế thấy:

  - Trạng thái ban đầu (initial state), hành động (actions) và mô hình chuyển tiếp (transition model) đã ngầm định nghĩa nên không gian trạng thái (state space) của vấn đề, tập hợp tất cả trạng thái cso thể truy cập được từ trạng thái ban đầu bằng bất kỳ hành động hay chuỗi hành động nào.
  - Không gian trạng thái tạo thành một **đồ thị (graph)** trong đó các node là các trạng thái và các liên kết giữa các node đó là các hành động chuyển tiếp từ trạng thái trước đến trạng thái sau, hoặc ngược lại.
  - **Đường đi (path)** trong không gian trạng thái là một chuỗi các trạng thái được kết nối bằng một chuỗi hành động.

- **Chi phí đường dẫn (path cost function)** chỉ định chi phí bằng một giá trị số nào đó cho mỗi đường dẫn. Xét chi phí của hành động a đi từ trạng thái s đến trạng thái *s'*, ta có hàm **c(s,a,s')**. Các tác nhân giải quyết vấn đề (**problem-solving agent**) nên sử dụng hàm này để phản ánh được hiệu suất của chính nó. Ví dụ với bài toán tìm đường đi, chi phí của hành động có thể là chiều dài được tính bằng số km hoặc thời gian đi đường đến đích. Chi phí của một đường đi có thể được tính bằng tổng chi phí của các hành động riêng lẻ dọc theo đường đi.

Giải pháp (solution) cho một vấn đề là một **chuỗi các hành động (sequence of actions)**, được gọi là **kế hoạch (plan)**. chuyển từ trạng thái ban

đầu đến trạng thái mục tiêu. Một giải pháp đạt được thông qua các **thuật toán tìm kiếm (search algorithms)**.

Chúng ta giả định rằng chi phía hành động có nghĩa là cộng theo vào, tức chi phí của một đường đi là tổng chi phí của từng hành động riêng lẻ trên đường đi đó.

Chất lượng giải pháp được đo bằng hàm chi phí và một **giải pháp tối ưu (optimal solution)** có chi phí đường đi thấp nhất trong tất cả các giải pháp.

2.1.3 Mã giả (pseudo code) chung để giải quyết một bài toán tìm kiếm

Algorithm 1 Mã giả chung để giải quyết bài toán tìm kiếm [3]

```
1:  function SIMPLE-PROBLEM-SOLVING-AGENT (percept) returns an action
2:      static: seq, an action sequence, initially empty
3:              state, some description of the current world state
4:              goal, a goal, initially null
5:              problem, a problem formulation
6:      state ← updateSate(state, percept).
7:      if seq is empty then do
8:          goal ← formulateGoal (state).
9:          problem ← formulateProblem (state, goal).
10:         seq ← search (problem).
11:     action ← first(seq).
12:     seq ← reset(seq).
13:     return action
```

2.1.4 Phân biệt Uninformed Search và Informed Search

Tìm kiếm không có thông tin (Uninformed Search) là chiến lược tìm kiếm mà trong đó chúng ta không có hiểu biết gì về các đối tượng để có hướng dẫn tìm kiếm mà chỉ đơn thuần xem xét các đối tượng theo một hệ thống nào đó để phát hiện ra đối tượng cần tìm. Chiến lược này còn được gọi là **Tìm kiếm mù (Blind Search).** Không có yêu cầu về thông tin để chọn ra các trạng thái để duyệt qua. Thuật toán chỉ có thể tạo ra những điểm kế thừa và phân biệt trạng thái mục tiêu với những trạng thái không phải mục tiêu. Tất cả

những thuật toán theo kiểu này được phân biệt bởi thứ tự mở rộng của các node.

Tìm kiếm có thông tin (Informed Search) là chiến lược tìm kiếm có thông tin về trạng thái mục tiêu, sử dụng kiến thức về vấn đề cụ thể, giúp cho việc tìm kiếm đạt hiệu quả hơn. Những thông tin đó có được bởi **heuristic.** Chiến lược này còn được gọi là **Tìm kiếm kinh nghiệm (Heuristic Search)**. Heuristic là một hàm ước tính mức độ gần của các trạng thái với trạng thái mục tiêu. Chiến lược tìm kiếm này sẽ có hiệu quả hơn so với **Uninformed Search** khi có thông tin về trạng thái mục tiêu.

Ví dụ cho việc tìm kiếm thông tin có hiểu biết và không có thông tin trong cuộc sống. Việc tìm kiếm không có thông tin thường thấy khi bạn đọc các trang web này đến trang web khác hoặc đọc các bài báo, quyển sách nào đó mà không thực sự có nhu cầu tìm kiếm thông tin. Ngược lại tìm kiếm hiểu biết thường xảy ra khi bạn không hiểu về một vấn đề nào đó, sẽ bạn tìm kiếm trực tiếp vấn đề mà bản thân chưa hiểu. [4]

Uninformed Search (Blind Search)	Informed Search (Heuristic Search)
Không yêu cầu thông tin cho việc thực hiện tìm kiếm	Yêu cầu thông tin cho việc thực hiện tìm kiếm
Đánh đổi tốc độ và thời gian cho độ chính xác	Đánh đổi độ chính xác cho tốc độ và thời gian
Có thể tìm được giải pháp tốt nhất cho bài toán	Một giải pháp tốt được được xem là giải pháp tối ưu, nhưng có thể không phải là giải pháp tốt nhất.
Yêu cầu nhiều tính toán	Không yêu cầu nhiều tính toán
Không thực tế cho các bài toán lớn	Có thể xử lý những bài toán lớn
Nhiều tốt kém nhưng đạt ít hiệu quả	Ít tốn kém nhưng đạt hiệu quả cao hơn
Ví dụ: Breadth-First Search, Depth-First Search, Uniform Cost Search	Ví dụ: Best First Search, Greedy Search, A* Algorithm

Bảng 2: Phân biệt Uninformed Search và Informed Search [4]

2.2 Thuật toán Depth First Search (DFS)

2.2.1 Ý tưởng chung

- **Tìm kiếm theo chiều sâu (Depth-first Search)** luôn mở rộng các node chưa được mở rộng ở sâu nhất (deepest unexpanded node) trong tập hợp các node đang mở trong *frontier*. Khi các node được mở, chúng sẽ bị loại khỏi *frontier*, giải thuật tiếp tục cho tới khi gặp được đỉnh cần tìm hoặc tới một node sâu nhất. Khi đó giải thuật quay lui về đỉnh vừa mới tìm kiếm ở bước trước.
- Trong quá trình mở rộng để ghi nhớ đỉnh liền kề, thuật toán sử dụng Stack (ngăn xếp) theo nguyên lý **LIFO (Last in First Out)** - nghĩa là node được tạo mới nhất sẽ được chọn để mở rộng. Vì vậy, node được chọn để mở phải là node sâu hơn.

2.2.2 Mã giả

```
function DFS(G, u):
    let St be stack
    Push u in the stack
    mark u as visited.
    while ( St is not empty)
        v = Node at the top of stack
        remove the node from stack
        for all neighbors adj_node of v in Graph G:
            if adj_node is not visited :
                mark adj_node as visited
                push adj_node in stack
```

2.2.3 Đánh giá thuật toán

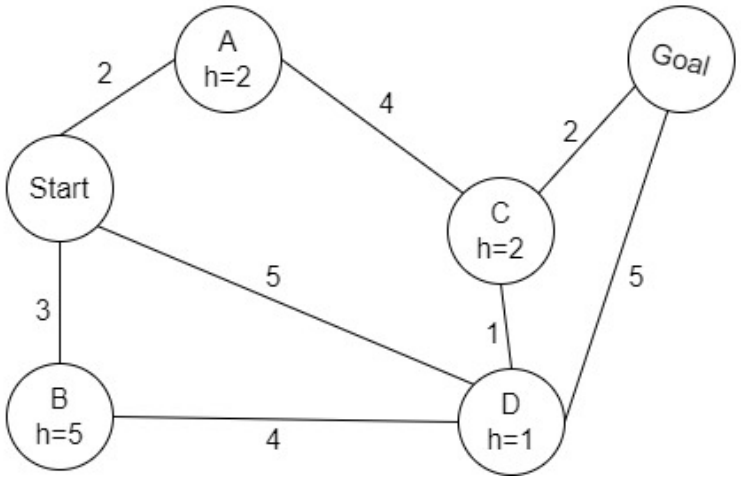
- Ta có:
  - m là độ sâu của cây tìm kiếm.
  - b là số nhánh tối đa của một node.
  - Giả sử với một cây tìm kiếm đầy đủ thì số node tại độ cao i là b^i.
- **Tính đầy đủ (completeness):** Bởi vì thuật toán thực hiện theo tư duy vét cạn, nên nó sẽ hoàn tất nếu cây tìm kiếm là hữu hạn và trả về lời giải cho bài toán nếu lời giải đó có tồn tại.

Ngoài ra, việc hoàn thành sẽ thất bại trong trường hợp cây tìm kiếm vô hạn (hay độ sâu vô hạn và không gian có vòng lặp). Để giảm thiểu trường hợp đó, ta sẽ sửa đổi để tránh các trạng thái lặp lại trên đường đi để biến không gian đó thành không gian hữu hạn.

