TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẠI HỌC QUỐC GIA TP.HCM

KHOA CÔNG NGHỆ THÔNG TIN

MÔN CƠ SỞ TRÍ TUỆ NHÂN TẠO

Lab01 - Uninformed Search & Informed Search

MỤC LỤC

THÔNG TIN SINH VIÊN THỰC HIỆN

- Họ tên: Khúc Khánh Đăng
- Mã số sinh viên: 20120262
- Khoá: K20

I. CÁC CÔNG VIỆC CẦN THỰC HIỆN

- Tìm hiểu thông qua tài liệu tham khảo.
- Cài đặt các công cụ, môi trường hỗ trợ cho việc lập trình Python (ngôn ngữ sử dụng trong bài lab) và các thư viện bổ trợ
- Đưa ra các giải thuật giả (pseudo-code).
- Cài đặt các thuật giải tìm kiếm (giải thuật chính) vào file đã được đưa.
- Xử lý sửa lỗi chương trình.
- Viết báo cáo, thực hiện video demo về quy trình làm việc và cài đặt chương trình, chạy chương trình.

II. ĐÁNH GIÁ MỨC ĐỘ HOÀN THIỆN YÊU CẦU ĐỒ ÁN

Yêu cầu	Đánh giá mức độ hoàn thiện
Tìm hiểu và trình bày được các thuật toán tìm kiếm trên đồ	100 %
So sánh các thuật toán	100 %
Cài đặt các thuật toán	100 %
Tìm hiểu thêm các thuật toán khác	60 %

III. CÁC THÀNH PHẦN CỦA 1 BÀI TOÁN TÌM KIẾM VÀ CÁCH GIẢI

1. Các thành phần

Một bài toán tìm kiếm có thể được định nghĩa bằng 5 thành phần:

- Trạng thái bắt đầu (Initial state)
- Mô tả các hoạt động (action) có thể thực hiện
- Mô hình di chuyển (transition model): mô tả kết quả của các hành động
    - Thuật ngữ successor tương ứng với các trạng thái có thể di chuyển được với một hành động duy nhất.
    - Trạng thái bắt đầu, hành động và mô hình di chuyển định nghĩa không gian trạng thái (state space) của bài toán.
    - Không gian trạng thái hình thành nên một đồ thị có hướng với đỉnh là các trạng thái và cạnh là các hành động.
    - Một đường đi trong không gian trạng thái là một chuỗi các trạng thái được kết nối bằng một chuỗi các hành động.
- Kiểm tra đích (goal test): Xác định một trạng thái có là trạng thái đích.
- Một hàm chi phí đường đi (path cost) gán chi phí với giá trị số cho mỗi đường đi

2. Cách giải một bài toán tìm kiếm nói chung

Một lời giải (solution) là một chuỗi hành động di chuyển từ trạng thái bắt đầu cho đến trạng thái đích (một lời giải tối ưu có chi phí đường đi thấp nhất trong số tất cả lời giải).

Các bước chính để giải một bài toán tìm kiếm:

- Xác định mục tiêu cần đạt đến (goal formulation)
    - Là tập hợp các trạng thái (đích).
    - Dựa trên: trạng thái hiện tại (của môi trường) và đánh giá hiệu quả hành động (của tác tử).
- Phát biểu bài toán (problem formulation)

  - Với một mục tiêu thì các hành động hay trạng thái đều cần phải được xem xét
- Quá trình tìm kiếm (search process)
  - Xem xét các chuỗi hành động có thể
  - Chọn chuỗi hành động tốt nhất

Giải thuật tìm kiếm

- Đầu vào là một bài toán cần phải giải quyết
- Đầu ra là một giải pháp được đưa ra, dưới dạng một chuỗi các hành động cần thực hiện.

IV. PHÂN LOẠI CÁC PHƯƠNG THỨC TÌM KIẾM

1. Uniformed Search

Uniformed Search (hay Tìm Kiếm Mù) là chiến lược tìm kiếm không dựa trên thông tin nào khác ngoài định nghĩa căn bản, thông tin có sẵn về các trạng thái kế tiếp.

Các chiến thuật tìm kiếm mù chỉ có khả năng sản sinh successor và phân biệt trạng thái đích

Mỗi chiến lược tìm kiêm là một thể hiện (đồ thị/cây) của bài toán tìm kiếm tổng quát

Một số thuật toán Uniformed Search phổ biến: BFS, DFS, UCS,…

2. Informed Search

Là tìm kiếm có định hướng hay tìm kiếm heuristic

Các thuật toán có thông tin về trạng thái mục tiêu, có khả năng tìm lời giải hiệu quả hơn so với chiến lược tìm kiếm mù (Uniformed Search)

Thông tin này được thu thập bằng phương pháp heuristic (ước lượng trạng thái hiện tại só với trạng thái đích)

Một số thuật toán Informed Search phổ biến: A*, Best-first Search, Greedy Best-first Search,…

V. DIỄN GIẢI CÁC THUẬT TOÁN TÌM KIẾM

A. THUẬT TOÁN TÌM KIẾM DFS

a) Định nghĩa

Thuật toán ưu tiên chiều sâu hay tìm kiếm theo chiều sâu (Depth-first Search:DFS) là một thuật toán duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị. Thuật toán khởi đầu tại gốc (hoặc chọn một đỉnh nào đó coi như gốc) và phát triển xa nhất có thể theo mỗi nhánh.

