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_1

Sinh viên thực hiện : Phan Nhật Triều

Mã số sinh viên : 20120605

TP Hồ Chí Minh, tháng 03 năm 2023

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:

  i. Trạng thái bắt đầu (initial state).

  ii. Mô tả các hành động (action) có thể thực hiện.

  iii. 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.

  iv. Kiểm tra đích (goal test): xác định một trạng thái có là trạng thái đích.

  v. 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

❖ Có 2 loại bài toán tìm kiếm là **uninformed search** và **informed search**

- *Uniformed search:*
    - không được cung cấp manh mối về mức độ gần của các trạng thái với mục tiêu
    - 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
    - Bao gồm các thuật toán tiêu biểu như: DFS, BFS, UCS, Depth-first search, Dijkstra,…

- *Informed search:*
    - 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
    - Bao gồm các thuật toán tiêu biểu như: Greedy best-first search, A* search, Bidirectional heuristic search,…

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

a. DFS

- **Ý tưởng chung:** việc triển khai DFS tiêu chuẩn là đặt mỗi đỉnh của đồ thị vào một trong 2 loại: ***visited*** và ***not visited***

  + Mục tiêu của thuật toán là đánh dấu mỗi đỉnh là đã ***visited*** trong khi tránh các chu trình

  + Thuật toán DFS làm việc như sau:
    - Bắt đầu bằng cách đặt bất kì của đỉnh nào lên đỉnh của stack
    - Lấy ra đỉnh của stack và đánh dấu đã visited

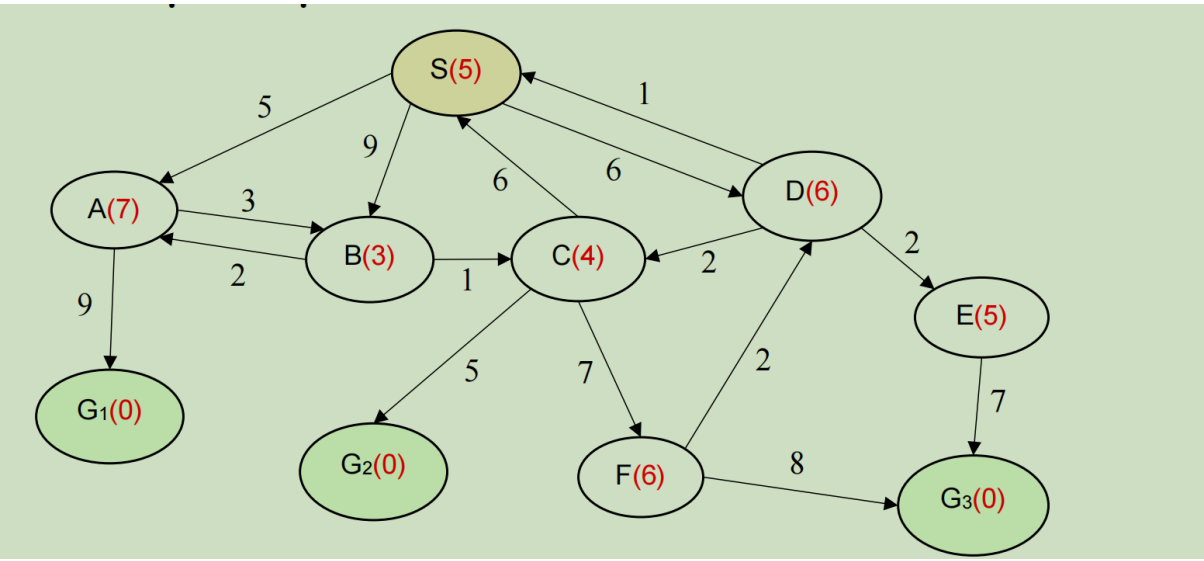
- Tạo ra danh sách của các đỉnh kề với đỉnh đang xét. Đỉnh nào chưa được visited thì thêm vào stack
- Lặp lại bước 2 và bước 3 cho đến khi tìm thấy mục tiêu hoặc stack trống.

- **Mã giải**

```
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
```

- **Đánh giá thuật toán**
  - Complete: (có nếu với đồ thị hữu hạn)
  - Time complexity: $O(b^m)$
  - Space complexity: O(bm)

    (b là hệ số phân nhánh, m là độ dài tối đa của cây tìm kiếm)
  - Optimal: Không
- **Minh họa:** Ta có đồ thị:

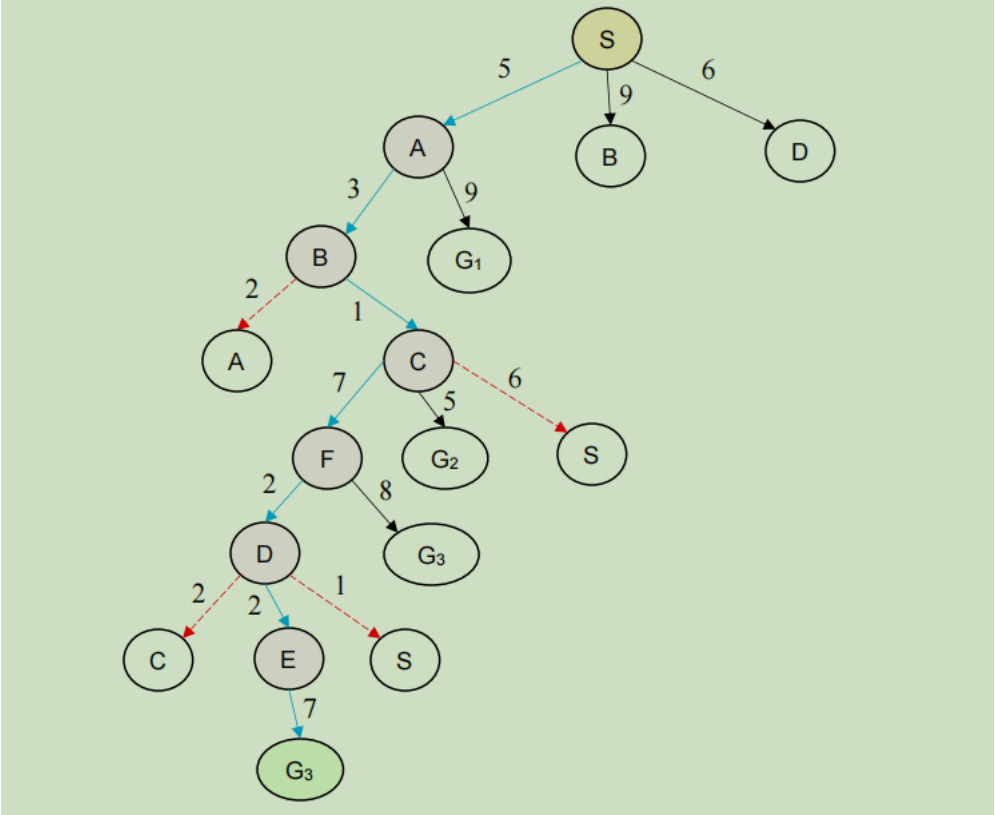


+ Các đỉnh được thăm lần lượt theo thứ tự là: S → A → B → C → F → D → E → G3

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

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

+ Cây tìm kiếm:



b. **BFS**

- **Ý tưởng chung:** việc triển khai thuật toán BFS đặt mỗi đỉnh trong đồ thị vào 2 trạng thái: visited và not visited

  + Mục tiêu đánh dấu mỗi đỉnh là đã thăm tránh các chu kì

  + Thuật toán BFS làm việc như sau:

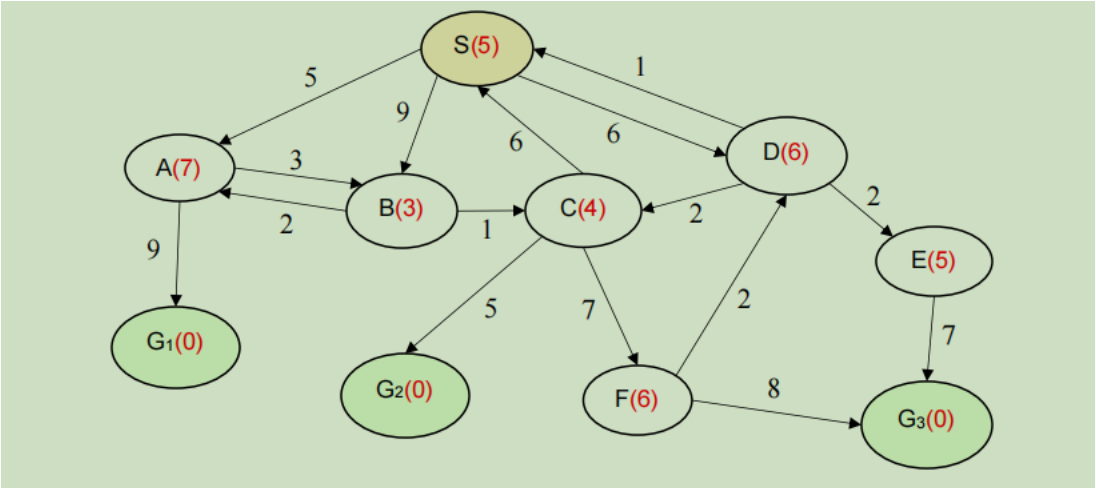
    - Đặt bất kì một đỉnh nào của đồ thị vào 1 hàng đợi queue
    - Lấy ra đỉnh được đưa vào sớm nhất so với các đỉnh khác trong queue theo cơ chế FIFO và đánh dấu đã **visited**
    - Tạo ra một danh sách các đỉnh kề với đỉnh đang xét hiện tại và đẩy vào hàng chờ nếu đỉnh đó chưa được **visited.**
    - Lặp lại bước 2 và 3 cho đến khi tìm được đích cuối cùng học hàng đợi queue trống.

- **Mã giải**

```
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
```

- **Đánh giá thuật toán**
  - Complete: Có
  - Time complexity: $O(b^d)$
  - Space complexity: $O(b^d)$

    (b là hệ số phân nhánh, d là độ sâu nông nhất cây tìm kiếm)
  - Optimal : Có trong trường hợp các chi phí đến các đỉnh là như nhau
- **Minh họa:** Ta có đồ thị sau:

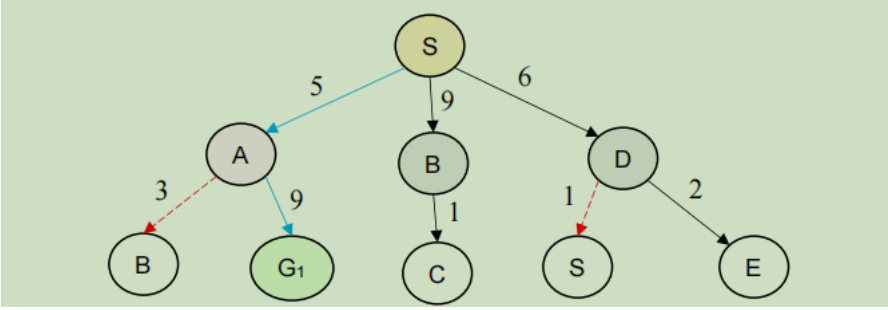


+ Các đỉnh được thăm theo thuật toán BFS: S → A → B → D → G1

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

+ Chi phí đường đi: $5 + 9 = 14$

+ Cây tìm kiếm:



c. UCS

- **Ý tưởng chung:** UCS(Uniform-Cost Search) là một biến thể của thuật toán Dijikstra. Từ trạng thái ban đầu chúng ta lần lượt duyệt qua các trạng thái liền kề chưa được duyệt qua (**not visited**) và chọn ra trạng thái có chi phí nhỏ nhất để duy chuyển từ trạng thái hiện tại đến trạng thái đó và đánh dấu đã **visited.** Việc lặp lại cho việc tìm kiếm này sẽ kết thúc khi tìm thấy trạng thái cần tìm.

- **Mã giải**

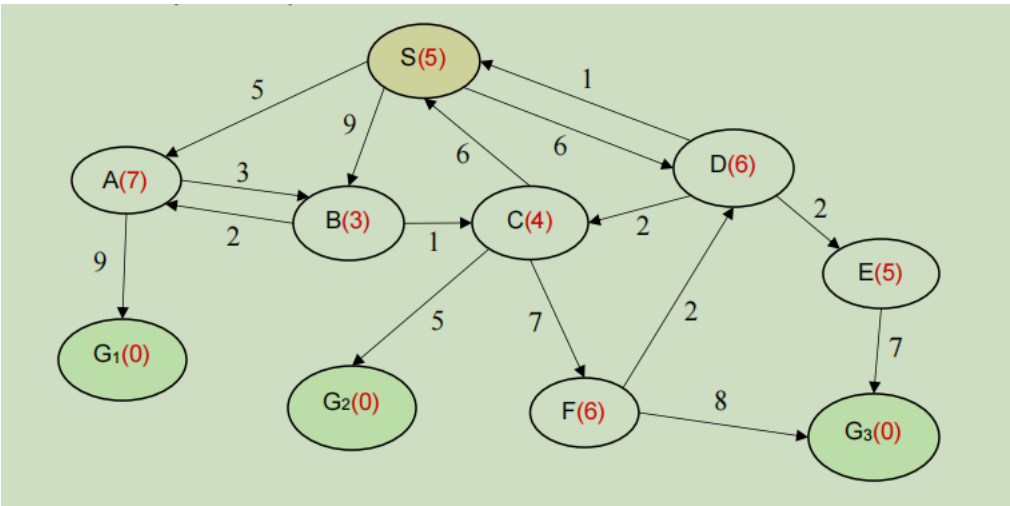
```
function UNIFORM-COST-SEARCH(problem) returns a solution node or
failure
    node ← NODE(STATE=problem.INITIAL)
    frontier ← a priority queue ordered by PATH-COST, with node
as an element
```

```
    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
```

```
    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
```

- **Đánh giá thuật toán**
  - Complete: Có (nếu không gian trạng thái hữu hạn và không có vòng lặp với chi phí bằng không)
  - Time complexity: $O(b^{1+[C^*/e]})$
  - Space complexity: $O(b^{1+[C^*/e]})$

    (b là hệ số phân nhánh, C* chi phí của giải pháp tối ưu là , e là chi phí của mỗi bước đi)
  - Optimal: Có (nếu không có chi phí âm)
- **Minh họa:** Ta có đồ thị

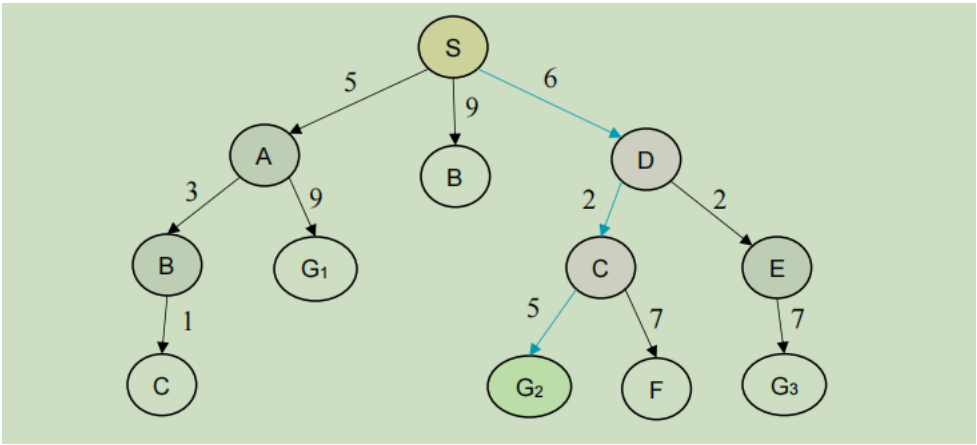


+ Các đỉnh được thăm theo thuật toán UCS lần lượt là: $S \rightarrow A \rightarrow D \rightarrow B \rightarrow C \rightarrow E \rightarrow G2$

+ Đường đi từ đỉnh bắt đầu đến đích: $S \rightarrow D \rightarrow C \rightarrow G2$.