- **Tính tối ưu (optimality):** Thuật toán không tối ưu vì nó có thể tạo ra một số lượng lớn các bước hoặc chi phí cao để đạt đến node mục tiêu song không phải lúc nào cũng tìm được đường đi ngắn nhất cho bài toán.

- **Tính phức tạp (complexity):**
  - Độ phức tạp về thời gian: $T(b) = 1 + b^2 + b^3 + ... + b^m = O(b^m)$
  Tương ứng với số lượng node được duyệt qua. Thuật toán sẽ hoạt động không tốt nếu m lớn hơn độ sâu của một giải pháp "cạn". Nhưng nếu các giải pháp dày đặc hơn thì thuật toán sẽ nhanh hơn so với Breadth First Search (BFS).
  - Độ phức tạp về không gian: $S(b) = O(bm)$ (không gian tuyến tính)

2.2.4 Minh họa



- Đỉnh bắt đầu: Start
- Đỉnh kết thúc: Goal
- Đường đi tìm được là: $S \to A \to C \to G$

Current Node	Open Set	Close Set
	{S}	∅
S	{A, B, D}	{S}
A	{C, B, D}	{S, A}
B	{C, D}	{S, A, B}
C	{D, G}	{S, A, B, C}
D	{G}	{S, A, B, C, D}
G	{ ∅}	{S, A, B, C, D, G}

Bảng 3: Các bước thực hiện thuật toán DFS

2.3 Thuật toán Breadth First Search (BFS)

2.3.1 Ý tưởng chung

- **Tìm kiếm theo chiều rộng (Breadth-first Search)** duyệt qua đồ thị theo chiều rộng trong đó node gốc được mở rộng trước, sau đó tất cả các node kế thừa của node gốc được mở rộng tiếp theo, sau đó là các node kế thừa của chúng đến khi tìm được mục tiêu (goal). Tất cả các node lân cận được mở cùng ở độ sâu hiện tại với node đang xét trước khi chuyển sang các node ở cấp độ tiếp theo.
- Trong quá trình thực hiện, thuật toán sử dụng hàng đợi (queue) theo nguyên lý **FIFO (First In First Out)** cho *frontier*. Theo đó, các node mới sẽ chuyển về phía sau hàng đợi và các node cũ sẽ được mở rộng trước tiên..

2.3.2 Mã giả

```
1: function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
2:     node ← NODE(problem.INITIAL).
3:     if problem.IS-GOAL(node.STATE) then return node
4:     frontier ← a FIFO queue, with node as an element.
5:     reached ← problem.INITIAL.
6:     while not IS-EMPTY(frontier) do
7:         node ← POP (frontier).
8:         for each child in EXPAND(problem, node) do
9:             s ← child.STATE
```

```
10:            if problem.IS-GOAL(s) then return child
11:            if s is not in reached then
12:              add s to reached
13:              add child to frontier
14:    return failure
```

Mã giả của thuật toán BFS [5]

2.3.3 Đánh giá thuật toán

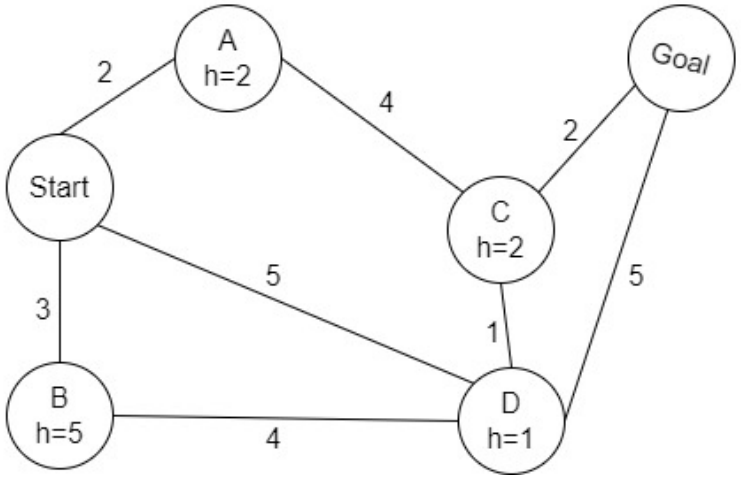
- Ta có:
  - m là độ sâu của cây tìm kiếm.
  - d là độ sâu của trạng thái đích - b là số nhánh tối đa của một node.
  - Giả sử với một cây tìm kiếm đầy đủ thì số node tại độ cao i là b^i. Gỉa sử với một cây tìm kiếm đầy đủ thì số node tại độ cao i là b^i.

- **Tính đầy đủ (completeness):** Thuật toán được thực hiện theo tư duy vét cạn do đó nó có thể trả về kết quả cho vấn đề được nếu ra nếu kết quả có tồn tại hay nói cách khác là hệ số phân nhánh của b là hữu hạn.

- **Tính tối ưu (optimality):** Thuật toán đưa ra kết quả với ít trạng thái tối ưu nhất. Thuật toán sẽ tối ưu nếu chi phí là không đổi ở mỗi bước, nhưng thường thì sẽ không tối ưu.

- **Tính phức tạp (complexity):**
  - Độ phức tạp về thời gian: $T(b) = 1 + b^2 + b^3 + ... + b^d = O(b^d)$
  Tương ứng với số lượng node được duyệt qua đến khi tìm được đích
  - Độ phức tạp về không gian: $S(b) = O(b^d)$ - mỗi node đều được lưu trong bộ nhớ.
  Yêu cầu về bộ nhớ là vấn đề lớn hơn so với thời gian thực hiện trong thuật toán này.

2.3.4 Minh họa



- Đỉnh bắt đầu: Start
- Đỉnh kết thúc: Goal
- Đường đi tìm được là: $S \to A \to C \to G$

Current Node	Open Set	Close Set
	{S}	∅
S	{A, B, D}	{S}
A	{B, D, C}	{S, A}
B	{D, C}	{S, A, B}
C	{D, G}	{S, A, B, C}
D	{G}	{S, A, B, C, D}
G	∅	{S, A, B, C, D, G }

Bảng 4: Các bước thực hiện thuật toán BFS

2.4 Thuật toán Uniform-Cost Search (UCS)

2.4.1 Ý tưởng chung

- **Uniform-cost search** thường được sử dụng cho các đồ thị có n node với các trọng số (chi phí) khác nhau nhằm tìm ra đường đi có tổng trọng số (chi phí) nhỏ nhất *g(n)*. Điều này được thực hiện bằng cách lưu trữ như một hàng đợi ưu tiên theo thứ tự của g.
- Việc tìm kiếm tiếp tục bằng cách duyệt các node tiếp theo với trọng số (chi

phí) thấp nhất tính từ node gốc. Khởi tạo mảng chứa các trọng số (chi phí) của đường đi tính từ trạng thái bắt đầu đến những trạng thái đang xét. Thuật toán sẽ mở rộng đến một node nếu node đó có trọng số (chi phí) thấp nhất đồng thời cập nhật lại chi phí đường đi tính từ trạng thái bắt đầu sau mỗi lần mở rộng.

2.4.2 Mã giả

Algorithm 2 Mã giả của thuật toán UCS

```
1:  function UNIFORM-COST SEARCH(problem) returns a solution, or failure
2:      node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0).
3:      frontier ← a priority queue ordered by PATH-COST, node as the only element
4:      explored ← an empty set
5:      loop do
6:          if EMPTY?(frontier) then return failure
7:          node ← POP(frontier) /*chooses the lowest-cost node in frontier*/
8:          if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
9:          add node.STATE to explored
10:         for each action in problem.ACTIONS(node.STATE) do
11:           child ← CHILD-NODE(problem, node, action
12:           if child.STATE is not in explored or frontier then
13:             frontier ← INSERT(child,frontier)
14:           else if child.STATE is in frontier with higher PATH-COST then
15:             replace that frontier node with child
```

2.4.3 Đánh giá thuật toán

- Với:
  - b là số nhánh tối đa của một node.
  - ϵ là chi phí tối thiểu tại mỗi bước.
  - C là chi phí của mỗi bước đi.
  - C^* là chi phí của biện pháp tối ưu nhất.

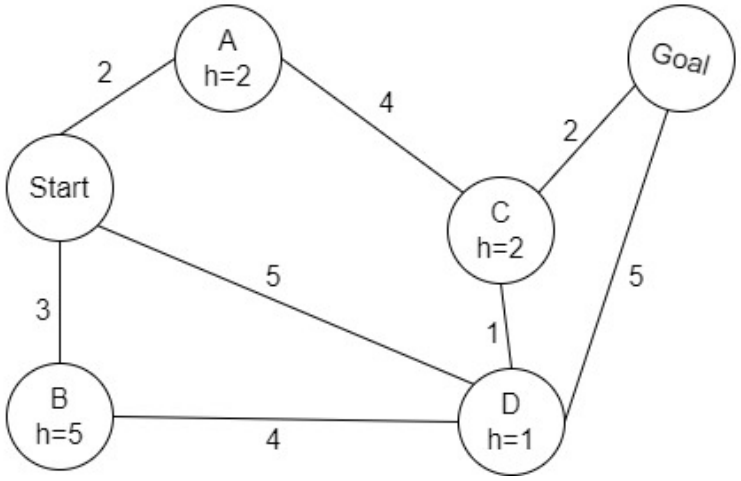
- **Tính đầy đủ (completeness):** Thuật toán UCS hoàn thành khi hệ số phân nhánh b là hữu hạn. chẳng hạn như nếu có giải pháp, UCS sẽ tìm ra giải pháp đó.
  Điều kiện: $C > \epsilon$ trong đó chi phí của mỗi bước lớn hơn hằng số dương ϵ

- **Tính tối ưu (optimality):** Thuật toán trả về lời giải tối ưu nhất với đường đi có chi phí thấp nhất đồng thời bởi vì các node được mở rộng theo thứ tự chi phí đường đi tăng dần.

