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

Nguyễn Phương Nam - 21120504

LAB 1
TÌM KIẾM TRÊN ĐỒ THỊ

BỘ MÔN CƠ SỞ TRÍ TUỆ NHÂN TẠO
CHƯƠNG TRÌNH CHÍNH QUY

GIÁO VIÊN HƯỚNG DẪN
Nguyễn Bảo Long

Tp. Hồ Chí Minh, tháng 10/2023

Lời cam đoan

Tôi xin cam đoan đây là bài làm của riêng tôi. Các phần code được chinh tôi viết. Các phần tài liệu tham khảo đều được đề cập cụ thể.

Lời cảm ơn

Tôi xin chân thành cảm ơn thầy Nguyễn Bảo Long đã tạo điều kiện để tôi có cơ hội tìm hiểu chi tiết về các thuật toán tìm kiếm cũng như thông qua pygame tạo một môi trường thú vị để thực hành các thuật toán này.

Mục lục

Chương 1

Tìm hiểu và trình bày các thuật toán

1.1 Giới thiệu chung về thuật toán tìm kiếm

1.1.1 Thuật toán tìm kiếm là gì?

- Trước khi hiểu về thuật toán tìm kiếm ta sẽ tìm hiểu sơ lược về khái niệm "Agent": Agent là một thực thể có những mục tiêu cụ thể và mục đích của agent là cố gắng đạt được những mục tiêu đó bằng một chuỗi hành động với hiệu quả cao nhất. [4]

- Trong bài toán tìm kiếm, mục tiêu của agent là từ một hoặc nhiều điểm bắt đầu có thể đi đến một hoặc nhiều điểm mục tiêu. Ta sẽ đặt agent ở các điểm bắt đầu đó, cho nó một môi trường để có thể di chuyển và sẽ lập trình cho agent một hoặc nhiều thuật toán nào đó để nó có thể tìm đến được đích đến ta muốn một cách hiệu quả nhất.

- Các thành phần của một bài toán tìm kiếm: [5]

  - Không gian trạng thái: Tất cả các trạng thái của môi trường có thể biến đổi thành trong quá trình agent di chuyển.
  - Trạng thái khởi đầu.
  - Trạng thái đích.
  - Các hành động mà agent có thể thực hiện: cho trạng thái s, ACTION(s) trả về một tập các hành động xuất phát từ s (ví dụ như trái, phải, lên, xuống, ...).

- Mô hình chuyển tiếp: mô tả kết quả của hành động, RESULT(s, a) trả về trạng thái là kết quả sau khi thực hiện hành động a tại trạng thái s.
- Hàm chi phí của hành động: ACTION-COST(s, a, s') trả về chi phí cho việc thực hiện hành động a để đi từ trạng thái s đến s'.

- Pseudo code:
  - Node: được tạo thành từ một trạng thái trong không gian trạng thái và một đường trên đồ thị.
  - Hàm f: trả về một số mô tả "độ tốt" tại node hiện tại. Có thể không xét trong BFS, DFS, hàm cost path trong UCS hay cost path + hàm heuristic trong AStar, ...

Best first search sẽ là khung xây dựng các thuật toán phía sau.

Algorithm 1 Best first search [5]

```
function BestFirstSearch(problem, f) return a node solution or failure
    start_node ← NODE(state=problem.INITAL)
    open_set ← a priority queue ordered by f , with node as an element
    open_set.add(start_node)
    closed_set ← a lookup table holds states has been reached
    while not IsEmpty(open_set) do
        node ← POP(open_set)
        closed_set.add(node.STATE)
        if problem.ISGOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            if child.STATE not in closed_set then
                if child in open_set then
                    if f(child) < f(child in open_set) then
                        replace child in open_set by child
                else open_set.add(child)
    return failure
```

```
function EXPAND(problem, node) yield nodes
    s' ← node.STATE
    for each action in problem.ACTIONS(s) do
        s' ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action,s)
        yield NODE(STATE=s', PARENT=node, PATH-COST=cost)
```

Trong đó

- node.STATE: trạng thái của node;
- node.PARENT: node đã mở ra node hiện tại
- node.PATH-COST: tổng chi phi đi từ gốc đến node hiện tại

1.1.2 Phân biệt tìm kiếm có thông tin và tìm kiếm không có thông tin

Bảng 1.1: Tìm kiếm có thông tin và tìm kiếm không có thông tin [3]

	Tìm kiếm có thông tin	**Tìm kiếm không có thông tin**
Tên gọi khác	Tìm kiếm Heuristic	Tìm kiếm mù
Thông tin	Sử dụng thông tin mô tả mối tương quan giữa trạng thái hiện tại và trang thái đích	Không sử dụng thông tin
Kết quả	Có thể không cho ra kết quả nếu thông tin sai lệch	Luôn cho ra kết quả
Hiệu quả	Hiệu quả cao hơn vì tốn ít tài nguyên chạy, giảm thời gian cho việc tìm kiếm	Tốn nhiều chi phí, thời gian cho việc vét cạn hết các khả năng xảy ra
Thuật toán	Greedy Search, A* Search, AO* Search, Hill Climbing Algorithm, ...	Depth First Search (DFS), Breadth First Search (BFS), Branch and Bound, ...

1.2 DFS- Tìm kiêm theo chiều sâu

- Ý tưởng: Đi sâu vào các nhánh đồ trước khi quay sang nhánh khác. Thuật toán lúc này sử dụng cấu trúc stack, các node thêm vào sau sẽ được lấy ra trước.