Thông thường, DFS là một dạng tìm kiếm thông tin không đầy đủ mà quá trình tìm kiếm được phát triển tới đỉnh con đầu tiên của nút đang tìm kiếm cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó giải thuật quay lui về đỉnh mới được tìm kiếm ở bước trước. Trong dạng không đệ quy, tất cả các đỉnh chờ được phát triển được bổ sung vào một ngăn xếp LIFO.

b) Nguyên lý hoạt động và đánh giá

(a) Ý tưởng DFS:

Chọn một đỉnh tuỳ ý của đồ thị làm gốc

Xây dựng đường đi từ đỉnh này bằng cách lần lượt ghép các cạnh sao cho mỗi cạnh mới ghép sẽ nối đỉnh cuối cùng trên đường đi với một đỉnh còn chưa thuộc đường đi. Tiếp tục ghép thêm cạnh vào đường đi chừng nào không thể thêm được nữa

Nếu đường đi qua tất cả các đỉnh của đồ thị thì cây do đường này tạo nên là cây khung.

Nếu chưa thì lùi lại đỉnh trước đỉnh cuối cùng của đường đi và xây dựng đường đi mới xuất phát từ đỉnh này đi qua các đỉnh còn chưa thuộc đường đi.

Nếu điều đó không thể làm được thì lùi thêm một đỉnh nữa trên đường đi và thử xây dựng đường đi mới. Tiếp tục quá trình như vậy cho đến khi tất cả các đỉnh của đồ thị được ghép vào cây. Cây T có được là cây khung của đồ thị.

(b) Mã giả

```
Function Depth-Search(problem,Stack) return a solution, or failure
    Stack ← make-queue(make_node(initial-state[problem]));
    father(initial-state[problem]) = empty;
    while(1)
        if Stack is empty then return failure;
        node = pop(Stack);
        if test(node, Goal[problem]) then return path (node, father);
        expand-nodes ← adjacent-nodes(node,Operators[problem]);
        push(Stack, expand-nodes);
        for each ex-node in expand-nodes
            father(ex-node) = node;
end
```

(c) Đánh giá thuật toán

- Về tính đầy đủ: Không được đảm bảo trong mọi trường hợp. Phụ thuộc vào không gian tìm kiếm, nếu không gian là đồ thị liên thông và không có chu trình thì DFS có thể tìm được tất cả đường đi giữa các đỉnh của đồ thị. Ngược lại, nếu đồ thị có chu trình thì DFS có thể bị mắc kẹt vào vòng lặp vô hạn và không thể hoàn thành được việc tìm kiếm.

  Hơn thế, nếu đồ thị không liên thông thì việc tìm được đường đi trong đồ thị thì DFS phải thực hiện rời rạc trên từng thành phần liên thông.

- Tính tối ưu: DFS không phải là thuật toán tìm kiếm đường đi ngắn nhất và không sử dụng bất kỳ tiêu chí nào để xác định đường đi ngắn nhất. Nếu có nhiều đường

đi giữa hai đỉnh, DFS chỉ tìm được một đường đi nào đó mà nó duyệt được trước các đường đi còn lại, và không đảm bảo đường đi đó là đường đi ngắn nhất => không đưa ra được lời giải tối ưu, số bước duyệt và chí phí bỏ ra là khá lớn.

- Độ phức tạp về thời gian: $O(b^m)$
- Độ phức tạp về không gian: O(bm) kích thước không gian tuyến tính

B. THUẬT TOÁN TÌM KIẾM BFS

a) Định nghĩa

Trong lý thuyết đồ thị, tìm kiếm theo chiều rộng (BFS) là một thuật toán tìm kiếm trong đồ thị trong đó việc tìm kiếm chỉ bao gồm 2 thao tác:

(a) Cho trước một đỉnh của đồ thị;

(b) Thêm các đỉnh kề với đỉnh vừa cho vào danh sách có thể hướng tới tiếp theo.

Có thể sử dụng thuật toán tìm kiếm theo chiều rộng cho hai mục đích: tìm kiếm đường đi từ một đỉnh gốc cho trước tới một đỉnh đích, và tìm kiếm đường đi từ đỉnh gốc tới tất cả các đỉnh khác. Trong đồ thị không có trọng số, thuật toán tìm kiếm theo chiều rộng luôn tìm ra đường đi ngắn nhất có thể.

Thuật toán BFS bắt đầu từ đỉnh gốc và lần lượt nhìn các đỉnh kề với đỉnh gốc. Sau đó, với mỗi đỉnh trong số đó, thuật toán lại lần lượt nhìn trước các đỉnh kề với nó mà chưa được quan sát trước đó và lặp lại.

Ta có thể sử dụng cấu trúc dữ liệu hàng đợi (queue) để cài đặt thuật toán này.

b) Nguyên lý hoạt động và đánh giá

(a) Ý tưởng BFS

Cho G là đồ thị liên thông với tập đỉnh $\{v_1, v_2, \dots, v_n\}$

- Thêm v_1 như là gốc của cây rỗng
- Thêm vào các đỉnh kề v_1 và các cạnh nối v_1 với chúng. Những đỉnh này là đỉnh mức 1 trong cây

- Đối với mọi đỉnh v mức 1, thêm vào các cạnh kề với v vào cây sao cho không tạo nên chu trình. Ta thu được các đỉnh mức 2

Tiếp tục quá trình này cho tới khi tất cả các đỉnh của đồ thị được ghép vào cây. Cây T có được là cây khung của đồ thị.

(b) Mã giả

```
Function Breadth-Search(problem, Queue) returns a solution, or failure
    Queue ← make-queue(make-node(initial-state[problem]));
    father(initial-state[problem]) = empty;
    while (1)
        if Queue is empty then return failure;
        node = pop(Queue) ;
        if test(node,Goal[problem]) then return path(node,father);
        expand-nodes ← adjacent-nodes(node, Operators[problem]);
        push(Queue, expand-nodes );
        foreach ex-node in expand-nodes
            father(ex-node) = node;
end
```

(c) Đánh giá thuật toán

- Tính đầy đủ: Thuật toán có tính đầy đủ, nếu áp dụng đúng cách và đủ thời gian thì BFS sẽ duyệt qua tất cả các đỉnh có thể đến từ đỉnh xuất phát. Do đó nếu bài toán có lời giải, tìm kiếm theo chiều rộng đảm bảo tìm ra lời giải.
- Tính tối ưu: : Có giải thuật tìm kiếm theo chiều rộng sẽ tìm ra lời giải với ít trạng
- thái trung gian nhất nhưng sẽ có chi phí tính toán và lưu trữ rất lớn, đặc biệt là khi đồ thị có kích thước lớn hoặc không xác định trước.