- **Tính phức tạp (complexity):**

  - Độ phức tạp về thời gian: $T(b) = O(b^{C^*/\epsilon})$.
  Chi phí tạo ra tất cả các node có chi phí $\leq$ chi phí của giải pháp tối ưu.
  - Độ phức tạp về không gian: $S(b) = O(b^{C^*/\epsilon})$.
  Tương tự như độ phức tạp về thời gian.

2.4.4 Minh họa



- Đỉnh bắt đầu: Start
- Đỉnh kết thúc: Goal
- Đường đi tìm được là: $S \to A \to C \to G$

Current Node	Open Set	Close Set
	{S(0)}	∅
S(0)	{A(2), B(3), D(5)}	{S(0)}
A(2)	{(B(3), D(5), C(2 + 4)}	{(S(0), A(2)}
B(3)	{ D(5), C(6)}	{(S(0), A(2), B(3)}
D(5)	{C(6), G(10)}	{(S(0), A(2), B(3), D(5)}
C(6)	{G(8)}	{(S(0), A(2), B(3), D(5), C(6)}
G(8)	∅	{(S(0), A(2), B(3), D(5), C(6), G(8)}

Bảng 5: Các bước thực hiện thuật toán UCS

2.5 Thuật toán A* (AStar)

2.5.1 Ý tưởng chung

- **AStar** là hình thức **best-first search** được biết đến rộng rãi. Thuật toán mở rộng các node gần trạng thái đích nhất.
- Thuật toán đánh giá các node là gần hay xa so với trạng thái đích dựa trên ahfm **heuristic**, là sự kết hợp **g(n)** và **h(n)**. Cụ thể như sau:

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

Trong đó:
+ f(n) là chi phí ước tính của giải pháp ít tốt kém nhất thông qua n.
+ g(n) là chi phí để tiếp cận node.
+ h(n) là chi phí đi từ node đó đến trạng thái đích.
- Chiến lược tìm kiếm là thử các node với giá trị $f(n)$ thấp nhất và *heuristic* được cung cấp phải đáp ứng những điều kiện cơ bản. [6]

2.5.2 Mã giả

- Goal state: *nodeGoal*
- Start state: *nodeStart*
- **OPEN:** bao gồm những node đã được truy cập nhưng chưa được mở rộng (các node kế tiếp nó chưa được thăm).
- **CLOSE:** bao gồm những node đã được truy cập và mở rộng (các node kế sau nó đã được thăm và đưa vào danh sách mở).
- Mã giả của thuật toán AStar [7]

```
1  Put nodeStart in the OPEN list with f(nodeStart) = h(nodeStart) (initialization)
2  while the OPEN list is not empty {
3      Take from the open list the node nodeCurrent with the lowest
4          f(nodeCurrent) = g(nodeCurrent) + h(nodeCurrent)
5      if nodeCurrent is nodeGoal we have found the solution; break
6      Generate each state nodeSuccessor that come after nodeCurrent
7      for each nodeSuccessor of nodeCurrent {
8          Set successorCurrentCost = g(nodeCurrent) + w(nodeCurrent, nodeSuccessor)
9          if nodeSuccessor is in the OPEN list {
10             if g(nodeSuccessor)  successorCurrentCost continue (to line 20)
11         } else if nodeSuccessor is in the CLOSED list {
12             if g(nodeSuccessor)  successorCurrentCost continue (to line 20)
```

```
            Move nodeSuccessor from the CLOSED list to the OPEN list
        } else {
            Add nodeSuccessor to the OPEN list
            Set h(nodeSuccessor) to be the heuristic distance to nodeGoal
        }
        Set g(nodeSuccessor) = successorCurrentCost
        Set the parent of nodeSuccessor to nodeCurrent
    }
    Add nodeCurrent to the CLOSED list
}
if(nodeCurrent != nodeGoal) exit with error (the OPEN list is empty)
```

2.5.3 Đánh giá thuật toán

- **Tính đầy đủ (completeness):** Thuật toán AStar hoàn thành khi hệ số phân nhánh là hữu hạn và chi phí cho mọi hành động là cố định, trừ trường hợp có vô số node với $f \leq f(G)$.

- **Tính tối ưu (optimality):** Thuật toán tìm kiếm AStar là tối ưu nếu nó tuân theo hai điều kiện sau: Một là *h(n)* phải là một heuristic có thể chấp nhận được. Hai là tính nhất quán. Không có thuật toán nào với cùng một heuristic được đảm bảo sẽ mở rộng ít node hơn. Thuật toán sẽ cho được một trong những kết quả tối ưu có thể xảy ra nhưng chưa chắc được rằng đó là kết quả tối ưu nhất.

- **Tính phức tạp (complexity):**
  - Độ phức tạp về thời gian: Phụ thuộc vào hàm heuristic và số lượng node được mở rộng theo cấp số nhân với độ sâu d là độ sâu của trạng thái đích trong cây, với b là số nhánh tối đa của một node. Trường hợp xấu nhất, độ phức tạp về thời gian là $O(b^d)$.
  - Độ phức tạp về không gian: Giống với UCS, độ phức tạp về không gian là $O(b^d)$.

- Một số hàm heuristic:

  – **Khoảng cách Euclid:**
  Công thức: $h(n) = \sqrt{(x_{goal} - x_{current})^2 + (y_{goal} - y_{current})^2}$

Dựa trên khoảng cách thẳng giữa node hiện tại và node đích trong không gian Euclid.

- **Khoảng cách Manhattan:**
  Công thức: $h(n) = |x_{goal} - x_{current}| + |y_{goal} - y_{current}|$
  Đây là khoảng cách giữa hai điểm được đo theo các trục vuông góc. Thường được sử dụng trong môi trường dựa trên lưới.
- **Khoảng cách Chebyshev:**
  Công thức $h(n) = max(|x_{goal} - x_{current}|, |y_{goal} - y_{current}|)$
  Đây là mức chênh lệch tuyệt đối lớn nhất giữa các tọa độ. Nó cũng được sử dụng trong môi trường dựa trên lưới.
- **Khoảng cách Octile:**
  Công thức: $h(n) = max(|x_{goal} - x_{current}|, |y_{goal} - y_{current}|) + (\sqrt{2} - 1) * min(|x_{goal} - x_{current}|, |y_{goal} - y_{current}|)$
  Di chuyển theo đường chéo nhưng với chi phí là $\sqrt{2}$ lần của môi trường
- **Hàm Heuristic tùy chỉnh:** Tùy thuộc vào vấn đề và lĩnh vực bạn gặp phải, có thể có những phướng pháp lựa chọn heuristic phù hợp với vấn đề được đề cập. Điều này có thể liên quan đến kiến thức về lĩnh vực cụ thể.

- Một hàm heuristic chấp nhận được (admissible heuristic) nếu với mọi node n, giá trị $h(n)$ nằm trong khoảng:

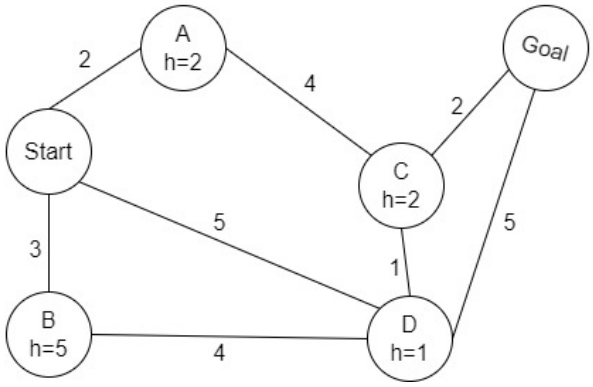
$$0 \leq h(n) \leq h^*(n)$$

Trong đó: $h^*(n)$ là chi phí thực để đạt được trạng thái đích bắt đầu từ n.

- Một hàm heuristic chấp nhận được không bao giờ có chi phí quá cao khi đạt được trạng thái đích, tức độ tối ưu.

- Ngoài ra, nếu h(n) là một heuristic chấp nhận được thì ASatr sẽ sử dụng tìm kiếm trên cây mới mức tối ưu cao nhất.