- Pesuodo Code:

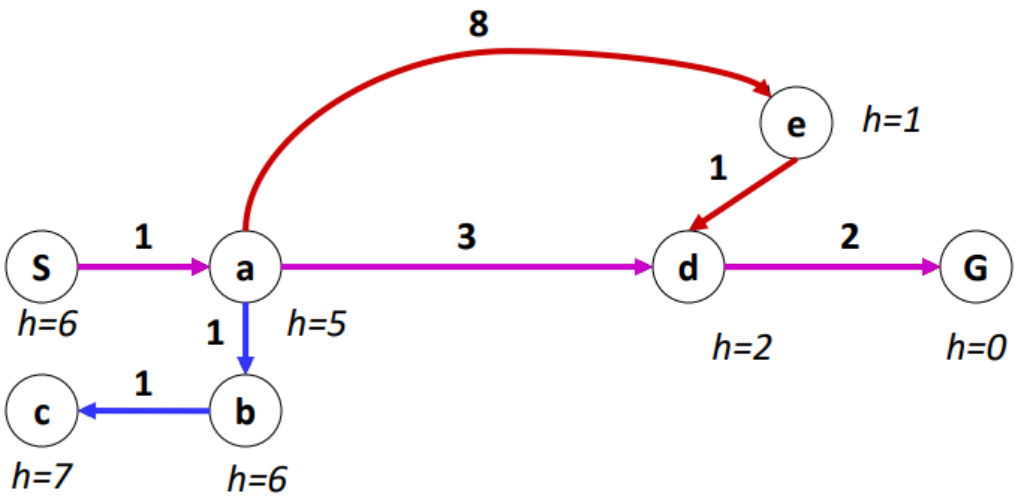
Algorithm 2 Depth first search [5]

```
function DFS(problem) return a node solution or failure
    start_node ← NODE(state=problem.INITAL)
    open_set ← a stack, with node as an element
    open_set.add(start_node)
    closed_set ← a lookup table holds states has been reached
    while not IsEmpty(open_set) do
        node ← POP(open_set)
        closed_set.add(node.STATE)
        if problem.ISGOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            if child not in closed_set and open_set then
                open_set.add(child)
    return failure
```

- Phân tích thuật toán: [2]

  - Độ hoàn chỉnh: DFS không phải là một thuật toán hoàn chỉnh vì nếu trong đồ thị xuất hiện vòng lặp, có khả năng thuật toán sẽ mắc kẹt trong vòng lặp này
  - Tối ưu: DFS không phải là một thuật toán tôi ưu. Nếu có nhiều con đường đến đích, có khả năng DFS sẽ chọn con đường dài hơn
  - Độ phức tạp:
    - Thời gian: $O(b^m)$ với b là số nhánh từ mỗi node, m là độ sâu tối đa của cây
    - Không gian: $O(bm)$ vì DFS chỉ giũ lại các node lân cận của các node đang được xét

- Ví dụ: cho đồ thị bên dưới với trạng thái bắt đầu là S, trạng thái đích là G



  - **B1** open_set: [S]
  - **B2** open_set: [a] → mở rộng node S được node a
    closed_set : [S]
  - **B3** open_set: [e, d, b] → mở rộng node a được node e, d, b
    closed_set : [S, a]
  - **B4** open_set: [e, d, c] → mở rộng node b được node c
    closed_set : [S, a, b]
  - **B5** open_set: [e, d] → mở rộng node c không có node mới
    closed_set : [S, a, b, c]
  - **B6** open_set: [e, G] → mở rộng node d được node G
    closed_set : [S, a, b, c, d]
  - **B7** open_set: [e] → lấy được node G
    closed_set : [S, a, b, c, d]
    → dừng

1.3 BFS - Tìm kiếm theo chiều rộng

- Ý tưởng: Đi hết từng tầng của đồ thị trước khi xuống tầng tiếp theo. Thuật toán lúc này sử dụng cấu trúc queue, các node thêm vào trước

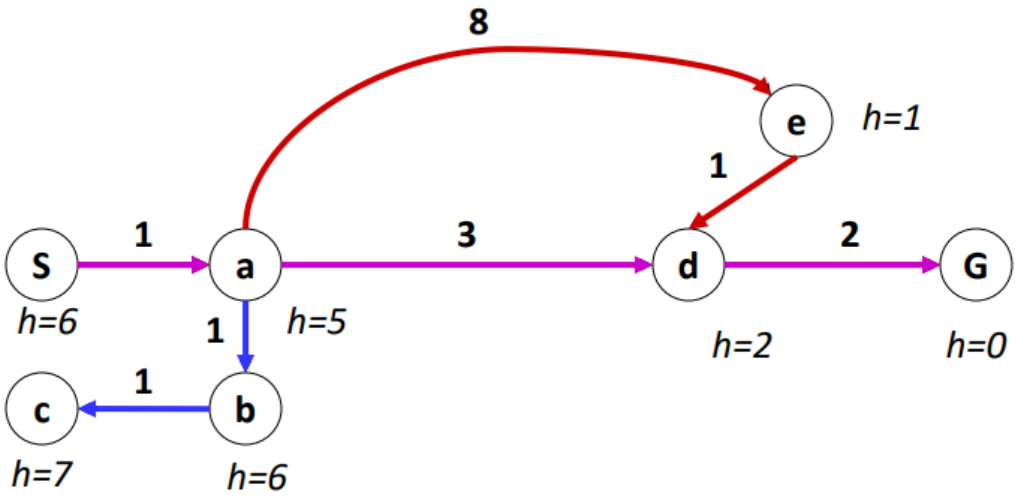
sẽ được lấy ra trước.

- Pesuodo Code: hoàn toàn tương tự DFS nhưng thay stack bằng queue.

