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

-----o0o----

BÁO CÁO THỰC HÀNH

CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

Giáo viên thực hành : Nguyễn Bảo Long
Lớp : 20_22
Sinh viên thực hiện : Nguyễn Ngọc Thùy
Mã số sinh viên : 20120206

Thành phố Hồ Chí Minh, tháng 10 năm 2022

MỤC LỤC

1. Tìm hiểu và trình bày thuật toán

1.1. Bài toán tìm kiếm

Một bài toán tìm kiếm gồm 5 thành phần:

1. Trạng thái bắt đầu (initial state)
2. Mô tả các hành động (action) có thể thực hiện
3. 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
4. Kiểm tra đích (goal test): xác định một trạng thái có là trạng thái đích
5. Một hàm chi phí đường đi (path cost) gán chi phí với giá trị số cho mỗi đường đi
   - Chi phí đường đi khi thực hiện hành động a từ trạng thái s để đến trạng thái s' ký hiệu $c(s, a, s')$

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 các lời giải.

Cách giải một bài toán tìm kiếm nói chung gồm các bước sau:

- Xác định vấn đề
- Phân tích vấn đề
- Xác định các giải pháp khả thi
- Lựa chọn giải pháp tối ưu

– Thực thi giải pháp

Có 2 loại bài toán tìm kiếm là: tìm kiếm mù (uninformed search) và tìm kiếm có định hướng (informed search).

– *Tìm kiếm mù*
  - Chiến lược tìm kiếm không dựa trên thông tin nào khác ngoài định nghĩa bài toán
  - Các thuật toán tìm kiếm mù chỉ có khả năng sinh successor và phân biệt trạng thái đích
  - Mỗi chiến lược tìm kiếm là một thể hiện (đồ thị/cây) của bài toán tìm kiếm tổng quát
– *Tìm kiếm có định hướng (tìm kiếm heuristic)*
  - Bên cạnh định nghĩa còn sử dụng tri thức cụ thể về bài toán
  - Có khả năng tìm lời giải hiệu quả hơn so với các chiến lược tìm kiếm mù
  - Heuristic là tri thức về bài toán được truyền vào thuật toán tìm kiếm, ước lượng khoảng cách giữa trạng thái hiện tại so với trạng thái đích

1.2. Trình bày về 4 thuật toán DFS, BFS, UCS, A*

1.2.1. Ý tưởng chung

a. DFS (tìm kiếm mù)

Thuật toán tìm kiếm theo chiều sâu (thường gọi là DFS) là một thuật toán duyệt khởi đầu tại nút gốc và đi xa nhất có thể theo một nhánh.

b. BFS (tìm kiếm mù)

Thuật toán tìm kiếm theo chiều rộng (thường gọi là BFS) là một thuật toán duyệt bắt đầu từ gốc, sau đó lần lượt xét qua các nút của một cây theo ưu tiên về độ sâu từ nhỏ đến lớn.

c. UCS (tìm kiếm mù)

Thuật toán UCS là một thuật toán duyệt, tìm kiếm trên một cấu trúc cây, hoặc đồ thị có trọng số (chi phí). Việc tìm kiếm bắt đầu tại nút gốc và tiếp tục bằng cách duyệt các nút tiếp theo với trọng số hay *chi phí thấp nhất* $g(n)$ tính từ nút gốc. Thuật toán UCS tương tự với thuật toán Dijkstra.

d. A (tìm kiếm có định hướng)*

Thuật toán A* là dạng tìm kiếm Best-first Search phổ biến nhất.

Best-first Search là thuật toán tìm kiếm tổng quát trên cây và đồ thị. Một nút được chọn mở dựa trên *hàm đánh giá* $f(n)$, được xây dựng dựa trên ước lượng chi phí.

- Nút có ước lượng thấp nhất được mở trước
- Cài đặt tương tự như UCS ngoại trừ việc sử dụng f thay vì g cho việc sắp xếp hàng đợi ưu tiên
- Lựa chọn f định nghĩa *chiến lược tìm kiếm*

Hầu hết hàm đánh giá f đều chứa *heuristic* - h.

Ý tưởng của A* search:

- Tích hợp heuristic vào quá trình tìm kiếm
- Tránh các đường đi có chi phí lớn

Hàm đánh giá của A* search: $f(n) = g(n) + h(n)$

- $g(n)$: chi phí đường đi đến n
- $h(n)$: ước tính khoảng cách đến đích
- $f(n)$: ước tính chi phí đến đích

Gần như tương tự với UCS ngoại trừ $f(n) = g(n) + h(n)$ thay vì $f(n) = g(n)$.