2.5.4 Minh họa



- Đỉnh bắt đầu: Start
- Đỉnh kết thúc: Goal
- Đường đi tìm được là: $S \rightarrow A \rightarrow C \rightarrow G$

Current	Open Set	Close Set
	{S}	∅
S	$\{B_A(3+5), A_A(2+2), D_A(5+1)\}$	{ S }
A(4)	$\{B_A(8), C_A(2+4+2), D_A(6)\}$	{S, A(4)}
D(6)	$\{B_A(8), C_A(8), G_A(10)\}$	{S, A(4), D(6)}
B(8)	$\{ C_A(8), G_A(10)\}$	{S, A(4), D(6), B(8) }
C(8)	$\{G_A(8)\}$	{S, A(4), D(6), B(8), C(8) }
G(8)	∅	{S, A(4), D(6), B(8), C(8), G(8) }

Bảng 6: Các bước thực hiện thuật toán AStar

3 So sánh các thuật toán khác nhau

3.1 So sánh UCS, Greddy và A-Star

So sánh	UCS	A-Star [8]	Greddy [8]
Loại tìm kiếm	Tìm kiếm mù	Tìm kiếm kinh nghiệm	Tìm kiếm kinh nghiệm
Độ phức tạp thời gian	$O(b^{C^*})$	$O(b^d)/\epsilon)$	$O(b^m)$
Độ phức tạp không gian	$O(b^{C^*})$	$O(b^{C^*})$	$O(b^m)$

So sánh	UCS	A-Star	Greddy
Tính hoàn thành	Đầy đủ khi không có trọng số âm và hệ số phân nhánh b là hữu hạn	Thuật toán A * hoàn thành miễn khi Hệ số phân nhánh là hữu hạn và chi phí cho mọi hành động là cố định.	Không đầy đủ, vì có thể bị mắc kẹt trong các vòng lặp.
Tính tối ưu	Tối ưu, vì nó tìm kiếm theo chi phí thấp nhất đầu tiên và có chi phí > 0	Thuật toán tìm kiếm AStar là tối ưu nếu nó tuân theo hai điều kiện đã đề cập ở trên.	Không tối ưu. Greedy tập trung vào việc tìm node tiếp theo gần mục tiêu nhất
Ưu điểm	Tìm ra đường đi có chi phí nhỏ nhất. Hoạt động tốt trong các trường hợp không có thông tin đặc biệt về vấn đề.	Kết hợp cả ưu điểm của UCS và Greedy: sử dụng thông tin về mục tiêu và đảm bảo tối ưu hóa chi phí. Admissible heuristic đảm bảo tìm ra đường đi tối ưu.	Sử dụng thông tin về mục tiêu để chọn node tiếp theo, điều này có thể dẫn đến tốc độ tìm kiếm nhanh. Tốt trong các vấn đề mà việc tối ưu hóa không quan trọng vì tìm kiếm nhanh đến vị trí gần đích, giúp tìm kiếm nhanh đến vị trí tiếp cận đích.
Nhược điểm	Không sử dụng thông tin về mục tiêu, có thể dẫn đến mở nhiều node không cần thiết. Không phù hợp với các vấn đề có không gian trạng thái lớn. Không hoạt động đúng trên đồ thị có trọng số âm.	Hiệu suất có thể bị ảnh hưởng bởi chất lượng của heuristic. Cần phải chọn một heuristic admissible (không bao giờ đánh giá quá cao) để đảm bảo tối ưu. Heuristic không admissible có thể dẫn đến kết quả không tối ưu.	Không đảm bảo tìm ra đường đi tối ưu. Có thể dẫn đến các vòng lặp và không đúng đắn.

Bảng 7: So sánh ba thuật toán UCS, Greddy và A-Star

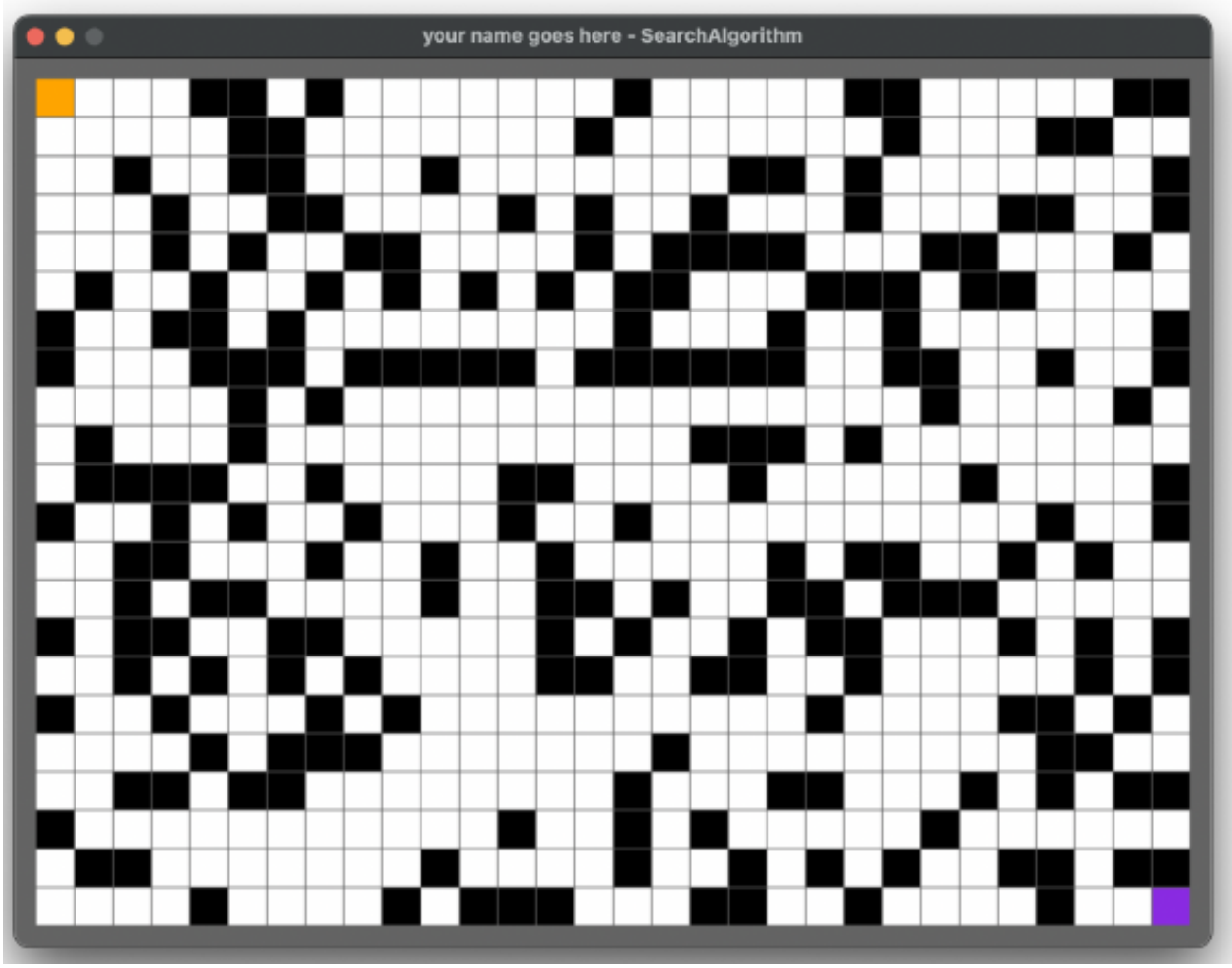
3.2 So sánh UCS và Dijkstra

So sánh	Dijkstra	UCS
Định nghĩa	Thuật toán Dijkstra tìm đường đi ngắn nhất từ node gốc đến mọi node khác	Tìm các đường dẫn ngắn nhất về mặt chi phí từ node gốc đến node mục tiêu, UCS là thuật toán tập trung vào việc tìm một đường đi ngắn nhất đến một điểm kết thúc duy nhất thay vì tìm đường đi ngắn nhất đến mọi điểm.
Loại tìm kiếm	Tìm kiếm kinh nghiệm	Tìm kiếm mù
Thuật toán	Dijkstra sử dụng chi phí để chọn node tiếp theo, nhưng nó cũng duy trì một tập hợp các node đã được duyệt để đảm bảo rằng chỉ các node chưa duyệt mới được xem xét. Dijkstra tập trung vào việc tìm kiếm đường đi ngắn nhất từ một đỉnh gốc đến tất cả các đỉnh còn lại trong đồ thị.	UCS sử dụng một tiêu chí chọn node tiếp theo dựa trên chi phí đến node đó. Đặc điểm của UCS là chỉ sử dụng thông tin về chi phí và không sử dụng bất kỳ thông tin gì về đỉnh đích.
Phù hợp với	Dijkstra được sử dụng trên đồ thị chung	UCS thường được sử dung với cây
Tốc độ	Dijkstra tốn nhiều thời gian hơn UCS	UCS ít tốn thời gian hơn Dijkstra
Đồ thị có trọng số không âm	Hoạt động tốt	Hoạt động tốt
Đồ thị có trọng số âm	Có thể áp dụng nhưng có thể cho kết quả sai	Không hoạt động tên đồ thị trọng số âm
Tính hoàn thành	Đầy đủ khi không có trọng số âm.	Đầy đủ khi không có trọng số âm và hệ số phân nhánh b là hữu hạn
Tính tối ưu	Tối ưu, vì tìm kiếm đường đi ngắn nhất từ điểm gốc tới tất cả các điểm khác.	Tối ưu, vì nó tìm kiếm theo chi phí thấp nhất đầu tiên và có chi phí > 0

