TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM

KHOA CÔNG NGHỆ THÔNG TIN

CƠ SỞ TRÍ TUỆ NHÂN TẠO

LAB 01

Giáo viên hướng dẫn: *Thầy Lê Hoài Bắc*

Thầy Nguyễn Bảo Long

Lớp: *20_1*

Sinh viên:

Nguyễn Hữu Phúc – 20120161

Năm học 2022-2023

Mục Lục

I. Bài toán tìm kiếm

1. Các thành phần của một bài toán tìm kiếm

Một bài toán tìm kiếm có năm thành phần chính: Q, S, G , succs, cost.

- **Q**: là một tập hữu hạn các trạng thái của bài toán.
- **S**: là một tập khác rỗng các trạng thái ban đầu.
- **G**: là tập hữu hạn các trạng thái đích.
- **Succs:** Q -> P(Q) là một hàm có đầu vào là trạng thái hiện tại và trả về một kết quả là trạng thái tiếp theo (điểm kề với điểm hiện tại).
- **Cost:** Nhận đầu vào là hai trạng thái s và s', trả về chi phí để đi từ s đến s'.

Ngoài các thành phần kể trên, bài toán tìm kiếm còn có thể có thêm các thành phần như:

- **Hành động (action)**: đây là một tập hợp các hành động để từ trạng thái hiện tại đạt được trạng thái tiếp theo.
- **Hàm chuyển tiếp (transition function)**: đây là một hàm giúp ánh xạ trạng thái hiện tại và hành động tương ứng để đến được trạng thái tiếp theo.
- **Tập mở (open set)**: đây là tập hợp chứa các trạng thái đã được khám phá bởi thuật toán và đang chờ được duyệt xem phải là đích đến hay không.
- **Tập đóng (closed set)**: đây là tập hợp chứa các trạng thái đã được thuật toán duyệt qua và không phải là đích.

2. Cách giải một bài toán tìm kiếm

<u>Các bước giải một bài toán tìm kiếm:</u>

Bước 1: Khởi tạo tập mở (open_set) và tập đóng (close set).

Bước 2: Thêm trạng thái xuất phát vào tập mở.

Bước 3: Lặp cho đến khi tập mở rỗng:

- Chọn 1 trạng thái trong tập mở (cách chọn tùy theo từng thuật toán) để kiểm tra.
- Kiểm tra nếu trạng thái đó là đích -> Kết thúc.
- Nếu không là đích -> Mở rộng khám phá đến các trạng thái kề với trạng thái đang được chọn.
- Với mỗi trạng thái kề với trạng được chọn: Nếu không có trong tập đóng và tập mở -> Thêm trạng thái kề đó vào tập mở.
- Thêm trạng thái hiện tại đang xét vào tập đóng.

Bước 4: Kết thúc vòng lặp, trả kết quả không tìm thấy đường đi.

Các ký hiệu thành phần có trong mã giả:

- *S: trạng thái xuất phát.*
- *G: trạng thái đích.*
- *Graph: đồ thị, không gian tìm kiếm.*
- *Open set: tập hợp chứa các trạng thái đang chờ được duyệt.*
- *Closed set: tập hợp chứa các trạng thái đã duyệt qua*

Mã giả:

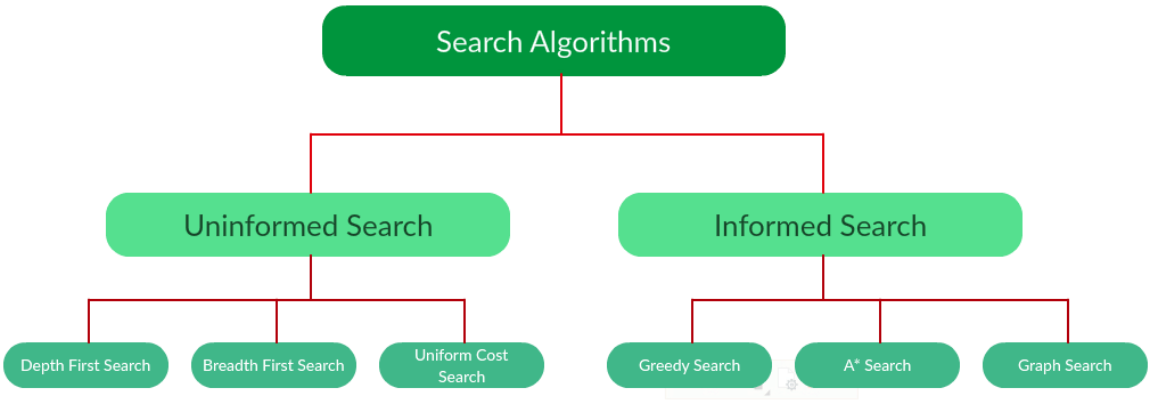
```
function Search (problem):
    // Khởi tạo
    open_set = []
    closed_set = []
    open_set.append(problem.S)// Thêm trạng thái bắt đầu vào danh sách

    // Bắt đầu tìm kiếm
    while open_set:
        current_state = open_set.pop()

        // Kiểm tra trạng thái hiện tại phải là đích đến hay không
        if current_state == problem.G
            return solution
        // Mở rộng phạm vi tìm kiếm
        next_actions = problem.get_next_actions(current_state)
        for action in next_actions:
            next_state = problem.get_next_state(current_state, action)

            if next_state not in open_set:
                open_set.append(next_state)
```

```
                    // Thêm trạng thái hiện tại vào tập đóng
                    closed_set.append(current_state)
                return failure // Không tìm được
```

3. Phân loại các thuật toán tìm kiếm

3.1. Phân loại

Các thuật toán tìm kiếm được chia làm hai nhóm chính: Informed search và Uninformed search.