Algorithm 3 Breadth first seach[5]

function BFS(problem) **return** a node solution or failure
Similar to DFS but change the stack to the queue

- Phân tích thuật toán: [2]
  - Độ hoàn chỉnh: BFS là một thuật toán hoàn chỉnh.
  - Tối ưu: BFS là một thuật toán tôi ưu. Nó luôn chọn được con đường có số bước đi là ngắn nhất.
  - Độ phức tạp:
    * Thời gian: $O(b^d)$ với b là số nhánh từ mỗi node, d là độ sâu của node đích.
    * Không gian: $O(b^d)$ vì BFS giữ lại tất cả các node ở cùng một tầng. Tại tầng thứ d có b^d node.
- Ví dụ: cho đồ thị bên dưới với trạng thái bắt đầu là S, trạng thái đích là G



  - **B1** open_set: [S]
    closed_set : []

- **B2** open_set: [a] → mở rộng node S được node a
  closed_set : [S]
- **B3** open_set: [e, d, b] → mở rộng node a được node e, d, b
  closed_set : [S, a]
- **B4** open_set: [d, b] → mở rộng node e được không được node mới
  closed_set : [S, a, e]
- **B5** open_set: [b, G] → mở rộng node d được node G
  closed_set : [S, a, e, d]
- **B6** open_set: [G, c] → mở rộng node b được node c
  closed_set : [S, a, e, d, b]
- **B7** open_set: [c] → lấy đươc node G
  closed_set : [S, a, e, d, b]
  → dừng

1.4 USC - Tìm kiếm chi phí đồng nhất

- Ý tưởng: Mỗi bước agent di chuyển sẽ chọn node mà ở đó đường đi ngược về start là ngắn nhất. Thuật toán sử dụng cấu trúc hàng đợi ưu tiên, các node có độ ưu tiên cao hơn sẽ được lấy ra trước. Cụ thể độ ưu tiên ở đây là chi phí đi ngược về start là nhỏ nhất.
- Pesuodo Code: gọi lại hàm best first search đã cài đặt ở trên, thay f bằng hàm g - path cost

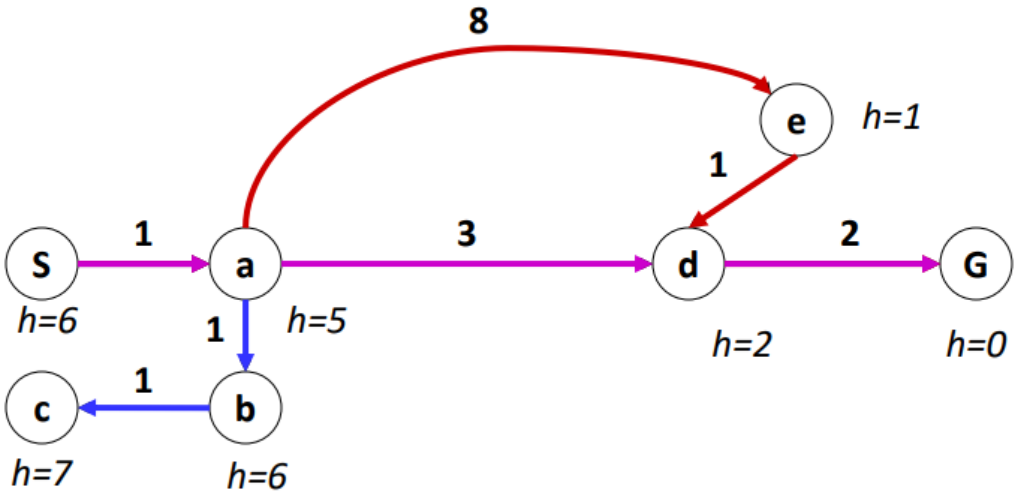
Algorithm 4 Uniform cost search[5]

function UCS(problem) **return** a node solution or failure
  return BESTFIRSTSEARCH(problem, PATH-COST)

- Phân tích thuật toán: [5]
  - Độ hoàn chỉnh: UCS là một thuật toán hoàn chỉnh
  - Tối ưu: UCS luôn cho ra con đường có chi phí là thấp nhất nếu chi phí trên đường đi là dương.

- Độ phức tạp:
  - Không gian: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
  - Thời gian: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

- Ví dụ: cho đồ thị bên dưới với trạng thái bắt đầu là S, trạng thái đích là G.



  - **B1** open_set: [(0,S)]
    closed_set : []
  - **B2** open_set: [(1, a)] → mở rộng node S được node a
    closed_set : [S]
  - **B3** open_set: [(9, e), (4, d), (2,b)] → mở rộng node a được node e, d, b
    closed_set : [S, a]
  - **B4** open_set: [(9, e), (4, d), (3,c)] → mở rộng node b được node c
    closed_set : [S, a, b]
  - **B5** open_set: [(9, e), (4, d)] → mở rộng node c không được node nào
    closed_set : [S, a, b, c]
  - **B6** open_set: [(9, e), (6, G)] → mở rộng node d được node G
    closed_set : [S, a, b, c, d]

- **B7** open_set: [(9, e)] → lấy được node G với chi phí đường đi thấp nhất là 6
  closed_set : [S, a, b, c, d]
  → dừng

1.5 AStar

- Ý tưởng: mỗi bước đi của agent giờ đây ngoài việc so sánh chi phi đi ngược về start còn được chỉ dẫn bởi một giá trị heuristic h(s). Giá trị này cho agent biết node hiện tại gần hay xa so với đích đến cũng có thể xem là chi phí ước tính từ trạng thái hiện tại cho tới đích. Tùy vào bài toán khác nhau mà hàm heuristic này sẽ có cách tính khác nhau cho phù hợp.