Bảng 8: So sánh hai thuật toán UCS và Dijkstra

4 Thực hiện các thuật toán trên

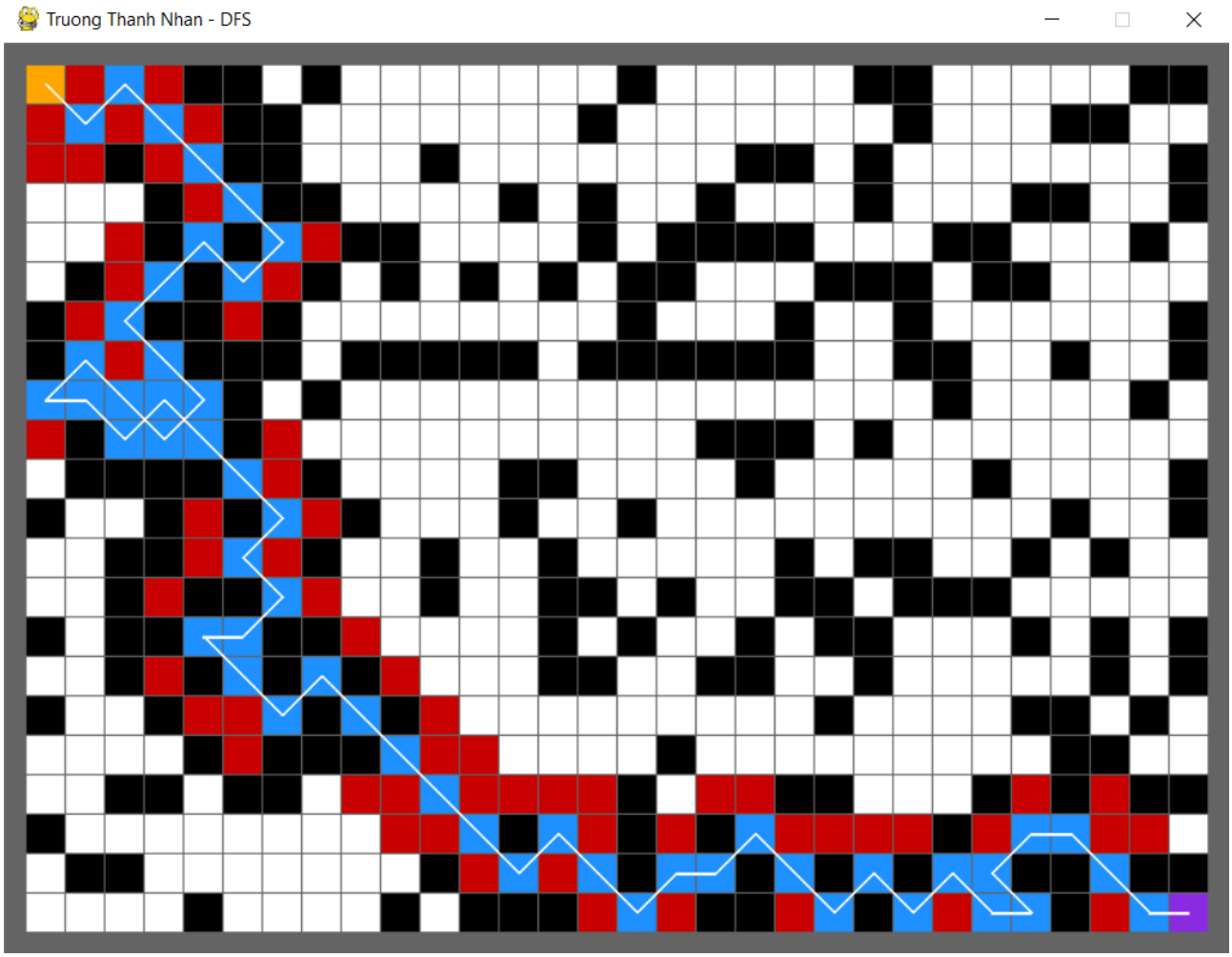
4.1 Yêu cầu



- Tìm đường đi từ node màu cam đến node màu tím trong mê cung
- Các ô màu đỏ: Node trong open_set
- Các ô màu xanh: Node trong close_set
- Dường màu trắng: Đường đi cuối cùng
- Ô màu vàng: Node hiện tại

4.2 DFS

4.2.1 Hình ảnh kết quả



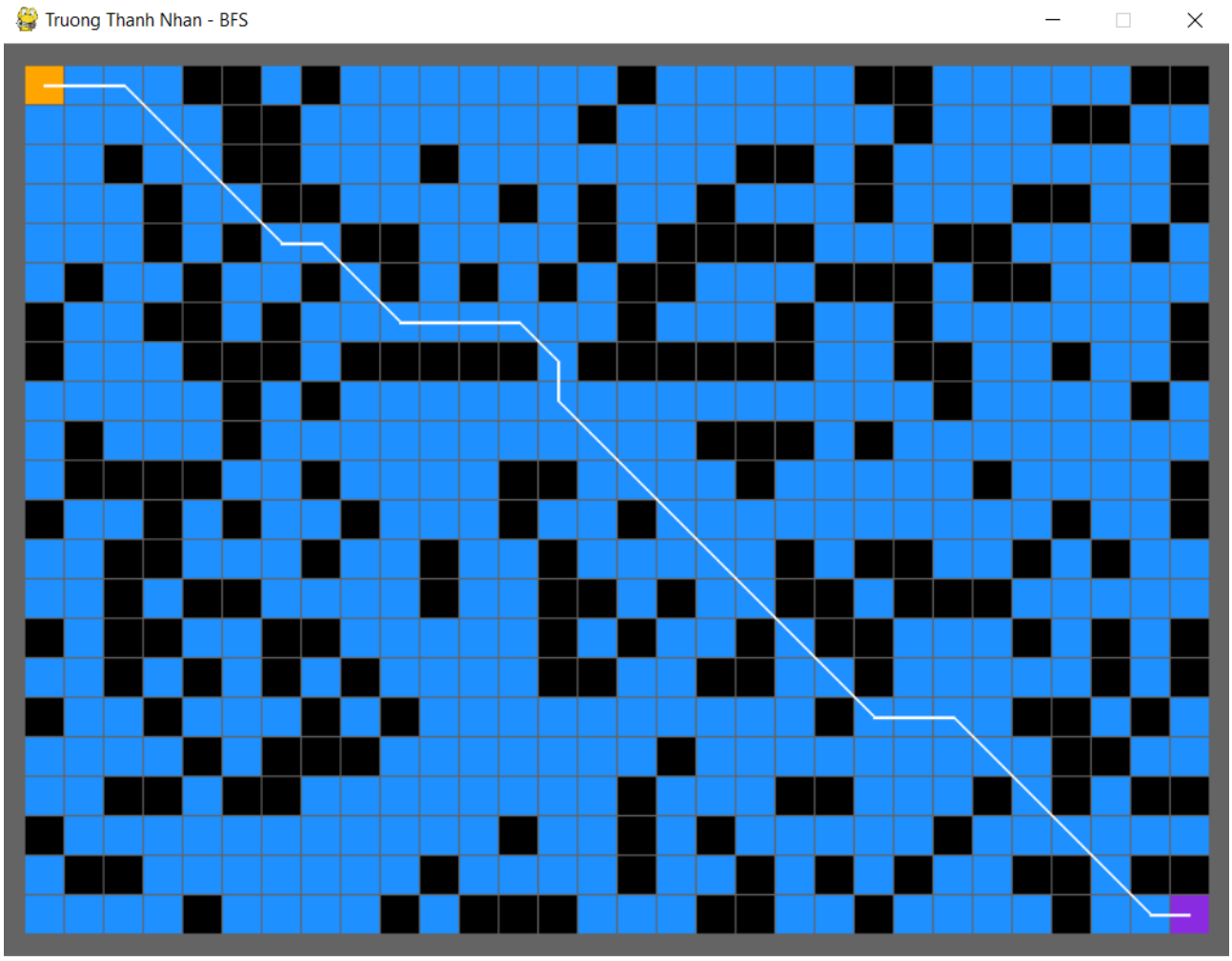
Kết quả tìm kiếm bằng thuật toán DFS

4.2.2 Nhận xét

- Thuật toán không quan tâm đến chi phí.
- Thuật toán tìm theo chiều sâu đến khi tìm được đích, kết quả đường đi tương đối xa hơn so với những thuật toán khác.
- Phụ thuộc vào cách thiết kế thuật toán của mỗi người mà có được những kết quả khác nhau.
- Sử dụng ít bộ nhớ, do số node được mở tương đối ít, số lượng node đang nằm trong *open_set* tương đương số lượng node trong *close_set*.