+ Chi phí đường đi: $6 + 2 + 5 = 13$

+ Cây tìm kiếm:



d.* A

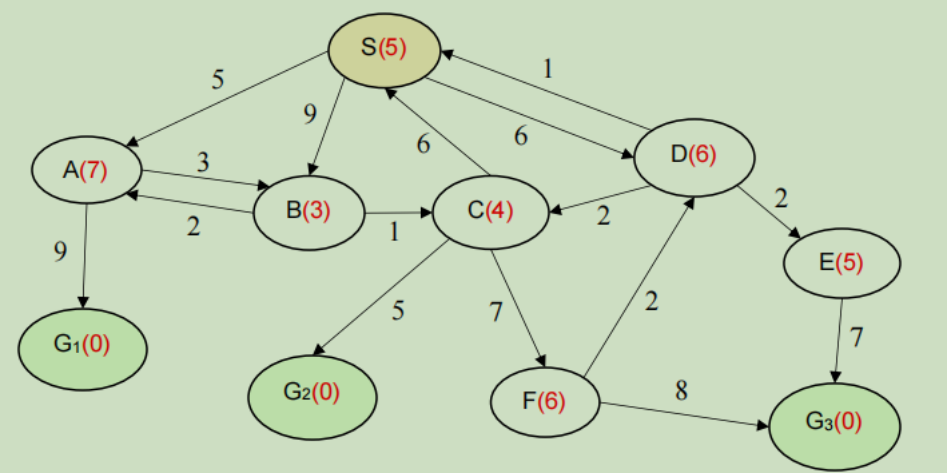
- **Ý tưởng chung**
  - Tích hợp heuristic vào quá trình tìm kiếm
  - Tránh các đường đi có chi phí lớn
  - Hàm đánh giá của A* search: $f(n) = g(n) + h(n)$
  - Quá trình diễn ra tương tự như UCS ngoại trừ $f(n) = g(n) + h(n)$ thay vì $f(n) = g(n)$
- **Mã giải:**

+ Các đỉnh được thăm lần lượt theo thứ tự:

```
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.
```

- **Đánh giá thuật toán**
  - Complete: Có (nếu chi phí ϵ của mỗi bước đi thì ϵ>0, và đồ thị hữu hạn)
  - Time complexity: $O(b^d)$
  - Space complexity: $O(b^d)$

    (b là hệ số phân nhánh, d là độ sâu của giải pháp)
  - Optimal: Có (nếu heuristic hợp lí và nhất quán)
- **Minh họa:** Ta có đồ thị

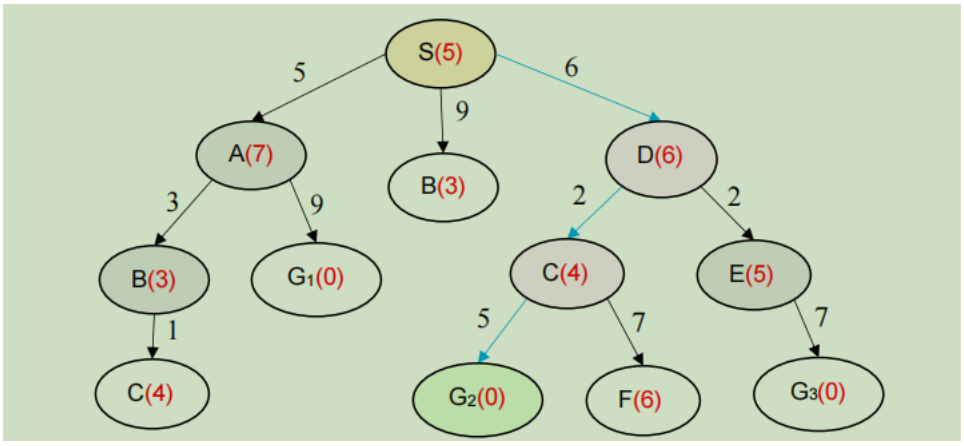


+ Các đỉnh được thăm theo thuật toán A* lần lượt là: $S(5) \rightarrow A(12) \rightarrow B(11) \rightarrow D(12) \rightarrow C(12) \rightarrow E(13) \rightarrow G2(13)$

+ Đường đi từ đỉnh bắt đầu đến đích: $S(5) \rightarrow D(12) \rightarrow C(12) \rightarrow G2(13)$

+ Chi phí đường đi: $6 + 2 + 5 = 13$

+ Cây tìm kiếm:



2. So sánh:

2.1. Sự khác biệt giữa 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: $O(b_{1+\lfloor C_*/\epsilon \rfloor})$
- Greedy: $O(b^m)$
- A*: $O(b^d)$

2.1.4. 2.1.4.Độ phức tạp không gian

- UCS: $O(b_{1+\lfloor C_*/\epsilon \rfloor})$
- 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 âm)
- 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. Sự khác biệt giữa UCS, 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