- Pesuodo Code: gọi lại hàm best first search đã cài đặt ở trên, f lúc này sẽ băng tổng của hàm cost path g(s) và hàm heuristic h(s)

Algorithm 5 AStar[5]

function ASTAR(problem) **return** a node solution or failure
return BESTFIRSTSEARCH(problem, PATH-COST + HEURISTIC-FUNCTION)

- Phân tích thuật toán:
  - Độ hoàn chỉnh: AStar là một thuật toán hoàn chỉnh
  - Tối ưu: Độ tôi ưu của AStar có thể đảm bảo nếu hàm heuristic đảm bảo admissible và chi phí đường đi là dương
  - Độ phức tạp: độ phức tạp của AStar cũng phụ thuộc rất nhiều vào hàm heuristic. Và có thể được cải thiện rất nhiều nếu sử dụng các admissible heuristic.

- Một vài hàm heuristic xấp xỉ [1]:
  - Manhattan Distance: dùng cho agent có thể đi 4 hướng.
  - Diagonal Distance: dùng cho agent có thể đi 8 hướng.

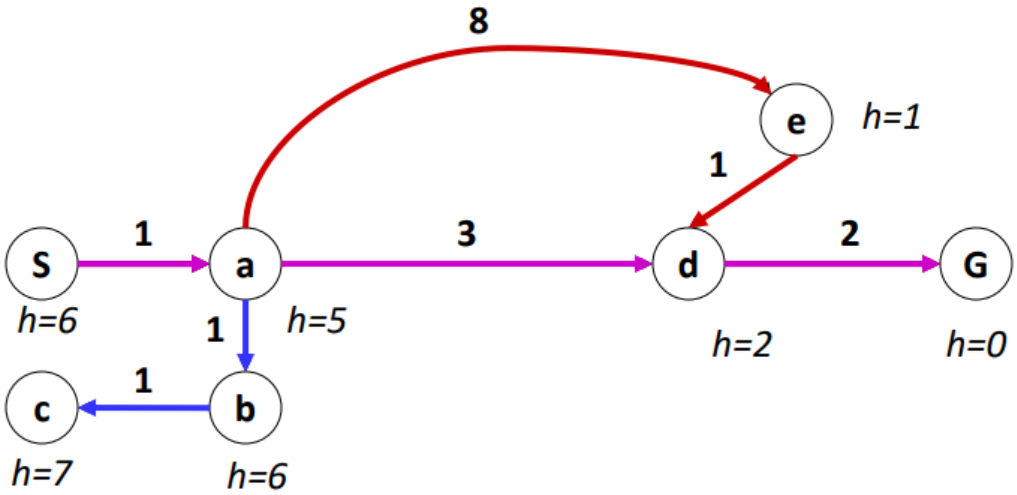
- Euclidean Distance: dùng cho agent có thể đi theo mọi hướng.

- Admissible heuristic [4]: là hàm heuristic mà chi phí ước tính tại trạng thái s (h(s)) luôn không âm và không lớn hơn chi phí thực tế để đi từ s đến đích.

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

  Với h*(s) là chi phí thực tế từ s về đích

- Ví dụ: cho đồ thị bên dưới với trạng thái bắt đầu là S, trạng thái đích là G.



  - **B1** open_set: [(h=6+g=0, S)]
    closed_set : []
  - **B2** open_set: [(h=5+g=1, a)] → mở rộng node S được node a
    closed_set : [S]
  - **B3** open_set: [(h=1+g=9, e), (h=6+g=2, b), (h=2+g=4, d)]
    → mở rộng node a được node e, d, b
    closed_set : [S, a]
  - **B4** open_set: [(h=1+g=9, e), (h=6+g=2,b), (h=0+g=6, G)]
    → mở rộng node d được node G
    closed_set : [S, a, d]
  - **B4** open_set: [(h=1+g=9, e), (h=6+g=2,b)] → lấy được node G với chi phí là 6

closed_set : [S, a, d]
→ dừng

1.6 Greedy best first search

- Ý tưởng: mỗi bước đi của agent giờ chỉ sủ dụng giá trị heuristic h(s) để chỉ dẫn.
- Pesuodo Code: gọi lại hàm best first search đã cài đặt ở trên, f lúc này sẽ được gán bằng hàm heuristic h(s). Lúc này phần so sánh và cập nhập f cho các node trong open_set là dư thừa vì hàm heuristic tại cùng một node lúc nào cũng bằng nhau. Ta sẽ lược bỏ phần đó trong phần cài đặt.

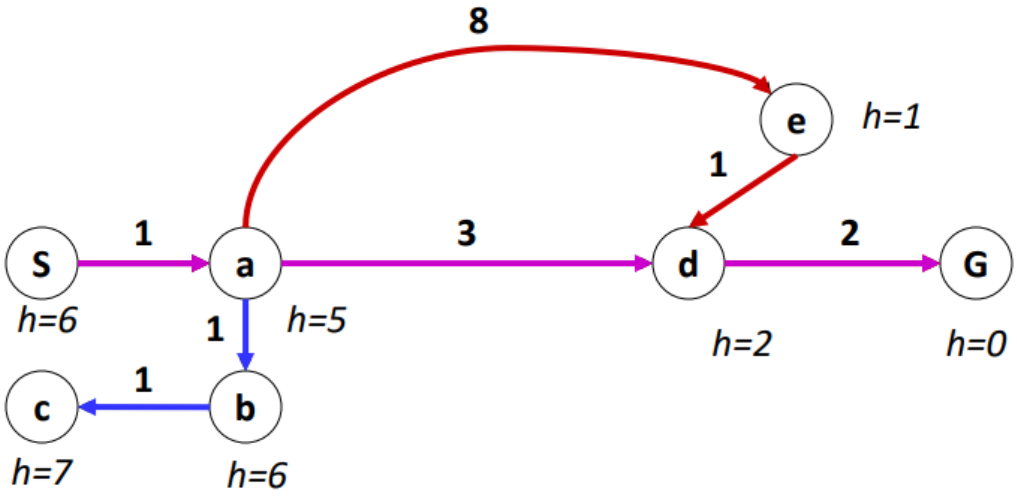
Algorithm 6 Greedy

function GREEDY(problem) **return** a node solution or failure
return BESTFIRSTSEARCH(problem, HEURISTIC-FUNCTION)