3.2. Informed search

- **<u>Định nghĩa:</u>** Informed search là một loại thuật toán tìm kiếm có sử dụng hàm Heuristic để định hướng việc tìm kiếm từ điểm hiện tại đến điểm đích. Thuật toán được gọi là "Informed" vì thuật toán này sử dụng thêm thông tin "information" liên quan đến bài toán hiện tại (ví dụ như tọa độ điểm đích) để đưa ra quyết định hướng di chuyển tiếp theo.
  Trong Informed search có sự xuất hiện của hàm Heuristic để tính toán chi phí từ điểm hiện tại đến điểm đích, mức chi phí cao hay thấp bao nhiêu sẽ phụ thuộc vào cách đánh giá của người lập trình. Hàm Heuristic lấy đầu vào là điểm hiện tại và đầu ra là đánh giá mức chi phí để đi được đến điểm đích. Các thuật toán Informed search sử dụng hàm này nhằm tìm và duyệt các điểm/trạng thái được đánh giá là ưu tiên, gần đích đến hơn.
- **<u>Các thuật toán Informed search:</u>**

  Một số thuật toán informed search thường gặp:

  - A*.
  - Hill climbing.
  - Greedy best-first search.

3.3. Uninformed search

- **Định nghĩa:** Uninformed search hay còn gọi là tìm kiếm mù, là các thuật toán tìm kiếm không sử dụng thông tin thêm về đích đến hay vấn đề tìm kiếm để tối ưu hóa việc tìm đường đi. Các thuật toán này từng bước khám phá không gian tìm kiếm một cách có hệ thống bằng cách duyệt các điểm tiếp theo của điểm hiện tại và kiểm tra xem có phải là đích không theo một thuật toán cố định được cài đặt trước.
- **Các thuật toán Uninformed search:**
  - BFS.
  - DFS.
  - UCS.

3.4. Khác nhau giữa Informed search và Uniformed search

Bảng so sánh:

Tiêu chí	Informed search	Uninformed search
Hàm Heuristic	Sử dụng hàm Heuristic để ước tính chi phí đến đích.	Không sử dụng hàm Heuristic.
Phương hướng tìm kiếm	Có phương hướng cụ thể cho quá trình tìm kiếm.	Tìm kiếm không có phương hướng cụ thể.
Khả năng hoàn thành bài toán	Không đảm bảo hoàn thành bài toán nếu không có hàm Heuristic phù hợp.	Đảm bảo hoàn thành nếu tồn tại một đường đi tới điểm đích và không gian tìm kiếm là hữu hạn.
Tối ưu	Vì tính chất vốn có của Heuristic, thuật toán không đảm bảo tìm được giải pháp tối ưu, nếu tìm được giải pháp tối ưu nhất thì chỉ đúng với duy nhất một trường hợp.	Đảm bảo tìm được giải pháp tối ưu nếu chi phí của mỗi hành động là không âm.
Độ phức tạp thời gian	Có thể có độ phức tạp thời gian thấp hơn do sử dụng hàm Heuristic.	Có thể có độ phức tạp thời gian cao hơn vì thuật toán tìm và khám phá từng điểm một cách có hệ thống.

Độ phức tạp bộ nhớ	Có thể có độ phức tạp bộ nhớ cao hơn do sử dụng hàm Heuristic.	Có thể có độ phức tạp bộ nhớ thấp hơn, không cần lưu trữ các kết quả tính toán từ hàm Heuristic.

II. Một số thuật toán tìm kiếm thường gặp

1. DFS (Depth-First Search)

1.1. Ý tưởng

DFS (Depth-First Search) là một thuật toán duyệt đồ thị bằng cách đi theo một hướng xa nhất có thể trước khi quay lui để duyệt theo các hướng đi khác.

Ý tưởng chung của thuật toán tìm kiếm DFS là xuất phát từ một điểm được chọn (được gọi là điểm gốc/nút gốc (root node)) và khám phá, duyệt càng xa càng tốt theo mỗi nhánh của đồ thị trước khi quay lui. Thuật toán kết thúc khi tìm được điểm đích hoặc duyệt hết đồ thị trong trường hợp không tồn tại điểm cần tìm.

1.2. Mã giả

Để cài đặt thuật toán DFS, ta sử dụng cấu trúc dữ liệu ngăn xếp (stack) để lưu danh sách những điểm đã được khám phá và đang đợi để được ghé thăm. Bắt đầu thuật toán, ta sẽ thêm vào stack điểm xuất phát (root node) và lặp lại các bước sau cho tới khi stack hoàn toàn rỗng:

- Pop điểm tiếp theo ra khỏi stack.
- Nếu điểm đó chưa được duyệt, thêm điểm đó vào danh sách các điểm đã được duyệt và tiến hành xử lý kiểm tra xem phải điểm đích không.
- Thêm các điểm kề với điểm hiện tại vào stack.

Mã giả:

```
function DFS(graph, start_node, goal_node):
    // Khởi tạo
    visited = array[]       // Mảng chứa các điểm đã duyệt qua
    stack = []              // Stack chứa các điểm đang đợi được duyệt

    stack.append(start_node)    // Thêm điểm xuất phát vào stack

    while stack:
        current_node = stack.pop()    // Duyệt điểm vừa thêm vào stack
```

```
if current_node == goal_node:  // Xử lý, kiểm tra xem có phải điểm đích
        return True

if current_node not in visited:
            visited.append(current_node)   // Đánh dấu điểm này đã được duyệt

// Lấy danh sách các điểm kề, mở rộng phạm vi tìm kiếm
for neighbor in graph.get_neighbors(current_node):
        if neighbor not in visited:
            stack.append(neighbor)    // Thêm các điểm chưa được duyệt vào stack
return False // Không tìm được kết quả
```