- Độ phức tạp về thời gian: $1 + b + b^2 + \ldots + b + b^d$ (số vòng lặp khi gặp trạng thái đích) = $O(b^d)$
- Độ phức tạp không gian: $O(b^{d-1})$ cho tập mở và $O(b^d)$ cho biên

C. THUẬT TOÁN TÌM KIẾM UCS

a) Định nghĩa

UCS (Uniform-Cost Search) là một thuật toán tìm kiếm trên đồ thị được sử dụng để tìm kiếm đường đi có chi phí nhỏ nhất từ nút gốc đến một nút bất kỳ trên đồ thị. Điểm mạnh của UCS là đảm bảo tìm kiếm đường đi có chi phí nhỏ nhất, tức là tối ưu hóa chi phí trong quá trình tìm kiếm.

Thuật toán UCS hoạt động bằng cách duyệt qua các nút trên đồ thị theo thứ tự tăng dần của chi phí của đường đi từ nút gốc đến nút đó. Tức là, các nút có chi phí thấp nhất sẽ được duyệt trước. Nếu có nhiều đường đi có chi phí bằng nhau, UCS sẽ tiếp tục duyệt các đường đi đó theo thứ tự độ ưu tiên đã được xác định trước đó.

b) Nguyên lý hoạt động và đánh giá

(a) Ý tưởng UCS

1. Thêm node bắt đầu start vào opened set với chi phí đường đi là cost[start] = 0

2. Đẩy node có chi phí thấp nhất lúc này ra khỏi opened set và gọi node này là current. Kiểm tra current có là goal hay không nếu là goal thì kết thúc, không thì tiếp tục

3. Thêm node mới là các node liên kết với current vào opened set (các node này đều chưa được duyệt tức không nằm trong closed set). Các node này được thêm vào với thứ tự ưu tiên cho các node có chi phí thấp. Nếu node được thêm đã tồn tại trong opened set ta sẽ giữ node có chi phí thấp hơn và bỏ node còn lại.

4. Đồng thời cập nhật lại giá trị trong father và cost

5. Thêm current tức node vừa xét xong vào closed set (các node đã duyệt)

6. Kiểm tra opened set có rỗng không. Rỗng thì kết thúc, không thì lặp lại bước 2.

(b) Mã giả

```
Function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    if problem's initial state is a goal then return empty path to initial state
    frontier ← a priority queue ordered by pathCost, with a node for the initial state
        reached ← a table of {state: the best path that reached state}; initially empty
        solution ← failure
        while frontier is not empty and top(frontier) is cheaper than solution do
            parent ← pop(frontier)
            for child in successors(parent) do
                s ← child.state
                if s is not in reached or child is a cheaper path than reached[s] then
                    reached[s] ← child
                    add child to the frontier
                    if child is a goal and is cheaper than solution then
                        solution = child
    return solution
```

(c) Đánh giá thuật toán

Với chi phí di chuyển thấp nhất là ε, C* là chi phí lời giải tối ưu

Tính đầy đủ: Có tính đầy đủ nếu đảm bảo tính hoàn chỉnh của tập hợp các hành động và trạng thái (nếu chi phí ở mỗi bước ≥ ε)

Tính tối ưu: Tính tối ưu là rất cao, đảm bảo tìm kiếm đường đi tối ưu với chi phí nhỏ nhất từ nút gốc đến một nút bất kì trên đồ thị. Tuy nhiên để đạt được tính tối ưu cảu UCS thì cần đảm bảo các hành động và trạng thái là đầy đủ và chi phí di chuyển không âm. USC có thể tốn nhiều thời gian và bộ nhớ khi đồ thị có kích thước lớn, không thể xử lí trong thời gian hợp lí.

Độ phức tạp về thời gian: Phụ thuộc vào tổng số các nút có chi phí ≤ chi phí của lời giải tối ưu: $O(b^{1+[\frac{C^*}{\varepsilon}]})$

Độ phức tạp về không gian: Phụ thuộc vào tổng số các nút có chi phí ≤ chi phí của lời giải tối ưu: $O(b^{1+[\frac{C^*}{\varepsilon}]})$

D. THUẬT TOÁN TÌM KIẾM A*

a) Định nghĩa

A* (đọc là A sao) là thuật toán tìm kiếm trong đồ thị. Thuật toán này tìm một đường đi từ một nút khởi đầu tới một nút đích cho trước (hoặc tới một nút thỏa mãn một điều kiện đích). Thuật toán này sử dụng một "đánh giá heuristic" để xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó. Thuật toán này duyệt các nút theo thứ tự của đánh giá heuristic này. Do đó, thuật toán A* là một ví dụ của tìm kiếm theo lựa chọn tốt nhất (best-first search).

Thuật toán A* được mô tả lần đầu vào năm 1968 bởi Peter Hart, Nils Nilsson, và Bertram Raphael. Trong bài báo của họ, thuật toán được gọi là thuật toán A; khi sử dụng thuật toán này với một đánh giá heuristic thích hợp sẽ thu được hoạt động tối ưu, do đó mà có tên A*.