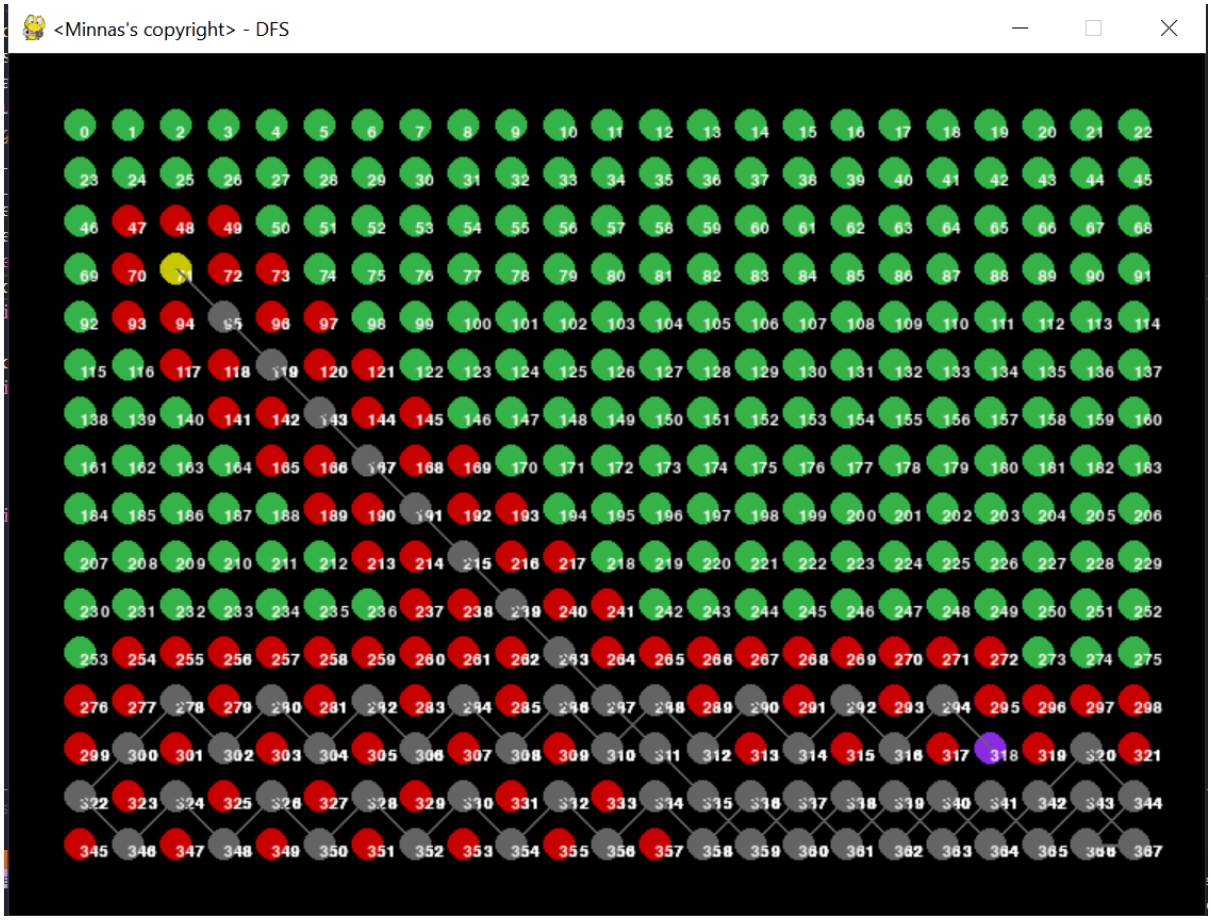
- Dijkstra: lưu toàn bộ đỉnh của đồ thị trong bộ nhớ ngay từ lúc bắt đầu
- UCS: chỉ lưu đỉnh bắt đầu khi bắt đầu tìm kiếm và dừng mở rộng bộ nhớ khi đã tìm được đỉnh đích
  - Dijkstra thường sẽ tốn nhiều không gian lưu trữ hơn so với UCS vì UCS có thể

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.