1.2.2. Mã giả

a. DFS

Mã giả thuật toán DFS dùng stack:

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

function DEPTH-FIRST-SEARCH(problem) returns a node or failure
    frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as
an element
    result ← failure
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        else if not IS-CYCLE(node) do
            for each child in EXPAND(problem, node) do
                add child to frontier
    return result
```

b. BFS

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or
failure
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue, with node as an element
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s is not in reached then
                add s to reached
                add child to frontier
    return failure
```

c. UCS

```
function UNIFORM-COST-SEARCH(problem) returns a solution node or
failure
    node ← NODE(STATE=problem.INITIAL)
    frontier ← a priority queue ordered by PATH-COST, with node
as an element
```

```
    reached ← a lookup table, with one entry with key
problem.INITIAL and value node
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
            for each child in EXPAND(problem, node) do
                s ← child.STATE
                if s is not in reached or child.PATH-COST <
reached[s].PATH-COST then
                    reached[s] ← child
                    add child to frontier
    return failure
```

*d. A**

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

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

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

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

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

```
    fScore := map with default value of Infinity
    fScore[start] := h(start)

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

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

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

1.2.3. Đánh giá về thuật toán

a. DFS

– Độ phức tạp thời gian: $O(b^m)$ ⇒ Tệ nếu m lớn hơn nhiều so với b

– Độ phức tạp không gian: $O(bm)$, kích thước không gian tuyến tính

– Hoàn thành: có (nếu không gian hữu hạn)

– Tối ưu: lời giải “phải nhất”

b. BFS

– Độ phức tạp thời gian: $O(b^d)$

– Độ phức tạp không gian: $O(b^{d-1})$ cho tập mở và $O(b^d)$ cho biên

– Hoàn thành: có (nếu b hữu hạn)

– Tối ưu: có (nếu chi phí di chuyển là như nhau)

c. UCS

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

– Độ phức tạp thời gian: $O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$

– Độ phức tạp không gian: $O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$

– Hoàn thành: có (nếu không gian trạng thái hữu hạn và không có vòng lặp với chi phí bằng không)

– Tối ưu: có (nếu không có chi phí âm)

*d. A**

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

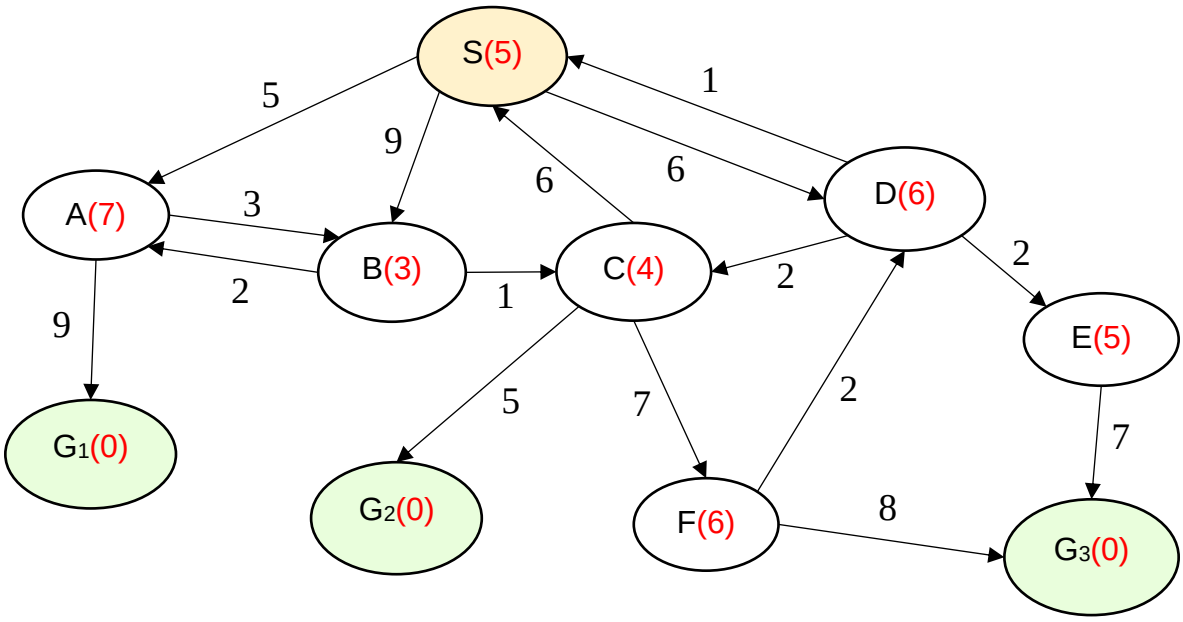
– Độ phức tạp thời gian: $O(b^d)$

– Độ phức tạp không gian: $O(b^d)$

– Hoàn thành: có (nếu $\epsilon > 0$ và không gian trạng thái hữu hạn)