1.3. Đặc điểm thuật toán

- **Khả năng hoàn thành**: Thuật toán DFS không đảm bảo tính hoàn thành khi không gian tìm kiếm là vô hạn hoặc khi duyệt vào một chu trình của đồ thị, thuật toán sẽ lặp vô tận. Tuy nhiên, nếu không gian tìm kiếm là hữu hạn thì thuật toán DFS đảm bảo tìm được đích đến nếu tồn tại. Bên cạnh đó, để khắc phục trường hợp lặp vô tận, ta thường dùng một mảng closed_set [] để lưu danh sách những điểm đã duyệt qua, tránh lặp lại vô tận.
- **Tính tối ưu**: Thuật toán DFS có thể tìm được đường đi tối ưu (trong trường hợp sau khi tìm được điểm đích, vẫn tiếp tục vét cạn đồ thị để tìm đường đi tối ưu) nhưng không đảm bảo. Thuật toán có thể tìm được đường đi tuy nhiên không tối ưu về chi phí vì thuật toán duyệt theo một hướng sâu nhất mà không quay lui cho đến khi tất cả đường đi được duyệt qua.
- **Độ phức tạp thời gian**: **$O(B^{LMAX})$**. (Mỗi nút có B nút con, một đường đi có Lmax nút).
- **Độ phức tạp không gian**: **O(LMAX).** (Mỗi lần duyệt một nhánh, thuật toán sẽ đi đến sâu nhất với độ dài Lmax).
  Trong đó:
  - B là thừa số phân nhánh trung bình (số con trung bình).
  - LMAX là độ dài đường đi dài nhất từ điểm xuất phát đến bất cứ đâu.

2. BFS (Breadth-First Search)

2.1. Ý tưởng

BFS là thuật toán duyệt, tìm kiếm đồ thị theo chiều rộng, có nghĩa là nó khám phá tất cả các nút ở cùng một mức độ sâu trước khi chuyển sang các nút ở mức độ sâu tiếp theo. BFS bắt đầu tại một nút được chỉ định, được gọi là gốc và khám phá tất cả các nút có thể truy cập được từ gốc lần lượt theo chiều rộng trước khi sang độ sâu tiếp theo.

2.2. Mã giả

Để cài đặt thuật toán BFS, ta sử dụng cấu trúc dữ liệu hàng đợi (queue). Bắt đầu thuật toán, ta thêm vào hàng đợi nút xuất phát đầu tiên và lặp lại các bước sau cho đến khi hàng đợi hoàn toàn rỗng:

- Pop điểm tiếp theo ra khỏi queue.
- Nếu điểm đó chưa được duyệt, thêm điểm đó vào danh sách các điểm đã được duyệt và tiến hành xử lý kiểm tra xem phải điểm đích không.
- Thêm các điểm kề với điểm hiện tại vào queue.

Mã giả:

```
function BFS(graph, start_node, goal_node):
    // Khởi tạo
    visited = array[]       // Mảng chứa các điểm đã duyệt qua
    queue = []              // Queue chứa các điểm đang đợi được duyệt

    queue.append(start_node)    // Thêm điểm xuất phát vào queue

    while queue:
        current_node = queue.pop()    // Duyệt điểm vừa thêm vào queue

        if current_node == goal_node:  // Xử lý, kiểm tra xem có phải điểm đích
            return True

        if current_node not in visited:
            visited.append(current_node)   // Đánh dấu điểm này đã được duyệt
```

```
            // Lấy danh sách các điểm kề, mở rộng phạm vi tìm kiếm
            for neighbor in graph.get_neighbors(current_node):
                if neighbor not in visited:
                    queue.append(neighbor)    // Thêm các điểm chưa được duyệt vào queue
    return False // Không tìm được kết quả
```

2.3. Đặc điểm thuật toán

- **Khả năng hoàn thành**: Tương tự DFS, thuật toán BFS không đảm bảo tìm được kết quả nếu không gian tìm kiếm là vô hạn. Tuy nhiên, trong một không gian tìm kiếm hữu hạn, thuật toán BFS đảm bảo tìm được kết quả nếu tồn tại, vì thuật toán sẽ duyệt qua toàn bộ đồ thị theo từng cấp, từng độ sâu và nếu tồn tại điểm đích thì cuối cùng cũng sẽ được tìm thấy.
- **Tính tối ưu**: Thuật toán BFS không đảm bảo tìm được giải pháp tối ưu, mặc dù phần lớn kết quả thường được tối ưu. Lý do là vì thuật toán BFS sẽ khám phá và duyệt qua toàn bộ đường đi từ điểm xuất phát đến điểm đích, chính vì thế khi tìm được đích, đường đi tối ưu nhất cũng sẽ được tìm thấy (đường đi tối ưu về khoảng cách, không đảm bảo tối ưu về chi phí). Tuy nhiên, nếu có nhiều đường đi cùng độ dài thì BFS sẽ không phân biệt sự ưu tiên của mỗi đường đi.
- **Độ phức tạp thời gian**:**$O(\min(N,B^L))$.**
- **Độ phức tạp không gian: $O(\min(N,B^L))$.**
  Trong đó:
  - B là thừa số phân nhánh trung bình (số con trung bình).
  - L độ dài đường đi từ điểm xuất phát đến điểm đích với chi phí thấp nhất.
  - N là số trạng thái trong bài toán (số điểm).

3. UCS (Uniform Cost Search)

3.1. Ý tưởng

Thuật toán UCS là một biến thể của thuật toán Dijkstra, thuật toán này duyệt và tìm kiếm đường đi từ điểm xuất phát đến đích sao cho tối ưu nhất về mặt chi phí. Thuật toán tìm và duyệt các nút theo chiều rộng, khác với thuật toán BFS, UCS sử dụng hàng đợi ưu tiên để lưu trữ các nút đang đợi được duyệt. Độ ưu tiên của hàng đợi này chính là *chi phí tích lũy* đi từ điểm xuất phát đến điểm được xét, chi phí càng thấp, ưu tiên càng cao. Thuật toán UCS duyệt các nút có độ ưu tiên cao nhất trước tiên nên luôn đảm bảo đường đi tìm ra có chi phí tối ưu nhất.