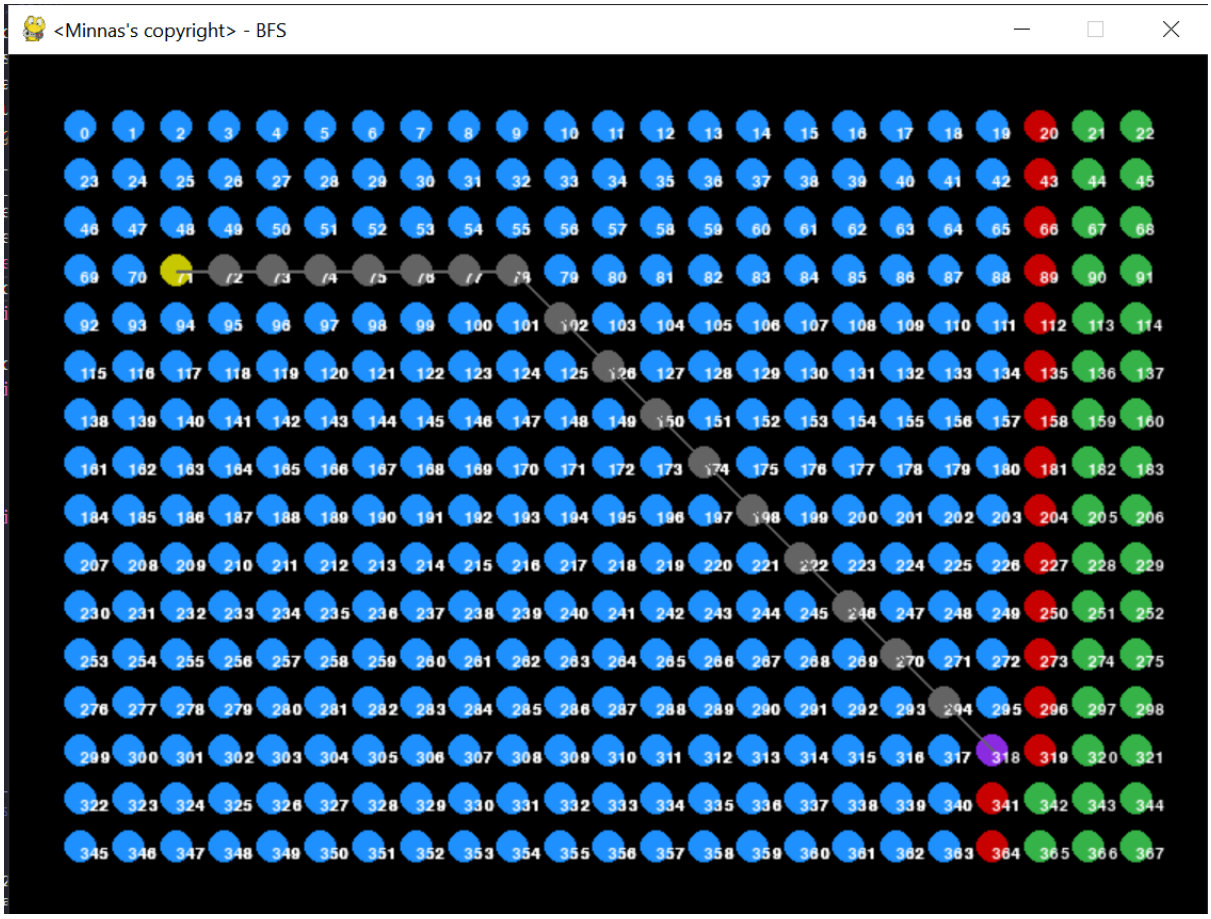
3. Cài đặt

3.1. DFS



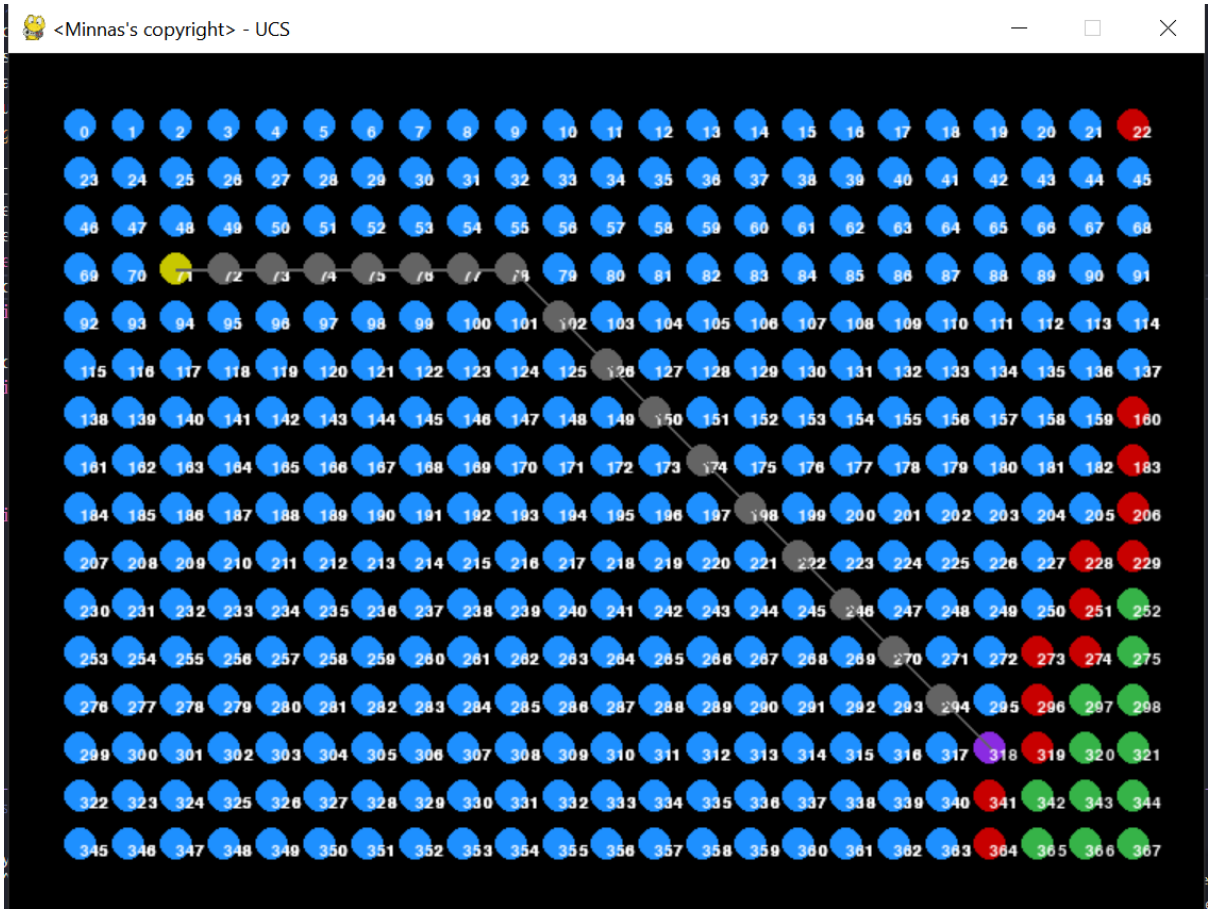
- Từ hình trên, dễ thấy thuật toán DFS sẽ ưu tiên tìm đường đi sâu nhất, luôn đi đến vị trí các node con của node vừa kiểm tra để tiếp tục kiểm tra. DFS không cân nhắc đến chi phí đường đi.
- Nhận xét: Thuật toán DFS có thời gian tìm kiếm dài nếu node bắt đầu và node đích không nằm trên đường thuận lợi 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 311).