– Tối ưu: có (nếu heuristic *hợp lý* và *nhất quán)*

1.2.4. Ví dụ minh họa



Đồ thị trên có S là đỉnh bắt đầu và G_1, G_2, G_3 là đỉnh đích. Số trong ngoặc là heuristic.

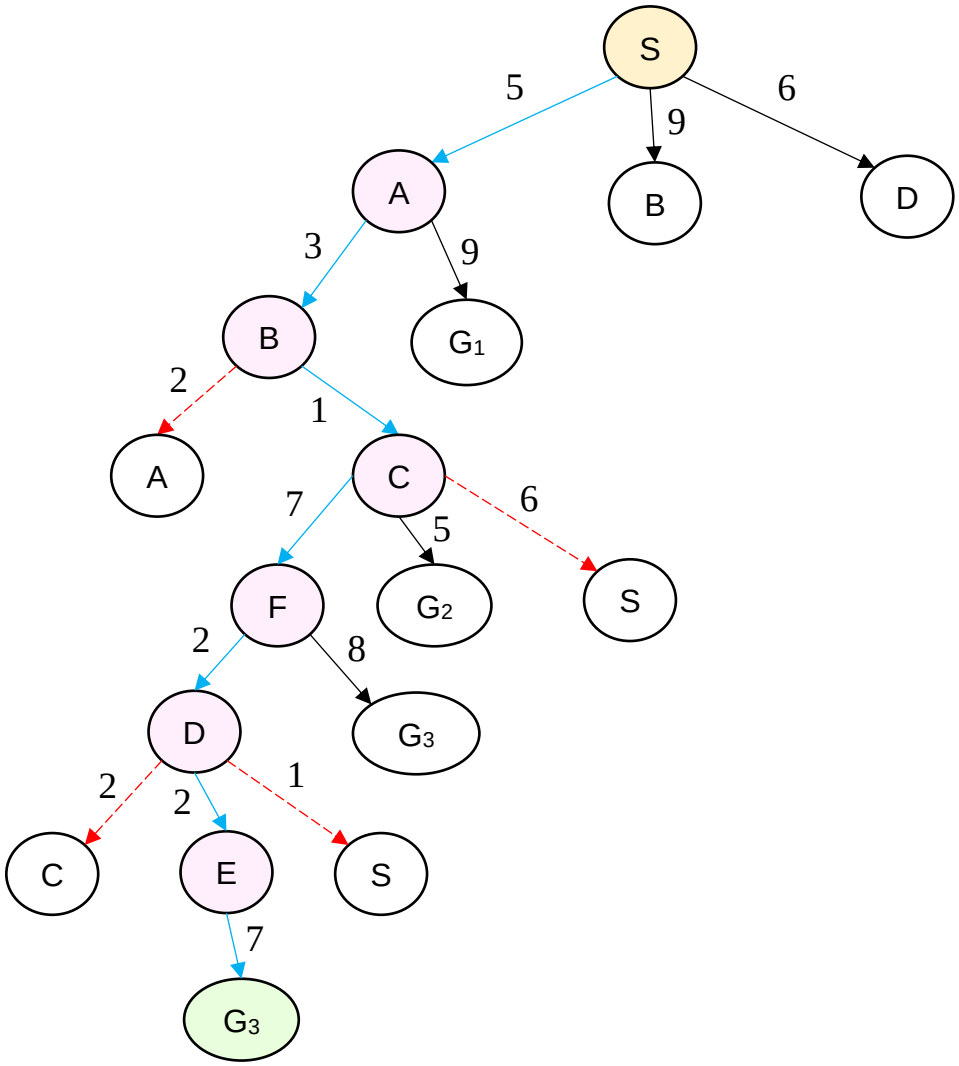
a. DFS

Các đỉnh được thăm theo thuật toán DFS lần lượt là: $S \to A \to B \to C \to F \to D \to E \to G_3$ (đến đích).

Đường đi từ đỉnh bắt đầu đến đích: $S \to A \to B \to C \to F \to D \to E \to G_3$.

Chi phí đường đi: $5 + 3 + 1 + 7 + 2 + 2 + 7 = 27$

Cây tìm kiếm:



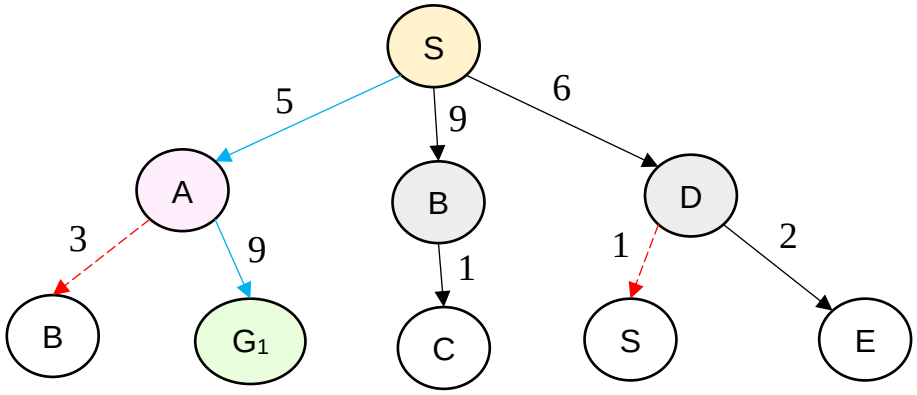
b. BFS

Các đỉnh được thăm theo thuật toán BFS lần lượt là: $S \to A \to B \to D \to G_1$ (đến đích).

Đường đi từ đỉnh bắt đầu đến đích: $S \to A \to G_1$.

Chi phí đường đi: $5 + 9 = 14$

Cây tìm kiếm:



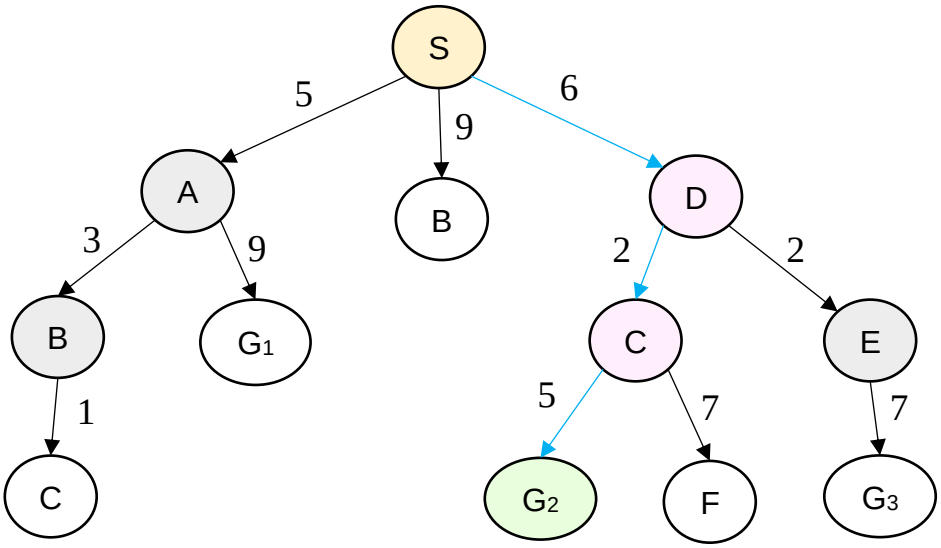
c. UCS

Các đỉnh được thăm theo thuật toán UCS lần lượt là: $S \rightarrow A \rightarrow D \rightarrow B \rightarrow C \rightarrow E \rightarrow G_2$ (đến đích).

Đường đi từ đỉnh bắt đầu đến đích: $S \rightarrow D \rightarrow C \rightarrow G_2$.

Chi phí đường đi: $6 + 2 + 5 = 13$

Cây tìm kiếm:



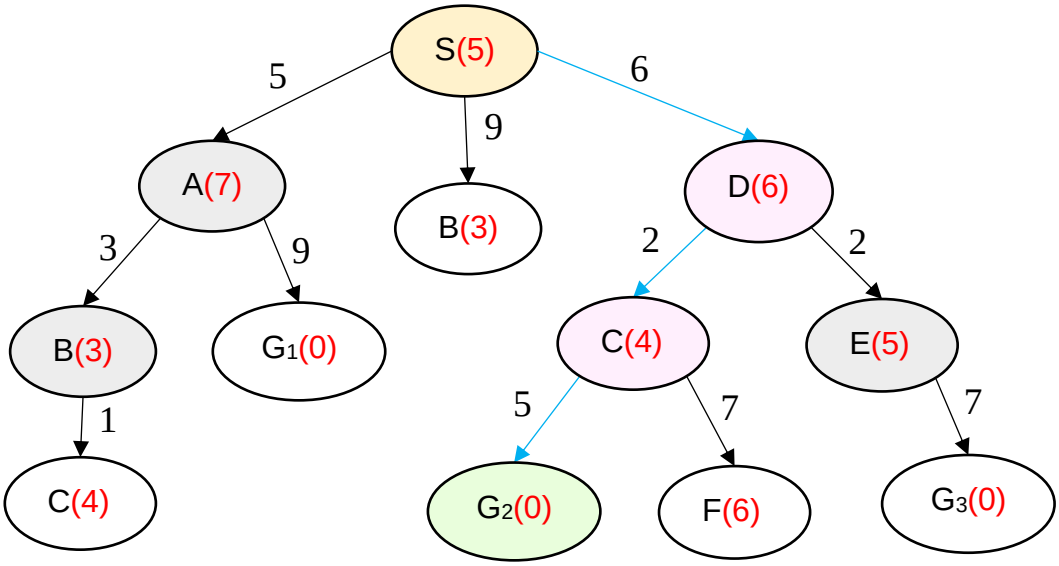
*d. A**

Các đỉnh được thăm theo thuật toán A* lần lượt là: $S(5) \rightarrow A(12) \rightarrow B(11) \rightarrow D(12) \rightarrow C(12) \rightarrow E(13) \rightarrow G_2(13)$ (đến đích).

Đường đi từ đỉnh bắt đầu đến đích: $S(5) \rightarrow D(12) \rightarrow C(12) \rightarrow G_2(13)$.

Chi phí đường đi: $6 + 2 + 5 = 13$

Cây tìm kiếm:



2. So sánh sự khác biệt

2.1. UCS, Greedy và A*

2.1.1. Ý tưởng