- Phân tích thuật toán: [0]
  - Độ hoàn chỉnh: Greedy không phải là một thuật toán hoàn chỉnh vì nó có thể mắc kẹt nếu đồ thị xuất hiện vòng lặp hay không gian tìm kiếm quá phức tạp.
  - Tối ưu: Greedy không phải thụật toán tối ưu vì nó thường chỉ quan tâm đến việc tối ưu cục bộ và điều này không phải lúc nào cũng dẫn đến tôi ưu toàn cục
  - Độ phức tạp: độ phức tạp của Greedy phụ thuộc rất nhiều vào hàm heuristic. Nếu hàm heuristic thích hợp Greedy có thể chạy khá nhanh và đòi hỏi ít không gian.

.

  - **B1** open_set: [(h=6, S)]
    closed_set : []



- **B2** open_set: [(h=5, a)] → mở rộng node S được node a
  closed_set : [S]
- **B3** open_set: [(h=6, b), (h=2, d), (h=1, e)] → mở rộng node a được node e, d, b
  closed_set : [S, a]
- **B4** open_set: [(h=6, b), (h=2, d)] → mở rộng node e không được node nào
  closed_set : [S, a, e]
- **B4** open_set: [(h=6, b), (h=0, G)] → mở rộng node d được node G closed_set : [S, a, e, d]
- **B5** open_set: [(h=6, b)] → lấy được node G
  closed_set : [S, a, e, d]
  → dừng

Chương 2

So sánh các thuật toán

2.1 So sánh UCS, Greedy, AStar

Bảng 2.1: UCS vs Greedy vs AStar

UCS	**Greedy**	**AStar**
Tìm kiếm mù	Tìm kiếm có thông tin	Tìm kiếm có thông tin
Sử dụng hàm g(s) tính chi phí từ start đi đến s	Sử dụng hàm h(s) ước lượng khoảng cách đi đến goal	Sử dụng cả hai hàm g(s) và h(s)
Cập nhập lại node trong open_set khi tìm đc đường đi tốt hơn	Không cần cập nhập lại open_set	Cập nhập lại node trong open_set khi tìm đc đường đi tốt hơn
Chạy khá chậm và tốn nhiều không gian	Chạy nhanh và tốn ít không gian	Chạy khá nhanh, đặc biệt nhanh khi hàm heuristic tốt
Đảm bảo tìm kiếm được đường đi ngắn nhất nếu chi phí trên đường đi đều dương	Không đảm bảo tìm được đường đi ngắn nhất, thường không đáng tin cậy	Đảm bảo tìm kiếm được đường đi ngắn nhất nếu chí phí đường đi dương, hàm heuristic đủ tốt và đảm bảo admissible

2.2 So sánh UCS, Dijkstra

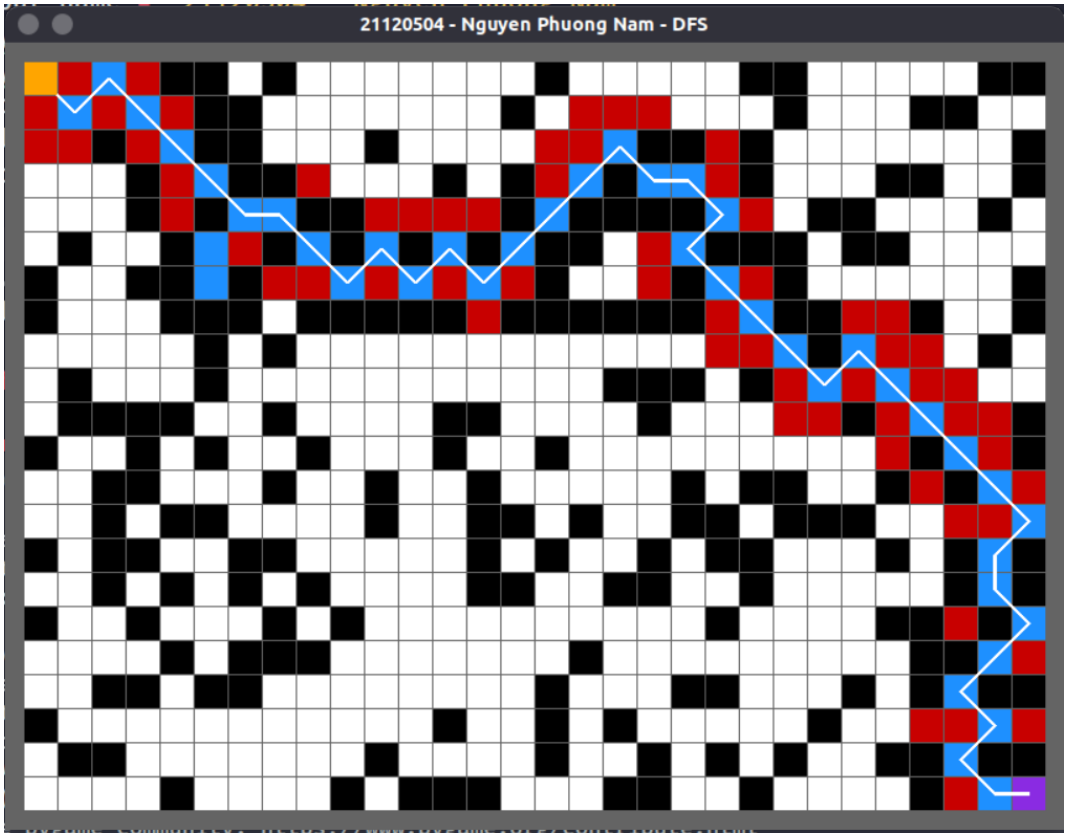
Bảng 2.2: UCS vs Dijkstra [6]