Năm 1964, Nils Nilsson phát minh ra một phương pháp tiếp cận dựa trên khám phá để tăng tốc độ của thuật toán Dijkstra. Thuật toán này được gọi là A1. Năm 1967 Bertram

Raphael đã cải thiện đáng kể thuật toán này, nhưng không thể hiển thị tối ưu. Ông gọi thuật toán này là A2. Sau đó, trong năm 1968 Peter E. Hart đã giới thiệu một đối số chứng minh A2 là tối ưu khi sử dụng thuật toán này với một đánh giá heuristic thích hợp sẽ thu được hoạt động tối ưu. Chứng minh của ông về thuật toán cũng bao gồm một phần cho thấy rằng các thuật toán A2 mới là thuật toán tốt nhất có thể được đưa ra các điều kiện. Do đó ông đặt tên cho thuật toán mới là A *(A sao, A-star).

b) Nguyên lý hoạt động và đánh giá

(a) Ý tưởng A*

A* (đọc là A sao) là dạng tìm kiếm Best-first Search phổ biến nhất. Thuật toán này tìm một đường đi từ một nút khởi đầu tới một nút đích cho trước (hoặc tới một nút thoả mãn một điều kiện đích). Thuật toán này sử dụng một "đánh giá heuristic" để xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó. Thuật toán này duyệt các nút theo thứ tự của đánh giá heuristic này. Ý tưởng trực quan xét bài toán tìm đường – bài toán mà A* thường được dung 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 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 được những tuyến đường nào sẽ có khả năng dẫn đến đích, A* sử dụng một "đánh giá heuristic" về khoảng cách từ điểm bất kỳ cho trước tới đích.

(b) Mã giả

```
Function Astar-Search(start, end)
    open_list ← {start}
    closed_list ← {}
    g(start) ← 0
```

```
h(start) ← heuristic_function(start,end)
f(start) ← g(start) + h(start)
while open_list.isEmpty():
        curr ← Node on top of open_list, with least f
        if curr == end then return curr
        open_list.remove(curr)
        closed_list.add(curr)
        for each n in child(curr)
                if n in closed_list
                        continue
                cost ← g(curr) + distance(curr, n)
                if n in open_list and cost < g(n)
                        open_list.remove(n)
                if n in closed_list and cost < g(n)
                        closed_list.remove(n)
                if n not in open_list and n not in closed_list
                        open_list.add(n)
                        g(n) ← cost
                        h(n) ← heuristic_function(n,end)
                        f(n) ← g(n) + h(n)
return failure
```

end

(c) Đánh giá thuật toán

Với chi phí di chuyển thấp nhất là ε, C* là chi phí lời giải tối ưu

Tính đầy đủ: Thuật toán có tính đầy đủ (nếu ε > 0 và không gian trạng thái hữu hạn)

Tính tối ưu: Thuật toán có tính tối ưu nếu "đánh giá heuristic" là hợp lí và nhất quán.

Độ phức tạp về thời gian: cấp số mũ

Độ phức tạp về không gian: cấp số mũ

Đề xuất heuristic: Các hàm heuristic đóng vai trò rất quan trọng. Hàm heuristic tốt đảm bảo tính tối ưu cho những thuật toán như A*. Bên cạnh đó hàm heuristic tốt cho phép hướng thuật toán theo phía lời giải, nhờ vậy giảm số trạng thái cần khảo sát và độ phức tạp của thuật toán. Các hàm heuristic h(n) được xây dựng tùy thuộc vào bài toán cụ thể. Cùng một loại bài toán chúng ta có thể có rất nhiều hàm heuristic khác nhau. Chất lượng hàm heuristic ảnh hưởng rất nhiều đến quá trình tìm kiếm. Hàm heuristic h(n) được gọi là chấp nhận được khi:

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

Trong đó h*(n) là giá thành đường đi thực từ n đến node đích

(d) Heuristic trong tìm kiếm

Heuristic là một phương pháp giải quyết vấn đề dựa trên kinh nghiệm và tri thức đã có sẵn, thường được sử dụng khi giải quyết vấn đề mà không có một giải pháp chính xác hoặc không thể tìm ra giải pháp tối ưu. Heuristic thường được sử dụng để tìm ra một giải pháp gần đúng hoặc tạm thời cho vấn đề cần giải quyết.

Heuristic có thể được áp dụng trong nhiều lĩnh vực khác nhau, từ khoa học máy tính đến y học và tâm lý học. Với các vấn đề phức tạp và không thể giải quyết được bằng các phương pháp tiếp cận truyền thống, heuristic có thể cung cấp các giải pháp chấp nhận được trong thời gian ngắn.

Một số heuristic được áp dụng trong A*:

- Heuristic Manhattan: Đây là heuristic phổ biến nhất được sử dụng trong A*. Nó tính khoảng cách giữa vị trí hiện tại và vị trí đích bằng cách lấy tổng tuyệt đối của khoảng cách theo trục x và trục y. Heuristic này cho phép A* tìm kiếm nhanh hơn so với việc không sử dụng heuristic.
- Heuristic Euclid: Heuristic này tính khoảng cách giữa vị trí hiện tại và vị trí đích dựa trên định lý Pythagoras. Heuristic này chính xác hơn heuristic Manhattan nhưng có thể tính toán tốn kém hơn.
- Heuristic Diagonal: Heuristic này kết hợp hai heuristic Manhattan và Euclid. Nó tính khoảng cách giữa vị trí hiện tại và vị trí đích bằng cách lấy tổng của heuristic Manhattan và heuristic Euclid. Heuristic này cho phép A* tìm kiếm nhanh hơn heuristic Euclid trong một số trường hợp.
- Heuristic Octile: Đây là heuristic được thiết kế để xử lý tốt các trường hợp khi di chuyển theo đường chéo trong một lưới ô vuông. Heuristic này tính khoảng cách giữa vị trí hiện tại và vị trí đích bằng cách lấy tổng của khoảng cách theo trục x và trục y, nhưng tính toán lại khoảng cách theo đường chéo bằng cách sử dụng định lý Pythagoras.