(Chi phí tích lũy từ A đến B = chi phí từ điểm xuất phát đến A + chi phí từ A đến B).

3.2. Mã giả

Thuật toán UCS sử dụng cấu trúc dữ liệu hàng đợi ưu tiên. Bắt đầu thuật toán, ta thêm điểm xuất phát vào hàng đợi ưu tiên và bắt đầu lặp các bước sau cho đến khi hàng đợi hoàn toàn rỗng:

- Pop điểm có chi phi tích lũy thấp nhất ra khỏi queue.
- Nếu điểm đó chưa được duyệt, thêm điểm đó vào danh sách các điểm đã được duyệt và tiến hành xử lý kiểm tra xem phải điểm đích không.
- Từ điểm đang xét, mở rộng ra các điểm kề. Với mỗi điểm kề, nếu điểm đó chưa được duyệt và không nằm trong queue thì tiến hành tính chi phí từ điểm hiện tại đến điểm kề đó và thêm vào queue.

Mã giả:

```
function UCS(graph, start_node, goal_node):
    // Khởi tạo
    visited = array[]       // Mảng chứa các điểm đã duyệt qua
    open = {}               // Dictionary chứa key-value là cặp điểm-chi phí
    open[start_node.value] = 0    // Thêm điểm xuất phát
    cost = []               // Mảng chứa chi phí của mỗi điểm
    cost[start_node.value] = 0     // cost của điểm bắt đầu là 0

    while open:
        current_node = min(open)    // lấy điểm (key) có chi phí (value) nhỏ nhất

        if current_node == goal_node:  // Xử lý, kiểm tra xem có phải điểm đích
            return True

        if current_node not in visited:
            visited.append(current_node)   // Đánh dấu điểm này đã được duyệt

        // Lấy danh sách các điểm kề, mở rộng phạm vi tìm kiếm
        for neighbor in graph.get_neighbors(current_node):
```

if neighbor not in visited and not in open:

cost[neighbor.value] = cost[current_node.value] + cost_from_current_node_to_neighbor // Tính chi phí

open[neighbor.value] = cost[neighbor.value] // Thêm các điểm chưa được duyệt vào mảng

return False // Không tìm được kết quả

3.3. Đặc điểm thuật toán

- **Khả năng hoàn thành:** Tương tự như thuật toán BFS sử dụng hàng đợi queue, thuật toán UCS sẽ tìm được đường đi đến điểm đích trong trường hợp không gian tìm kiếm là hữu hạn. Hơn hết, kết quả đường đi tìm được luôn luôn tối ưu về mặt chi phí.
- **Tính tối ưu:** UCS là thuật toán tối ưu vì nó luôn tìm đường đi rẻ nhất tới mục tiêu vì thuật toán này khám phá các đường đi với chi phí thấp nhất đến từng nút trong đồ thị và mở rộng các nút theo thứ tự tăng dần của chi phí tích lũy của chúng.
- **Độ phức tạp thời gian: $O(\log(Q) * \min(N,B^L))$.**
- **Độ phức tạp không gian: $O(\min(N,B^L))$.**

  Trong đó:

  - B là thừa số phân nhánh trung bình (số con trung bình).
  - L độ dài đường đi từ điểm xuất phát đến điểm đích với chi phí thấp nhất.
  - N là số trạng thái trong bài toán (số điểm).
  - Q là kích cỡ hàng đợi ưu tiên trung bình.

4. A* Search

4.1. Heuristic trong A* search

Trong thuật toán UCS, khi duyệt đến một nút mà các nút kề tiếp nó đều có chi phí là tối ưu nhất, thuật toán sẽ không có cơ sở để đánh giá độ ưu tiên và chọn ngẫu nhiên một trong các nút kề để tiếp tục duyệt và tìm kiếm. Vì thế cần cung cấp thêm cho thuật toán một tiêu chí nhằm đánh giá mức độ ưu tiên và nên chọn nhánh nào khi đứng tại ngã rẽ các nút đều có chi phí tối ưu. Hàm Heuristic ra đời để phục vụ cho mục đích này. Hàm Heuristic trong tìm kiếm đồ thị sẽ tính toán chi phí đi từ điểm nút bất kỳ để đi được đến điểm đích.

Với hàm này, khi UCS gặp các nút có cùng chi phí tối ưu sẽ tiếp tục có thêm tiêu chí để chọn nút nào là nút cần đi (nút có chi phí đi đến đích thấp nhất). Vì hàm Heuristic sử dụng thông tin biết trước từ bài toán (vị trí điểm đích) nên thuật toán A* thuộc nhóm Informed search.

Tuy nhiên, hàm Heuristic mang tính chất của Heuristic nên việc chọn cách đánh giá một hàm Heuristic sẽ quyết định thuật toán A* search có tối ưu về mặt chi phí, có tìm được điểm đích hay không.