4.3 BFS

4.3.1 Hình ảnh kết quả



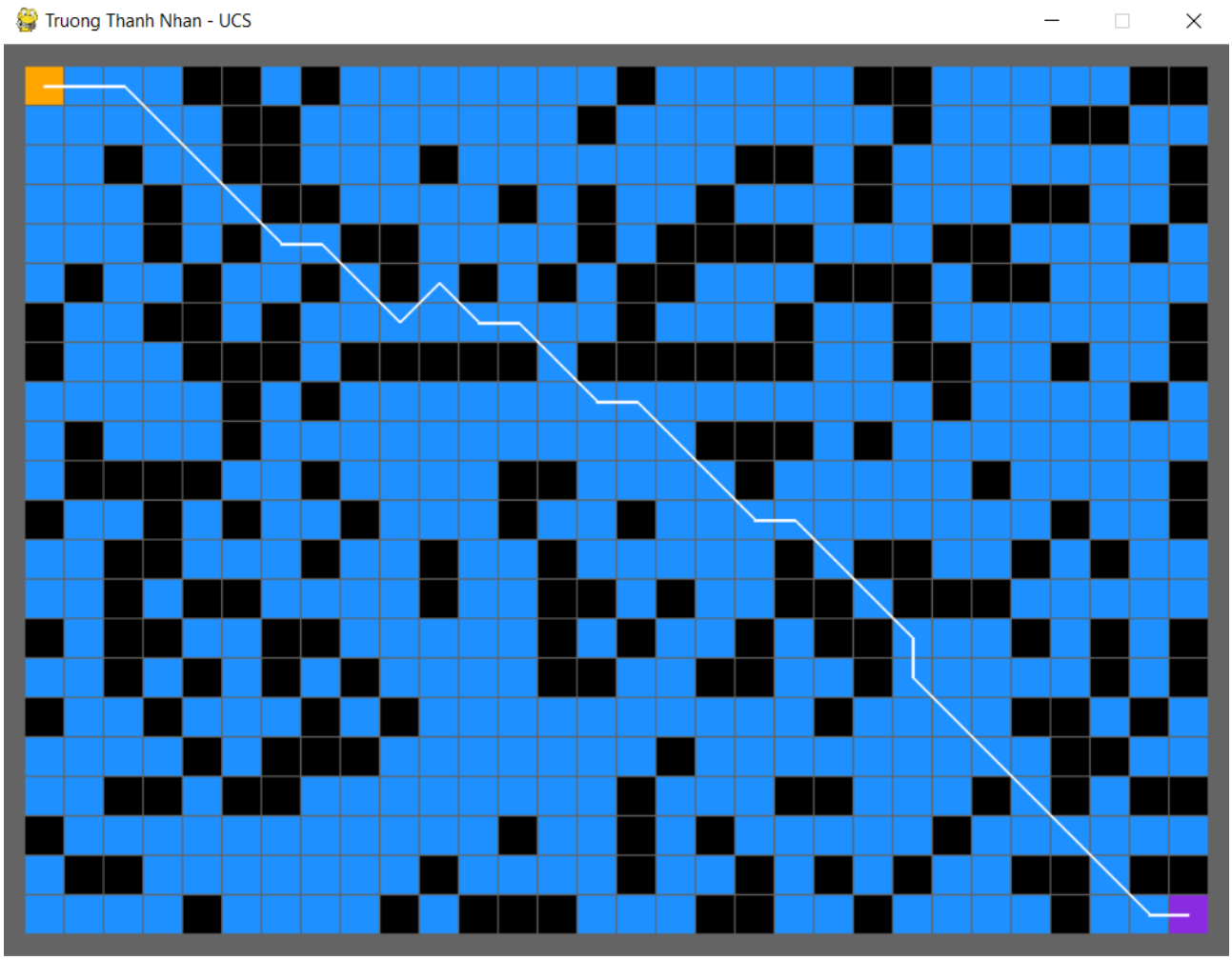
Kết quả tìm kiếm bằng thuật toán BFS

4.3.2 Nhận xét

- Không quan tâm đến chi phí.
- Sử dụng nhiều bộ nhớ hơn (thăm hết tất cả các ô trong mê cung) nên hiệu quả không cao.
- Đường đi tìm được ngắn hơn so với DFS.
- Qúa trình tìm kiếm theo chiều rộng, mở các node lân cận nhau trước khi tìm được đích, vì node bắt đầu và kết thúc nằm ở 2 góc của mê cung do vậy mọi node/ô trong mê cung đều được thăm.

4.4 UCS

4.4.1 Hình ảnh kết quả



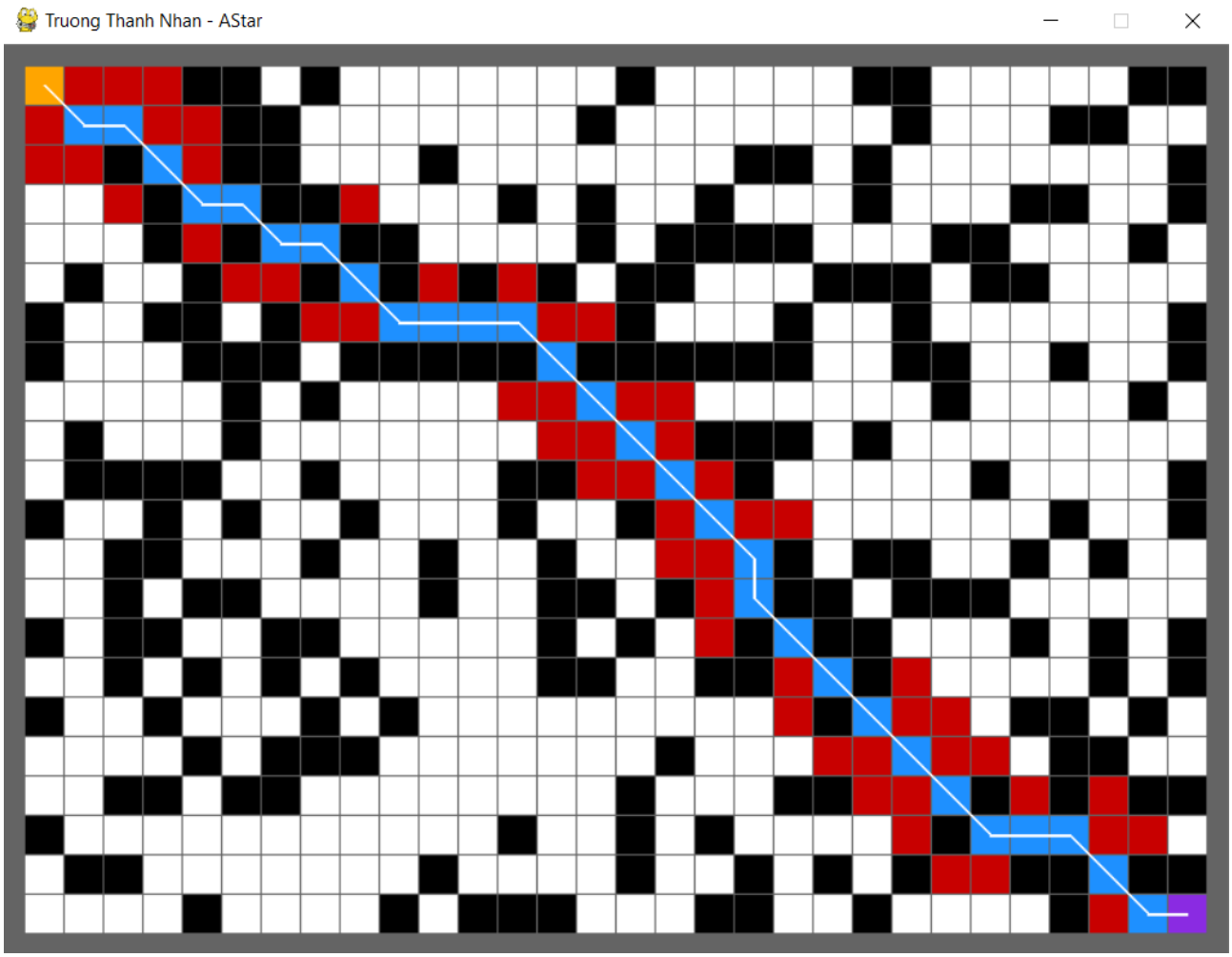
Kết quả tìm kiếm bằng thuật toán UCS

4.4.2 Nhận xét

- Quan tâm đến chi phí trong quá trình tìm kiếm, ở đâu là khoảng cách giữa các ô với nhau.
- Kết quả tìm kiếm tương tự BFS, các node/ô đều được thăm, tuy nhiên đường đi lại không giống nhau, do sự khác biệt giữa chi phí của các node trong quá trình tìm kiếm.
- Xét về quảng đường thì tương đối bằng với BFS và ngắn hơn DFS.
- Trong một số trường hợp, độ phức tạp của UCS có thể lớn hơn BFS.