3.2. BFS



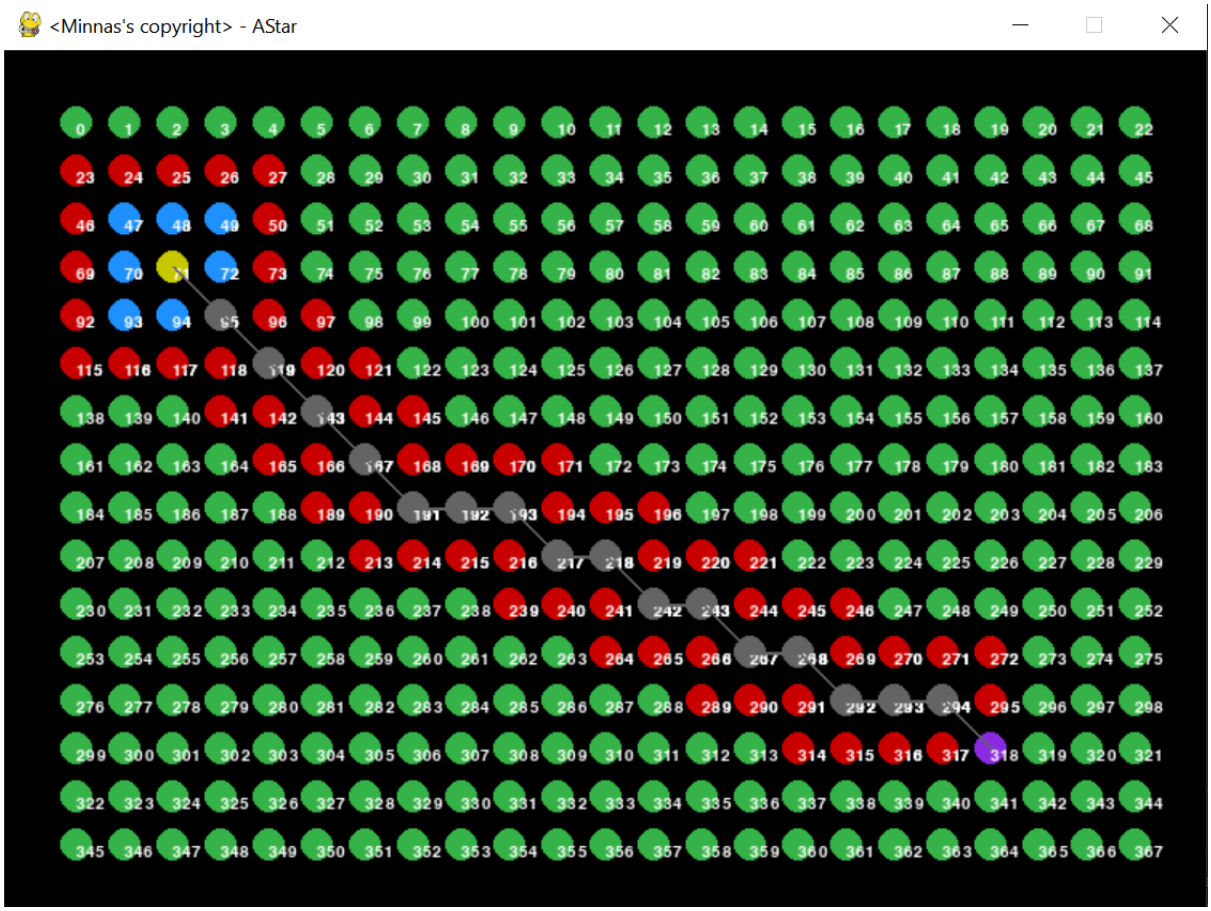
- Từ hình trên, dễ thấy thuật toán BFS sẽ ưu tiên tìm kiếm theo chiều rộng, mở các node con ở cùng độ sâu trước. Vùng tìm kiếm của BFS mở rộng có dạng như một hình vuông (hoặc hình chữ nhật) và BFS không cân nhắc đến chi phí đường đi.
- Nhận xét: Thuật toán BFS có thời gian tìm kiếm khá dài vì cần phải mở rộng nhiều node, tuy nhiên nó sẽ cho chúng ta đường đi với chi phí tối ưu

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. Thuật toán tìm kiếm khác

❖ ***Iterative Deepening A* algorithm (IDA*) – Artifical Intelligence***

4.1. Giới thiệu

- Iterative Deepiening A*(IDA*) là một phương thức duyệt đồ thị và tìm kiếm đường đi, nó có thể xác định đường đi ngắn nhất trong một đồ thị có trọng số giữ điểm xuất phát xác định trước và bất cứ điểm nào trong nhóm các mục tiêu. Nó là một tìm kiếm

theo chiều sâu đào sâu lặp đi lặp lại áp dụng ý tưởng của thuật toán tìm kiếm A* về việc sử dụng một hàm heuristic để đánh giá chi phí còn lại để đi đến đích.

- IDA* là một phiên bản giới hạn bộ nhớ của A*. Thực hiện tất cả các hoạt động mà A* thực hiện và có tính năng tối ưu cho việc xác định đường đi ngắn nhất, nhưng nó chiếm ít bộ nhớ hơn
- IDA* sử dụng hàm heuristic để chọn ra node kề để khám phá và dừng ở độ sâu nào. Trái ngược với Iterative Deepending DFS sử dụng độ sâu đơn giản để xác định khi nào kết thúc quá trình lặp hiện tại và tiếp tục với độ sâu cao hơn
- IDA* sử dụng f-score được gọi là "threshold" thứ sẽ tăng lên khi mà đạt đến nút có f-score lớn hơn và thuật toán bắt đầu lại từ đầu cho đến độ sâu mới.