Trong thuật toán A* search, chi phí tính bởi hàm Heuristic thường được ký hiệu là *h*, chi phí tích lũy của thuật toán UCS ký hiệu là *g*, và chi phí để đánh giá tìm được đi tối ưu trong A* là *f = g + h.*

4.2. Một số hàm Heuristic trong A* search

- Trong trường hợp tại một nút trong đồ thị, ta chỉ có thể di chuyển theo các phương dọc và ngang (trên, dưới, trái, phải), hàm Heuristic dùng khoảng cách Manhattan để đánh giá khoảng cách giữa một điểm bất kỳ và điểm đích.

  h = abs (current_cell.x – goal.x) + abs (current_cell.y – goal.y)
- Trong trường hợp tại một nút, ta có thể di chuyển theo 4 phương (8 hướng), hàm Heuristic sử dụng khoảng cách Diagonal để đánh giá khoảng cách từ nút đó đến điểm đích.

  dx = abs(current_cell.x – goal.x)
  dy = abs(current_cell.y – goal.y)

  *h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)*

  D là độ dài của một nút (thường = 1) và D2 là khoảng cách Diagonal giữa mỗi nút (thường = sqrt(2)).
- Trong trường hợp tại một nút, ta có thể di chuyển tự do theo bất kỳ hướng nào trong không gian, hàm Heuristic sử dụng khoảng cách Euclide để đánh giá khoảng cách từ nút đó đến đích:

  h = sqrt ((current_cell.x – goal.x)2 + (current_cell.y – goal.y)2).

4.3. Ý tưởng

Thuật toán tìm kiếm A* là một trong những kỹ thuật tốt nhất và phổ biến được sử dụng trong tìm đường và duyệt đồ thị. Khác với các thuật toán tìm kiếm các (DFS, BFS, UCS) thuộc loại uninformed search, thuật toán A* là thuật toán tìm kiếm thông minh, có "bộ não" riêng để tìm đường đi, nhờ sử dụng hàm Heuristic. Đây là thuật toán thuộc loại informed search.

Về mặt ý tưởng, khi duyệt đến nút ở vị trí hiện tại, để xác định nút tiếp theo cần duyệt, thuật toán A* dựa vào giá trị f của nút tiếp theo phải đi. Giá trị này là tổng của g và h, trong đó g là chi phí tích lũy để đi từ điểm xuất phát đến điểm tiếp theo đó, và h là hàm Heuristic đánh giá khoảng cách từ điểm đó đến điểm đích. Thuật toán A* sẽ lựa chọn điểm tiếp theo sao cho giá trị f của điểm đó là nhỏ nhất trong tất cả các sự lựa chọn mà thuật toán có thể chọn.

4.4. Mã giả

Thuật toán A* sử dụng cấu trúc dữ liệu hàng đợi. Bắt đầu thuật toán, ta thêm điểm xuất phát vào hàng đợi và bắt đầu lặp các bước sau cho đến khi hàng đợi hoàn toàn rỗng:

- Pop điểm tiếp theo ra khỏi queue, điểm này có chi phí f là thấp nhất.
- Nếu điểm đó chưa được duyệt, thêm điểm đó vào danh sách các điểm đã được duyệt và tiến hành xử lý kiểm tra xem phải điểm đích không.
- Từ điểm đang xét, mở rộng ra các điểm kề. Với mỗi điểm kề, nếu điểm đó chưa được duyệt và không nằm trong queue thì tiến hành tính chi phí f của điểm đó và thêm điểm đó vào queue.

Mã giả:

```
function UCS(graph, start_node, goal_node):
    // Khởi tạo
    visited = array[]       // Mảng chứa các điểm đã duyệt qua
    open = {}                       // Dictionary chứa key-value là cặp điểm-chi phí f
    open[start_node.value] = 0          // Thêm điểm xuất phát
    g_cost = []             // Mảng chứa chi phí tích lũy của mỗi điểm như ucs
    g_cost[start_node.value] = 0        // cost của điểm bắt đầu là 0
    f_cost = []                         // Mảng chứa chi phí f của mỗi điểm
    f_cost[start_node.value]

    while open:
        current_node = min(open)    // lấy điểm (key) có chi phí (value) nhỏ nhất

        if current_node == goal_node:  // Xử lý, kiểm tra xem có phải điểm đích
            return True

        if current_node not in visited:
            visited.append(current_node)   // Đánh dấu điểm này đã được duyệt

        // Lấy danh sách các điểm kề, mở rộng phạm vi tìm kiếm
```

```
            for neighbor in graph.get_neighbors(current_node):
                if neighbor not in visited and not in open:
                    g_cost[neighbor.value] = g_cost[current_node.value] +
                    cost_from_current_node_to_neighbor // Tính chi phí
                    f_cost[neughbor.value] = g_cost[neighbor.value] +
                    h_cost(neighbor, goal_node)
                    open[neighbor.value] = f_cost[neighbor.value]    // Thêm các
                    điểm chưa được duyệt vào mảng
        return False // Không tìm được kết quả
```

4.5. Đặc điểm thuật toán:

- **Khả năng hoàn thành:** Thuật toán A* là thuật toán có khả năng hoàn thành kể cả trong không gian tìm kiếm vô tận. Với hàm Heuristic giúp xác định khoảng cách từ điểm đang xét đến điểm đích, chỉ cần tồn tại đích đến, thuật toán luôn cho ra kết quả là đường đi tối ưu.
- **Tính tối ưu:** Thuật toán A* là thuật toán tối ưu. Chỉ cần hàm Heuristic đánh giá đủ tốt thì thuật toán cho ra kết quả đường đi không bao giờ vượt quá chi phí thực tế.
- **Độ phức tạp thời gian, không gian:** Độ phức tạp của thuật toán phụ thuộc vào hàm Heuristic được sử dụng.

III. Các thuật toán tìm kiếm khác

1. Dijikstra Search

Thuật toán Dijkstra là thuật toán duyệt và tìm kiếm đường đi ngắn nhất trong đồ thị. Thuật toán là một biến thể của BFS, sử dụng hàng đợi ưu tiên (priority queue) để lưu trữ danh sách các điểm cần duyệt. Tại mỗi ngã rẽ, thuật toán lựa chọn điểm có độ ưu tiên cao nhất: điểm có chi phí tích lũy (chi phí từ điểm xuất phát đến điểm đó) là thấp nhất. Với cách lựa chọn như vậy, thuật toán đảm bảo tìm được đường đi từ điểm xuất phát đến điểm đích với chi phí là tối ưu nhất.