4.5 AStar

4.5.1 Hình ảnh kết quả



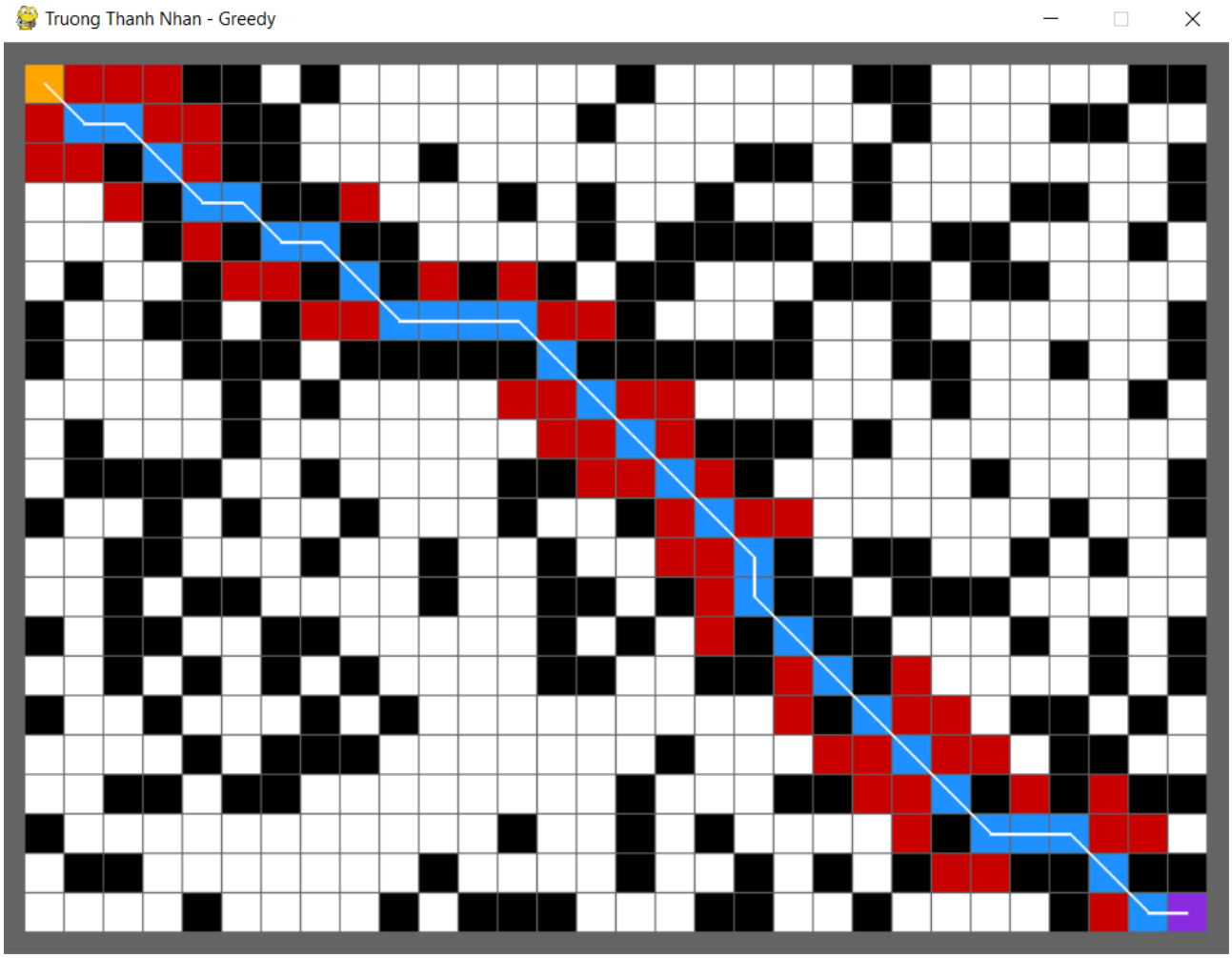
Kết quả tìm kiếm bằng thuật toán AStar

4.5.2 Nhận xét

- Thuật toán phụ thuộc nhiều hàm hàm *heuristic*, mỗi hàm *heuristic* khác nhau có thể cho những kết quả khác nhau. Tùy vào từng bài toán, vấn đề được đặt ra ta lựa chọn hàm *heuristic* phù hợp để đạt hiệu quả tốt nhất.
- Với hình ảnh trên, thuật toán sử dụng hàm **Khoảng cách Euclidean**
  Công thức: $h(n) = \sqrt{(x_{goal} - x_{current})^2 + (y_{goal} - y_{current})^2}$
- Kết quả của thuật toán A^* cho thấy số lượng các node/ô được thăm tương đối ít hơn so với BFS và UCS nhưng đường đi tìm được lại tương đương với hai thuật toán còn lại.

4.6 Greedy

4.6.1 Hình ảnh kết quả



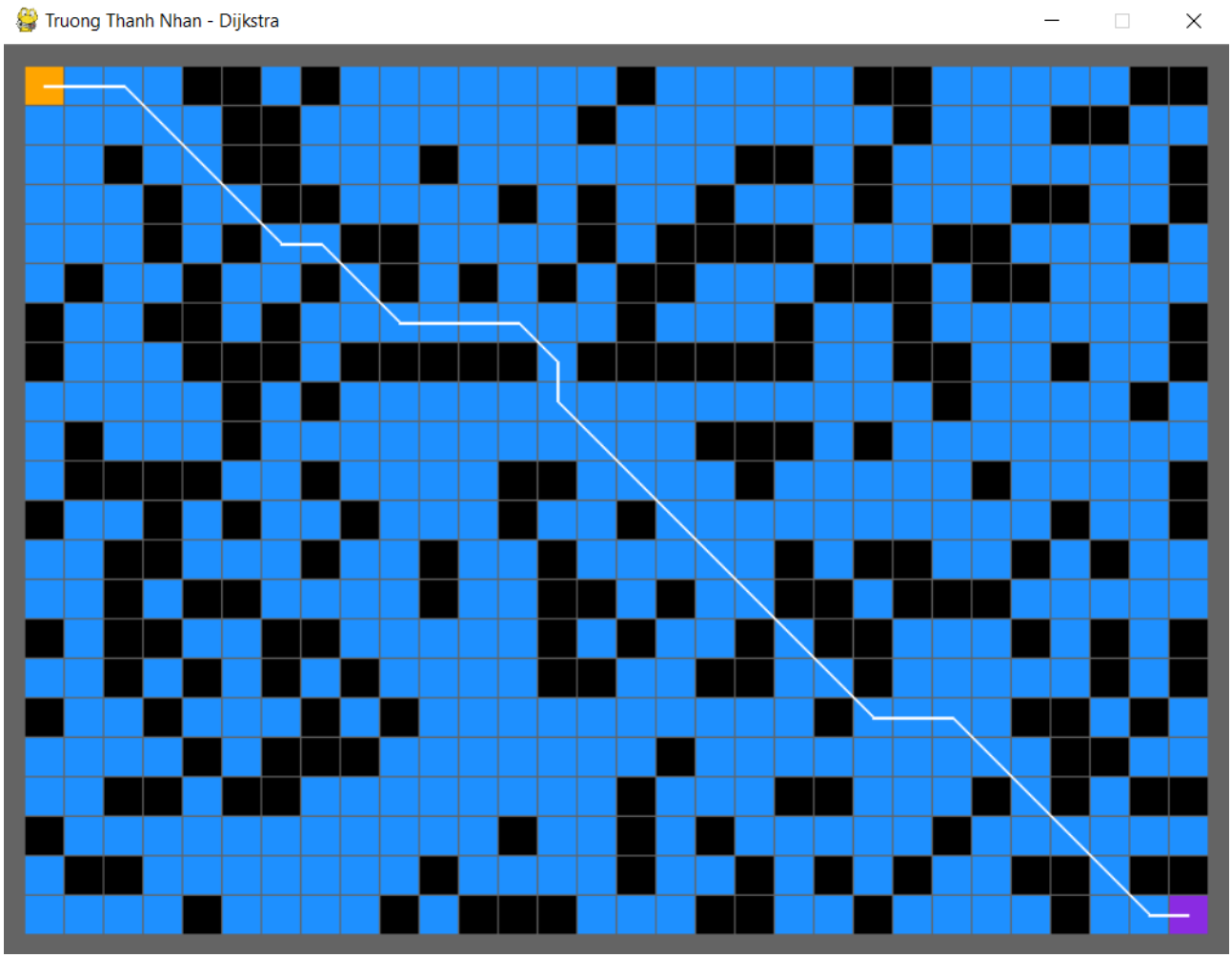
Kết quả tìm kiếm bằng thuật toán Greedy

4.6.2 Nhận xét

- Greedy hay còn gọi là thuật toán tìm kiếm đầu tiên tốt nhất, cũng tương tự với AStar. Thuật toán có xu hướng cho ra kết quả trong thời gian nhanh nhất nhưng không phải lúc nào cũng là tối ưu nhất.
- Greedy tập trung tìm kiếm các node gần với mục tiêu nhất. Tuy nhiên, đôi khi thuật toán sẽ không tối ưu do bị mắc kẹt và rơi vào ngõ cụt. Kết quả tìm kiếm của thuật toán trong bài toán này giống vơi AStar, vì chỉ tìm kiếm những node gần với mục tiêu nên số lượng các node được thăm ít hơn so với BFS và UCS.

4.7 Dijkstra

4.7.1 Hình ảnh kết quả



Kết quả tìm kiếm bằng thuật toán Dijkstra

4.7.2 Nhận xét

- Thuật toán Dijkstra là thuật toán tìm đường đi ngắn nhất giữa 2 điểm trong không gian đồ thị. Thuật toán sẽ không khả thi trong trường hợp không gian được cho quá lớn.
- Theo như kết quả, tuy đường đi tìm được ngắn nhưng thuật toán đã mở rộng phạm vi tìm kiếm, thăm tất cả các node có trong ma trận trước khi tìm được mục tiêu.

5 Một số thuật toán khác

5.1 Greedy

- **Greedy** [6] là một loại tìm kiếm thuộc **best-first search**, là loại tìm kiếm tham lam trong *Imformed search*, tìm kiếm những node gần với mục tiêu nhất.
- Sử dụng hàm đánh giá f(n) = h(n) (heuristic), ước tính chi phí từ n đến mục tiêu.
- Người ta thường sử dụng heuristic là một khoảng cách thẳng: $h_{SLD}(n)$.
- Ngoài ra, cần có kiến thức về các vấn đề thực tế để thấy được h_{SLD} có sự tương quan với khoảng cách ngoài thực tế, giúp cho heuristic hiệu quả hơn.
- Các vấn đề tính hoàn thành, tính tối ưu và độ phức tạp đã được so sánh ở trên.