- UCS: mở nút n với chi phí đường đi thấp nhất
- Greedy: mở các nút được ước lượng gần với đích nhất
- A*: tích hợp heuristic vào quá trình tìm kiếm và tránh các đường đi có chi phí lớn

2.1.2. Hàm đánh giá

$f(n)$: ước tính chi phí đến đích

$g(n)$: chi phí đường đi đến n

$h(n)$: ước tính khoảng cách đến đích

- UCS: đánh giá dựa trên chi phí đường đi ⇒ $f(n) = g(n)$
- Greedy: đánh giá dựa trên heuristic ⇒ $f(n) = h(n)$
- A*: đánh giá dựa trên chi phí đường đi và heuristic ⇒ $f(n) = g(n) + h(n)$

2.1.3. Độ phức tạp thời gian

- UCS: $O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$
- Greedy: $O(b^m)$
- A*: $O(b^d)$

2.1.4. Độ phức tạp không gian

- UCS: $O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$
- Greedy: $O(b^m)$
- A*: $O(b^d)$

2.1.5. Hoàn thành

- UCS: có (nếu không gian trạng thái hữu hạn và không có vòng lặp với chi phí bằng không)
- Greedy: không (có thể bị kẹt trong vòng lặp)
- A*: có (nếu $\epsilon > 0$ và không gian trạng thái hữu hạn)

2.1.6. Tối ưu

- UCS: có (nếu không có chi phí âm)
- Greedy: không (không đảm bảo sẽ tìm được đường đi với chi phí thấp nhất)
- A*: có (nếu heuristic *hợp lý* và *nhất quán)*

2.2. UCS và Dijkstra

2.2.1. Ý tưởng

- Dijkstra: tìm đường đi ngắn nhất từ đỉnh bắt đầu đến tất cả các đỉnh khác trong đồ thị ⇒ chỉ sử dụng được với đồ thị tường minh, đồ thị mà ta biết được các đỉnh và cạnh của nó (số đỉnh của đồ thị phải giới hạn).
- UCS: tìm đường đi ngắn nhất từ đỉnh bắt đầu đến đỉnh đích ⇒ có thể sử dụng cho cả đồ thị tường minh và không tường minh.