2. Greedy Search

Thuật toán Greedy search là thuật toán thuộc nhóm các thuật toán informed search có sử dụng hàm Heuristic.Thuật toán này hoạt động như sau: tại mỗi điểm, thuật toán sẽ đánh giá chi phí của điểm đang xét so với điểm đích, sau đó chọn điểm có chi phí thấp nhất làm hướng đi tiếp theo mà không quan tâm đến chi phí đi từ điểm xuất phát đến điểm hiện tại. Với mỗi hành động/bước đi, sự lựa chọn của thuật toán này là tối ưu nhất về mặc chi phí (đảm bảo tối ưu cục bộ). Tuy nhiên khi xét về cả hành trình, đường đi được tìm thấy bởi thuật toán không đảm bảo tối ưu nhất về mặt chi phí (không đảm bảo tối ưu toàn cục). Thuật toán thường được sử dụng cho các bài toán tìm kiếm trong đó

giải pháp toàn cục xấp xỉ với giải pháp tối ưu cục bộ và không quan tâm đến tối ưu toàn cục. Khác với thuật toán A* có hàm chi phí f = g + h, Greedy search có hàm chi phí f = h.

Khác với thuật toán Dijkstra và UCS tính chi phí tích lũy từ điểm xuất phát đến điểm đang xét, chi phí ở thuật toán Greedy là chi phí giữa điểm hiện tại và điểm đích.

IV. So sánh các thuật toán

1. UCS, Greedy và A*

	UCS	**Greedy**	**A***
Hàm Heuristic	Không sử dụng.	Có sử dụng.	Có sử dụng.
Loại tìm kiếm	Uninformed Search.	Informed Search.	Informed Search.
Cách tính chi phí	Từ điểm xuất phát đến điểm hiện tại.	Từ điểm hiện tại đến điểm đích.	Tổng chi phí từ điểm xuất phát đến hiện tại với chi phí từ điểm hiện tại đến điểm đích.
Khả năng hoàn thành bài toán	Không đảm bảo tìm thấy nếu không gian là vô hạn.	Có thể tìm thấy.	Có thể tìm thấy điểm đích.
Tối ưu	Nếu tìm được đích đến thì đường đi là tối ưu về chi phí, không tối ưu về thời gian.	Tối ưu về chi phí, thời gian nếu có hàm Heuristic hợp lý.	Tối ưu về chi phí, thời gian nếu có hàm Heuristic hợp lý.
Độ phức tạp thời gian	$O(\log(Q) * \min(N,B^L))$.	$O(B^{LMAX})$ với hàm Heuristic tốt, độ phức tạp có thể giảm còn $O(B.h)$ với h là khoảng cách từ điểm đang xét so với đích.	Phụ thuộc vào hàm Heuristic.
Độ phức tạp không gian	$O(\min(N,B^L))$.	$O(B^{LMAX})$.	Phụ thuộc hàm Heuristic.

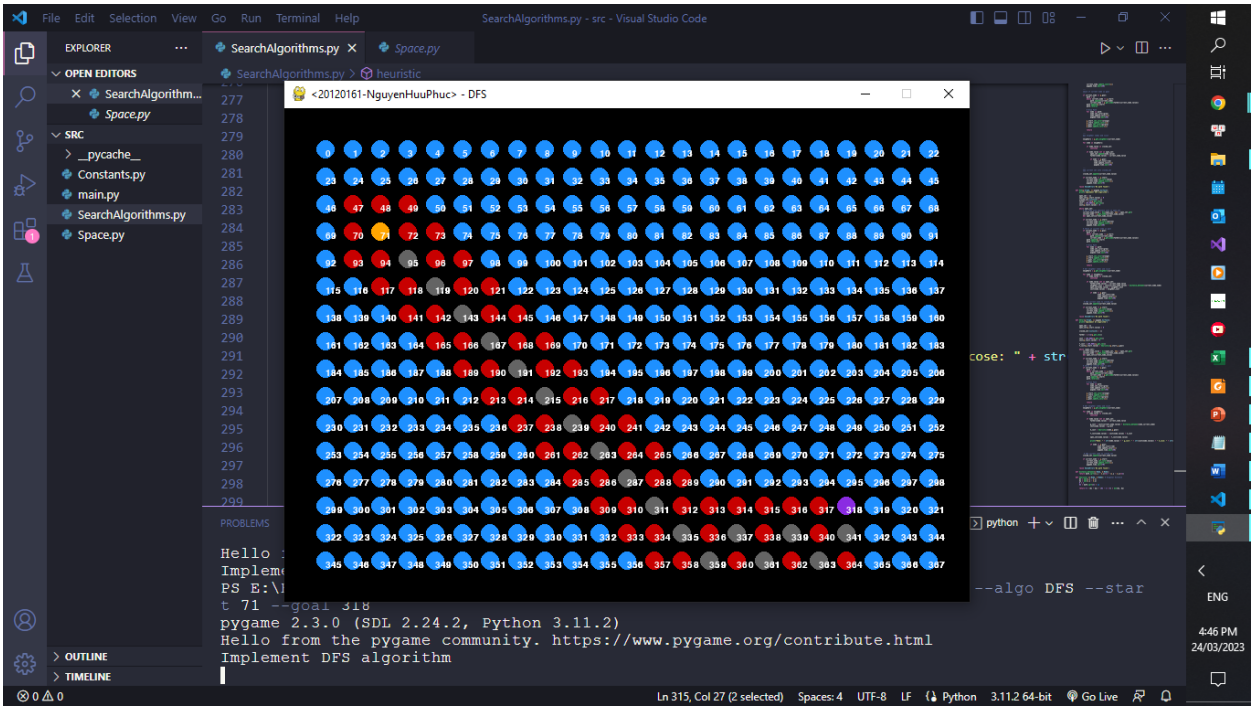
Tốc độ	Rất chậm và tốn bộ nhớ.	Nhanh và hiệu quả về mặt bộ nhớ. Tuy nhiên không đảm bảo về đường đi tối ưu.	Nhanh hơn UCS và Greedy, đảm bảo đường đi tối ưu nhất.
Cấu trúc dữ liệu	Hàng đợi ưu tiên.	Hàng đợi ưu tiên.	Hàng đợi ưu tiên.
Cài đặt	Trung bình – Khá.	Dễ cài đặt.	Trung bình – Khá.