VI. SO SÁNH CÁC THUẬT TOÁN TÌM KIẾM

A. So sánh sự khác biệt giữa UCS,Greedy và A*

Cơ sở so sánh	UCS	Greedy	A*
Đặc điểm	Thuật toán UCS chọn nút n có chi phí thấp nhất để mở rộng đường đi.	Thuật toán Greedy Search là chọn trong tập nút biên thì nút nào có khoảng cách tới đích là nhỏ nhất để mở rộng. Trong phương pháp này thì để đánh giá một nút là tốt ta sử dụng hàm heuristic	Tương tự về cách hoạt động của thuật toán Greedy nhưng điểm khác là sử dụng hàm đánh giá f(n) với hai thành phần, thành phần thứ nhất là đường đi từ nút đang xét tới nút xuất phát g(n); thành phần thứ hai

		(hàm ước lượng chi phí từ một nút đến đỉnh). Ở đây ta có f(n) = g(n).	là khoảng cách ước lượng tới đích h(n). f(n) = g(n) + h(n)
Tính đầy đủ	Có tính đầy đủ	Không có tính đầy đủ do có khả năng tạo ra lặp vô hạn ở một số nút.	Có tính đấy đủ nếu trong môi trường không gian trạng thái hữu hạn.
Tính tối ưu	Có tính tối ưu	Thuật toán không tối ưu.	Thuật toán sẽ có tính tối ưu nếu hàm heuristic hợp lý và nhất quán
Độ phức tạp về thời gian	$O(b^{1+[\frac{C^*}{\varepsilon}]})$	Trong trường hợp xấu nhất là $O(b^m)$	Cấp số mũ
Độ phức tạp về không gian	$O(b^{1+[\frac{C^*}{\varepsilon}]})$	Trong trường hợp xấu nhất là $O(b^m)$	Cấp số mũ

B. So sánh sự khác biệt giữa UCS và Dijkstra

a) Định nghĩa Dijkstra

Thuật toán Dijkstra là một thuật toán tìm đường đi ngắn nhất có trọng số trong đồ thị vô hướng hoặc có hướng, với độ phức tạp thời gian O(E + V log V), trong đó E là số cạnh, V là số đỉnh của đồ thị. Thuật toán này được đặt tên theo nhà toán học Edsger W. Dijkstra và được phát minh vào năm 1956.

Thuật toán Dijkstra sử dụng một tập hợp các đỉnh đã được xác định nhãn tối ưu để tính toán đường đi ngắn nhất từ đỉnh xuất phát đến các đỉnh khác trong đồ thị. Ban đầu, chỉ có đỉnh xuất phát có nhãn tối ưu bằng 0, và tất cả các đỉnh khác đều có nhãn tối ưu bằng vô cực. Tiếp theo, thuật toán duyệt qua các đỉnh kề của đỉnh hiện tại để cập nhật nhãn tối ưu

của chúng, nếu nhãn mới tốt hơn nhãn hiện tại. Thuật toán tiếp tục lặp lại quá trình này cho đến khi tất cả các đỉnh đều có nhãn tối ưu hoặc không còn đỉnh nào có thể được cập nhật.

b) Sự khác biệt giữa UCS và Dijkstra

Thuật toán Dijkstra tìm đường đi ngắn nhất từ nút gốc đến mọi nút khác. Còn thuật toán UCS tìm kiếm các đường đi ngắn nhất về chi phí từ nút gốc đến nút đích. Thuật toán UCS là trường hợp riêng của thuật toán Dijkstra tập trung vào việc tìm kiếm một con đường ngắn nhất duy nhất đến một điểm kết thúc duy nhất thay vì con đường ngắn nhất đến mọi điểm.

Thuật toán UCS thực hiện tìm kiếm bằng cách dừng lại ngay khi tìm thấy điểm kết thúc. Đối với thuật toán Dijkstra, không có trạng thái mục tiêu và quá trình xử lý vẫn tiếp tục cho đến khi tất cả các nút được loại bỏ khỏi hang đợi ưu tiên, tức là cho đến khi xác định được đường đi ngắn nhất đến tất cả các nút (không chỉ một nút mục tiêu).

Thuật toán UCS có ít yêu cầu không gian hơn, trong đó hàng đợi ưu tiên được lấp đầy dần dần so với thuật toán Dijkstra thêm tất cả các nút vào hàng đợi ngay từ đầu với chi phí vô hạn

Thuật toán Dijkstra cũng tốn nhiều thời gian do yêu cầu về bộ nhớ so với thuật toán UCS.

Thuật toán Dijkstra được sử dụng trên đồ thị chung còn thuật toán UCS thì thường được xây dựng trên đồ thị dạng cây.

Thuật toán Dijkstra chỉ có thể áp dụng trong các đồ thị tường minh mà chúng ta biết tất cả các cạnh và đỉnh, trong đó đưa toàn bộ đồ thị vào input. Còn UCS bắt đầu với đỉnh nguồn và đi qua các phần tử cần thiết của đồ thị. Do đó có thể áp dụng vào cả đồ thị tường minh và không tường minh.

Một sự khác biết nữa giữa hai thuật toán là các giá trị khoảng cách cuối cùng của các đỉnh không thể truy cập từ đỉnh nguồn. Trong thuật toán Dijkstra, nếu không có đường đi

giữa đỉnh nguồn S và đỉnh V thì giá trị khoảng cách của nó (dist[v]) là ∞. Tuy nhiên trong thuật toán UCS thì dist[v] không tồn tại.

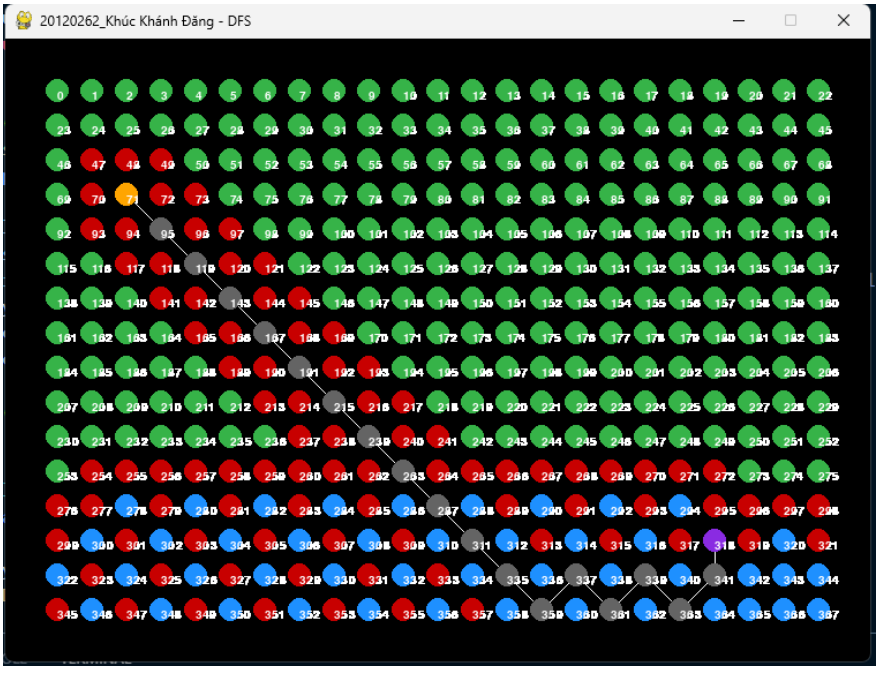
Bảng so sánh 2 thuật toán:

	Khởi tạo	Yêu cầu bộ nhớ	Thời gian chạy	Ứng dụng
Dijkstra	Một tập hợp tất cả các đỉnh có trong đồ thị	Cần phải lưu trữ toàn bộ đồ thị	Chậm hơn do yêu cầu bộ nhớ lớn	Chỉ cho những đồ thị tường minh
UCS	Một tập hợp chỉ có các đỉnh nguồn	Chỉ lưu trữ các đỉnh cần thiết	Nhanh hơn do không phải yêu cầu bộ nhớ lớn	Cả đồ thị tường minh và không tường minh

VII.MÔ TẢ CÁC THUẬT TOÁN ĐÃ CODE

1. DFS

Kết quả:



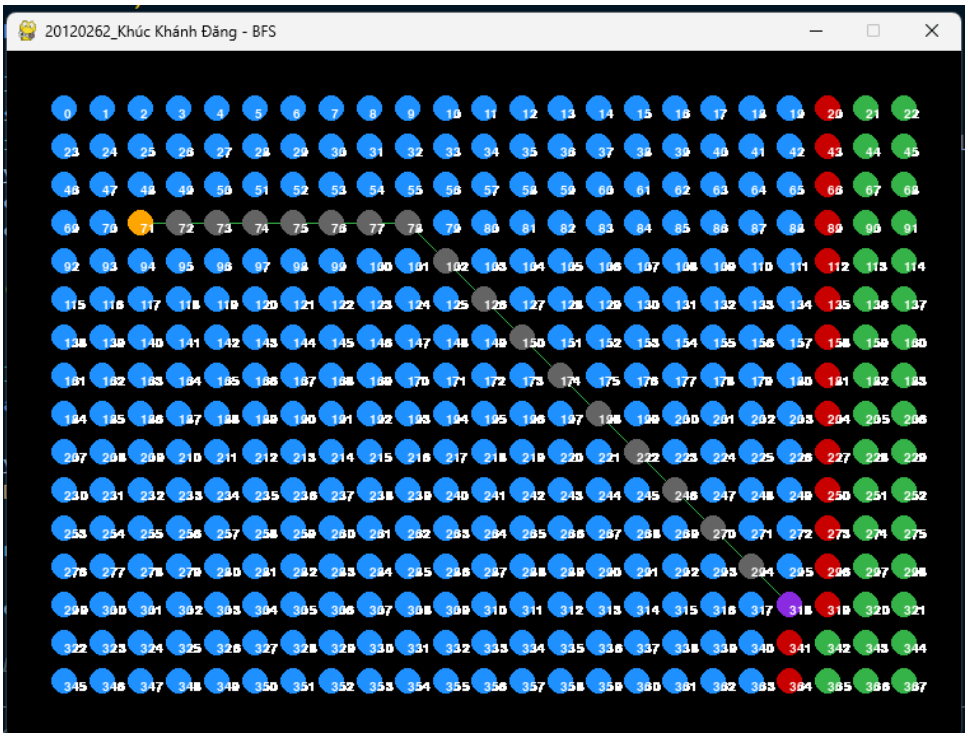
Mô tả: Quá trình tìm kiếm của thuật toán DFS.

Duyệt các node theo chiều sâu cho đến khi tìm được đích nên đường đi dài và không tuân theo một cấu trúc nhất định.

Nhận xét về kết quả: Kết quả cho thấy việc tìm kết quả của DFS là không tối ưu. Mặc dù vẫn tìm ra kết quả nhưng thời gian tìm kiếm lâu và số lượng node cần lưu trữ nhiều

2. BFS

Kết quả:

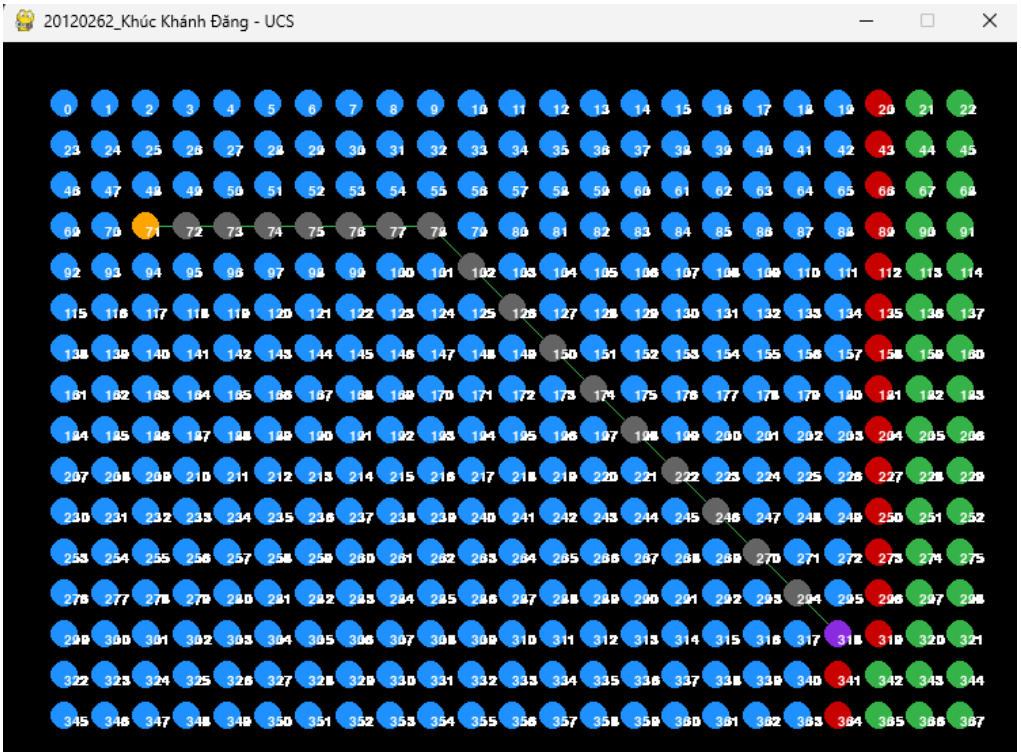


Mô tả: Vì thuật toán BFS duyệt qua tất các đỉnh kề nên nó sẽ tỏa ra như ngọn lửa đang cháy cho đến khi nào tìm được node đích.

Nhận xét về kết quả: Kết quả của tìm kiếm BFS cho ra kết quả tối ưu nhưng thời gian chạy khá lâu vì số lượng node cần lưu trữ lớn.

3. UCS

Kết quả:

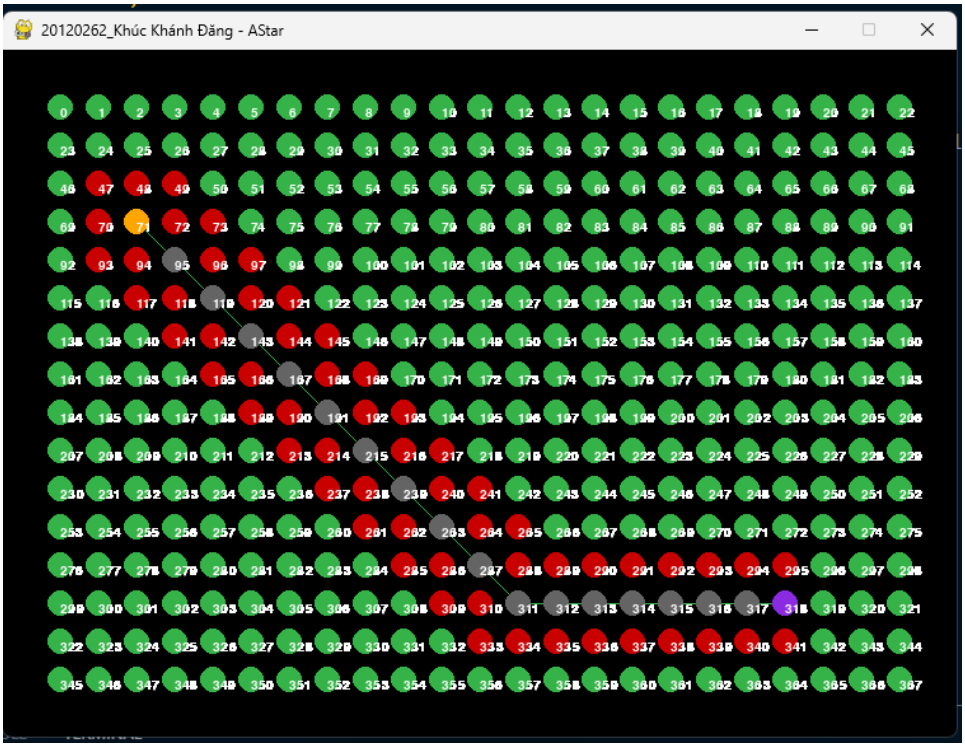


Thuật toán UCS ưu tiên mở những đường có chi phí di chuyển thấp nhất. Cho tới khi đường đi có chi phí thấp nhất là điểm dích thì thuật toán sẽ dừng lại.

Nhận xét: Ta thấy thuật toán UCS có cách mở rộng khá giống với BFS do trọng số các cạnh được cho là ≃ 1 nên UCS cũng mở rộng về nhiều phía. Đường đi tìm được là đường đi tối ưu. Không gian lưu trữ và thời gian tìm kiếm lớn.

4. A*

Kết quả:



Thuật toán A* sử dụng hàm đánh giá $f(n) = g(n) + h(n)$ cho từng đỉnh.

Trong đó, $g(n)$ là chi phí thực tế đi từ đỉnh đến điểm đang xét, $f(n)$ là chi phí ước tính đi từ đỉnh đang xét đến đích.

Khi hoạt động, A* đưa các đỉnh mới vào một hàng đợi chung với các đỉnh cũ, dựa vào hàm đánh giá $f(n)$ để quyết định sẽ mở đỉnh nào tiếp theo.