Dijkstra	ten**UCS**
Tất cả các node đều được khởi tạo vào trong queue	Chỉ khởi tạo node start vào trong queue
Dừng khi tất cả các node được lấy ra khỏi queue	Dừng khi node goal được tìm thấy
Tìm kiếm đường đi ngắn nhất từ đỉnh start đến tất cả các đỉnh trong đồ thị.	Chỉ quan tâm đường đi ngắn nhất đi từ start đến goal
Tốn nhiều chi phi cũng như thời gian để tìm đường ngắn nhất cho tất cả các node	Tốn ít chi phí và thời gian hơn
Thường dùng trong đồ thị	Thường dùng trong cây
Chỉ dùng cho đồ thị đẩy đủ thông tin về các cạnh và các node	Cỏ thể dùng được cho cả đồ thị đầy đủ và không đầy đủ (node và cạnh phát sinh trên đường đi)

Chương 3

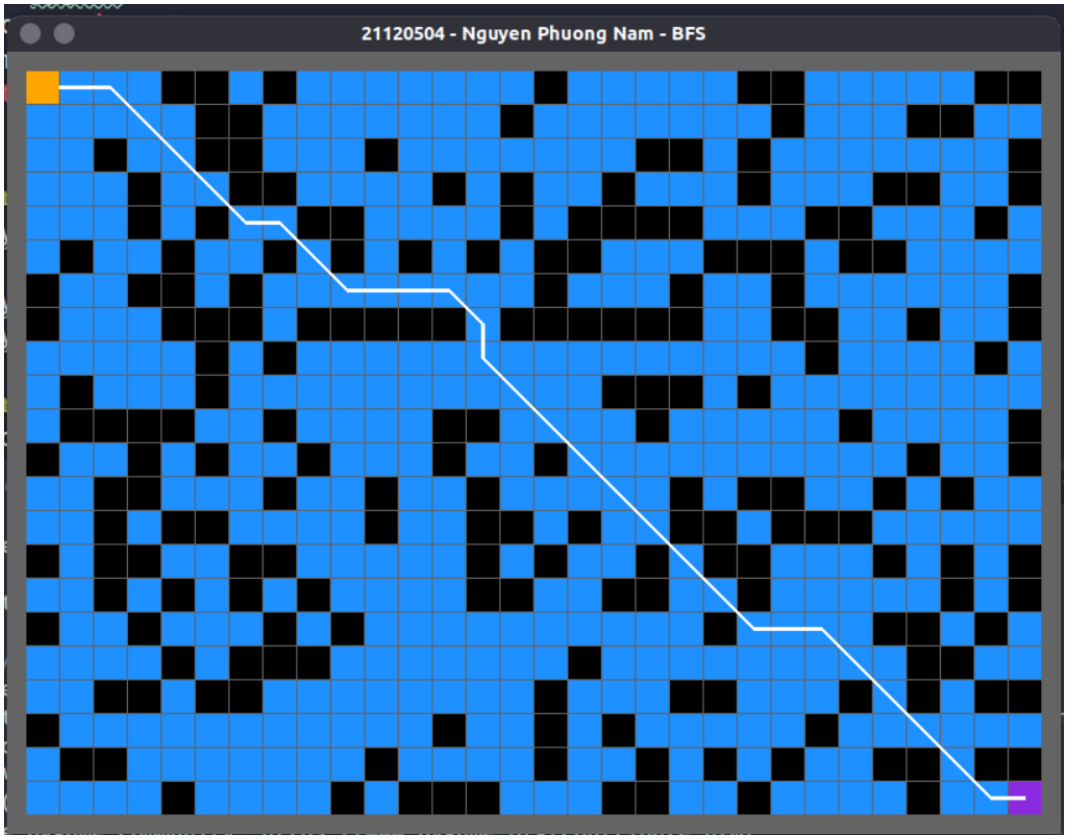
Cài đặt các thuật toán

3.1 DFS



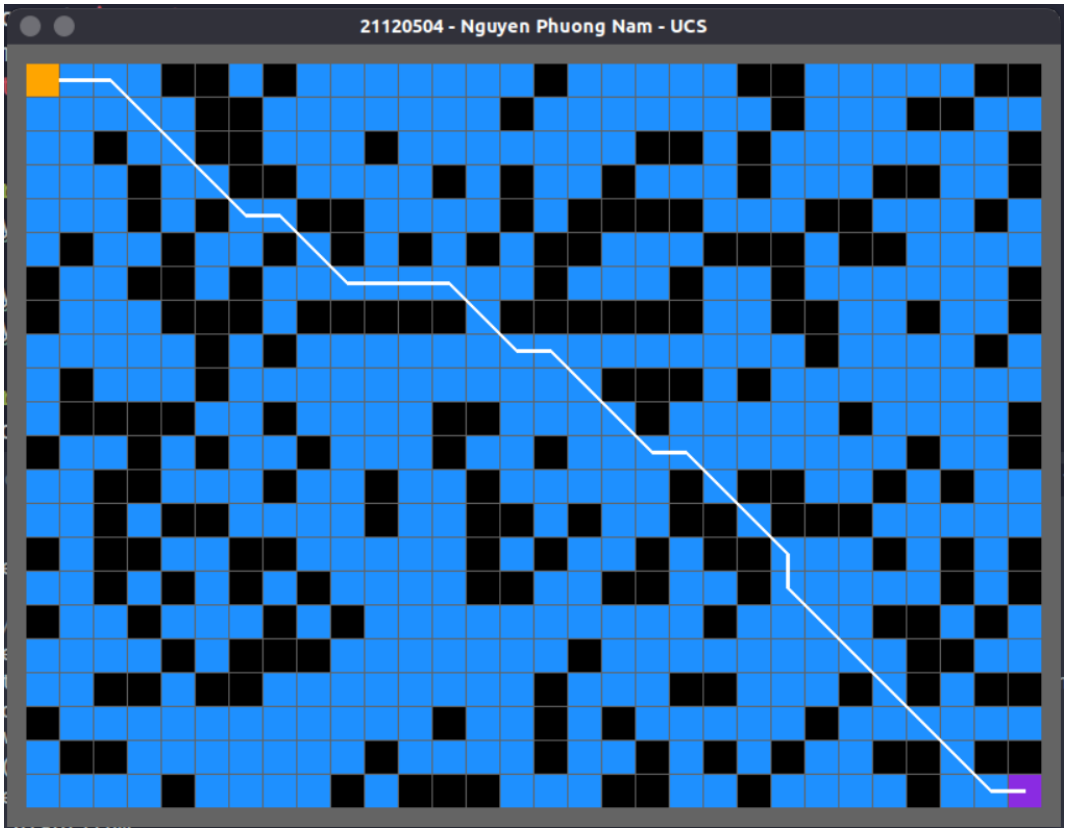
- Thuật toán chạy theo ý tưởng LILO, nên các node vào sau sẽ được mở rộng trước. Trong chương trình thì 2 node kề nằm ở hướng right_up, right_down được thêm vào cuối nên agent hầu như sẽ ưu tiên đi theo 2 hướng này. Do đó đường đi của agent hầu như là đi đường chéo.