5.2 Thuật toán Dijkstra

- Thuật toán Dijkstra, mang tên của nhà khoa học máy tính người Hà Lan Edsger Dijkstra vào năm 1956 và ấn bản năm 1959, là một thuật toán giải quyết bài toán đường đi ngắn nhất từ một đỉnh đến các đỉnh còn lại của đồ thị có hướng không có cạnh mang trọng số không âm.
- Mã giả: [9]

```
1   function Dijkstra(Graph, source):
2       for each vertex v in Graph.Vertices:
3           dist[v] <- INFINITY
4           prev[v] <- UNDEFINED
5           add v to Q
6       dist[source] <- 0
7
8       while Q is not empty:
9           u <- vertex in Q with min dist[u]
10          remove u from Q
11
```

```
12            for each neighbor v of u still in Q:
13                alt <- dist[u] + Graph.Edges(u, v)
14                if alt < dist[v]:
15                  dist[v] <- alt
16                  prev[v] <- u
17        return dist[], prev[]
```

- Thuật toán Dijkstra là một thuật toán Greedy (Thuật toán tham lam)
  - Thuật toán Dijkstra bình thường sẽ có độ phức tạp là $O(n^2 + m)$
  - Mặt khác, ta có thể sử dụng kế hợp với cấu trúc heap, khi đó độ phức tạp sẽ là $O((m + n)log(n))$
  - Nếu dùng *Fibonacci Heap* thì độ phức tạp giảm xuống còn $O(m + nlogn)$

  Trong đó, **m** là số cạnh, **n** là số đỉnh của đồ thị đang xét.
- Ưu điểm và hạn chế: Thuật toán Dijkstra là thuật toán đơn giản, dễ hiểu, dễ cài đặt. Tuy nhiên chỉ áp dụng cho những đồ thị có trọng số không âm hoặc đồ thị chứa chu trình âm. Mà các thuật toán Floyd – Warshall, Bellman-Ford có thể làm được.
- Ứng dụng:
  - Tìm đường đi ngắn nhất trên bản đồ.
  - Ứng dụng trong mạng xã hội.
  - Ứng dụng trong hệ thống thông tin di động.
  - Ứng dụng trong hàng không.

5.3 Thuật toán Bellman–Ford

Thuật toán Bellman–Ford là thuật toán để tìm đường đi ngắn nhất từ một đỉnh tới các đỉnh còn lại trong đồ thị có trọng số (trong đó một số cung có thể có trọng số âm).
Thuật toán được đề xuất lần đầu bởi **Alfonso Shimbel (1955)**, nhưng thay vào đó được đặt theo tên của **Richard Bellman** và **Lester Ford Jr.**, là hai

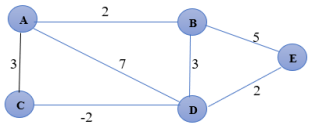
người đã công bố nó lần lượt vào năm 1958 và 1956. **Edward F. Moore** cũng công bố một biến thể của thuật toán vào năm 1959, chính vì thế đôi khi thuật toán này còn được gọi là thuật toán **Bellman–Ford–Moore**.

Thuật toán Dijkstra cũng giải quyết bài toán này và có thời gian chạy nhanh hơn nhưng đòi hỏi trọng số của các cạnh trong đồ thị phải có giá trị không âm. Do đó, ưu điểm của thuật toán Bellman-Ford so với Dijkstra là vẫn có thể giải bài toán tìm đường đi ngắn nhất với trọng số âm.

Mã giả: [10]

```
function BellmanFord(Graph, edges, source)
  distance[source] = 0
  for v in Graph
    distance[v] = inf
    predecessor[v] = undefind
  for i=1...num_vertexes-1
    for each edge (u, v) with wieght w in edges
      tempDist = distance[v]
      if tempDist < distance[v]
        distance[v] = tempDist
        predeccessor[v] = u
  for each edge (u, v) with weight w in edges
    if distance[u] + w < distance[v]
      error: "Negative cycle detected"
```

Ví dụ



Áp dụng thuật toán Bellman-Ford ta tìm được đường đi ngắn nhất từ đỉnh A đến đỉnh E là: A $\rightarrow C \rightarrow D \rightarrow E$.

5.4 Thuật toán Floyd-Warshall

- Thuật toán Floyd-Warshall còn được gọi là thuật toán Floyd được Robert Floyd tìm ra năm 1962 là thuật toán để tìm đường đi ngắn nhất giữa mọi cặp đỉnh. Floyd hoạt động được trên đồ thị có hướng, có thể có trọng số âm, tuy nhiên không có chu trình âm. Ngoài ra, Floyd còn có thể được dùng để phát hiện chu trình âm.

- Mã giả: [11]

```
Floyd-Warshall(W)
    n = W.rows
    D^(0) = W
    for k = 1 to n
        let D^(k) = (d^(k)ij) be a new nxn matrix
        for o = 1 to n do
            d^(k)_ij = min(d^(k-1)_ij, d^(k-1)_ij+d^(k-1)_ij
    return D(n)

```

- Thuật toán Floyd-Warshall bản chất là một thuật toán quy hoạch động.
- Độ phức tạp theo thời gian của thuật toán là: $O(n^3)$ với n là số đỉnh của đồ thị. Với các bài toán yêu cầu tìm đường đi ngắn nhất của toàn bộ cặp cạnh trong đồ thị, Floyd Warshall là một lựa chọn hợp lý so với các thuật toán tìm đường đi ngắn nhất trên nguồn đơn như Dijkstra và Bellman Ford.
- Ưu điểm:
  - Dễ cài đặt và tìm đường đi ngắn nhất giữa mọi cặp đỉnh hiệu quả hơn so với Dijkstra, và được sử dụng cả cho đồ thị có trọng số âm.
  - Thuật toán Floyd có thể dụng để tìm chu trình âm của độ thị (nếu có).
- Hạn chế: Thuật toán chiếm bộ nhớ và độ phức tạp thuật toán lớn nên không dùng để cài đặt với đồ thị phức tạp.
- Ứng dụng
  - Giải bài toán tìm đường đi ngắn nhất từ nguồn đơn với đồ thị kích thước nhỏ có trọng số.
  - In ra đường đi ngắn nhất.
  - Tìm chu trình nhỏ nhất hoặc chu trình âm.
  - Tìm đường đi ngắn nhất giữa các cặp đỉnh của đồ thị có giá trị lớn nhất.

Tài liệu

[1] Search problem. Online. Available. https://en.wikipedia.org/wiki/Search_problem. Accessed: Ngày 30 tháng 10 năm 2023

[2] Search Problem in AI. Online. Available. https://kartikkukreja.wordpress.com/2015/05/23/search-problems-in-ai/. Accessed: Ngày 30 tháng 10 năm 2023

[3] Min-Yen Kan. *Solving problems by searching*. Online. Available. https://web.pdx.edu/~arhodes/ai7.pdf?fbclid=IwAR3J8zbJYLbV-HS5c5XzjnZLuLs0r8UDBfEQk7xKmXAaT8hA4v6uVNngN1o. Accessed: Ngày 30 tháng 10 năm 2023

[4] Aayush Kumar Gupta. *Difference between Informed and Uninformed search in AI*. Online. Available. https://exploringbits.com/difference-between-informed-and-uninformed-search/#Difference_between_Informed_and_Uninformed_Search_in_AI. Accessed: Ngày 30 tháng 10 năm 2023

[5] Russell, Stuart; Norvig, Peter. *Artificial Intelligence_ A Modern Approach (4th edition)*. Online. Available. Accessed: Ngày 30 tháng 10 năm 2023

[6] Min-Yen Kan. *Informed search algorithms*. Online. Available. https://web.pdx.edu/~arhodes/ai8.pdf?fbclid=IwAR3UFzGbDOgsPKgA32WL3o1iuDkcXw4e_Tn4Cl_5Ndlk_kGiH-sUhkpqalEAccessed: Ngày 30 tháng 10 năm 2023

[7] *AStar-Algorithm*. Online. Available. https://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdfAccessed: Ngày 30 tháng 10 năm 2023

[8] *Các thuật toán Informed Search Algorithms* . Online. Available. https://websitehcm.com/cac-thuat-toan-informed-search-algorithms/Accessed: Ngày 30 tháng 10 năm 2023

[9] *Dijkstra's Algorithm – Explained with a Pseudocode Example* . Online. Available. https://www.freecodecamp.org/news/dijkstras-algorithm-explained-with-a-pseudocode-exampleAccessed: Ngày 30 tháng 10 năm 2023

[10] *bellman-ford.pseudo*. Online. Available. https://gist.github.com/travishen/82e7a8aadf38b46f27652d758dc75261Accessed: Ngày 30 tháng 10 năm 2023

[11] *floydWarshall.pdf*. Online. Available. https://www.cs.umd.edu/class/fall2021/cmsc351-0301/files/floydWarshall.pdfAccessed: Ngày 30 tháng 10 năm 2023