2.2.2. Không gian bộ nhớ

- Dijkstra: lưu toàn bộ đỉnh của đồ thị trong bộ nhớ ngay từ lúc bắt đầu
- UCS: chỉ lưu đỉnh bắt đầu khi bắt đầu tìm kiếm và dừng mở rộng bộ nhớ khi đã tìm được đỉnh đích

⇒ Dijkstra thường sẽ tốn nhiều không gian lưu trữ hơn so với UCS vì UCS có thể chỉ lưu một số đỉnh của đồ thị.

2.2.3. Thời gian chạy

Thuật toán Dijkstra thường sẽ tốn nhiều thời gian chạy hơn UCS vì cần phải lưu trữ nhiều hơn khi bắt đầu.

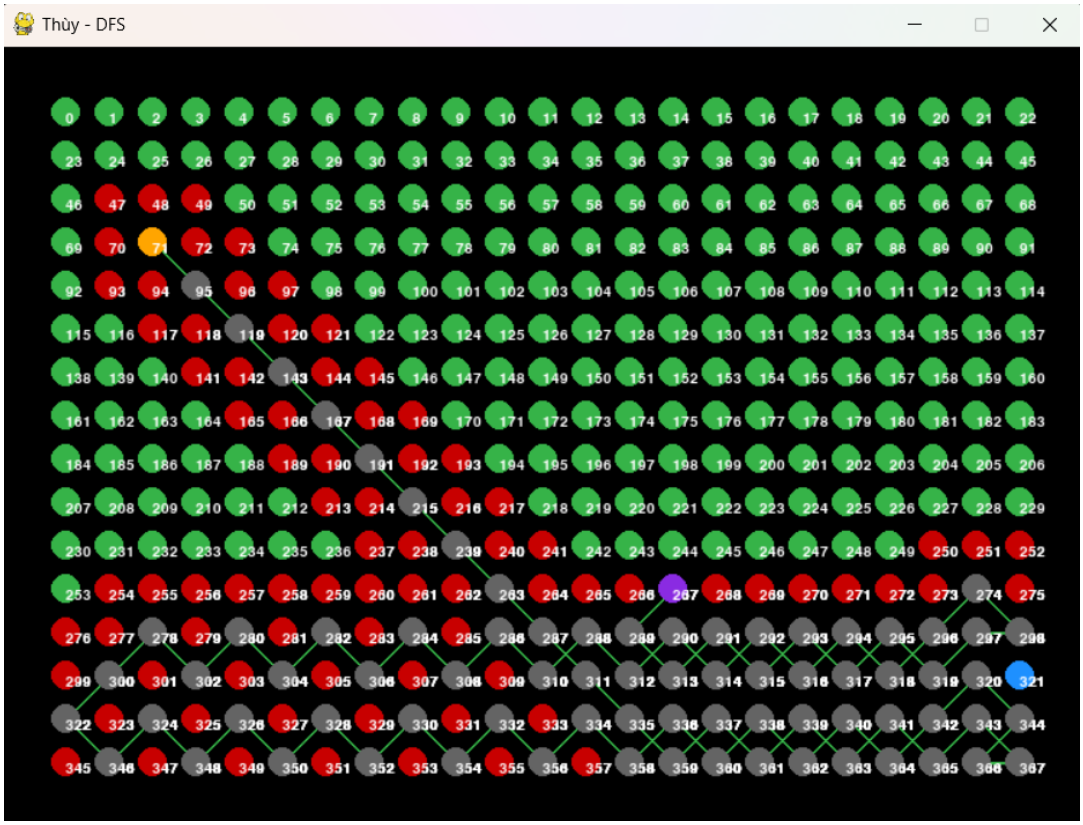
3. Cài đặt

Cài đặt 4 thuật toán tìm kiếm có đỉnh bắt đầu là 71, đỉnh đích là 267.

Chi phí đi từ một node đến node khác sẽ bằng khoảng cách Euclid giữa hai node. Vì thế nên chi phí đi đến các node ngang, dọc là như nhau và chi phí đi đến các node chéo là như nhau.

Heuristic của một node là khoảng cách Euclid từ node đó đến node đích.

3.1. DFS

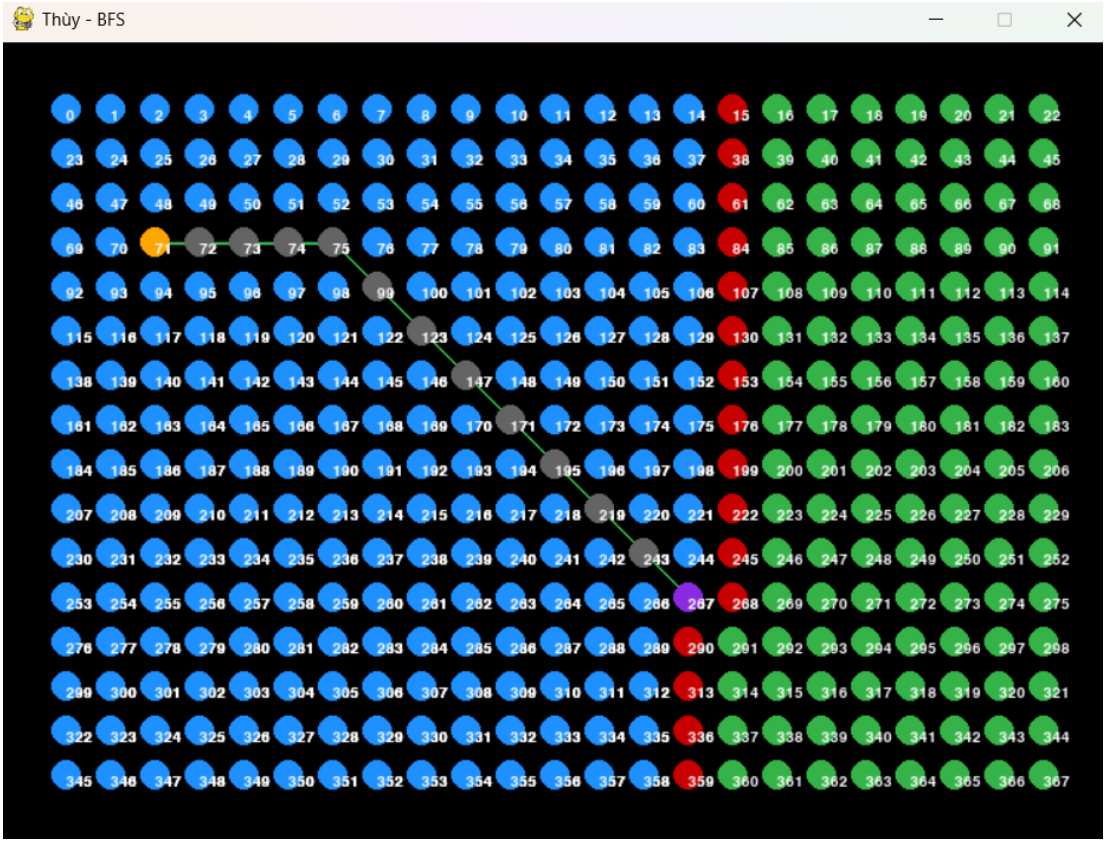


Từ hình trên, dễ thấy thuật toán DFS sẽ ưu tiên tìm đường đi sâu nhất, luôn đi đến vị trí các node con của node vừa kiểm tra để tiếp tục kiểm tra. DFS không cân nhắc đến chi phí đường đi.

Nhận xét:

- Thuật toán DFS có thời gian tìm kiếm dài nếu node bắt đầu và node đích không nằm trên đường thuận lợi để tìm kiếm. Ngược lại, nếu nằm trên đường tìm kiếm thuận lợi, DFS sẽ cho ra đường đến node đích khá nhanh (ví dụ: node bắt đầu là 71, node đích là 263).
- Hầu hết các trường hợp, DFS sẽ không cho con đường tối ưu đến node đích vì không cân nhắc đến chi phí đường đi.

3.2. BFS

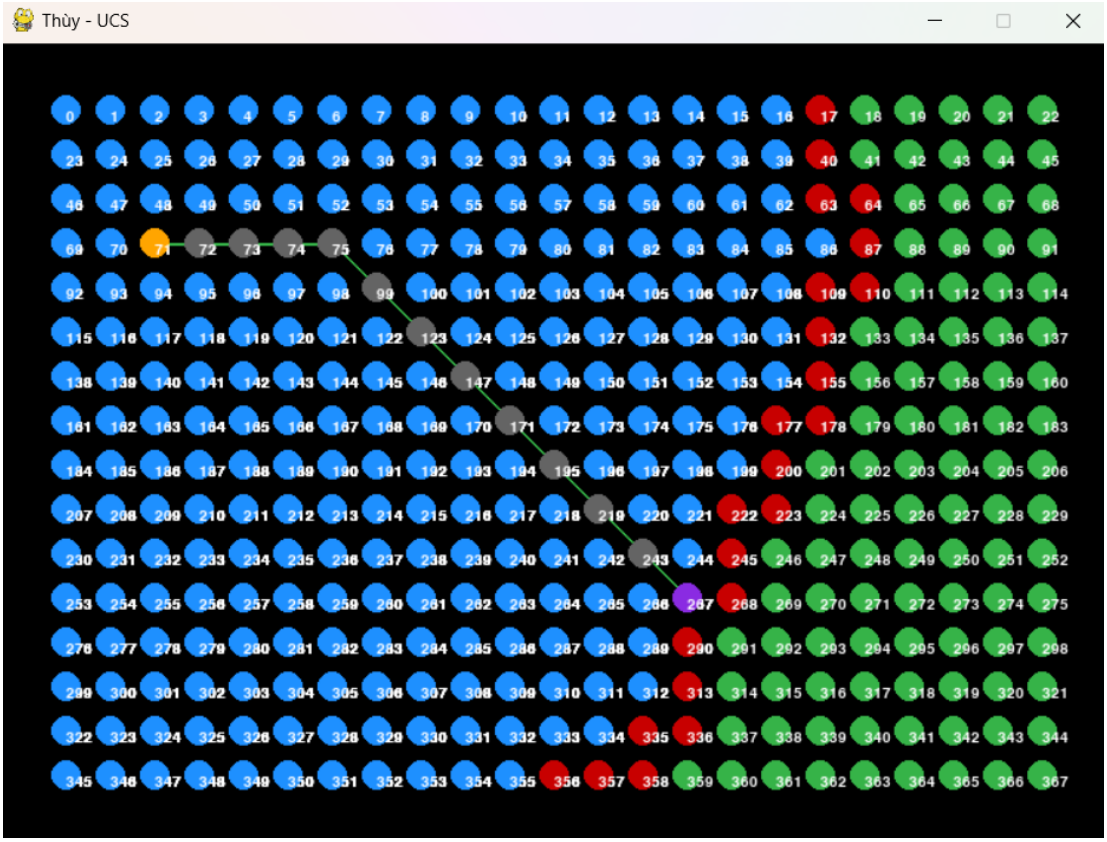