4.2. Ý tưởng chung

- Bước 1: (khởi tạo) đặt nút gốc làm nút hiện tại và tìm f-score
- Bước 2: (thiết lập threshold) đặt giới hạn chi phí làm threshold cho một nút
- Bước 3: (mở rộng nút) mở rộng node hiện tại cho các node con và tìm f-score
- Bước 4: (pruning) nếu có bất kì node con nào có f-score> threshold, tỉa node đó bởi chi phí được coi là lớn hơn so với ngưỡng.
- Bước 5: (đích) nếu là đỉnh mục tiêu thì trả về đường dẫn và kết thúc chương trình
- Bước 6: (cập nhật threshold) nếu node mục tiêu không được tìm thấy thì lập lại bước 2 và cập nhật threshold nếu giá trị của node bị tỉa đi nhỏ nhất

4.3. *Mã giải*

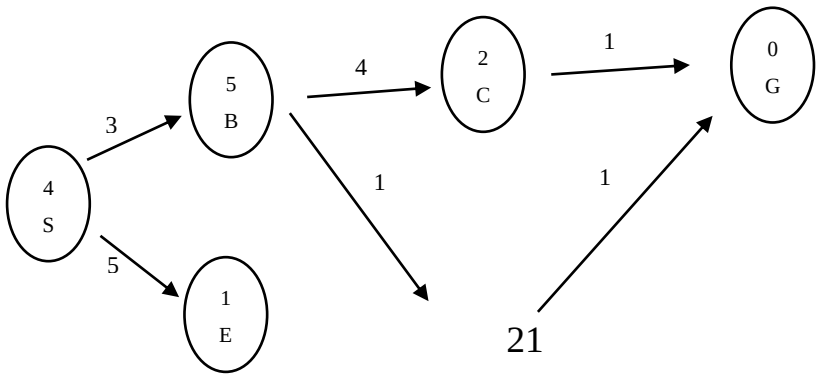
```
path              current search path (acts like a stack)
node              current node (last node in current path)
g                 the cost to reach current node
f                 estimated cost of the cheapest path (root..node..goal)
h(node)           estimated cost of the cheapest path (node..goal)
cost(node, succ)  step cost function
is_goal(node)     goal test
successors(node)  node expanding function, expand nodes ordered by g + h(node)
ida_star(root)    return either NOT_FOUND or a pair with the best path and its cost

procedure ida_star(root)
    bound := h(root)
    path := [root]
    loop
        t := search(path, 0, bound)
        if t = FOUND then return (path, bound)
        if t = ∞ then return NOT_FOUND
        bound := t
    end loop
end procedure

function search(path, g, bound)
    node := path.last
    f := g + h(node)
    if f > bound then return f
    if is_goal(node) then return FOUND
    min := ∞
    for succ in successors(node) do
        if succ not in path then
            path.push(succ)
            t := search(path, g + cost(node, succ), bound)
            if t = FOUND then return FOUND
            if t < min then min := t
            path.pop()
        end if
    end for
    return min
end function
```

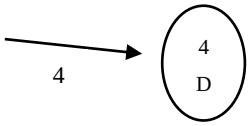
4.4. *Đánh giá thuật toán*

- *Complete:* Có ở đồ thị hữu hạn
- *Time complexity:* $O(b^d)$
- *Space complexity:* O(b*d)

  (b là hệ số phân nhánh, d là độ sâu giải pháp đầu tiên)
- *Optimal*: Có vì bản chất của nó là tìm kiếm dựa trên hàm đánh giá heuristic.

4.5. **Minh họa:** Ta có đồ thị: Điểm xuất phát S và điểm đích là G





STT	Nút được mở rộng	Tập biên O
Threshold=4		
0		S(4)
1	S	
Threshold = 8		
0		S(4)
1	S	Bs(8), Es(6)
2	B	Db(8),Es(6)
3	D	G(4),Es(6)
	G	Đích

- Đường đi từ đỉnh bắt đầu đến đích: S → B → D →G.
- Chi phí: 3+ 1+ 1=5

5. 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
- ✓ Tìm hiểu về thuật toán khác -1đ

⇨ Tự đánh giá 10đ

Tài liệu tham khảo

[1] S. J. R. a. P. Norvig, Artificial Intelligence - A modern approach - Fourth Edition - Global Edition.

[2] "Depth First Search (DFS)," [Online]. Available: https://www.programiz.com/dsa/graph-dfs.

[3] "Programiz," [Online]. Available: https://www.programiz.com/dsa/graph-bfs.

[4] "Stack Exchange," [Online]. Available: https://cs.stackexchange.com/questions/56176/a-graph-search-time-complexity.

[5] "Comparison Between Uniform-Cost Search and Dijkstra's Algorithm," [Online]. Available: https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras.

[6] "A* Search Algorithm," [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/.

[7] "GeeksforGeeks," [Online]. Available: https://www.geeksforgeeks.org/iterative-deepening-a-algorithm-ida-artificial-intelligence/.

[8] "Iterative deepening A*," [Online]. Available: https://en.wikipedia.org/wiki/Iterative_deepening_A*.

[9] "why is IDA* optimal, even without monotonicity?," [Online]. Available: https://cs.stackexchange.com/questions/16477/why-is-iterative-deepening-a-optimal-even-without-monotonicity.