Nhận xét: Đường đi tìm được là đường đi tối ưu. Không gian lưu trữ nhỏ và thời gian tìm kiếm ngắn.

VIII. TÌM HIỂU THÊM THUẬT TOÁN (GREEDY)

a) Định nghĩa

Thuật toán Greedy là một thuật toán tìm kiếm và giải quyết bài toán theo cách tiếp cận "tham lam". Cụ thể, thuật toán này luôn chọn phương án tốt nhất ở mỗi bước, hy vọng rằng nó sẽ dẫn đến kết quả tối ưu cho toàn bộ bài toán. Thuật toán Greedy thường được sử dụng trong các bài toán tối ưu hoặc trong việc xây dựng các hệ thống tự động quyết định. Tuy nhiên, không phải lúc nào thuật toán Greedy cũng cho ra kết quả tối ưu, do đó cần đánh giá kỹ càng trước khi sử dụng.

b) Nguyên lý hoạt động và đánh giá

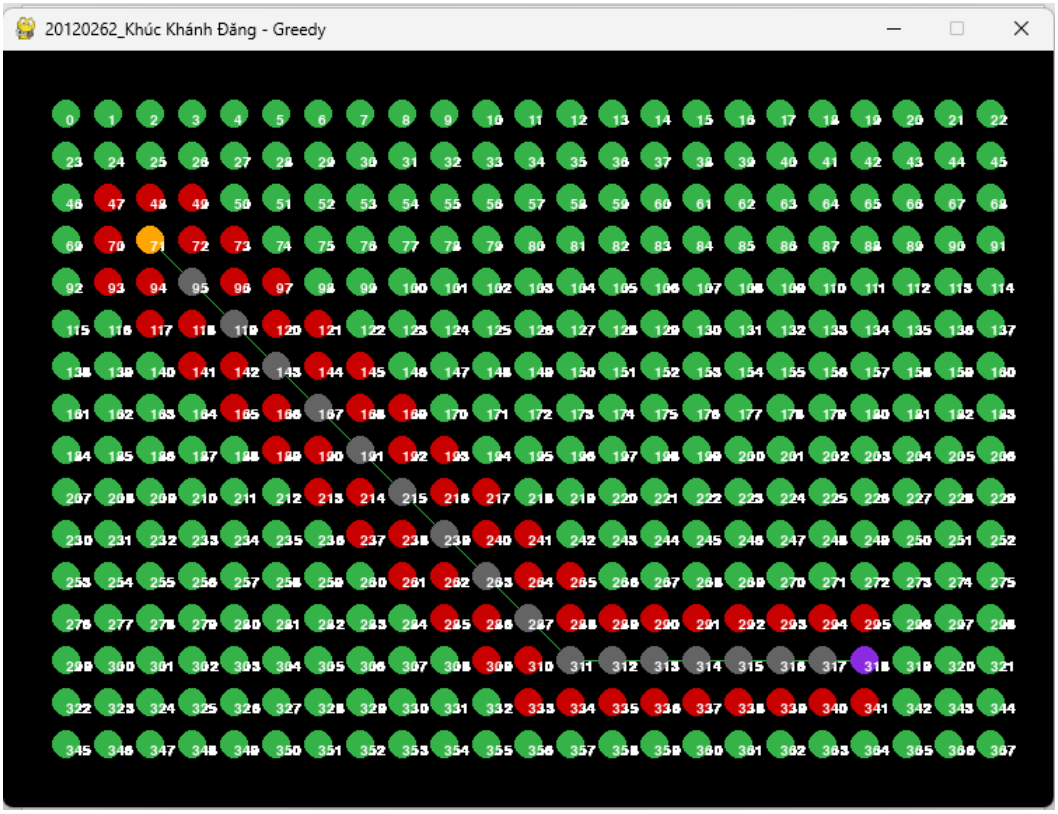
(a) Ý tưởng Greedy

chọn trong tập nút biên nút có khoảng cách tới đích nhỏ nhất để mở rộng. Lý do làm như vậy để dễ mở rộng nút gần đích có xu hướng dẫn tới lời giải nhanh hơn. Trong phương pháp này, để đánh giá độ tốt của một nút ta dùng hàm heuristic. Đối với thuật toán Greedy thì f(n) = h(n).

(b) Đánh giá thuật toán

- Tính đầy đủ: Không có tính đầy đủ do có khả năng tạo ra vòng lặp vô tận ở một số nút
- Tính tối ưu: Thuật toán là không phải là tối ưu
- Độ phức tạp về thời gian: Trong trường hợp xấu nhất là $O(b^m)$
- Độ phức tạp về không gian: Trong trường hợp xấu nhất là $O(b^m)$

Code tìm đường đi bằng thuật toán Greedy:



Mô tả: Vì các đỉnh kề nhau có khoảng cách như nhau nên đường đi của thuật toán Greedy giống với thuật toán A*.

Nhận xét: Vì các đỉnh kề nhau có khoảng cách như nhau nên thuật toán Greedy cho ra đường đi tối ưu.

IX. TÀI LIỆU THAM KHẢO

https://fit.lqdtu.edu.vn/files/FileMonHoc/1.%20TTNT.pdf

https://drive.google.com/file/d/14LlgPi84-GZdl-fc-KUAU7BIPl-VYqj8/view

https://vnoi.info/wiki/algo/graph-theory/Depth-First-Search-Tree.md

https://users.soict.hust.edu.vn/huonglt/AI/lecture%20notes.htm

https://github.com/VoNhatVinh/Introduce-to-AI

https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/depth-first-search-dfs-algorithm/

https://github.com/convitnhodev/Search-algorithm

https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras

https://stackoverflow.com/questions/12806452/whats-the-difference-between-uniform-cost-search-and-dijkstras-algorithm

- Link youtube: https://youtu.be/Jh_x4e7N6UY