Từ hình trên, dễ thấy thuật toán BFS sẽ ưu tiên tìm kiếm theo chiều rộng, mở các node con ở cùng độ sâu trước. Vùng tìm kiếm của BFS mở rộng có dạng như một hình vuông (hoặc hình chữ nhật) và BFS không cân nhắc đến chi phí đường đi.

Nhận xét:

– Thuật toán BFS có thời gian tìm kiếm khá dài vì cần phải mở rộng nhiều node.
– Vì chi phí đường đi giữa các node được cài đặt như mục 3. nên BFS đưa ra được đường đi tối ưu đến node đích.

3.3. UCS

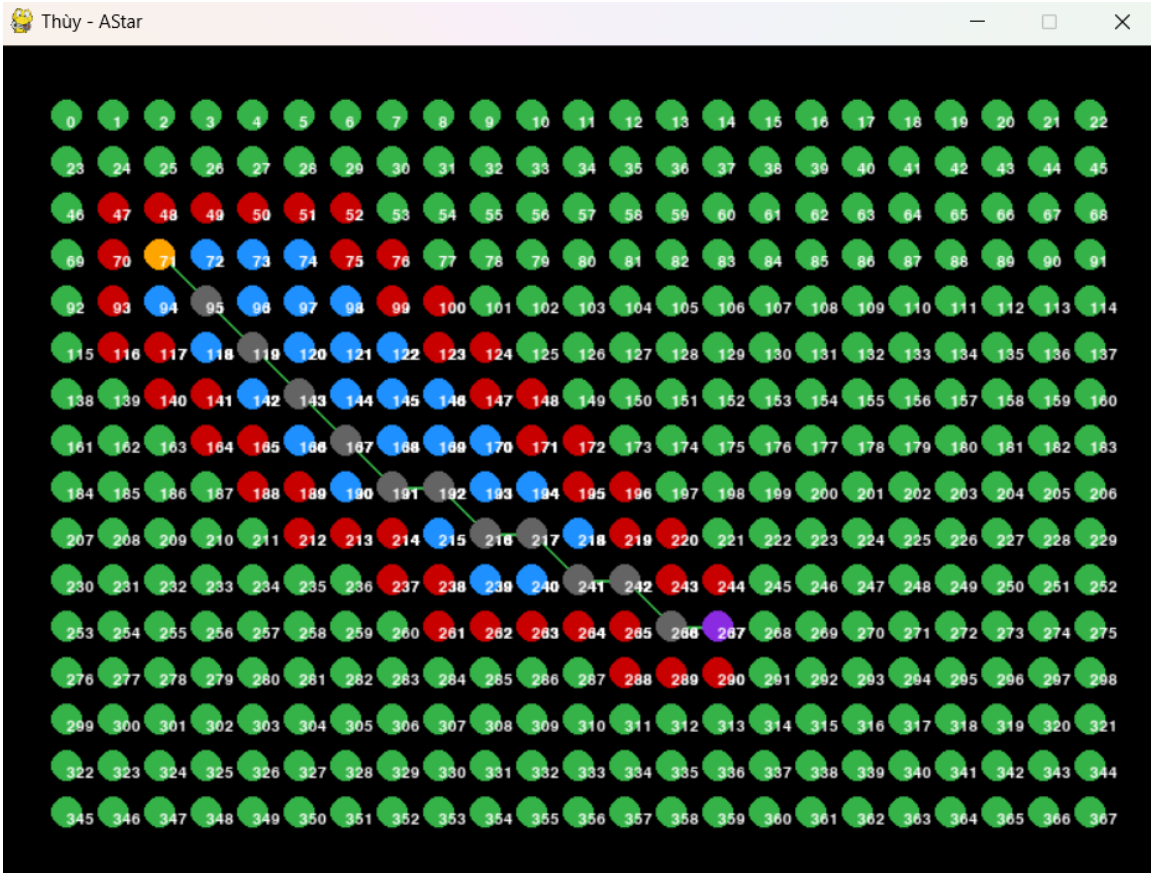


Thuật toán UCS sẽ ưu tiên tìm kiếm dựa trên chi phí đường đi ngắn nhất. Vì chi phí đường đi giữa các node được cài đặt như mục 3. nên dễ thấy trên hình, vùng tìm kiếm của thuật toán UCS mở rộng có dạng như một hình tròn.

Nhận xét:

- Thuật toán UCS có thời gian tìm kiếm khá dài vì cần phải mở rộng nhiều node.
- Vì UCS tìm kiếm dựa trên chi phí đường đi nên UCS đưa ra được đường đi tối ưu.

3.4. A*



Thuật toán A* sẽ ưu tiên tìm kiếm dựa trên chi phí đường đi và heuristics/ước tính khoảng cách đến đích. Trên hình, dễ thấy vùng tìm kiếm của A* mở rộng hướng đến node đích.

Nhận xét:

– Thuật toán A* có thời gian tìm kiếm ngắn so với các thuật toán trước vì A* mở rộng vùng tìm kiếm hướng đến node đích.

– Vì A* tìm kiếm dựa trên chi phí đường đi và heuristics nên A* đưa ra được đường đi tối ưu.

4. Tự đánh giá

✓ Tìm hiểu và trình bày các thuật toán trên đồ thị - 4đ

✓ So sánh các thuật toán với nhau - 2đ

✓ Cài đặt được các thuật toán - 3đ

⇒ Tổng điểm tự đánh giá: 9đ.

TÀI LIỆU THAM KHẢO