- Nhận xét: Ở trường hợp trên hình là do đích nằm ở phía right_down nên ta tìm được kết quả khá ổn (qua 40 ô). Nếu ta đổi thứ tự thêm các node kề vào thì DFS sẽ cho kết quả đường đi khá tệ.

3.2 BFS



- Cài đặt BFS hầu như tương tự DFS, chỉ thay stack bằng queue. Các node sẽ được duyệt theo từng tầng trong cây tìm kiếm.
- Nhận xét: Thuật toán chạy khá chậm do node đích nằm gần như ở tầng cuối của cây. Lúc tìm được node thì các ô trong mê cung đều đã được duyệt qua hết. Nhưng bù lại ta tìm được đường đi với số ô di chuyển là ít nhất (30 ô).

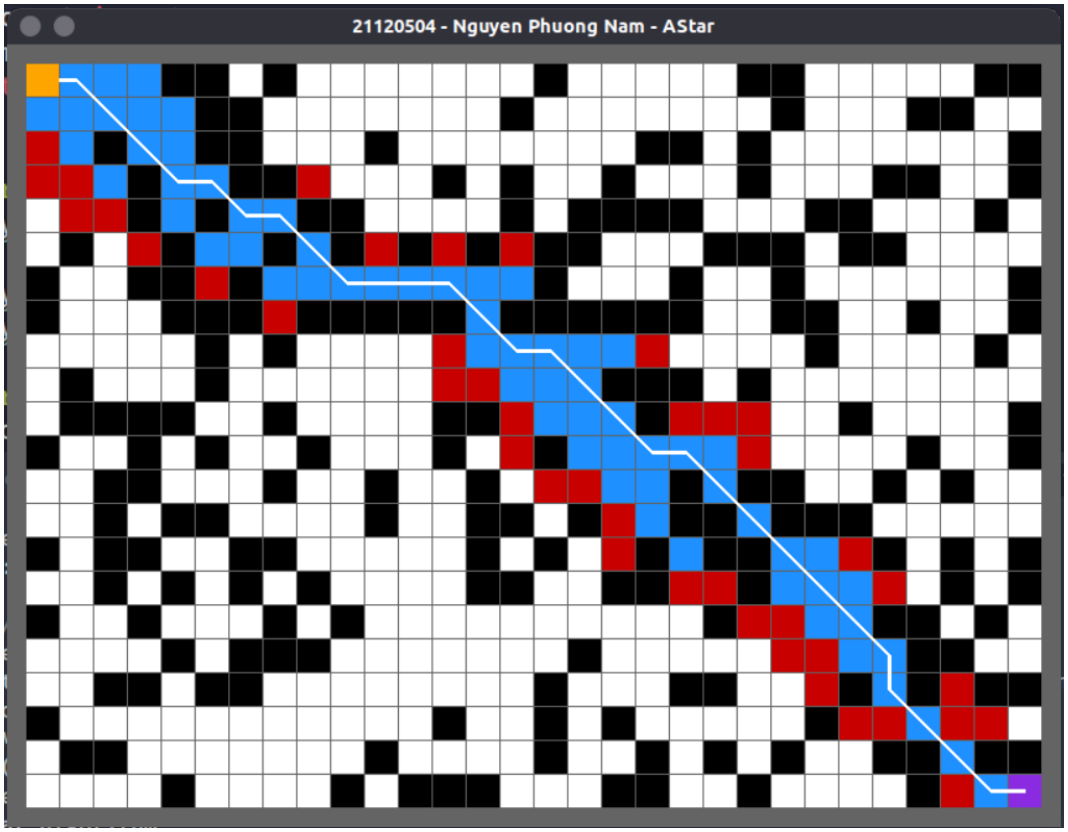
3.3 UCS



- Có 2 loại chi phí được sử dụng trong UCS: 1 cho di chuyển theo đường thẳng và $\sqrt{2}$ cho chi phí duyển theo đường chéo. Vì chi phí chênh lệch không quá lớn nên các duyển của agent có phần tương đồng với BFS (nếu sử dụng 1 - 1 thì thuật toán sẽ trở thành BFS nhưng có chi phí cao hơn)

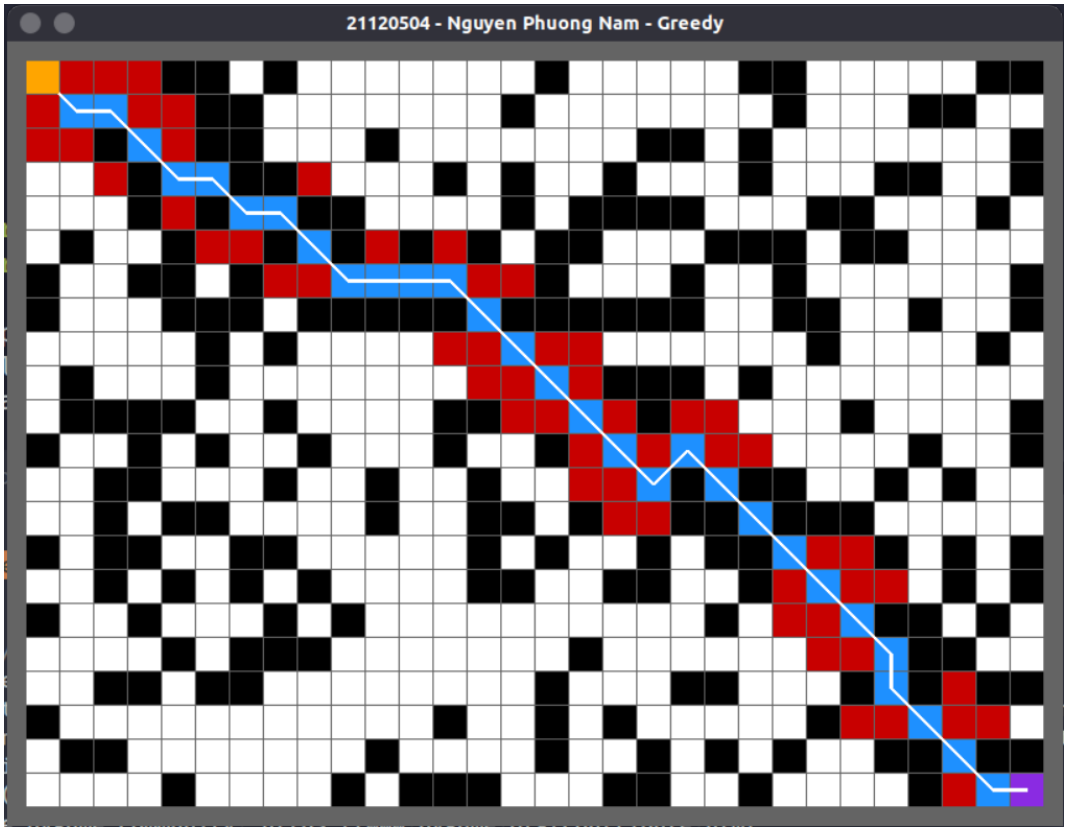
- Thuật toán chạy khá chậm. Lúc tìm được node đích thì các ô trong mê cung đều đã duyệt qua hết. Nhưng bù lại ta tìm được đường đi với số ô di chuyển là ít nhát (30 ô) cũng như chi phí cho đường đi là nhỏ nhất (38.28).

3.4 AStar



- AStar cũng sử dụng 2 loại chi phí như UCS cùng với 1 hàm admissible heuristic là Diagonal Distance. [0]
- Với hàm heuristic phù hợp, thuật toán chạy khá nhanh và mở rộng rất ít ô. Dù vậy thuật toán vẫn cho ra kết quả ngang ngữa với UCS (30 ô và chi phí 38.28)

3.5 Greedy



- Greedy sử dụng Euclidean Distance nên chủ yếu đi theo đường chéo hướng theo phương right_down. [0]
- Ở ví dụ này đường chéo hướng về đích khá rộng mở nên greedy cho kết quả khá tót. Nhưng nếu đích được bọc lại kĩ hơn greedy thường cho kết qua tệ hơn do hiện tượng overestimate (lúc này g(s) = 0)

Tài liệu tham khảo

Danh sách tài liệu

[1] *A* Search Algorithm.* URL: https://www.geeksforgeeks.org/a-search-algorithm/ (visited on 10/20/2023).

[0] *A* Search Algorithm.* URL: https://www.geeksforgeeks.org/a-search-algorithm/ (visited on 10/22/2023).

[2] *Depth-First Search vs. Breadth-First Search.* URL: https://www.baeldung.com/cs/dfs-vs-bfs (visited on 10/19/2023).

[3] *Difference between Informed and Uninformed Search in AI.* URL: https://www.geeksforgeeks.org/difference-between-informed-and-uninformed-search-in-ai/ (visited on 10/19/2023).

[0] *Greedy Best first search algorithm.* URL: https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/ (visited on 10/22/2023).

[4] *Introduction to Artificial Intelligence.* URL: https://courses.fit.hcmus.edu.vn/pluginfile.php/198612/mod_resource/content/1/n1.pdf (visited on 10/18/2023).

[5] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach.* Pearson, 2020.

[6] *What's the difference between uniform-cost search and Dijkstra's algorithm?* URL: https://stackoverflow.com/questions/12806452/whats-the-difference-between-uniform-cost-search-and-dijkstras-algorithm (visited on 10/21/2023).