2. UCS và Dijkstra

	UCS	**Dijkstra**
Mục tiêu	Tìm đường đi ngắn nhất từ điểm xuất phát đến điểm đích.	Tìm đường đi ngắn nhất đến mọi điểm trong đồ thị.
Phạm vi, quy mô tìm kiếm	Duyệt tìm đường đi đến điểm đích, tìm được thì dừng lại.	Duyệt cả đồ thị, không dừng lại khi tìm được đích.
Tốc độ	Nhanh hơn.	Chậm hơn vì duyệt toàn đồ thị.
Chiếm bộ nhớ	Ít hơn, chỉ lưu những điểm được duyệt trên đường tìm đến đích	Nhiều hơn vì phải lưu cả đồ thị

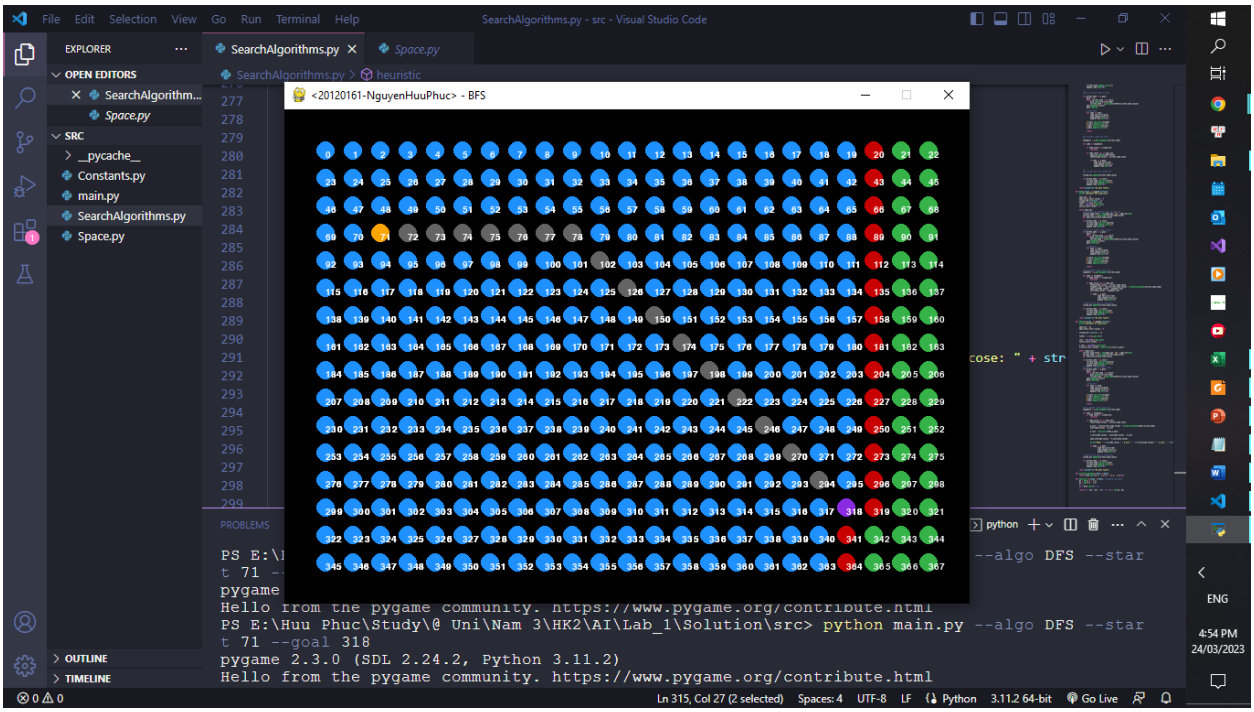
V. Cài đặt thuật toán

1. DFS



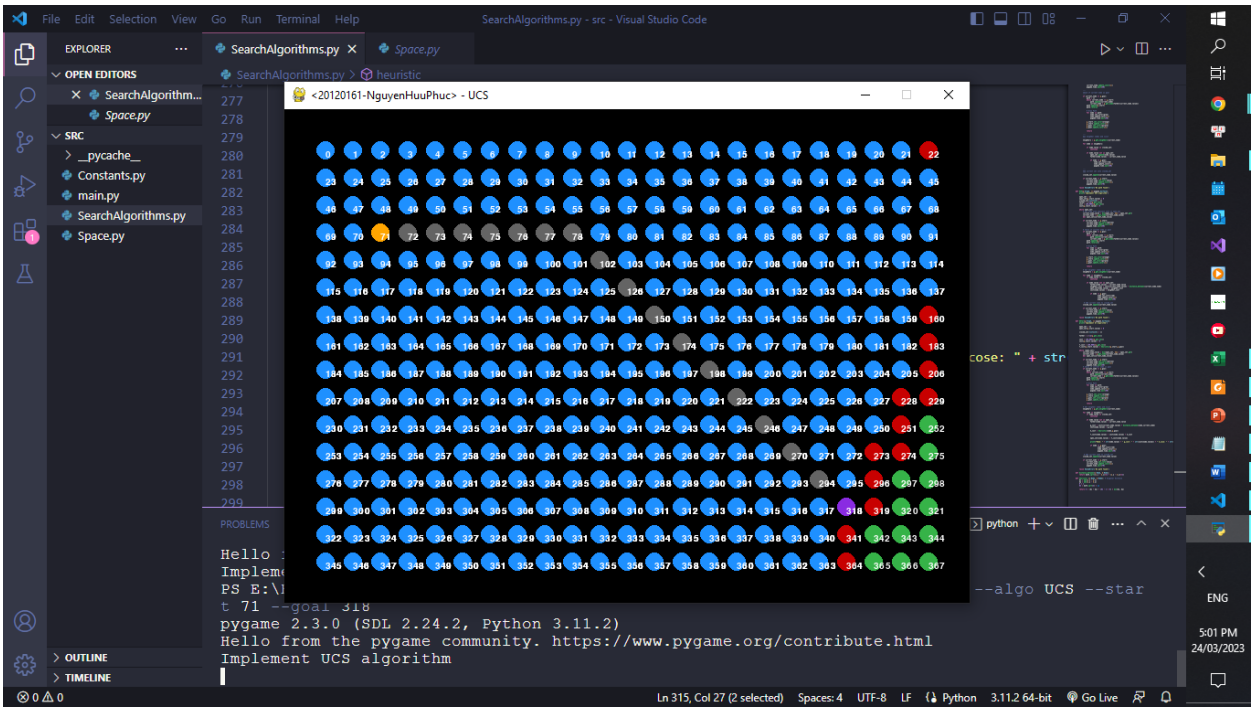
Theo như yêu cầu của đề bài, màu xanh dương là những điểm đã duyệt qua. Khi bắt đầu ,các điểm đã duyệt qua được đánh dấu thành màu xanh dương theo một đường dài khi không còn đường thì bắt đầu lan sang những hướng đi khác.

2. BFS



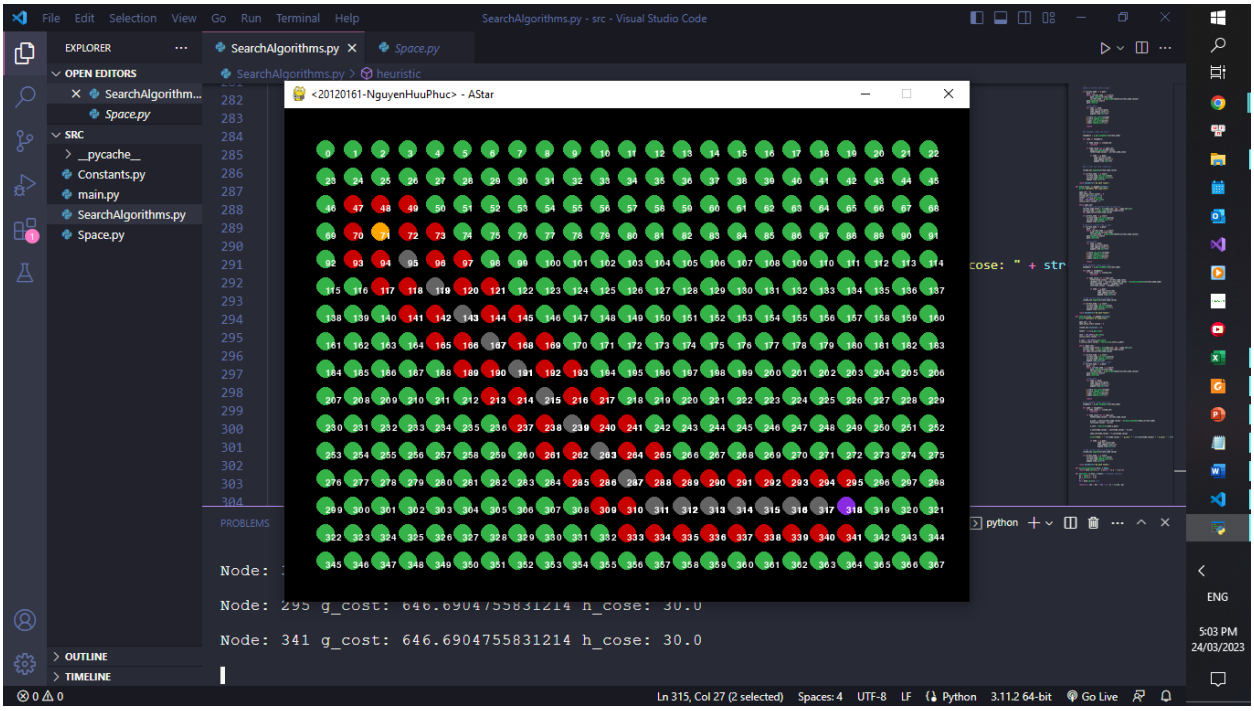
Bắt đầu từ điểm màu vàng, các ô lần lượt chuyển sang màu xanh theo dạng hình vuông bao quanh điểm xuất phát và mở rộng dần ra cho đến khi tìm được điểm đích.

3. UCS



Cách thức mở rộng tương tự như BFS, tuy nhiên cách điểm màu xanh mở rộng ra theo dạng hình tròn bào quanh điểm xuất phát, đến khi gặp điểm đích thì dừng lại.

4. A*



Các điểm màu xanh mở rộng theo hướng đi thẳng tới điểm đích, không đi loằng ngoằng.

VI. Tài liệu liên quan

1. Link Demo

Youtube: https://www.youtube.com/watch?v=bx9jcjms5GY

2. Tham khảo

Chat GPT.

Slide bài giảng của thầy Lê Ngọc Thành.

Trang bìa báo cáo của bạn Võ Duy Trường 20120232.

Heuristic trong A*: https://www.geeksforgeeks.org/a-search-algorithm/

Các thuật toán search: https://www.geeksforgeeks.org/search-algorithms-in-ai/