[1] Đức, N.N. (2021). *Bài toán tìm kiếm* [Slideshow]. Truy cập 07/10/2022, từ https://cloud.ducnn.com/s/RZSiwZbLeMTD32W?dir=undefined&openfile=128598.

[2] Đức, N.N. (2021). *Tìm kiếm mù* [Slideshow]. Truy cập 07/10/2022, từ https://cloud.ducnn.com/s/RZSiwZbLeMTD32W?dir=undefined&openfile=131571.

[3] Đức, N.N. (2021). *Tìm kiếm có định hướng* [Slideshow]. Truy cập 07/10/2022, từ https://cloud.ducnn.com/s/RZSiwZbLeMTD32W?dir=undefined&openfile=133787.

[4] Russell, S.J. & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach, Global Edition, 4th Edition*. London: Pearson.

[5] Levine, J. (2017). *A* Search*. [Video]. Youtube. Truy cập 06/10/2022, từ https://www.youtube.com/watch?v=6TsL96NAZCo.

[6] Levine, J. (2018). *Depth First Search*. [Video]. Youtube. Truy cập 06/10/2022, từ https://www.youtube.com/watch?v=h1RYvCfuoN4.

[7] Levine, J. (2018). *Breadth First Search – Part 1*. [Video]. Youtube. Truy cập 06/10/2022, từ https://www.youtube.com/watch?v=1wu2sojwsyQ.

[8] Levine, J. (2018). *Breadth First Search – Part 2*. [Video]. Youtube. Truy cập 06/10/2022, từ https://www.youtube.com/watch?v=n3fPL9q_Nyc.

[9] Levine, J. (2017). *Uniform Cost Search*. [Video]. Youtube. Truy cập 06/10/2022, từ https://www.youtube.com/watch?v=dRMvK76xQJI.

[10] MdRafiAkhtar. (2022). *Search Algorithm in AI*. Truy cập 05/10/2022, từ https://www.geeksforgeeks.org/search-algorithms-in-ai/?ref=lbp.

[11] Wu, G. (2021). *Comparison Between Uniform-Cost Search and Dijkstra's Algorithm*. Truy cập 05/10/2022, từ https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras.